FACILITATING DATA COMMUNICATION BETWEEN SUPERVISED MODULES IN A DISTRIBUTED UNMANNED AERIAL VEHICLE SOFTWARE ARCHITECTURE

By Matthew Amato-Yarbrough

A Thesis

Submitted in Partial Fulfillment

of the Requirement for the Degree of

Master of Science

in Computer Science

Northern Arizona University

December 2023

Approved:

Paul G. Flikkema, Ph.D., Chair

Michael W. Shafer, Ph.D.

Michael Gowanlock, Ph.D.

ABSTRACT

FACILITATING DATA COMMUNICATION BETWEEN SUPERVISED MODULES IN A DISTRIBUTED UNMANNED AERIAL VEHICLE SOFTWARE ARCHITECTURE

MATTHEW AMATO-YARBROUGH

This thesis addresses the challenges and inefficiencies associated with traditional very high frequency (VHF) radio tag-based wildlife tracking methods. Researchers have historically relied on manual tracking, involving extensive travel to remote locations and the use of handheld radios for signal detection. This approach consumes significant time and resources, hampering research efforts to understand animal behavior, demographics, and habitat utilization. To mitigate these challenges, this thesis proposes a robust and versatile distributed software architecture that leverages modern technologies to enhance the precision and effectiveness of VHF radio tagging data collection. This architecture is specifically tailored to operate on companion computers with hardware limitations, which are deployed on unmanned aerial vehicles (UAVs). UAVs offer a novel solution that not only enhances tracking efficiency but also overcomes obstacles related to terrain affordance and minimizing disturbances to local wildlife. The findings of this thesis will contribute to the broader field of wildlife research by demonstrating the feasibility and benefits of UAV-based VHF radio tagging, paving the way for more accurate and efficient data collection in the future.

ACKNOWLEDGEMENTS

First, I want to truly thank my advisor, Dr. Paul Flikkema, who introduced me to the wonders of conducting research and the field of signal processing. Paul, you did not have to take me on as a new advisee, but you did. You have been a constant source of wisdom and knowledge over these past several years. Thank you.

I would also like to thank Dr. Michael Shafer for his invaluable support throughout my research endeavors. Mike, I am incredibly thankful for how you have always been there when I was seeking advice or needed answers.

I express my gratitude to the professors who have imparted their knowledge to me during my undergraduate and graduate studies, as well as to the family and friends who have been my support system during my academic career.

Lastly, I gratefully acknowledge the financial support that I received from the National Science Foundation through grant number 2104570, as well as the financial support I received from the School of Informatics, Computing, and Cyber Systems (SICCS) at Northern Arizona University.

TABLE OF CONTENTS

Page

Page

Page

Page

LIST OF TABLES

Table Page

Page

Page

Page

LIST OF FIGURES

Page

Page

Page

Page

Page

Chapter 1

INTRODUCTION

1.1 Introduction

Wildlife tracking, particularly through very high frequency (VHF) radio tags, has long been integral to understanding animal behavior and habitat utilization. However, conventional methods relying on manual tracking and handheld radios have posed substantial challenges, consuming time and resources while impeding comprehensive research efforts. To address these inefficiencies, this thesis aims to introduce a robust and adaptable distributed software architecture tailored for companion computers on unmanned aerial vehicles (UAVs). This architecture seeks to enable UAV VHF radio tagging data collection by leveraging modern technologies to enhance precision and effectiveness, while also mitigating limitations related to terrain accessibility and reducing disturbances to wildlife.

The core of this thesis lies in the proposed distributed software architecture, centering on the implementation of the Unmanned Aerial Vehicle - Radio Telemetry (UAV-RT) software packages. These packages form the backbone of the architecture, managing real-time data processing, transmission, and efficient data management. The thesis describes the role of each package, from the uavrt_supervisor as the central control hub to the uavrt_connection facilitating communication with the flight controller and defining essential message structures through the uavrt_interfaces package. Notably, the implementation of the channelizer and uavrt_detection packages, rooted in the research of Dr. Michael Shafer and Dr. Paul Flikkema, stand as the core component for real-time pulse detection.

The significance of this project lies in demonstrating the real-time processing capability of the UAV-RT system in tracking animal movements through UAV-based VHF radio tags. Ultimately, the findings and methodologies outlined in this thesis mark an advancement in wildlife monitoring technology, offering feasible and advantageous alternatives to traditional methods for more precise and efficient data collection in the realm of environmental research and wildlife tracking.

1.2 System overview

Figure 1.1 describes the architecture of the UAV-RT system and illustrates the information and control flow among its components. It includes a user actor and four main components: the animal, the ground control station, the unmanned aerial vehicle, and the UAV-RT software packages. A user interacts with the ground control station that is comprised of a custom version of QGroundControl and a telemetry radio. Data is transmitted between custom QGroundControl and the unmanned aerial vehicle, which is attached to the VHF directional antenna and the companion computer. The companion computer is attached to a telemetry radio, a flight controller, and software defined radio. Additionally, the companion computer is responsible for housing the five packages of the UAV-RT software system: channelizer, uavrt_detection, uavrt_supervisor, uavrt_connection, and uavrt_interfaces. Detailed diagrams that showcase the processing of information within the UAV-RT system can be found in Section 2.10.

1.3 Literature review

Section 1.3 provides an analysis of literature in the field of UAV wildlife tracking, and explores the key concepts of the cited literature and the advancements made by the cited literature. The technical similarities and differences between the software

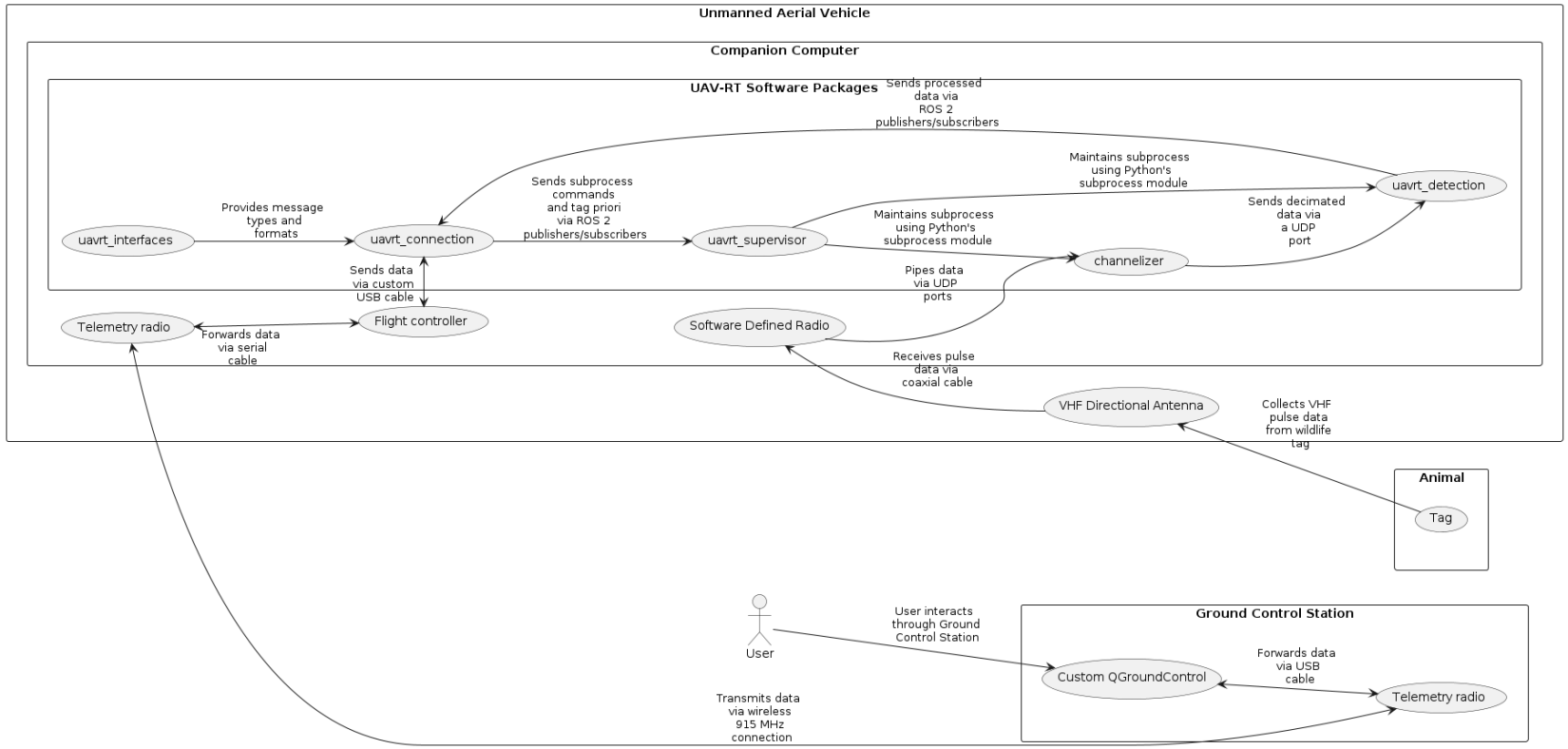

Figure 1.1: The flow of information and interactions between the components in the UAV-RT system.

and hardware components of the systems in the cited literature will also be presented. This analysis provides insight into the advancements that were made by the UAV-RT system and the software and hardware components discussed in this thesis.

Comparison criteria

The following criteria will be used to highlight the software and hardware similarities and differences of the systems proposed in the cited literature:

- Data processing: Whether systems perform real-time processing using the onboard computer and/or perform post-processing on the ground.
- User interface: The user interface for operators or researchers to interact with the collected data, onboard computer, and UAV.
- Communication systems and flight controller: The communication modules used by the system for data transfer, possibly using radio frequency, satellite, or other wireless technologies. This also includes the system's flight controller.
- Sensors and VHF transceivers: The various sensors, such as antennas, a GPS, cameras, accelerometers, and magnetometer, to collect data. This also includes VHF transceivers.
- Onboard computer: The onboard computer mounted to the UAV that was used for real-time data processing and/or decision-making.
- Inter-process communication (IPC): The methodology used for facilitating communication between processes running within the onboard computer.

Literature summary

Santos et al. [101] describes a novel approach to wildlife tracking using small UAVs equipped with low-cost off-the-shelf components, particularly a digital television receiver dongle as a software-defined radio (SDR). The primary innovation lies in leveraging the advancements in consumer technologies like digital television dongle receivers and hobbyist UAVs for the purpose of wildlife tracking. Below are the key innovations and advancements made by Santos et al. [101], as well as the software and hardware components used by Santos et al. [101] that meet the criteria listed in Subsubsection 1.3:

- Cost-effective system: The use of consumer-grade technologies, such as digital television dongles and a hobbyist UAV, contributes to a cost-effective solution compared to traditional telemetry methods.

- Multiple collar tracking: The proposed aerial system can track multiple collars simultaneously, providing real-time data that can be accessed after the UAV flight.

- Data processing: The system captures raw data during UAV flight, including global positioning system (GPS) data and downconverted signals from the SDR, and stores the data to a 64 GB Secure Digital (SD) card. Initial real-time signal processing occurs using the onboard computer. The primary data processing occurs post-flight, extracting stored data for analysis and generating a heat map of signal strength over the flight area.

- Sensors and VHF transceivers: The system incorporates a GPS unit, a television receiver dongle as an SDR unit, and an omnidirectional antenna. The television receiver dongle unit contains a Rafael Micro R820T tuner capable of

demodulating complex signals and a two-channel 8-bit analog to digital converter (ADC).

- Onboard computer: A Texas Instruments BeagleBone Black (BBB) board is used as the main control board, running a Linux operating system for data storage and processing.

Cliff et al. [9] discusses the development and implementation of a UAV system for the autonomous localization of radio-tagged wildlife. The system's innovations lie in the novel antenna design, real-time autonomous localization algorithm, and the application of greedy information gain planning. Below are the key innovations and advancements made by Cliff et al. [9], as well as the software and hardware components used by Cliff et al. [9] that meet the criteria listed in Subsubsection 1.3:

- Two-point phased array antenna design: The authors propose a novel approach using a two-point phased array antenna system for radio tag localization. This design involves mounting two monopole antennas on a carrier rail, carried by a UAV.

- Real-time autonomous localization: The system demonstrates real-time autonomous localization of live birds equipped with low-power radio tags. The proposed algorithm processes observations, updates a grid-based filter, and plans the next observation point online.

- Greedy information gain planning: The system incorporates a planning strategy based on greedy information gain, optimizing the selection of observation points to maximize the quality of data collected.

- Data processing: The received signal strength indicator (RSSI) Guidance Controller algorithm is executed on a ground-based laptop computer, where the

system processes observations online, updating the belief state and planning the next observation point in real-time. Initial real-time signal processing occurs using the onboard computer and an AGC circuit.

- User interface: The system uses Ascending Technologies' proprietary high-quality flight control and autonomous GPS waypoint-following software, designed to interface with the Falcon 8 UAV.

- Communication systems and flight controller: The system uses wireless communication between the UAV and a ground station. The ground station relays telemetry data and accepts control commands via USB. The wireless communication is handled using Digi XTend radio modems.

- Sensors and VHF transceivers: The primary sensor discussed is the two-point phased array antenna system for RF signal direction finding. A Radiometrix LMR1 receiver acts as the data link between the two-point phased array antenna and the onboard computer. A GPS unit is also used.

- Onboard computer: An ARM 32-bit Cortex-M3 microprocessor mounted to a custom miniaturized printed circuit board acts as the onboard computer.

- Inter-process communication: The RSSI Guidance Controller algorithm was implemented in Robot Operating System (ROS).

Consi et al. [10] details the integration of a radio tag telemetry receiver into a UAV based on a commercial radio-controlled (RC) model aircraft. This UAV system is designed for the autonomous localization of radio-tagged fish, specifically sturgeon, in freshwater lakes and rivers. The key innovation is the design of an airplane-style UAV for the purposing of automating the aerial monitoring of wildlife, particularly in aquatic environments. Below are the key innovations and advancements made by

Consi et al. [10], as well as the software and hardware components used by Consi et al. [10] that meet the criteria listed in Subsubsection 1.3:

- Customized UAV design: The authors detail the design of an Aero-Works Bravata airplane-style UAV tailored for efficient long-distance travel. The UAV was also modified to account for issues that arose during testing, such as radio frequency noise from the engine's ignition system and the impact of engine vibration on data logging. The solutions implemented include creating a Faraday cage and stabilizing components with epoxy.

- Underwater localization: The authors successfully demonstrated the localization of fish tags anchored underwater in an ice-covered lake.

- Data processing: The telemetry data and Pixhawk autopilot data are combined and processed using MATLAB after flights. Color surface plots are generated using the collected data and MATLAB.

- User interface: The Futaba model 14SG, 14-channel RC transmitter serves as the user interface for manual control, allowing a ground-based pilot to control the UAV. APM Mission Planner on a laptop serves as the ground station's user interface for setting up flight parameters, downloading flight data, and real-time monitoring.

- Communication systems and flight controller: The UAV uses a set of 3DR 915 MHz telemetry radios for communication between the flight controller and the ground station. A 3DR Pixhawk serves as the flight controller, enabling autonomous flight modes and waypoint navigation.

- Sensors and VHF transceivers: Sensors include a GPS unit and an electronic compass. An 3-axis accelerometer and a 3-axis gyroscope are integrated into

the 3DR Pixhawk flight controller.

VonEhr et al. [109] details the implementation of two SDR approaches, Pseudo Doppler (PD), and Yagi Rotation (YR), for assisted wildlife tracking using a multi-rotor UAV. The innovation lies in the development of a modular Radio Direction-Finding (RDF) system that can be adapted to various UAV platforms and RDF techniques. Below are the key innovations and advancements made by VonEhr et al. [109], as well as the software and hardware components used by VonEhr et al. [109] that meet the criteria listed in Subsubsection 1.3:

- Pseudo Doppler (PD): This approach uses the Doppler Effect and digitally samples multiple monopole antennas in a sequential and circular order, simulating the rotation of a single monopole antenna. Although not implemented by the authors, it highlights the potential for increased accuracy and reduced search time through advanced search patterns.

- Yagi Rotation (YR): This technique involves an ATS Track 13860 folding Yagi antenna rotating 360°to calculate Signal to Noise Ratio (SNR) and Angle of Arrival (AoA) of incoming radio signals. The authors successfully implemented and field-tested the YR system, demonstrating its ability to extend the search range and improve safety for wildlife researchers.

- Modularity: The system is designed to be adaptable to any UAV platform and RDF technique, showcasing a modular RDF system that extends flexibility for wildlife researchers.

- Data processing: The YR system performs real-time processing during the UAV's scan rotation. It calculates SNR and AoA while flying and sends the

data to the ground station during the scan. GNU Radio Companion was used to create the code for the implemented signal processing algorithms.

- User interface: The system uses 3DR Mission Planner v1.3.28 on the ground station laptop for user feedback and SDR control.

- Communication systems and flight controller: The YR system utilizes a communication architecture involving a Fun-cube Dongle Pro+ SDR, Raspberry Pi 2, 3DR Pixhawk flight controller, HopeRF HM-TRP XCVRs telemetry transceivers, and a ground station laptop. The communication is outlined for the YR system but can be applied to PD by changing the antenna and SDR signal processing. The 3DR Pixhawk flight controller is employed for UAV control.

- Sensors and VHF transceivers: The primary sensors mentioned are the Yagi antenna and 3DR u-blox GPS. The IMU embedded into the 3DR Pixhawk flight controller provides a 3-axis accelerometer and 3-axis gyroscope.

- Onboard computer: The onboard computer is a Raspberry Pi 2 (RPi2) running GNU Radio for signal detection, processing, and system communications.

Nguyen et al. [46] discusses a system called TrackerBots, an autonomous unmanned aerial vehicle (UAV) designed for real-time localization and tracking of multiple mobile radio-tagged animals. The innovation lies in the integration of a particle filter for tracking and localizing and a partially observable Markov decision process (POMDP) for dynamic path planning. Below are the key innovations and advancements made by Nguyen et al. [46], as well as the software and hardware components used by Nguyen et al. [46] that meet the criteria listed in Subsubsection 1.3:

- Sensor design: The sensor system includes a compact, lightweight VHF antenna geometry and a software-defined radio architecture for capturing RSSI values from multiple VHF radio tags.

- Real-time tracking and planning: The system integrates a particle filter for real-time tracking and localization and a POMDP for dynamic path planning. This allows the UAV to autonomously navigate in the direction of maximum information gain.

- Search termination criteria: The system introduces the concept of search termination criteria to maximize the number of located animals within power constraints of the aerial system.

- Data processing: The onboard sensor system handles real-time signal processing on the UAV, while results are transmitted to the ground control system for tracking, decision-making, and planning.

- User interface: The system employs QGroundControl to monitor and abort autonomous navigation.

- Communication systems and flight controller: The system uses communication channels between the UAV and the ground control system, employing the MAVLink protocol over 915 MHz and 2.4 GHz radio channels. The author's do not explictly state the equipment used to facilitate the duplex radio channels. The system uses ArduPilotMega (APM) on the IRIS+ UAV as its flight controller, transmitting GPS location data to the ground control system.

- Sensors and VHF transceivers: The system uses a HackRF One SDR device, a GPS, and a specially designed folded 2-element Yagi antenna for VHF signals.

- Onboard computer: An Intel Edison board is used as an embedded compute module for processing digital signals from the SDR receiver.

Shafer et al. [105] details the implementation of the initial UAV-RT system designed for wildlife radiotelemetry in ecological research. The key innovation is the integration of improved localization techniques and a novel direction of arrival (DOA) bearing estimation technique using principal component analysis (PCA). Below are the key innovations and advancements made by Shafer et al. [105], as well as the software and hardware components used by Shafer et al. [105] that meet the criteria listed in Subsubsection 1.3:

- Signal processing: The system employs a signal processing algorithm, including the use of Chebyshev type II bandpass filters and windowing techniques, to improve real-time processing.

- Amplitude-based DOA: The system introduces an amplitude-based DOA bearing estimation technique using PCA to improve the accuracy of localization. The technique incorporates pulse amplitudes as weights, placing emphasis on stronger signals when estimating tag positions. The localization technique can also adapt as the UAV rotates in place and scans for pulses.

- Data processing: Real-time data processing occurs on the onboard computer using signal processing algorithms implemented in Python using GNU radio. Post-processing using MATLAB was completed for analyzing the collected data.

- User interface: The system uses Arudcopter Mission Planner as the user interface on the ground control station (GCS). Mission Planner allows for flight plans to be created and uploaded prior to launch. It also enables autonomous operation mode using the flight controller.

- Communication System and Flight Controller: The flight controller is a Pixhawk Mini running Arducopter firmware, and communicates with the GCS via 915 MHz telemetry radio. A N3000 Wireless Wi-Fi transceiver module could also be used to communicate with the UAV when the TL-WR841HP Wi-Fi router was connected to the GCS.

- Sensors and VHF Transceivers: Sensors include a directional RA-23K H antenna, a set of 915 MHz SiK telemetry radios, and a Airspy R2 for collecting VHF pulse data.

- Onboard Computer: The UDOO x86 Ultra was used for real-time signal processing onboard the UAV.

Comparing systems

The software and hardware components of the cited literature in Section 1.3 are compared in Table 1.1 and Table 1.2 using the criteria listed in Subsubsection 1.3.

Author/ system	Data processing	User interface	Comm. systems and flight controller	Sensors and VHF transceivers	Onboard computer	IPC
Santos et al. [101]	Primarily post-processing	Not specified	Not specified	Omni-directional antenna, television receiver dongle, GPS	Texas Instruments BBB board	Not specified
Cliff et al. [9]	Real-time processing	Proprietary software	Radio modems	Two-point phased antenna, receiver, GPS	Cortex-M3 microprocessor	ROS

Table 1.1: The comparison of the systems discussed in the literature references.

Consi et al. [10]	Post-processing	APM Mission Planner	Telemetry radios, flight controller	IMU, GPS, compass, barometer	Not specified	Not specified
VonEhr et al. [109]	Real-time processing	3DR Mission Planner	Telemetry transceivers, flight controller	Yagi antenna, SDR, IMU, GPS, compass	Raspberry Pi 2	Not specified
Nguyen et al. [46]	Real-time processing	QGroundControl	Duplex radio channels, flight controller	Yagi antenna, SDR, IMU, GPS, barometer	Intel Edison board	Not specified
Shafer et al. [105]	Real-time processing and post-processing	Arudcopter Mission Planner	Telemetry radios, flight controller, Wi-Fi transceiver	Directional antenna, SDR, IMU, GPS, magnetometer, barometer, compass	UDOO X86 Ultra	Not specified

Table 1.2: The comparison of the systems discussed in the literature references (cont).

In the context of the cited literature in Section 1.3, the key innovation of the system discussed in this thesis is the introduction of a novel ROS 2 based communication

architecture. The primary function of the architecture is to facilitate inter-process communication for processes running on the onboard computer. The communication architecture is also responsible for supervising and managing the onboard computer's processes that are required for signal processing and pulse detection. Additionally, it provides a framework for transmitting and receiving tag information, system status messages, and user commands to and from the GCS. Lastly, the communication architecture also conducts interpolation on collected telemetry data and detected pulses, while saving generated data files to the onboard computer for extraction and post-processing.

1.4 Summary of contents

Chapter 2 of this thesis discusses the hardware and software that were included in the architectural design of the UAV-RT system. Chapter 3 provides a detailed description of the uavrt_supervisor, uavrt_connection, and uavrt_interfaces UAV-RT software packages. Chapter 4 covers the setup, installation, configuration, and demonstration of the UAV-RT system. Chapter 5 presents experiments, verification, and results associated with the UAV-RT system. Chapter 6 provides a conclusion, including a summary of this thesis and a discussion of potential future work.

Chapter 2

ARCHITECTURAL DESIGN

2.1 Chapter overview

Chapter 2 provides a comprehensive understanding of the software components, tools, and communication protocols that constitute the architecture of the UAV-RT system. The chapter explains the Robot Operating System (ROS) 2-based software components used in the UAV-RT system, including logging mechanisms, graphical user interface tools, and communication protocols. Additionally, the Robot Operating System Qt (RQt) and its "Node Graph" plugin are discussed in the context of visualizing and monitoring data flow between ROS 2 nodes.

Further, Chapter 2 outlines the integration of MATLAB Coder for code conversion, essential for deploying the channelizer and uavrt_detection packages to the companion computer. Custom QGroundControl software is introduced for configuring, monitoring, and controlling unmanned vehicles, as well as receiving data from UAV-RT software packages running on the companion computer. The Pixhawk 1 flight controller's role in UAV flight control, MAVLink communication protocol, and the MAVSDK library are explained. This chapter also details the use of the Airspy Mini software-defined radio, csdr, Netcat, and MATLAB-based channelization in the context of VHF data collection.

Chapter 2 concludes by summarizing the uavrt_supervisor, uavrt_connection, uavrt_interfaces, channelizer, and uavrt_detection packages, and briefly explaining the roles and interactions of each package within the UAV-RT system.

2.2 ROS 2

ROS 2 is an open-source framework that provides a set of tool, libraries, and conventions for developing and managing complex software and hardware systems [55]. It supports modular and distributed architectures, and incorporates real-time capabilities, software security, and communication middleware. The UAV-RT software system was developed using the the Galactic Geochelone distribution of ROS 2 [70].

The ROS Client Library (RCL) provides the programming interface for creating and maintaining ROS nodes and components, as well as handling communication between them [50]. rclcpp is the RCL for C++, offering tools and abstractions to create and manage nodes and components using C++, while rclpy serves the same purpose for Python. ROS 2 concepts like topics, publishers, subscribers, and executors are provided by the RCL.

The UAV-RT packages that use rclcpp are the uavrt_connection and uavrt_detection packages, while the uavrt_supervisor package uses rclpy. The uavrt_interfaces package uses both rclcpp and rclpy. The channelizer package does not use ROS 2. All of these packages are cloned into the uavrt_source directory, with the uavrt_source directory being inside the uavrt_workspace directory. The process of installing the UAV-RT software system is explained in Section 4.3.

2.2.1 ROS and ROS 2 nodes, nodelets, components, and executors

A ROS node is a fundamental computational unit in ROS and ROS 2 that performs specific tasks and communicates with other nodes via messages [58]. A ROS nodelet is a specialized type of node designed for efficient resource sharing and communication [57]. ROS nodes are compiled into an executable, whereas ROS nodelets are compiled into a shared library. Nodelets enhance communication efficiency and

resource sharing by operating within a shared memory space, enabling direct data transfer between nodelets without inter-process communication overhead. This results in reduced latency and more efficient utilization of computational resources compared to traditional nodes, making nodelets a lightweight and streamlined alternative for specific applications.

In ROS 2, a component is similar to a nodelet in that it provides an easy way to generate a custom executable with predefined nodes at compile time [48]. Components offer better encapsulation and reusability compared to nodes, enabling cleaner code organization and promoting a more efficient and modular architecture. In the uavrt_detection package, nodes were used. In the uavrt_supervisor and uavrt_connection packages, components were used. The functions for instantiating, interacting with, and controlling a component are the same functions as nodes. This includes functions such as "add_node".

Executors in ROS 2 manage the execution of nodes and components, handling the scheduling and coordination of their tasks and callbacks [53]. There are three different types of executors, each offering their own benefits and use cases:

- SingleThreadedExecutor: Process nodes or components sequentially in a single thread.
- MultiThreadedExecutor: Allows for parallel execution, potentially enhancing performance but requiring careful consideration of thread safety.
- StaticSingleThreadedExecutor: Static version of the SingleThreadedExecutor. Entities such nodes, subscribers, and publishers are created once and can not be modified until the executor has stopped. This executor is optimal for resource constrained systems.

For this project, SingleThreadedExecutors were used in the uavrt_ supervisor,

uavrt_connection, and uavrt_detection packages. This choice was made because of the low amount of complexity associated with SingleThreadedExecutors, and the large amount of new development that was happening for this iteration of the project. For future iterations of the UAV-RT project, it would be worth exploring the benefits of MultiThreadedExecutors and StaticSingleThreadedExecutors.

2.2.2 ROS 2 topics, subscribers, publishers, and timers

In ROS 2, topics are communication channels through which nodes exchanges messages [56]. Publishers are used by nodes and components to send messages on a specific topic, while subscribers are used by nodes and components to receive and process the published messages. Callbacks in ROS 2 refer to defined functions that are executed in response to received messages or timed events. Timers are mechanisms that enable nodes to schedule and execute functions at specified intervals, providing a way to perform tasks at regular time intervals. Together, these mechanisms and concepts form the foundation for asynchronous communication and event-driven programming in the UAV-RT packages. Actions and services are also communication types for nodes and components in ROS 2, but these functionalities were not used in the UAV-RT project.

A topic list for the UAV-RT project can be generated through RQt, as discussed in Subsection 2.2.5, or through the terminal by running the following commands:

```
source /opt/ros/galactic/setup.bash
cd ~/uavrt_workspace/
. install/setup.bash
ros2 topic list
ros2 topic echo pulse_pose
```

This will also print out an echo of the "pulse_pose" topic. Note that the topic list and ehco will only be populated if the UAV-RT software packages are running.

2.2.3 ROS 2 message types

ROS 2 messages [51] are data structures used for communicating information and data between nodes or components in a ROS 2 system. Custom message types can also be implemented, allowing developers to define their own data structures for specialized message formats [52]. Both standard and custom ROS 2 message types were used in the UAV-RT project.

ROS 2 standard message types

The following tables describe the data fields within the ROS 2 standard message types used in the UAV-RT project:

- Table 2.1 describes the Pose message type.
- Table 2.2 describes the Point message type.
- Table 2.3 describes the Quaternion message type.
- Table 2.4 describes the PoseStamped message type.
- Table 2.5 describes the Header message type.
- Table 2.6 describes the Time message type.
- Table 2.7 describes the DiagnosticArray message type.
- Table 2.8 describes the DiagnosticStatus message type.
- Table 2.9 describes the KeyValue message type.
- Table 2.10 describes the Bool message type.

Data field	Description
geometry_msgs/msg/ Point position	This contains the position of a point in free space.
geometry_msgs/msg/ Quaternion orientation	This represents an orientation in free space in quaternion form.

Table 2.1: Data fields within the Pose message type [63].

Data field	Description
float64 x	The longitude of a point in free space.
float64 y	The latitude of a point in free space.
float64 z	The altitude of a point in free space.

Table 2.2: Data fields within the Point message type [65].

Data field	Description
float64 x	The x orientation of a point in free space.
float64 y	The y orientation of a point in free space.
float64 z	The z orientation of a point in free space.
float64 w	The w orientation of a point in free space.

Table 2.3: Data fields within the Quaternion message type [66].

Data field	Description
std_msgs/msg/ Header header	Standard metadata for higher-level stamped data types.
geometry_msgs/msg/ Pose pose	This contains the position of a point in free space.

Table 2.4: Data fields within the PoseStamped message type [64].

Data field	Description
builtin_interfaces/msg/ Time stamp	Two-integer timestamp that is expressed as seconds and nanoseconds.
string frame_id	Transform frame with which this data is associated.

Table 2.5: Data fields within the Header message type [68].

Data field	Description
int32 sec	The seconds component, valid over all int32 values.
uint32 nanosec	The nanoseconds component, valid in the range [0, 10e9).

Table 2.6: Data fields within the Time message type [59].

Data field	Description
std_msgs/msg/Header header	Contains the timestamp and frame ID.
diagnostic_msgs/msg/ DiagnosticStatus[] status	The nanoseconds component, valid in the range [0, 10e9).

Table 2.7: Data fields within the DiagnosticArray message type [60].

Data field	Description
byte OK=0, WARN=1, ERROR=2, STALE=3	Possible levels of operations.
byte level	The nanoseconds component, valid in the range [0, 10e9).
string name	A description of the test/component reporting.
string message	A description of the status.
string hardware_id	A hardware unique string.
diagnostic_msgs/msg/ KeyValue[] values	An array of values associated with the status.

Table 2.8: Data fields within the DiagnosticStatus message type [61].

Data field	Description
string key	What to label this value when viewing.
string value	A value to track over time.

Table 2.9: Data fields within the KeyValue message type [62].

Data field	Description
bool data	True or false value.

Table 2.10: Data field within the Bool message type [67].

UAV-RT custom message types

Tables 2.11, 2.12, 2.13, and 2.14 describe the data fields within the PulsePose, Pulse, and Tag UAV-RT custom message types, respectively. These custom messages are included in msg files that are found in the "msg" directory of the uavrt_interfaces package.

Data field	Description
uavrt_interfaces/msg/ Pulse pulse	A representation of the pulse data associated with a pulse waveform object.
geometry_msgs/msg/ Pose antenna_pose	A representation of pose in free space, composed of position and orientation.

Table 2.11: Data fields within the PulsePose custom message type.

Data field	Description
string detector_dir	Directory from which the detector executable was executed from.
uint32 tag_id	The tag ID that was used for detection priori info. Useful for tractability.
float64 frequency	Frequency at which pulse was detected. 0 value indicates detector heartbeat.
builtin_interfaces/msg/ Time start_time	System time at rising edge of pulse time bin.
builtin_interfaces/msg/ Time end_time	System time at falling edge of pulse time bin.
builtin_interfaces/msg/ Time predict_next_start	This is the time that the next pulse is expected to occur based on the current pulse time and the priori pulse interval information.
builtin_interfaces/msg/ Time predict_next_end	This is the time that the next pulse is expected to end based on the current pulse time and the priori pulse interval information.
float64 snr	Estimated pulse SNR in dB. This is the SNR during the time of pulse transmission. Additionally, this is the ratio of the pulses peak power point to the estimated noise power at that same frequency.
float64 stft_score	Scoring value used during the pulse detection process.

Table 2.12: Data fields within the Pulse custom message type.

uint16 group_ind	If more than one pulse is used for incoherent summing, the pulse group will have up to K pulses.
float64 group_snr	Signal power ratio to noise power in its frequency bin for all pulses in group
bool detection_status	This property indicates if the pulse is a subthreshold pulse (0), superthreshold pulse (1), or confirmed pulse (2).
bool confirmed_status	This property indicates if the pulse has been confirmed (1), or is of yet unconfirmed (0).

Table 2.13: Data fields within the Pulse custom message type (cont).

Data field	Description
uint32 tag_id	The tag ID that was used for detection priori info. Useful for tractability.
float64 frequency	Expected frequency of tag.
float64 pulse_duration	Duration of pulse in seconds
float64 interpulse_time_1	Interpulse duration in seconds.
float64 interpulse_time_uncert	Interpulse duration uncertainty in seconds.
float64 interpulse_time_jitter	Interpulse duration jitter in seconds.
float64 interpulse_time_2	Interpulse duration in seconds. This field is used for multi-rate tags.
uint32 k	Number of pulses to integrate by.
float64 false_alarm_probability	Probability of a false alarm.

Table 2.14: Data fields within the Tag custom message type.

2.2.4 ROS 2 logging

ROS 2 logging [49] is a system for managing log messages in ROS 2, providing a standardized way to capture information, warnings, errors, and debug messages in systems. The logging functionality for the uavrt_supervisor package is provided for by rclpy, while rclcpp provides the logging functionality for uavrt_connection. The primary difference between logging in rclpy and rclcpp is the programming language used. Aside from specific functions and syntax that vary due to the differences between Python and C++, the core logging concepts are consistent.

2.2.5 ROS RQt

ROS Qt (RQt) [54] provides a collection of graphical user interface (GUI) tools and plugins for visualizing and interacting with ROS and ROS 2 related data. RQt's "Node Graph" plugin allows users to visualize and monitor the communication and data flow between ROS and ROS 2 nodes in a system. This is helpful in understanding the system's behavior and effectively diagnosing issues within a system. The "Topic Monitor" plugin for RQt displays information associated with ROS and ROS 2 topics and messages.

RQt Node Graph can be used for the UAV-RT project by opening a terminal window and running the following commands:

```
source /opt/ros/galactic/setup.bash
cd ~/uavrt_workspace/
. install/setup.bash
rqt
```

Figure 2.1 is an image of the RQt Node Graph for the UAV-RT system. Note that the UAV-RT software system must be running and actively tracking a tag in order to see the entire RQt Node Graph.

Figure 2.1 also illustrates the topics, publishers, and subscribers that connect the uavrt_connection, uavrt_supervisor, and uavrt_detection packages in Figure 1.1.

2.3 MATLAB Coder

MATLAB Coder [32] is a MathWorks product that allows users to convert MATLAB algorithms and code into standalone, optimized C/C++ code. The resulting C/C++ code maintains the same functionality as the original MATLAB algorithms, allowing for MATLAB-based applications to be deployed to embedded systems. A

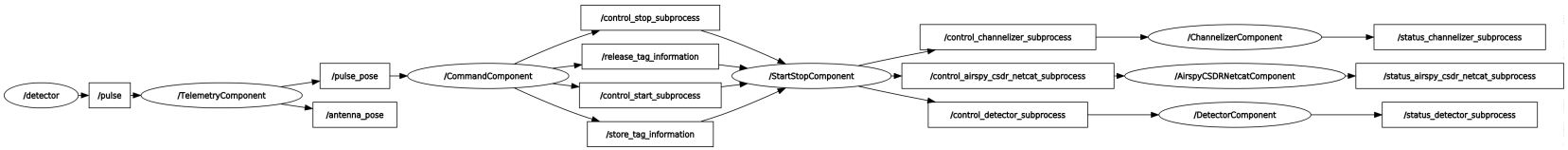

Figure 2.1: The RQt Node Graph for the UAV-RT system.

drawback of MATLAB Coder is the lack of readability associated with the generated code, making it more difficult to debug the generated code than the original MATLAB code. There is also additional configuration associated with the generated code that needs to be addressed prior to compilation and execution.

In this project, MATLAB Coder was used to convert the MATLAB code in the channelizer and uavrt_detection packages into C++ code. Dr. Michael Shafer provides an overview to the additional configuration necessary to compile and execute the generated code, as well MATLAB scripts that can be used to make the code generation step easier for these packages. This overview is provided in the READMEs of the channelizer and uavrt_detection packages. MATLAB Coder was not used to generate the code in the uavrt_supervisor, uavrt_connection, or uavrt_interfaces packages.

2.4 Custom QGroundControl

QGroundControl [97] is an open-source ground control station (GCS) software used for configuring, monitoring, and controlling unmanned vehicles, primarily drones and robots. QGroundControl offers a user-friendly interface for mission planning, telemetry visualization, and real-time vehicle control. In this project, a custom version of QGroundControl was developed and maintained by Don Gagne [23], and provides additional functionality that is not found in the main version of QGroundControl. This includes functionality such as processing tag input via text file, sending detection commands and tag information to UAV-RT software running on the companion computer, and receiving, processing, and printing out information for tags that are being tracked in real time.

2.5 Pixhawk 1 flight controller

The Pixhawk 1 flight controller [89] is an open-source autopilot system designed for UAVs and other robotic applications. Its primary function in the UAV-RT system is to control the flight of the UAV by processing data from various sensors and executing commands to control motors. The MAVLink communication protocol and C++ MAVSDK library are used to communicate with the Pixhawk 1 flight controller.

The Pixhawk 1 flight controller is equipped with an inertial measurement unit (IMU), enabling it to gather information about the UAV's orientation and velocity. A GPS module with an embedded digital magnetometer is connected to the Pixhawk 1 flight controller, which provides the flight controller with the UAV's position, and additional compass and orientation accuracy. In order to transmit information to and from the GCS, a 915 MHz telemetry radio is connected to a telemetry port on the Pixhawk 1 flight controller. A 915 MHz telemetry radio is also connected to the USB port on the GCS. Note that a reason the 915 MHz telemetry radio was used for the UAV-RT project is because of its compliance with FCC regulations for the 915 MHz ISM (Industrial, Scientific, and Medical) frequency band [22], which permits license-free operation for wireless communication devices. This ensures legal and interference-free operation in the United States.

Chapter 4 provides details on the equipment utilized in the UAV-RT project, along with instructions for configuring the said equipment.

2.6 MAVLink communication protocol

The MAVLink communication protocol [35] is a lightweight, open-source communication protocol that acts as a underlying communication framework within this project. The MAVLink protocol uses a packet-based structure to standardize the

way information is packaged and sent between the onboard flight controller and the ground control station (GCS). The MAVSDK library abstracts the complexities of the MAVLink protocol, and provides a user-friendly interface to the low-level details of the MAVLink protocol.

2.6.1 Tunnel protocol

The MAVLink Tunnel Protocol [36] is a protocol designed to encapsulate MAVLink messages in a structured message format, which can then be transmitted to other parts of the system via a flight controller and a connected radio. On the receiving end, the encapsulated MAVLink messages are extracted and the contents of the messages are forwarded to the intended software for processing. The structure of the Tunnel Protocol is described in Table 2.15.

Data field	Description
uint8_t target_system	System ID.
uint8_t target_component	Component ID.
uint16_t payload_type	An enum code that identifies the content of the payload.
uint8_t payload_length	Length of the data transported in payload.
uint8_t[128] payload	Variable length payload.

Table 2.15: Data fields within the Tunnel Protocol [36].

Don Gagne introduced the Tunnel Protocol as a solution to a data transmission problem identified during the project's development phase. This issue arose when using the MAVLink DEBUG_FLOAT_ARRAYS message type [34] and connecting to the flight controller through a USB cable. While this message type and connection

setup partially functioned, it resulted in messages being dropped and restricted control over what and how data that could be sent to the GCS. The Tunnel Protocol fixes this issue, simplifying the creation, configuration, and transmission of messages. Adopting the Tunnel Protocol necessitated changing the connection configuration from USB to a serial connection for the flight controller. This transition is detailed further in Subsection 4.4.1. The Tunnel Protocol also requires that an active connection be established to both the flight controller and custom QGroundControl before messages can been be exchanged.

The Tunnel Protocol was implemented into the project structs that are defined within the TunnelProtocol.h file. The structs that comprise the TunnelProtocol.h file are: HeaderInfo_t, AckInfo_t, TagInfo_t, StartDetectionInfo_t, StopDetectionInfo_t, StartTagsInfo_t, EndTagsInfo_t, and PulseInfo_t. These structs are packaged within the "payload" portion of the Tunnel Protocol. Figure 2.2 illustrates the data contained in each struct. How TunnelProtcol messages are packaged and sent is discussed in Subsection 3.3.5.

Don Gagne was a significant help when it came to implementing the structs within the TunnelProtocol.h file, laying out the framework for transmitting, receiving, and processing Tunnel Protocol messages, and communicating with custom QGroundControl running on the GCS.

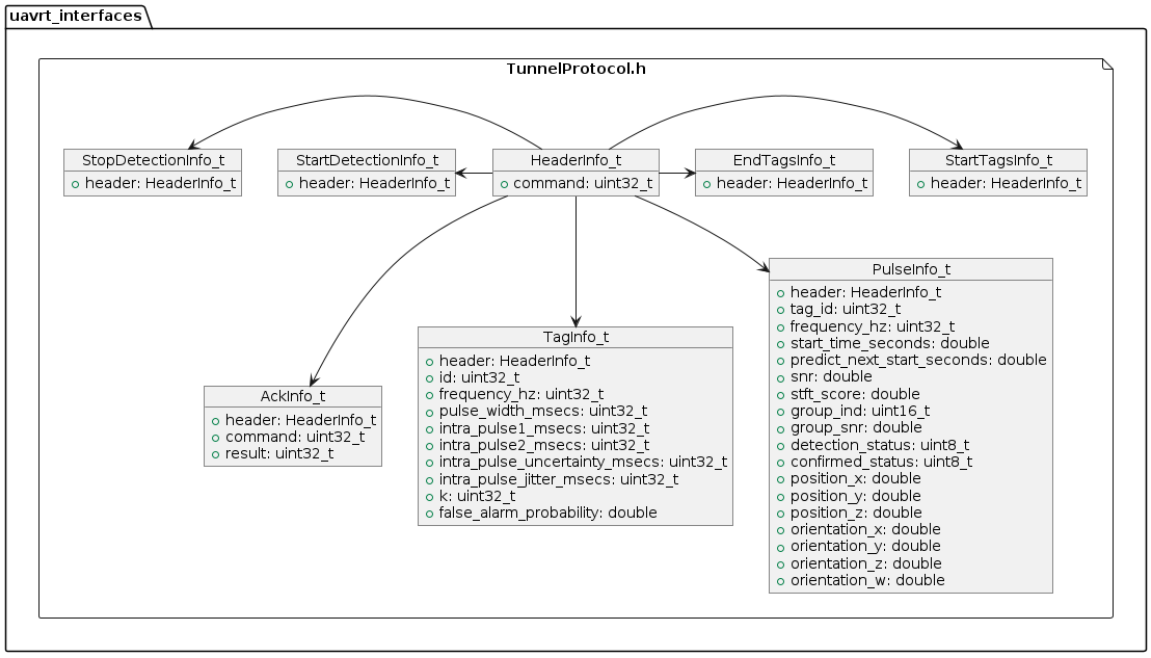

Figure 2.2: A class diagram that defines the structs that are defined within the TunnelProtocol.h file.

Table 2.16 describes the constants that are defined in TunnelProtocol.h. These constants are used in the structs that are also defined in TunnelProtocol.h, as well as functions in the command_component.cpp file discussed in Subsection 3.3.5.

Constant name	Value	Description
COMMAND_ID_ACK	1	Ack response to command.
COMMAND_ID_START_TAGS	2	Previous tag set should be cleared, new tags are about to be uploaded.
COMMAND_ID_END_TAGS	3	All new tags have been uploaded.
COMMAND_ID_TAG	4	Tag information.
COMMAND_ID_START_DETECTION	5	Start pulse detection.
COMMAND_ID_STOP_DETECTION	6	Stop pulse detection.
COMMAND_ID_PULSE	7	Detected pulse value.
COMMAND_RESULT_SUCCESS	1	Successful result.
COMMAND_RESULT_FAILURE	2	Failed result.

Table 2.16: Command constants within the TunnelProtocol.h file.

2.7 C++ MAVSDK library

The C++ MAVSDK library [38] is an open-source software development kit for building applications that interact with MAVLink-compatible flight controllers. It offers a set of C++ plugins to facilitate control, telemetry, and mission planning for vehicles via the onboard flight controller. Within this project, the C++ MAVSDK library is to used to access the Position [44] and Quaternion [45] telemetry data that is gathered by the onboard flight controller, a 3DR Pixhawk 1 Flight Controller. The Position and Quaternion telemetry structs are described in Tables 2.17 and 2.18, respectively.

Data field	Description
double latitude_deg	Latitude in degrees (range: -90 to +90).
double longitude_deg	Longitude in degrees (range: -180 to +180).
float absolute_altitude_m	Altitude AMSL (above mean sea level) in metres.
float relative_altitude_m	Altitude relative to takeoff altitude in metres.

Table 2.17: Data fields within the Position telemetry struct [44].

Data field	Description
float w	Quaternion entry 0, also denoted as a.
float x	Quaternion entry 1, also denoted as b.
float y	Quaternion entry 2, also denoted as c.
float z	Quaternion entry 3, also denoted as d.
uint64_t timestamp_us	Timestamp in microseconds.

Table 2.18: Data fields within the Quaternion telemetry struct [45].

2.8 Airspy library and airspy_rx tool

This project employs the use of an Airspy Mini software-defined radio (SDR) in order to collect pulse information from animal wildlife tags. The Airspy Mini is a compact SDR receiver designed for a wide frequency range from 24 MHz to 1,800 MHz. The device connects to a computer via USB and is interfaced via the Airspy library [1], which provides software tools and resources for interfacing with the device. Additionally, the airspy_rx tool [2] is a software application that is part of the Airspy ecosystem and enables users to control and configure their Airspy Mini SDR for various radio signal reception and processing tasks. A complete example of

the airspy_rx command call is as follows: `/usr/local/bin/airspy_rx -f 148.710 -r - -p 0 -a 3000000 -t 0 -d`. The individual commands that this command call is comprised of are described in Table 2.19.

Command	Provided functionality
Path to the installation of the airspy_rx executable	If the Airspy library was installed using the instructions listed in the airspyone_host installation instructions, then the tool should be accessible via this path: `/usr/local/bin/airspy_rx`.
[-f frequency_MHz]	Set frequency in MHz between 24MHz and 1900MHz.
[-r filename]	Receive data into this file. If a "-" is entered, then the data is read to stdout.
[-p packing]	Set packing for samples: 1 = enabled(12bits packed), 0 = disabled(default 16bits not packed).
[-a sample_rate]	Set sample rate.
[-t sample_type]	Set sample type: 0 = FLOAT32_IQ, 1 = FLOAT32_REAL, 2 = INT16_IQ(default), 3 = INT16_REAL, 4 = U16_REAL, 5 = RAW
[-d]	Verbose mode.

Table 2.19: Usage commands as described in the airspy_rx.c file [2].

2.9 csdr and Netcat

This section gives an overview of additional software tools used in this project: csdr and Netcat.

csdr, a command line tool and library for digital signal processing tasks, provides

the fir_decimate_cc function from its library [99]. This function acts as a decimator, retaining one sample out of every decimation_factor samples [100]. When collecting VHF data from the Airspy SDR using the airspy_rx tool, data is first passed to csdr via the command line to decimate the incoming VHF data.

Following decimation by csdr, the data is forwarded to Netcat, a networking utility for reading and writing data across network connections using TCP/IP or UDP [24]. Netcat facilitates the transmission of the decimated data from csdr to the Channelizer over the UDP protocol, writing it to a designated port on the machine.

2.10 Processing within the UAV-RT software package

This section contains a set of sequence diagrams demonstrating the interactions between the UAV-RT software packages and the processing that takes place among each component within the UAV-RT system. This section will also contain summaries of each UAV-RT package used within this iteration of the project. The files that comprise the uavrt_connection, uavrt_supervisor, and uavrt_interfaces packages will be explored in detail in Chapter 3.

Figures 2.3 and 2.4 provide an overview of the communication and actions involved in managing real-time tag data and detection processes. The interactions begin with the User starting the relevant packages and software. The User enters tag information, which is exchanged and acknowledged between custom QGroundControl and uavrt_connection. The uavrt_supervisor processes this information, and the User initiates a detection process, leading to the activation of subprocesses for data collection and processing. Data flows through various components and is eventually logged and transmitted back to custom QGroundControl. When the User stops the detection process, the subprocesses and packages are shut down. Note that specific details were omitted, such as telemetry data collection and transmitting data to

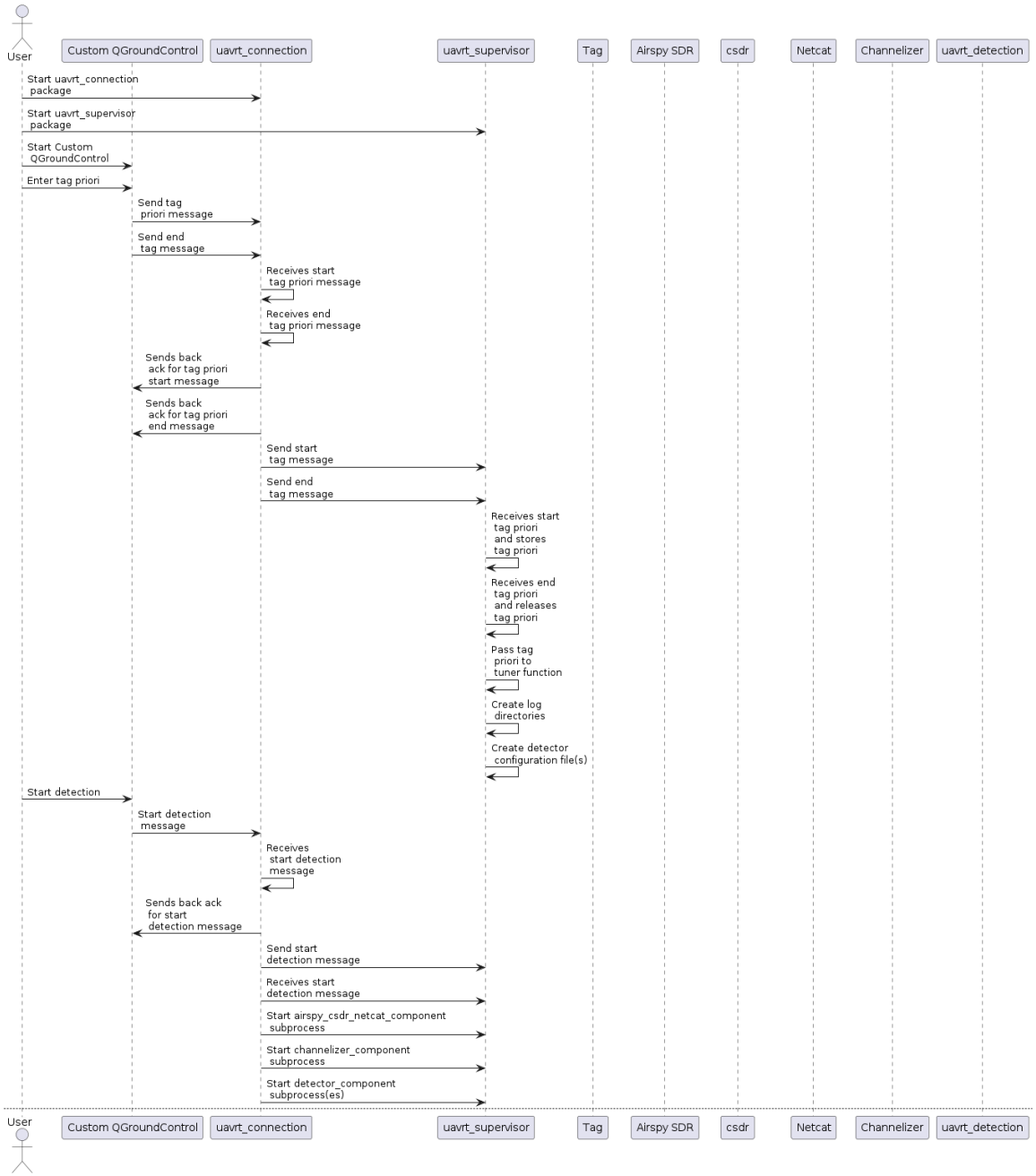

Figure 2.3: A sequence diagram that outlines the interactions between various components in the UAV-RT system.

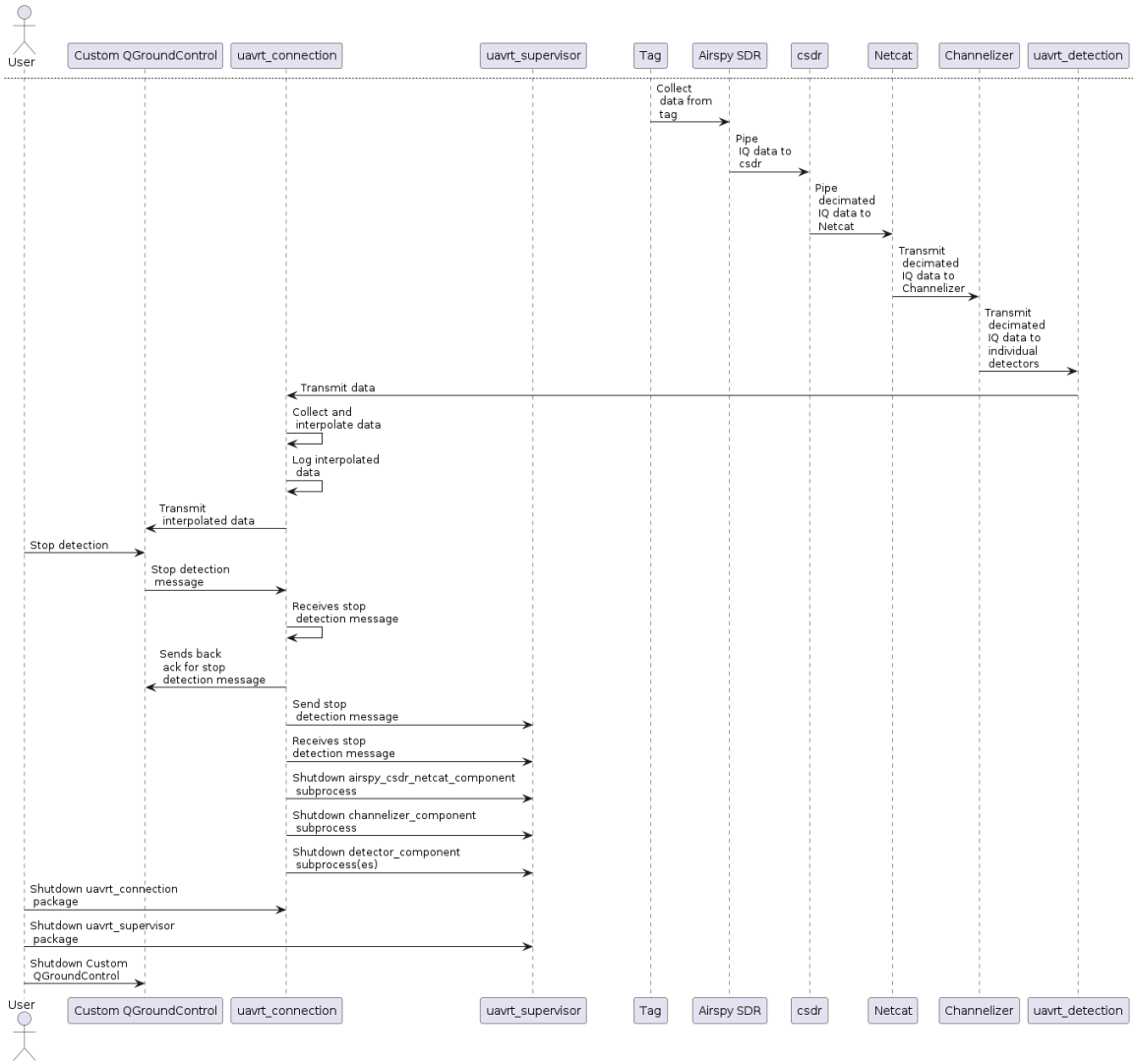

Figure 2.4: A sequence diagram that outlines the interactions between various components in the UAV-RT system. (cont)

custom QGroundControl through the flight controller and telemetry radios.

2.10.1 uavrt_source directory summary

The uavrt_source directory [5] acts as the source in which certain UAV-RT repositories/packages can be cloned into to be built using the version of colcon packaged with ROS 2 Galactic Geochelone. It also contains the logging structure that is built out by the uavrt_supervisor package. As explained in Subsubsection 4.3.2, additional

files have been included in the current iteration of the uavrt_source directory. The uavrt_source package is not explored in Chapter 3.

2.10.2 uavrt_supervisor package summary

The uavrt_supervisor package [6] is a ROS 2 package consisting of multiple Python files and classes designed for managing real-time operations within the UAV-RT software system. This package includes several components and scripts that work together to receive, transmit, and process ROS 2 messages, initialize, control, and monitor subprocesses, and create logging directories and configuration files based on user input.

2.10.3 uavrt_connection package summary

The uavrt_supervisor package [3] is a ROS 2 package consisting of multiple C++ files and classes designed for managing real-time communication within the UAV-RT software system. Its main functionality is to establish a connection and communicate with the PX4 flight controller using the MAVLink protocol and MAVSDK C++ API. It establishes a connection and communicates with custom QGroundControl using the Tunnel Protocol supplied by the MAVLink protocol. Lastly, the uavrt_supervisor package receives detected pulse data from processes that are a part of the uavrt_detection package and performs interpolation on the data received using telemetry data gathered from the PX4 flight controller.

2.10.4 uavrt_interfaces package summary

The uavrt_interfaces package [4] is a ROS 2 package consisting of a C style header file and several ROS 2 message files related to message transmission and reception in the context of the MAVLink Tunnel Protocol and ROS 2 custom messages. This package serves as a source for constants and structs required for sending and receiving

MAVLink Tunnel Protocol messages and defines the custom messages PulsePose, Pulse, and Tag.

2.10.5 channelizer package summary

The channelizer package [103] is a collection of MATLAB files designed to facilitate the channelization of incoming VHF data and make the channelized data accessible to other processes through UDP. It serves as an intermediary between high-sample-rate incoming data and processes running on a computer that may require access to one or more channels at a lower sample rate. Since the channelized data is delivered via UDP ports, it can be utilized by any program capable of handling UDP data. The code in the channelizer package is written in MATLAB and can be translated into an executable using MATLAB Coder. The main channelizer functionality was implemented using the Channelizer System Object in the MATLAB Digital Signal Processing System Object Toolbox [33].

2.10.6 uavrt_detection package summary

The uavrt_detection package [102] is a collection of MATLAB files designed to detect pulses within incoming VHF data given tag information, with the detected pulse information then made accessible to other processes through UDP ports. The uavrt_detection package was implemented in MATLAB by Dr. Michael Shafer, with the signal processing methodologies being researched, discussed, and written about by Dr. Michael Shafer and Dr. Paul Flikkema [104]. The code in the uavrt_detection package is written in MATLAB and can be translated into an executable using MATLAB Coder. Note that uavrt_detection processes are referred to as "detectors" within Chapter 3 to remain consistent with naming scheme of the files used within the UAV-RT software system.

Chapter 3

SOFTWARE DESCRIPTION

3.1 Chapter overview

Chapter 3 presents a detailed analysis of the uavrt_supervisor, uavrt_connection and uavrt_interfaces packages within the UAV-RT software system. Figures are used to explain the directory and file hierarchy of these packages, while tables and class and activity diagrams are used to explain the classes and functions found each file. The interaction between the uavrt_supervisor, uavrt_connection and uavrt_interfaces packages and different components of the UAV-RT are also explained in this chapter. For brief summaries of each of these packages, refer to Section 2.10.

Figure 3.1: A class diagram depicting the files and classes in the uavrt_supervisor package.

Figure 3.1 represents the directory and file hierarchy for the uavrt_supervisor package. ROS 2 directories and files that are automatically generated during the initial creation of ROS 2 package have been omitted from Figure 3.1, Figure 3.18, and Figure 3.25, as well as subsections in Chapter 3.

3.2.1 main.py

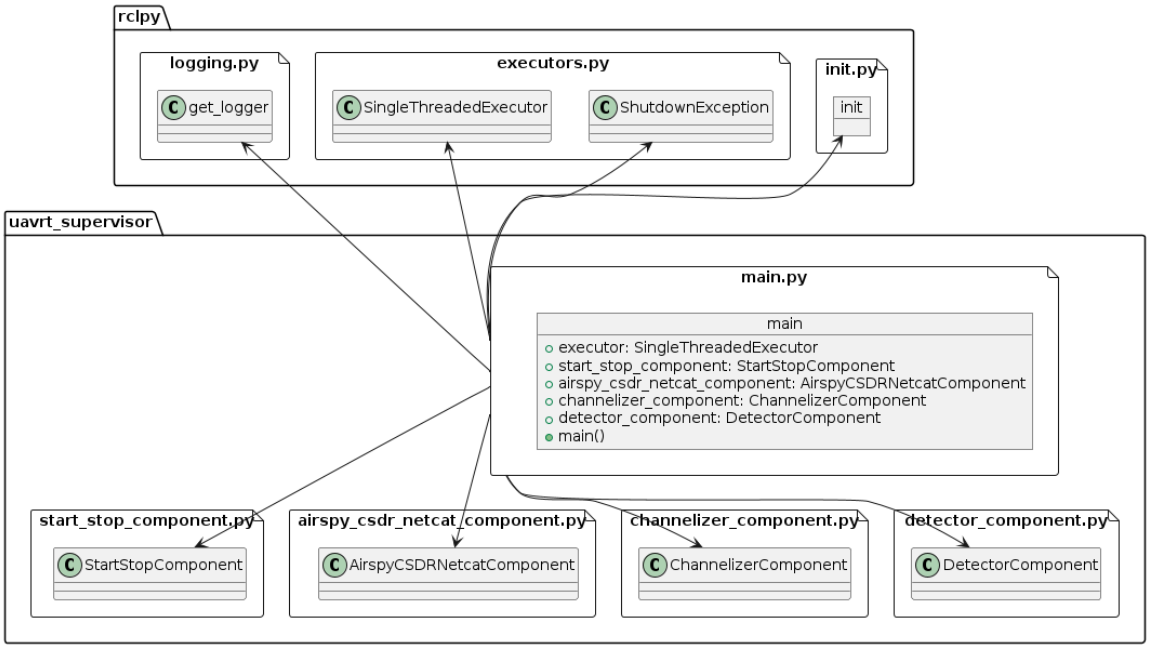

Figure 3.2: A class diagram illustrating the main.py file, its imports, and the variables declared in the main function.

The main.py file acts as the entry point to the uavrt_supervisor package. The main functionality of the main.py file is to initialize the ROS Client Library, create instances of uavrt_supervisor components, add those components to a single-threaded executor, and start the executor's event loop to execute the components' callbacks. It also handles exceptions and provides logging messages to indicate the program's status. Figure 3.2 represents the class diagram for the main.py file. The imports for the main.py file are described in Table 3.1 and Table 3.2.

Imported library or module	Provided functionality
rclpy	ROS Client Library for the Python language [88].
rclpy.executors	Executors are responsible for the actual execution of ROS callbacks [69].
rclpy.logging	The logging subsystem in ROS 2 delivers logging messages to set components [73].
uavrt_supervisor. start_stop_component	The the main purpose of this file is to create configuration files and directories, as well as start and stop components based on messages received from elsewhere in the UAV-RT software package.

Table 3.1: Imported rclpy and uavrt_supervisor modules used in the start_stop_component.py file.

uavrt_supervisor.airspy_csdr_netcat_component, uavrt_supervisor. channelizer_component, uavrt_supervisor. detector_component	The main purpose of these files is to implement ROS 2 components that manage and control subprocesses related to a specific software library, package, or tool.

Table 3.2: Imported rclpy and uavrt_supervisor modules used in the start_stop_component.py file. (cont.)

After the imports, the main function is defined as the entry point for the main.py file. Figure 3.3 is an activity diagram that describes the flow of the main function. This function initializes the ROS Client Library using the init function. It then creates an instance of the SingleThreadedExecutor class to handle the execution of callbacks. Instances of the imported uavrt_supervisor components are then created. These components include the StartStopComponent, AirspyCSDRNetcatComponent, ChannelizerComponent, and DetectorComponent. The created components are added to the executor using the add_node method. The executor will handle the execution of callbacks for these components.

The script sets up a try-except-finally block to handle exceptions and perform cleanup. Inside the try block, log messages are printed using the get_logger function to indicate that the package has started and provide instructions for stopping the program. The spin method of the executor is called in the try block. The spin method starts the event loop and executes ROS 2 callbacks until the executor is interrupted by an exception.

The except block catches the KeyboardInterrupt exception if the user presses

Ctrl+C to stop the program. The except block also catches the ShutdownException if the executor is shut down during runtime. This exception will be thrown in instances when the KeyboardInterrupt exception is not thrown. In both cases, a log message is printed to the terminal and provides information about the type of exception.

The finally block is always executed, regardless of whether an exception occurred. It logs a message indicating that the package is shutting down. Finally, the executor is shut down using the shutdown method to clean up and release the resources used by the executor.

/hlReview afterpage commands and determine if necessary. If yes, are they causing text to disappear.

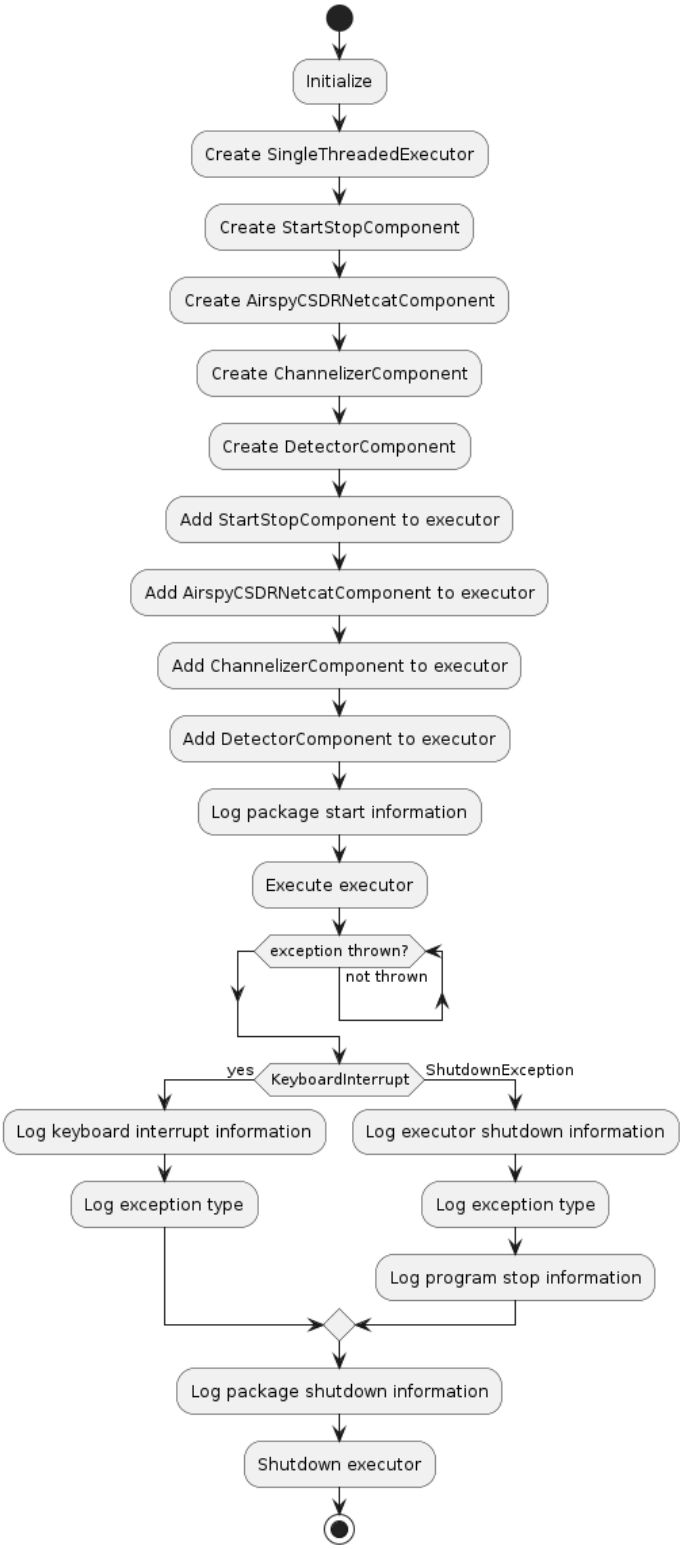

Figure 3.3: An activity diagram depicting the initialization and execution of the components in the main function.

3.2.2 start_stop_component.py

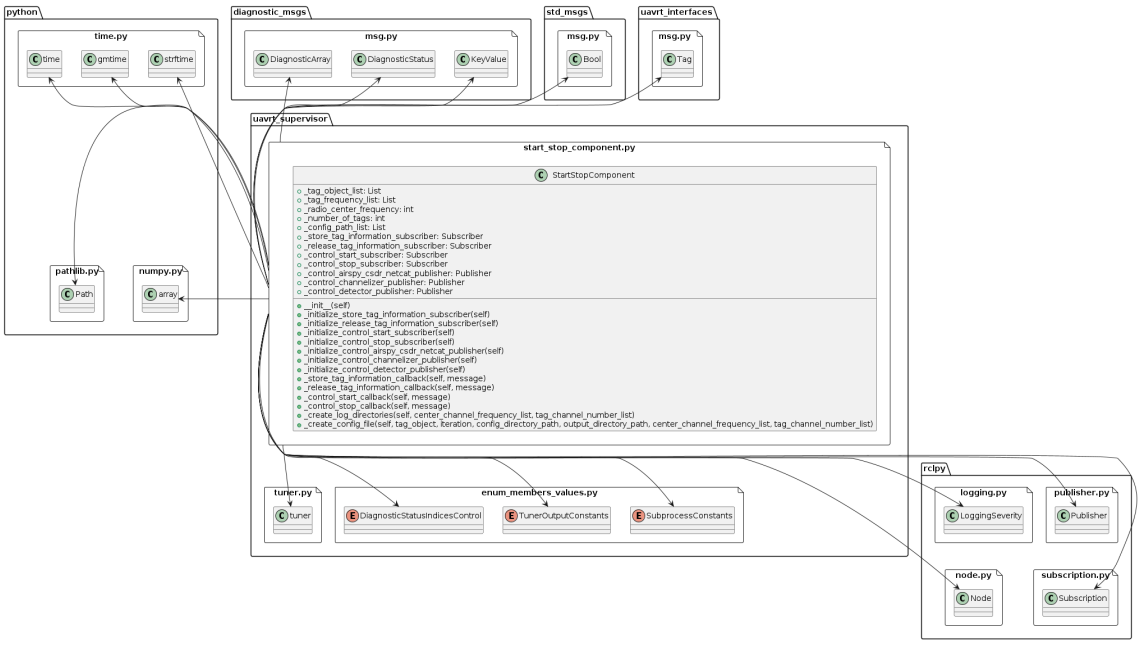

Figure 3.4: Class diagram illustrating the class structure and dependencies of the start_stop_component and its class variables and functions.

The start_stop_component.py file resides in the uavrt_supervisor package and defines the StartStopComponent class. The purpose of the StartStopComponent class is to implement subscribers and publishers in order to issue start and stop commands to other parts of the uavrt_supervisor package, as well as provide tag information and file paths. The class diagram for this Python script is shown in Figure 3.4.

Imported library or module	Provided functionality
pathlib	This module offers classes representing file system paths with semantics appropriate for different operating systems [92].
time	This module provides various time-related functions [93].
numpy	NumPy is the fundamental package for scientific computing with Python [47].
rclpy.node	A Node is the primary entry point in a ROS system for communication. It can be used to create ROS entities such as publishers, subscribers, services, etc. [74].
rclpy.logging	The logging subsystem in ROS 2 aims to deliver logging messages to a variety of targets [73].
rclpy.publisher	A publisher is used as a primary means of communication in a ROS system by publishing messages on a ROS topic [76].
rclpy.subscription	A publisher is used to catch messages that are published to topics [76].
diagnostic_msgs.msg	Diagnostic messages which provide the standardized interface for the diagnostic and runtime monitoring systems in ROS/ROS 2 [86].

Table 3.3: Imported rclpy, uavrt_supervisor, and uavrt_interfaces modules used in the start_stop_component.py file.

std_msgs.msg	Standard metadata for higher-level stamped data types. This is generally used to communicate timestamped data in a particular coordinate frame [82].
uavrt_interfaces.msg	Provides the ROS 2 custom messages used with the UAV-RT software package.
uavrt_supervisor.tuner	Selects the radio center frequency well.
uavrt_supervisor. enum_members_values	Enum values that are used in the uavrt_supervisor package to describe the index that is being accessed.

Table 3.4: Imported rclpy, uavrt_supervisor, and uavrt_interfaces modules used in the start_stop_component.py file (cont.).

The start_stop_component.py file begins by importing the necessary modules from rclpy, as well as modules from the uavrt_supervisor and uavrt_interfaces packages. These imports are described in Table 3.3 and in 3.4.

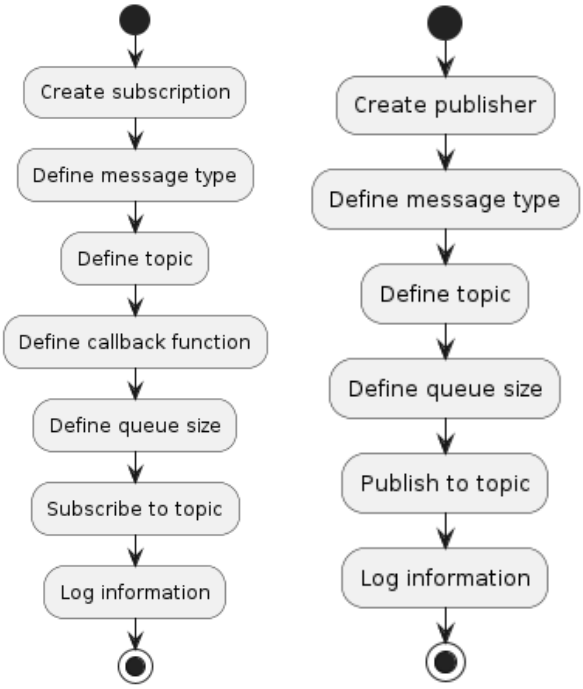

Figure 3.5: Left: The process of creating a subscription, defining a message type, topic, callback function, queue size, subscribing to the topic, and logging information. Right: The process of creating a publisher, defining a message type, topic, queue size, publishing to the topic, and logging information.

The StartStopComponent class begins by initializing its attributes, including lists to store tag objects and frequencies, a variable to store the radio center frequency, the number of tags currently being queued up for processing, and a list for holding detector configuration file paths. The class then defines several private methods prefixed with _initialize_ that are responsible for initializing the subscribers and publishers for different topics. These methods create instances of subscribers and publishers using the create_subscription and create_publisher methods provided by the Node class. Each method specifies the message type, topic name, callback function, and queue size for the subscriber or publisher. Informational log messages are also logged for each initialization. The left-hand side of Figure 3.5 depicts the activity diagram that an _initialize_subscriber method will follow, and the right-hand side of Figure 3.5 illustrates the activity diagram for the corresponding _initialize_publisher methods.

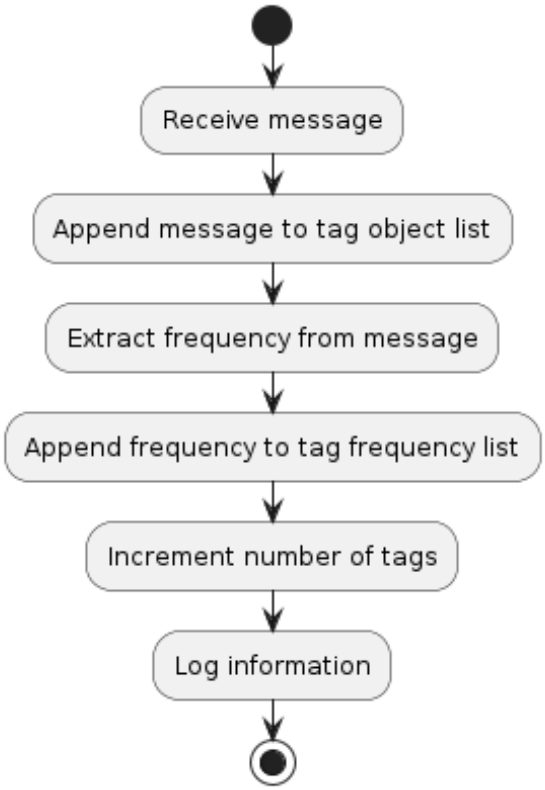

Figure 3.6: The process of receiving and processing a message with tag information.

Following the initialization methods, there are several callback methods prefixed with _store_, _release_, and _control_. These methods define the behavior when a message is received on the corresponding topic. The first method is the _store_tag_information_callback method. This method is called when a message of type Tag is received on the topic "store_tag_information". It appends the received tag object to the _tag_frequency_list class variable, increments the _number_of_tags class variables, and logs a success message. The list and number of tags counter are class variables. Figure 3.6 illustrates the flow of this method.

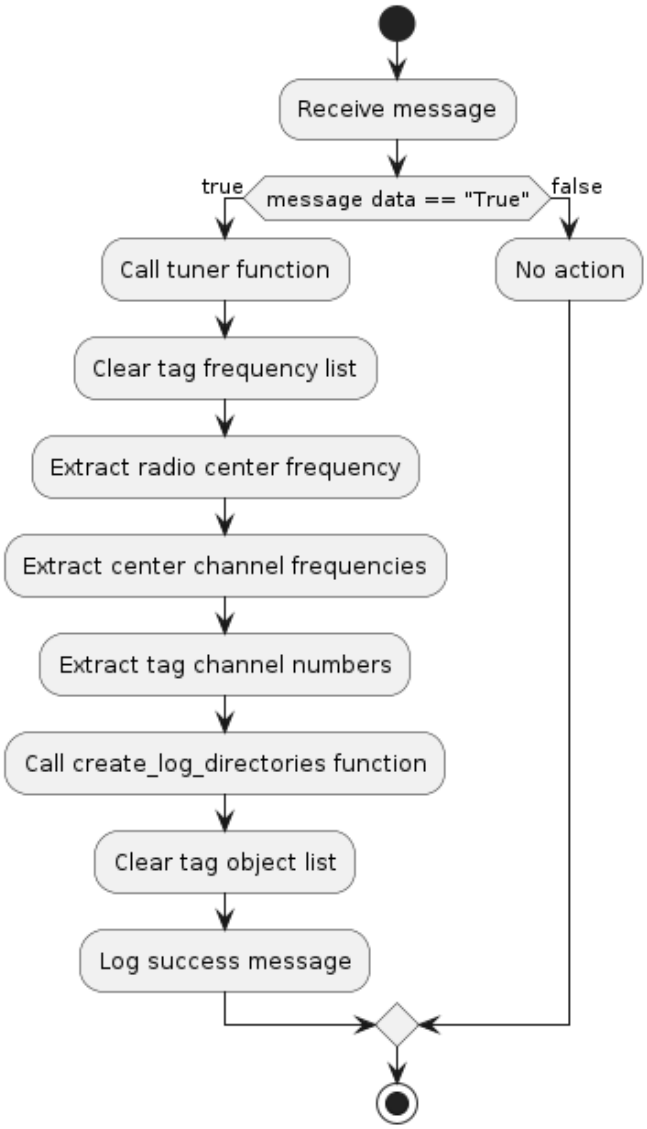

Figure 3.7: An activity diagram illustrating the processing of a received release message and subsequent actions based on its data.

The _release_tag_information_callback method is called when a message of type Bool is received on the topic "release_tag_information". If the variable data in the received message is set to True, the method performs the operations of calling the tuner function in tuner.py, clearing the _tag_frequency_list, storing the radio center frequency and other values from the tuner function call, creating log directories, and clearing the _tag_object_list variable. Success messages are logged accordingly. The

lists are cleared so that the method can be called again in future instances. Figure 3.7 demonstrates the sequence of steps in this method.

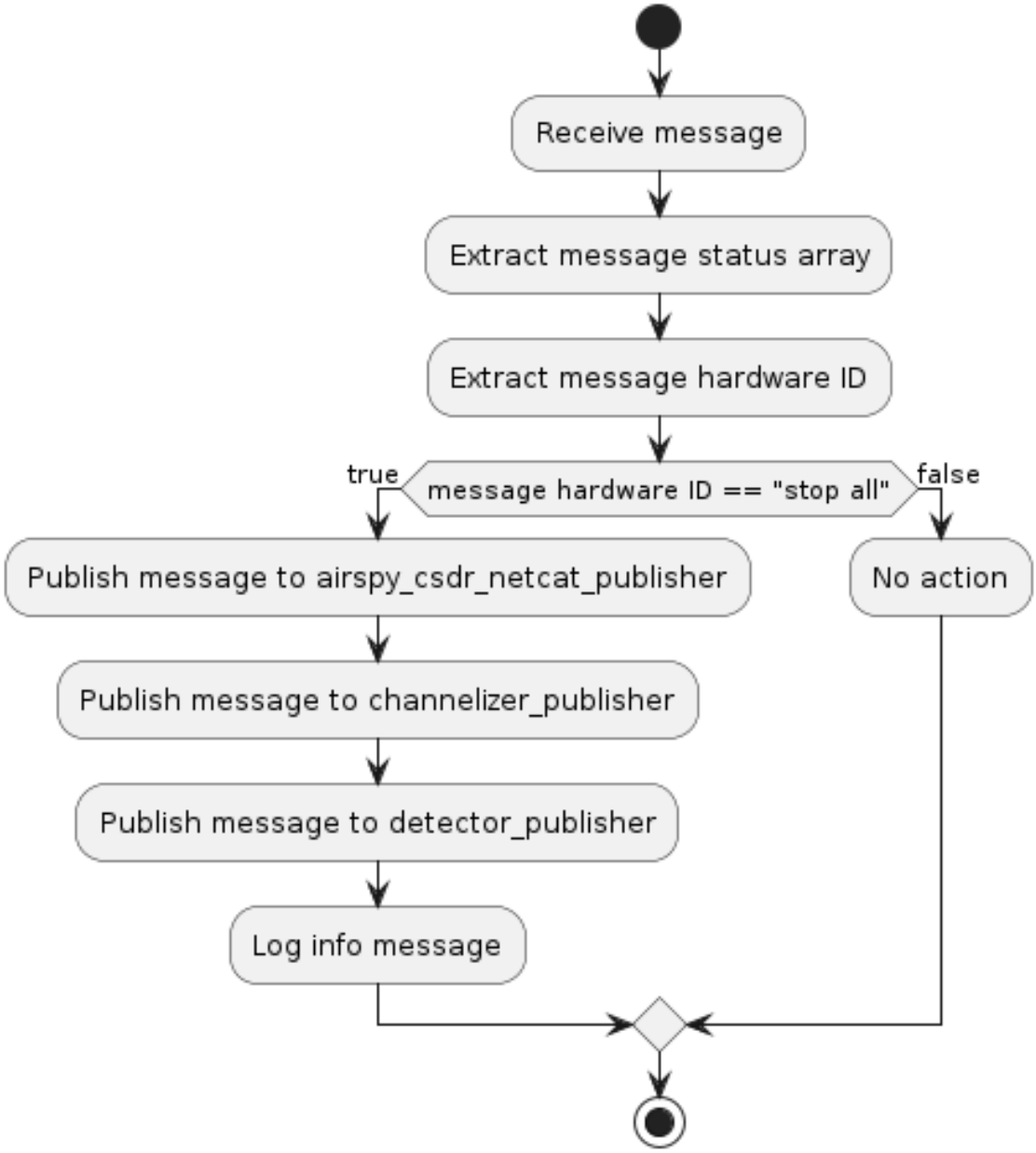

Figure 3.8: An activity diagram depicting the process of publishing stop messages to the different components.

Similarly, the _control_start_callback and _control_stop_callback methods handle messages received on the topics "control_start_subprocess" and "control_stop_subprocess",

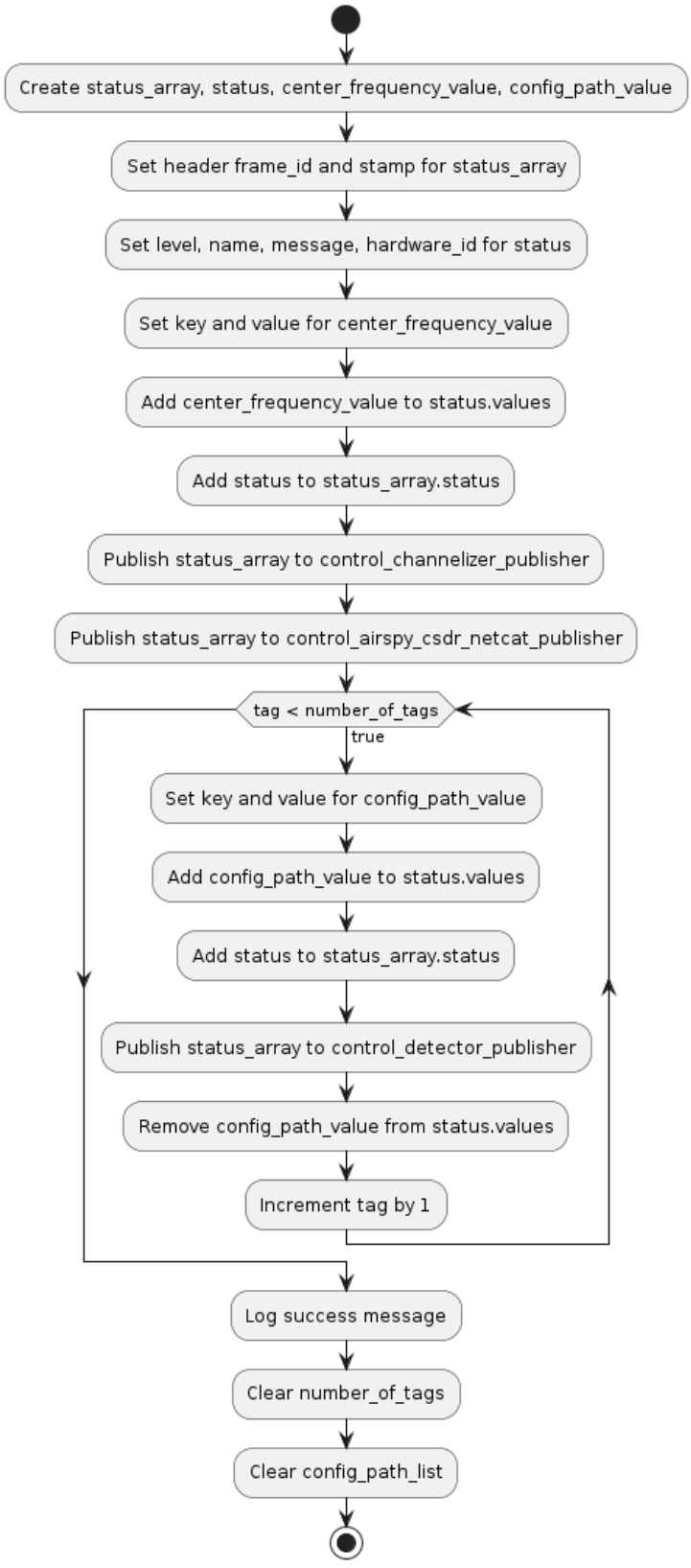

Figure 3.9: An activity diagram depicting the process of publishing start messages to the different components.

respectively. The main purpose of these methods is to start and stop the airspy_csdr_netcat_publisher, channelizer, and detector subprocesses via publishers. These methods create diagnostic messages, set values for the attributes associated with the diagnostic message, publish the diagnostic messages using the appropriate publishers, and log success messages. Figure 3.9 depicts the step-by-step process of _control_start_callback method. Figure 3.8 depicts the step-by-step process of _control_stop_callback method.

The _create_log_directories method is responsible for creating log directories for each flight. It creates directories for airspy_csdr_netcat logs, airspyhf_channelizer logs, and detector logs. It also creates individual detector directories, and then output and configuration directories inside each detector directory. The method then logs success messages and appends the configuration paths to the _config_path_list class variable to be published to the DetectorComponent class in the detector_component.py file. Figure 3.10 illustrates the flow of the _create_log_directories method execution.

The _create_config_file method is used to create a configuration file for each detector. It generates the path for the configuration file and the data record bin file, as well as calculates values such as the center frequency, port numbers, and sampling rate. The calculated values and necessary configuration information are then written to the configuration file. This process is demonstrated in Figure 3.11.

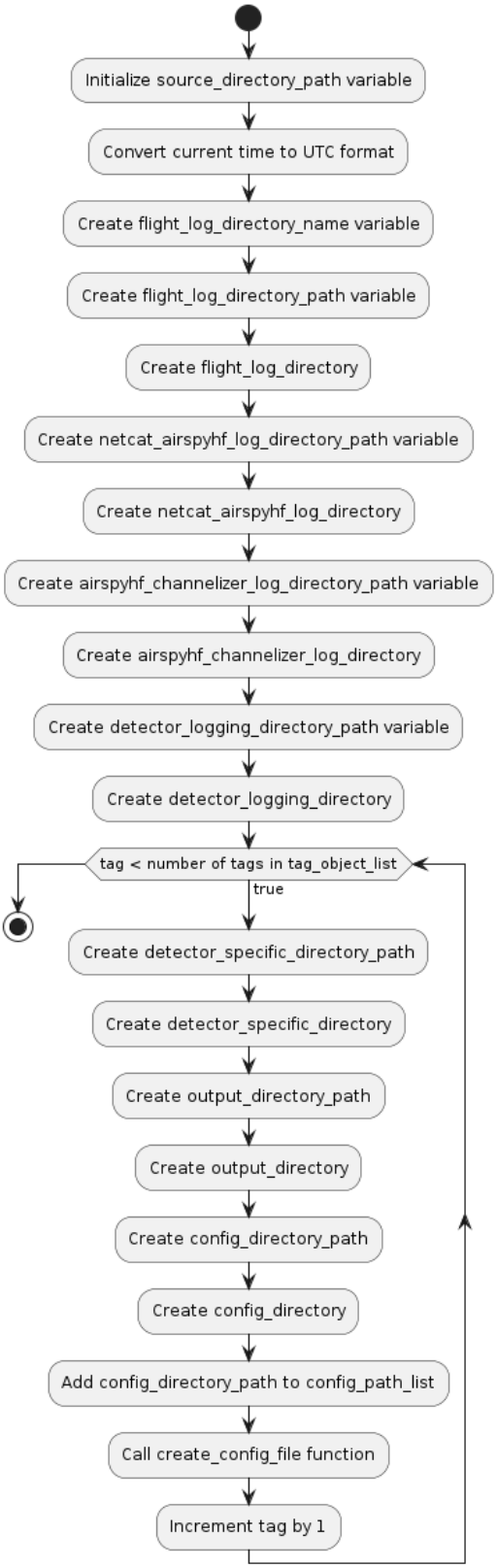

Figure 3.10: An activity diagram illustrating the log directory creation process.

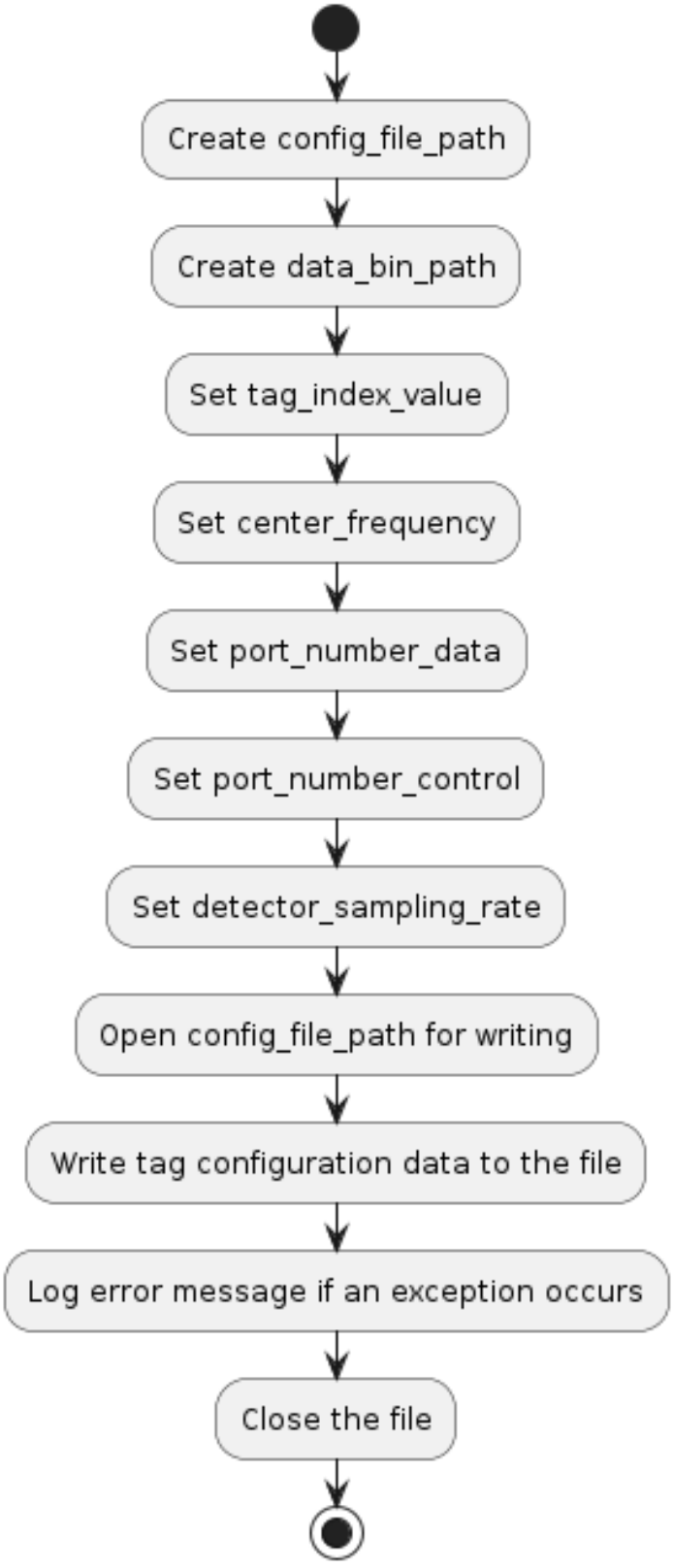

Figure 3.11: An activity diagram illustrating the configuration file creation process.

3.2.3 *airspy_csdr_netcat_component.py, channelizer_component.py, and detector_component.py*

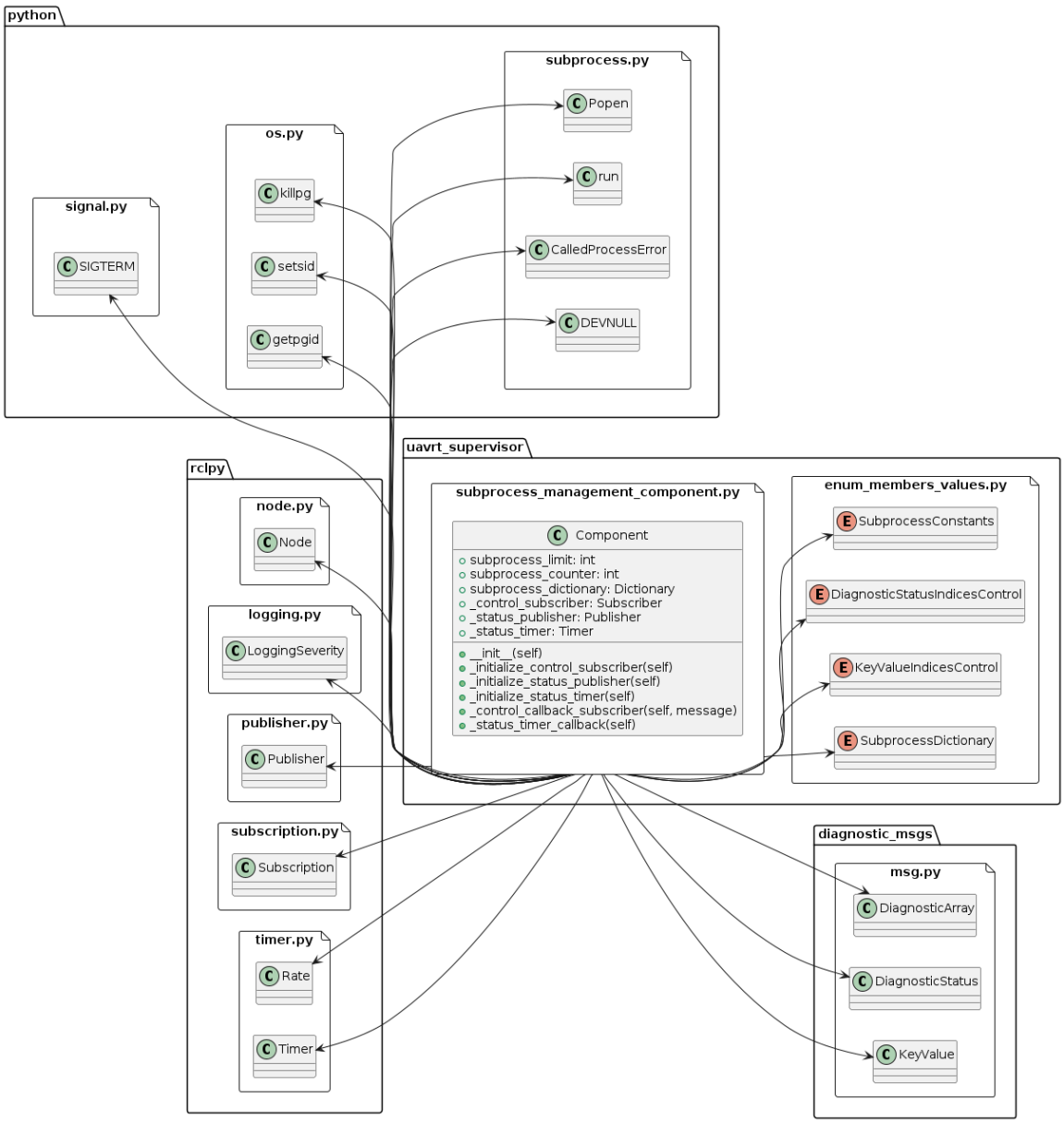

Figure 3.12: A class diagram representing the structure and interactions of the classes in each of the airspy_csdr_net_component.py, channelizer_ component.py, and detector_component files.

The airspy_csdr_netcat_component.py, channelizer_component.py, and detector_ component files reside in the uavrt_supervisor package and contain the AirspyCS-

DRNetcatComponent, ChannelizerComponent, and DetectorComponent classes, respectively. The primary function of these classes is to manage the initiation and termination of subprocesses via ROS 2 subscribers. The AirspyCSDRNetcatComponent class manages the airspy_rx, CSDR, and Netcat subprocesses, the ChannelizerComponent class manages the channelizer subprocess, and the DetectorComponent manages the detector subprocesses. Additionally, these classes publish status updates regarding these subprocesses through ROS 2 publishers. Lastly, a ROS 2 timer is employed in each of the classes to periodically monitor the status of the aforementioned subprocesses. These files are depicted in Figure 3.12.

The AirspyCSDRNetcatComponent, ChannelizerComponent, and DetectorComponent classes are similar enough in imports, structure, methods, and functionality that this section will provide an overview that is applicable to all three classes. Small differences between the classes will mentioned when necessary.

The airspy_csdr_netcat_component.py, channelizer_component.py, and detector_component files begin by importing the necessary modules from rclpy and other modules from the uavrt_supervisor package. These imports are described in Table 3.5 and Table 3.6. Certain files might import additional methods. For example, the run method from the subprocess module was imported into the airspy_csdr_netcat_component.py file to test and ensure that the Airspy SDR was plugged into the machine. The functionality from the run method was not necessary in the other component files.

The AirspyCSDRNetcatComponent, ChannelizerComponent, and DetectorComponent classes then initializes a limit for the number of subprocesses that can be active at once, a counter to keep track of the number of subprocesses, and a dictionary for storing the subprocess objects. Each of the classes then defines three private methods prefixed with _initialize_ that are responsible for initializing the subscriber,

Imported library or module	Provided functionality
subprocess	The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes [96].
os	This module provides a portable way of using operating system dependent functionality [94].
signal	This module provides mechanisms to use signal handlers in Python [95].
rclpy.node	A Node is the primary entry point in a ROS system for communication. It can be used to create ROS entities such as publishers, subscribers, services, etc. [74].
rclpy.logging	The logging subsystem in ROS 2 aims to deliver logging messages to a variety of targets [73].
rclpy.publisher	A publisher is used as a primary means of communication in a ROS system by publishing messages on a ROS topic [76].
rclpy.subscription	A publisher is used to catch messages that are published to topics [76].
rclpy.timer	Timers let you schedule a callback to happen at a specific rate [75].

Table 3.5: Imported rclpy and uavrt_supervisor modules used in the airspy_csdr_netcat_component.py, channelizer_component.py, and detector_component files.

diagnostic_msgs	Diagnostic messages which provide the standardized interface for the diagnostic and runtime monitoring systems in ROS/ROS 2 [86].
enum_members_values	Enum values that are used in the uavrt_supervisor package to describe the index that is being accessed.

Table 3.6: Imported rclpy and uavrt_supervisor modules used in the airspy_csdr_ netcat_component.py, channelizer_component.py, and detector_component files. (cont.)

publisher, and timer used in the component's class. It also sets the logging level to INFO and logs informational messages. The ChannelizerComponent class also initializes a variable for storing the path where the channelizer code was installed, while the DetectorComponent classes initializes a variable for storing the directory where the detector code has been installed.

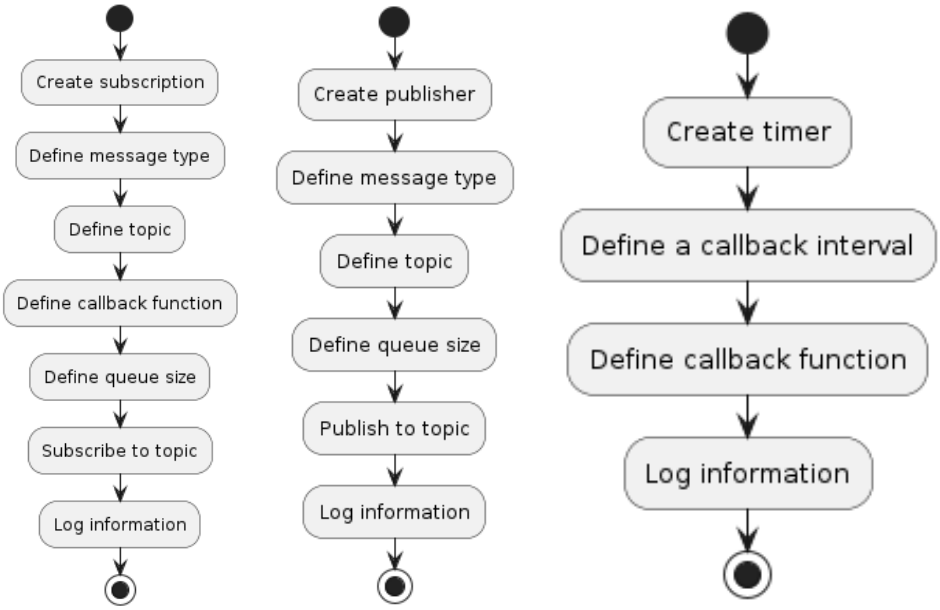

Figure 3.13: Left: The process of creating a subscriber, defining a message type, topic, callback function, queue size, subscribing to the topic, and logging information. Middle: The process of creating a publisher, defining a message type, topic, queue size, publishing to the topic, and logging information. Right: The process of creating a timer, defining a callback interval, callback function, and logging information.

The _initialize_control_subscriber method sets up a ROS 2 subscriber to listen to the "control_*__subprocess" topic, which receives control commands for starting and stopping the subprocess or subprocesses. For example, the ROS 2 subscriber for the AirspyCSDRNetcatComponent would be subscribed to the "control_airspy_csdr_netcat_subprocess" topic. The _initialize_status_publisher method sets up a ROS 2 publisher to send status information to the "status_*__subprocess" topic. The _initialize_status_timer method creates a ROS 2 timer that periodically triggers the _status_timer_ callback method responsible poll subprocesses in the subprocess_dictionary variable to determine the status of a subprocess object, and then publishing the status of that subprocesses object. This flow is shown in Figure 3.13.

The _control_callback_subscriber method serves as the callback for the ROS 2 _initialize_control_subscriber subscriber that receives control messages for stopping and starting subprocesses. The method begins by extracting various information from the received message object, such as message type, time, and status attributes. The _control_callback_subscriber method then checks the content of the control message to determine whether to start or stop the subprocess. This process is shown in Figure 3.14.

If the message contains the command "start," the _control_callback_subscriber method constructs a command string to start the subprocess using various parameters. For example, the AirspyCSDRNetcatComponent class would use the center frequency and sampling rate variables, along with hardcoded airspy_rx, CSDR and Netcat keywords, in order to create the command string. The _control_callback_subscriber method also performs checks to ensure that the maximum number of allowed subprocesses has not been been exceeded before attempting to start a new subprocess. If the maximum number of allowed subprocesses has not been exceeded, then a new subprocess is started using Popen. The subprocess is added to the subprocess_dictionary

Figure 3.14: A activity diagram illustrating the process of receiving and acting on control messages for stopping and starting subprocesses.

class variable, where the key of the dictionary entry is equal the original message's hardware ID and the value is equal to the subprocess object. The subprocess_counter is also incremented by one. Then the _control_callback_subscriber method logs a message indicating the start of a new subprocess.

In case of an exception or error during the subprocess start attempt, the _control_ callback_subscriber method catches the exception, and updates the status from the

originally received message to indicate an error. The exception is then logged along with the details of the error. Aside from an exception being raised for when the subprocess limit is hit, an exception can be raised in the AirspyCSDRNetcatComponent class when the run method is being used to test whether the Airspy SDR is plugged into the machine. If the Airspy SDR is not plugged in, a "CalledProcessError" exception will be raised.

If the message contains the command "stop" and the specific hardware ID "stop all", the _control_callback_subscriber method iterates through the subprocess_dictionary, terminating each subprocess using the killpg and getpgid functions, and removing them from the subprocess_dictionary. If any exceptions occur during this process (e.g., the number of subprocesses is already 0), they are caught and logged. The method also updates the status of the originally received message to reflect the subprocesses' being stopped.

Finally, regardless of whether a "start" or "stop" command was received, the _control_callback_subscriber method publishes the updated status message to the appropriate topic using the _status_publisher.

It is important to note that there are additional semantics associated with starting the channelizer and detector subprocesses. With both the channelizer and detector subprocesses, it is essential that the "LD_LIBRARY_PATH" environmental variable be set to where the channelizer installation directory resides. In the case of the detector subprocesses, the "HOME" environmental variable also needs to be set. Additionally, for the detector subprocesses, it is required that the machine's ROS 2 installation be sourced, the package's installation files are sourced, and then the run command is called from the appropriate detector configuration directory.

The _status_timer_callback method serves as the callback for the ROS 2 _initialize_status_timer timer, which periodically polls subprocesses in the subprocess_dictionary

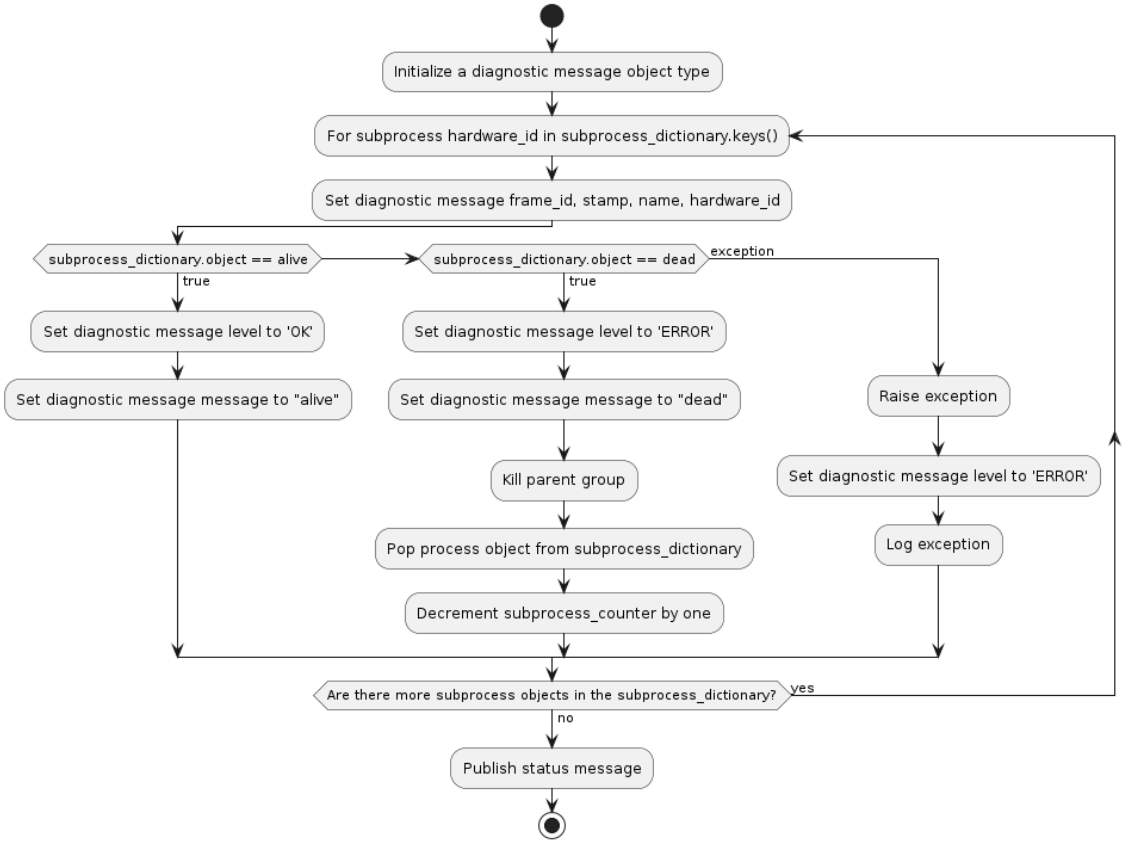

Figure 3.15: An activity diagram illustrating the process of periodically polling subprocesses in the subprocess_dictionary variable to determine the subprocess's status.

variable to determine the subprocess's status. This method begins by initializing a diagnostic message object type, which will contain the following attributes: a frame id, stamp, name, hardware id, level, and a message.

The _status_timer_callback method then begins to iterate over each of the objects in the subprocess_dictionary variable in a try block. For each iteration, the method polls the subprocess object at the current index and determines whether it is alive or dead. If it is alive, the level of the diagnostic message is set to "OK," and the message is set to "alive." If it is dead, the level is set to "ERROR," and the message is set to "dead." Additionally, the parent group associated with the process ID of the subprocess is killed, the subprocess object is removed from the subpro-

cess_dictionary variable, and the subprocess counter is decremented by one. Then, the _status_timer_callback method logs a message indicating that a subprocess has been stopped. This process is repeated for all subprocesses in the subprocess_dictionary variable.

If an exception is raised during this process, the level of the diagnostic message is set to "ERROR" and the exception is logged. Finally, the _status_timer_callback method publishes to the "status_*__subprocess" topic using the _status_publisher. This flow is shown in Figure 3.15.

3.2.4 enum_members_values.py

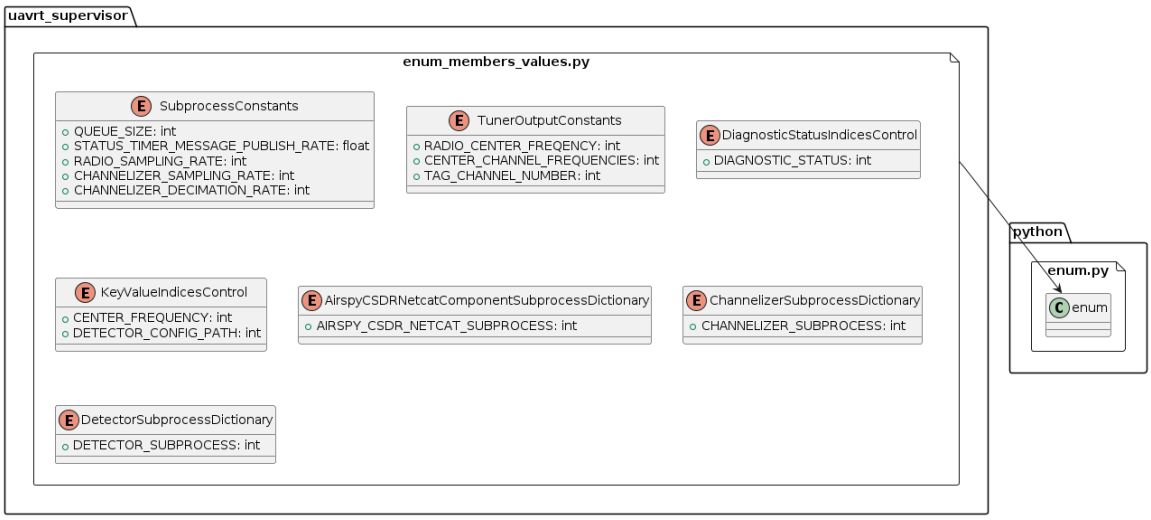

Figure 3.16: A class diagram representing the structure of the classes in the enum_members_values.py file.

The enum_members_values.py file defines seven classes containing enumerations with named constants. These constants are used to represent specific values that are used in the airspy_csdr_netcat_component.py, channelizer_component.py, and detector_component files. The classes in enum_members_values.py are represented in Figure 3.16.

The SubprocessConstants class defines the following constants: QUEUE_SIZE,

STATUS_TIMER_MESSAGE_PUBLISH_RATE, RADIO_SAMPLING_RATE, CHANNELIZER_SAMPLING_RATE, and CHANNELIZER_DECIMATION_RATE. The QUEUE_SIZE constant defines a value that is a required QoS (quality of service) setting for ROS 2 objects, such as publishers, subscribers, and timers. The QoS setting limits the amount of queued messages if a subscriber is not receiving them fast enough. The STATUS_TIMER_MESSAGE_PUBLISH_RATE constant defines the rate at which status timer messages will be checked and published, measured in seconds. The RADIO_SAMPLING_ RATE constant represents the sampling rate used by airspy_rx to determine the rate of which samples are collected. The CHANNELIZER_SAMPLING_RATE and CHANNELIZER_DECIMATION_RATE constants are used for operating the channelizer subprocess. The CHANNELIZER_SAMPLING_RATE determines the number of samples that will be collected by each channel in the channelizer, while the CHANNELIZER_DECIMATION_RATE specifies the number of channels that the channelizer will construct.

The TunerOutputConstants class defines the following constants: RADIO_ CENTER_FREQENCY, CENTER_CHANNEL_FREQUENCIES, and TAG_CHANNEL_NUMBER. These constants are used to determine where certain values are located in the output array of the tuner function. For example, the value for the optimal radio center frequency (as determined by the tuner function) is found at index 0 in the array of outputs.

The DiagnosticStatusIndicesControl class defines the DIAGNOSTIC_STATUS value, which signifies the index where the status portion of the diagnostic message is stored. The KeyValueIndicesControl class defines the CENTER_FREQUENCY and DETECTOR_CONFIG_PATH constants. These constants are similar to the DIAGNOSTIC_STATUS value, in that they signify the index where certain values are stored in the values array. For an overview of the ROS 2 Diagnostic Message

type, refer to Subsection 2.2.3.

The AirspyCSDRNetcatComponentSubprocessDictionary, ChannelizerSubprocess-Dictionary, and DetectorSubprocessDictionary classes each define a *_SUBPROCESS constant. This constant is used to signify where subprocess objects are stored in the subprocess_dictionary. These classes are kept separate in case additional information is stored in the subprocess_dictionary variable. For example, the center frequency that is used to start subprocesses in the AirspyCSDRNetcatComponent class can also be stored in the subprocess_dictionary. The center frequency would be part of the same key and value pair. In the the event that a subprocess needs to be reset, the center frequency would be easily accessible, as it would be paired with the subprocess object.

3.2.5 tuner.py

The tuner.py file resides in the uavrt_supervisor package and defines the tuner function, which is used in the start_stop_component.py to assist in the creation of configuration files. This function was written and maintained by Dr. Michael Shafer. The purpose of the tuner function is to optimize the selection of a radio's center frequency. Figure 3.17 describes the process followed by the tuner function.

This function aims to choose the center frequency that minimizes the presence of tags in each channel and subsequently reduces the edge cost. The edge cost for each channel is determined based on the proximity of tags to the channel edges, with tags closer to the edge incurring a higher cost. Tags located precisely at the channel edge are assigned an infinite cost. Minimizing this cost is crucial to position tag frequencies away from the shoulder roll-off of each channel.

Furthermore, the program generates various outputs related to the channels, as described in the outputs section. It is important to note that the program will raise

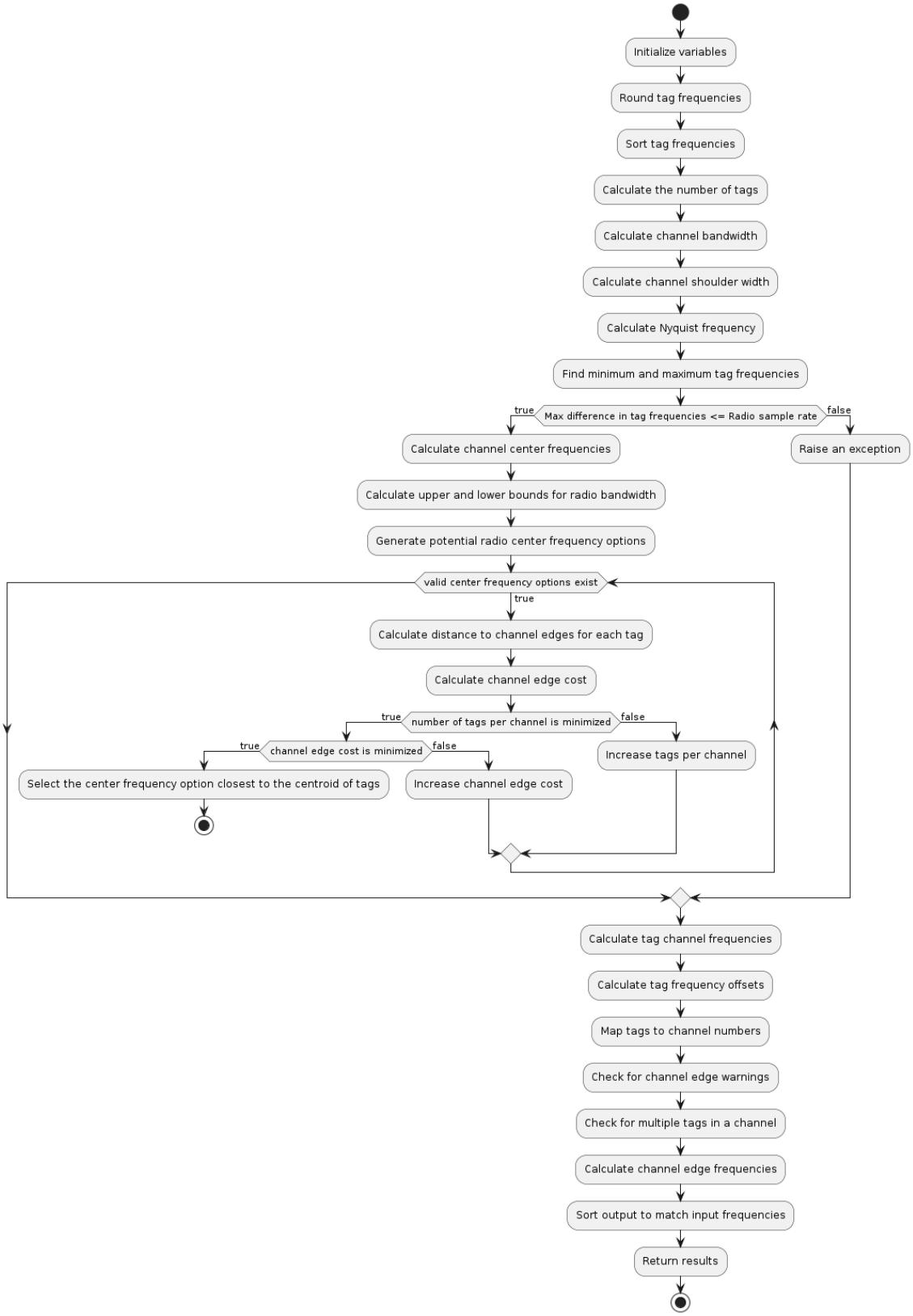

Figure 3.17: An activity diagram representing the flow of the tuner function in the tuner.py file.

an exception if the frequency range defined by the input tag specifications exceeds the radio's available bandwidth.

3.3 uavrt_connection

package

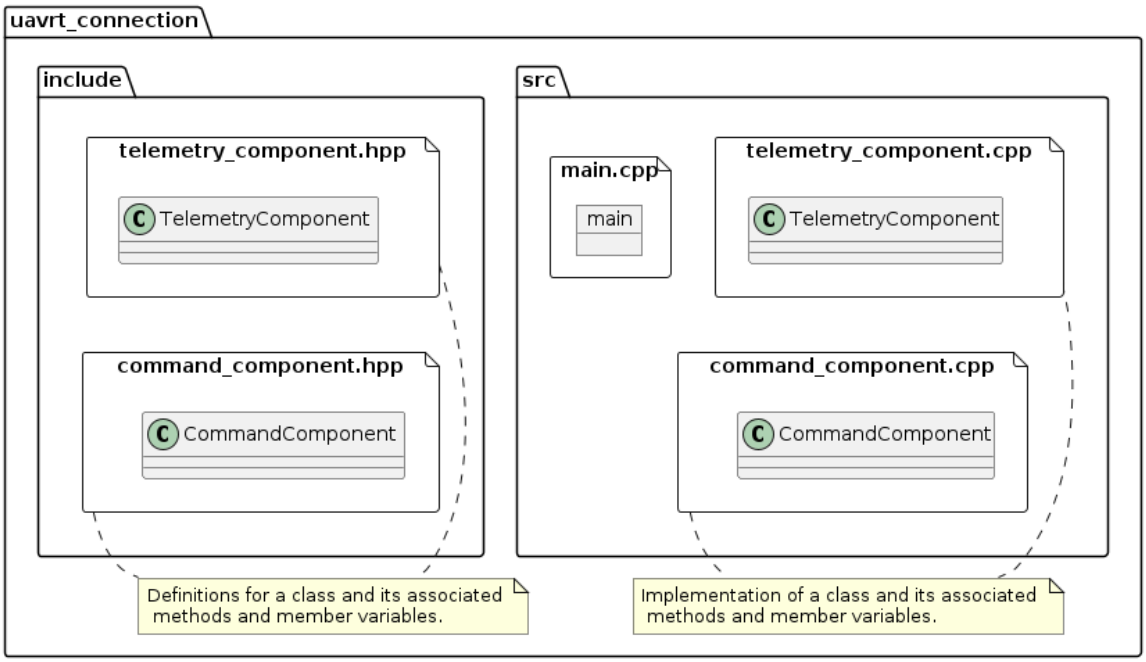

Figure 3.18: A class diagram depicting the files and classes in the uavrt_connection package.

Figure 3.18 represents the directory and file hierarchy for the uavrt_connection package.

3.3.1 main.cpp

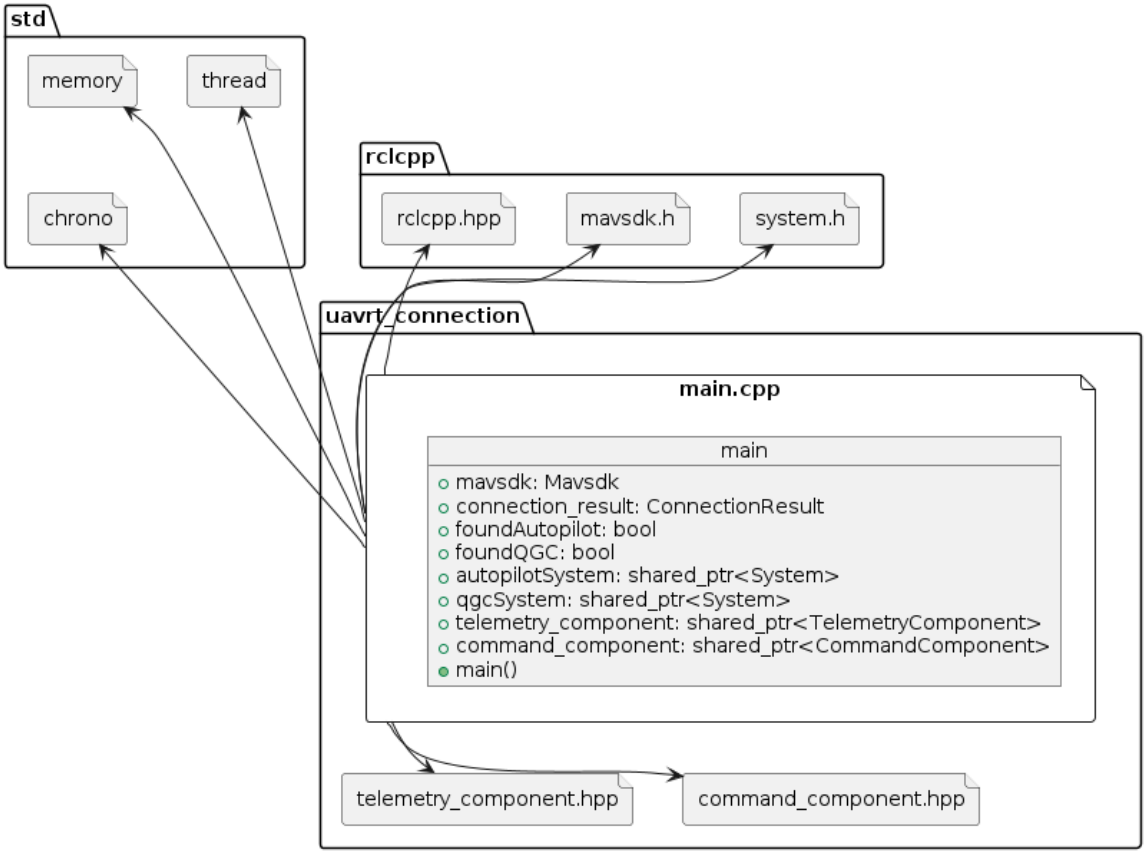

Figure 3.19: A class diagram illustrating the main.py file and its imports and the variables declared in the main function.

The main.cpp file acts as the entry point to the uavrt_connection package. Figure 3.19 represents the structure of the main.cpp file. The main functionality of this file is to gather input from the user, establish a connection with the flight controller and QGroundControl, and spin an executor that is operating two uavrt_connection components: the TelemetryComponent and the CommandComponent.

The main.cpp file begins by importing the necessary C++ standard library header files and other header files. These imports are described in Table 3.7.

After the imports, the variables and functions in the main.cpp file are defined. Figure 3.20 is an activity diagram that describes the flow of the main function. This

Imported library or header	Provided functionality
memory	This header defines general utilities to manage dynamic memory [17].
chrono	The chrono library defines three main types as well as utility functions and common typedefs: clocks, time points, and durations [13].
thread	This header is part of the thread support library and defines support for threads, atomic operations, mutual exclusion, condition variables, and futures [19].
rclcpp/rclcpp.hpp	rclcpp provides the canonical C++ API for interacting with ROS 2 [87].
mavsdk/mavsdk.h	This is the header file for the main class of MAVSDK (a MAVLink API Library) [41].
mavsdk/system.h	This class represents a system, made up of one or more components (e.g. autopilot, cameras, servos, gimbals, etc) [42].
uavrt_connection/ telemetry _component.hpp	The primary function of this file is to provide the TelemetryComponent class.
uavrt_connection/ command _component.hpp	The primary function of this file is to provide the CommandComponent class.

Table 3.7: Imported C++ standard library header files and other header files used in the main.cpp file.

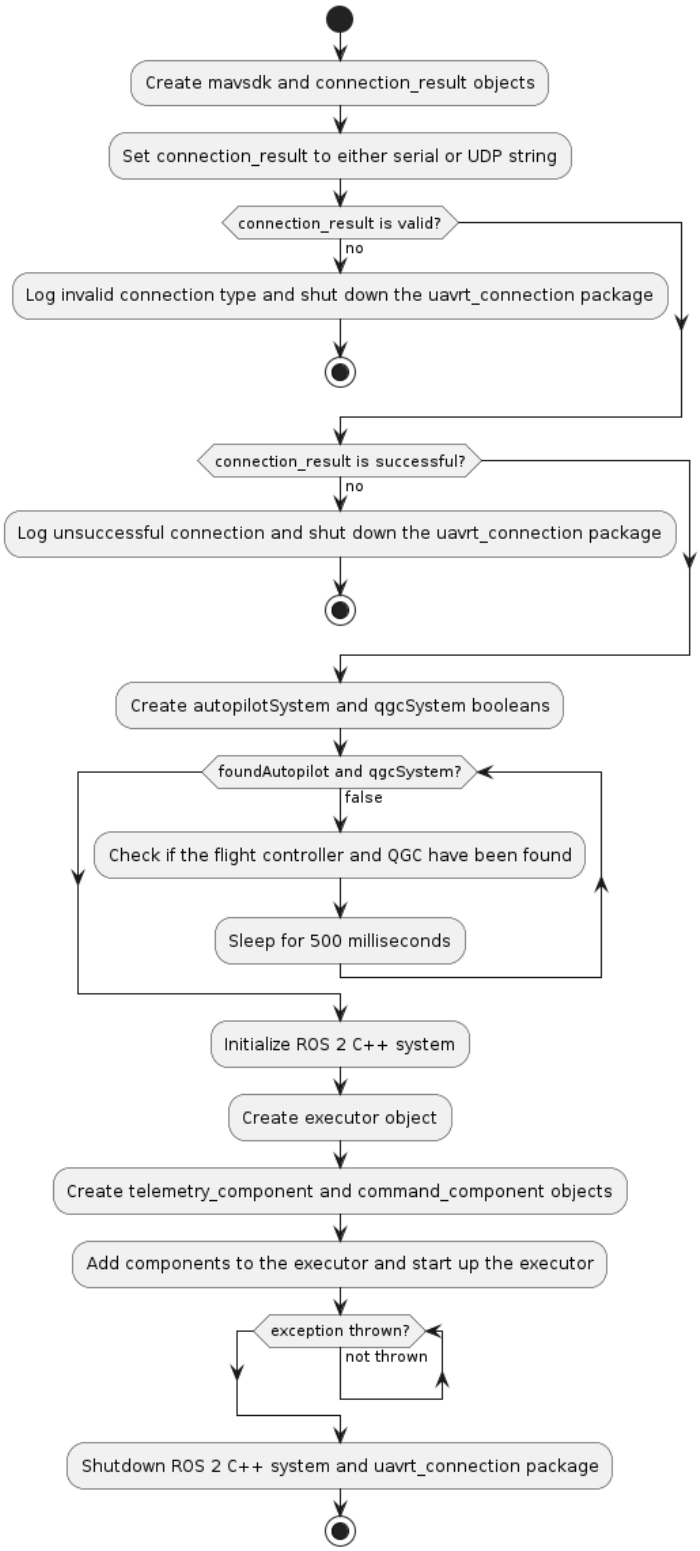

Figure 3.20: An activity diagram for the main function with main.cpp.

function first asks the user whether the connection to the PX4 flight controller is serial or UDP. For example, if the user wants to connect to the PX4 flight controller via serial, the string passed to the MAVSDK add_any_connection would be "serial:///dev/ttyUSB0". After a check to see whether the serial or UDP connection is alive, the main function will attempt to create two shared pointers: one shared pointer for the flight controller system object and one shared pointer for the custom QGroundControl system object. System objects are used to interact with UAVs (including their components), standalone cameras, and mission planners. They are not created directly by application code, but are returned by the Mavsdk class, which is a part of the MAVSDK library [42]. The main function will remain at this point until it is able to create the two shared pointers, as they are necessary for communication later. It is necessary to have QGroundControl running in the background at this point; otherwise, the main function will not progress.

Once the shared pointers have been created for both system objects, the main function can progress onto initializing ROS 2 for C++. This begins by calling rclcpp's init function and created a single threaded executor (similar to rclpy). This executor is used to operate two components in the uavrt_connection package: the Telemetry and Command components. After two shared pointers are created for the TelelmetryComponent and ConnectionComponent classes, those pointers are passed to the executor using the add_node function to begin operation. The executor is then spun up using the spin() function. The executor will continue to execute work as it becomes available on the two components, and will remain blocking in the main function. The executor can only be interrupted using Ctrl-C, an uncaught exception occurs with the TelelmetryComponent or ConnectionComponent objects, or the exectutor is shut down from in the TelelmetryComponent or ConnectionComponent objects. Once the executor is interrupted, the main function will shut down.

3.3.2 *telemetry_component.hpp*

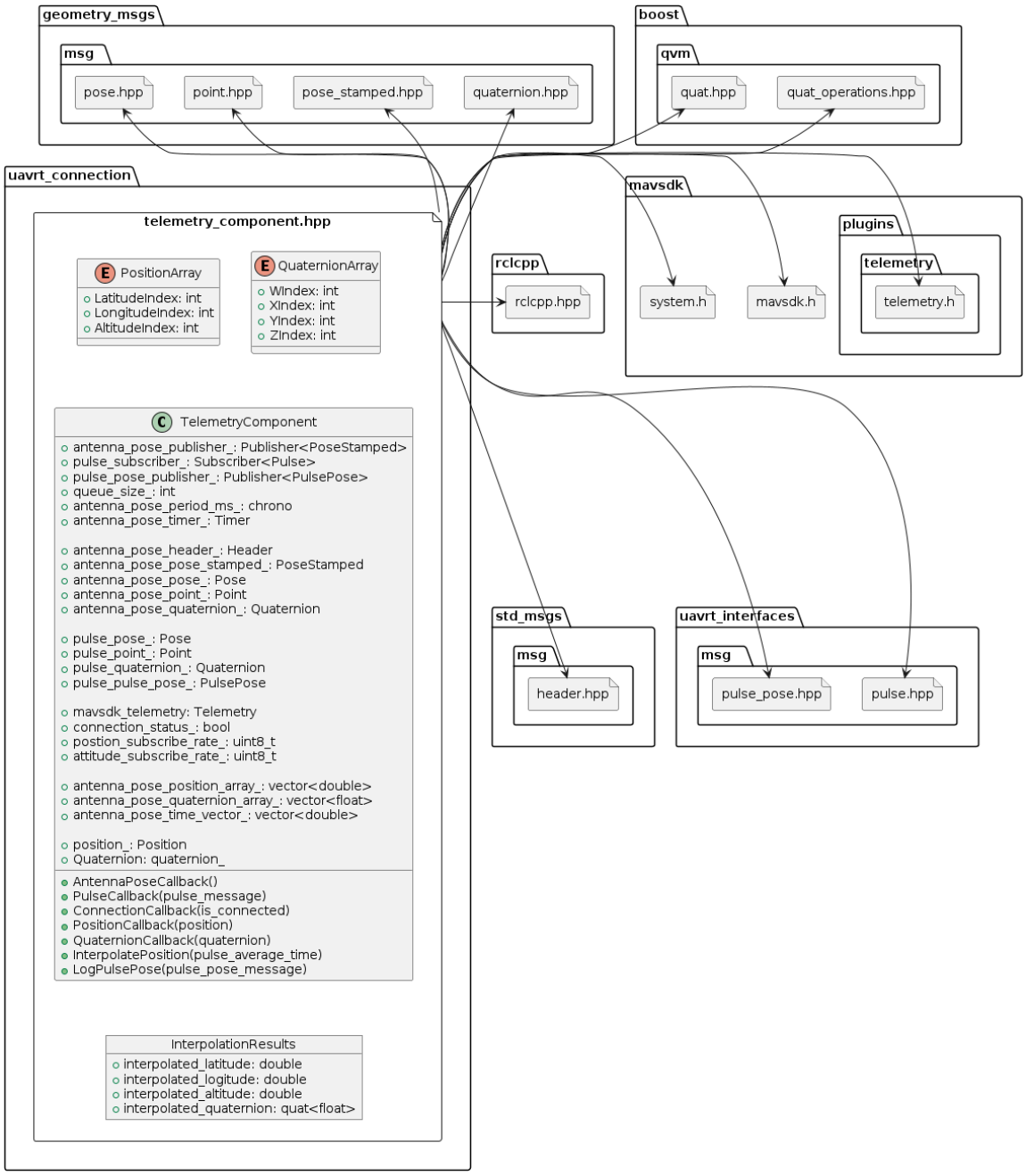

Figure 3.21: A class diagram illustrating the telemetry_component.hpp file, its imports, and the variables and functions declared in the TelemetryComponent class.

The telemetry_component.hpp file resides in the uavrt_connection package. The primary function of this file is to provide a class declaration for the TelemetryCompo-

nent class, as well as defining additional variables used in the telemetry_component.cpp file. Header files from ROS 2, MAVSDK, and Boost libraries, and the uavrt_interfaces package are imported in and the telemetry_component.hpp file. The structure of the telemetry_component.hpp file is described in Figure 3.21. The imports are described in Table 3.8 and Table 3.9.

Imported library or header	Provided functionality
rclcpp/rclcpp.hpp	rclcpp provides the canonical C++ API for interacting with ROS 2 [87].
std_msgs/msg/ header.hpp	Standard metadata for higher-level stamped data types. This is generally used to communicate timestamped data in a particular coordinate frame [82].
geometry_msgs/msg/ pose.hpp	A representation of pose in free space, composed of position and orientation [78].
geometry_msgs/msg/ point.hpp	This contains the position of a point in free space [77].
geometry_msgs/msg/ pose_stamped.hpp	This represents a Point with reference coordinate frame and timestamp [79].
geometry_msgs/msg/ quaternion.hpp	This represents an orientation in free space in quaternion form [80].

Table 3.8: Imported ROS 2, MAVSDK, and Boost library header files and uavrt_ interfaces package header files used in the telemetry_component.hpp file.

mavsdk/mavsdk.h	This is the header file for the main class of MAVSDK (a MAVLink API Library). It is used to discover vehicles and manage active connections [41].
mavsdk/system.h	This class represents a system, made up of one or more components (e.g. autopilot, cameras, servos, gimbals, etc) [42].
mavsdk/plugins/ telemetry/telemetry.h	Allow users to get vehicle telemetry and state information (e.g. battery, GPS, RC connection, flight mode etc.) and set telemetry update rates [43].
boost/qvm/quat.hpp	This header defines the struct quat template, which acts as a generic quaternion type [7].
boost/qvm/ quat_operations.hpp	This header defines all available function overloads that operate on quaternion objects [8].
uavrt_interfaces/msg/ pulse.hpp	This file provides the structure and data types for the custom "pulse" message.
uavrt_interfaces/msg/ pulse_pose.hpp	This file provides the structure and data types for the custom "pulse_pose" message.

Table 3.9: Imported ROS 2, MAVSDK, and Boost library header files and uavrt_ interfaces package header files used in the telemetry_component.hpp file (cont).

3.3.3 *telemetry_component.cpp*

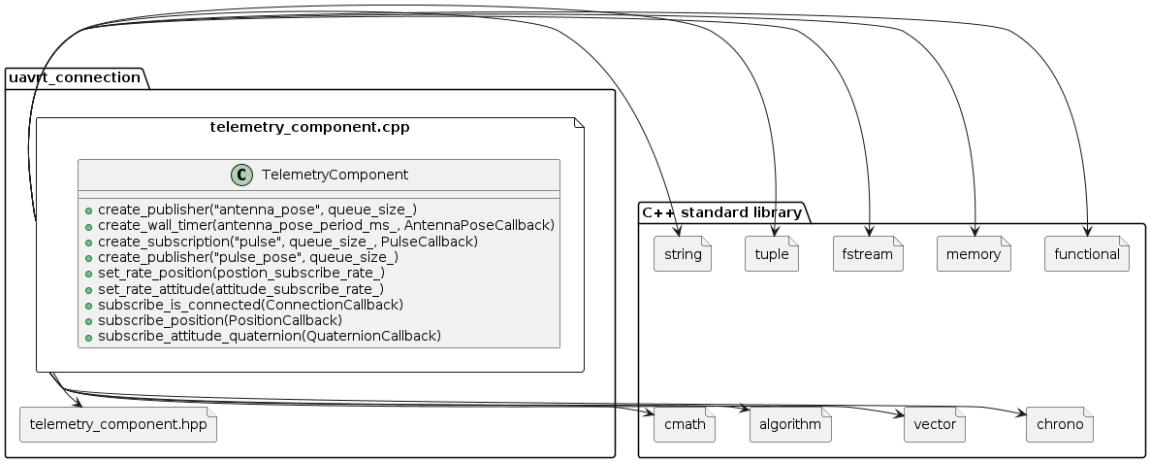

Figure 3.22: A class diagram illustrating the telemetry_component.cpp file, its imports, and the variables and functions declared in the TelemetryComponent class.

The telemetry_component.cpp file resides in the uavrt_connection package. Figure 3.22 represents the structure of the telemetry_component.cpp file. The primary function of this file is to provide the TelemetryComponent class. The TelemetryComponent class manages the connection between other parts of the system, including the flight controller (via MAVSDK), custom QGroundControl, and the uavrt_detection package. The TelemetryComponent class will collect position and quaternion data from the flight controller (via MAVSDK) and keep a constantly updating record of the global coordinates (position data) and position in free space (quaternion data) of the UAV. While collecting detected pulses from the uavrt_detection package, the TelemetryComponent class will interpolate the position and quaternion data associated with the detected pulses. The detected pulses, along with the interpolated data, will then be published under the "pulse_pose" topic so that it can be collected by the CommandComponent. The pulse_pose information will also be logged to a text file and stored on the disk of the companion computer.

Imported library or header	Provided functionality
chrono	The chrono library defines three main types as well as utility functions and common typedefs: clocks, time points, and durations [13].
functional	This header is part of the function objects library and provides the standard hash function. [16].
memory	This header defines general utilities to manage dynamic memory [17].
vector	This header is part of the containers library and defines the vector container [20].
tuple	This header is part of the general utility library and defines the tuple container [11].
algorithm	This header is part of the algorithm library defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements [12].
string	This header is part of the strings library and defines a container for storing a series of characters [18].
cmath	This header is part of the numeric library and includes common mathematical functions and types, as well as optimized numeric arrays and support for random number generation. [14].

Table 3.11: Imported C++ standard library header files and other header files used in the telemetry_component.cpp file.

fstream	This header is part of the Input/Output library and is an OOP-style stream-based I/O library, print-based family of functions (since C++23), and the standard set of C-style I/O functions [15].
uavrt_connection/ telemetry _component.hpp	The header file for the telemetry_component.cpp file.

Table 3.10: Imported C++ standard library header files and other header files used in the telemetry_component.cpp file. (cont.)

The telemetry_component.cpp file begins by importing the necessary C++ standard library header files and the telemetry_component.hpp header file. These imports are described in Table 3.11 and Table 3.10.

3.3.4 *command_component.hpp*

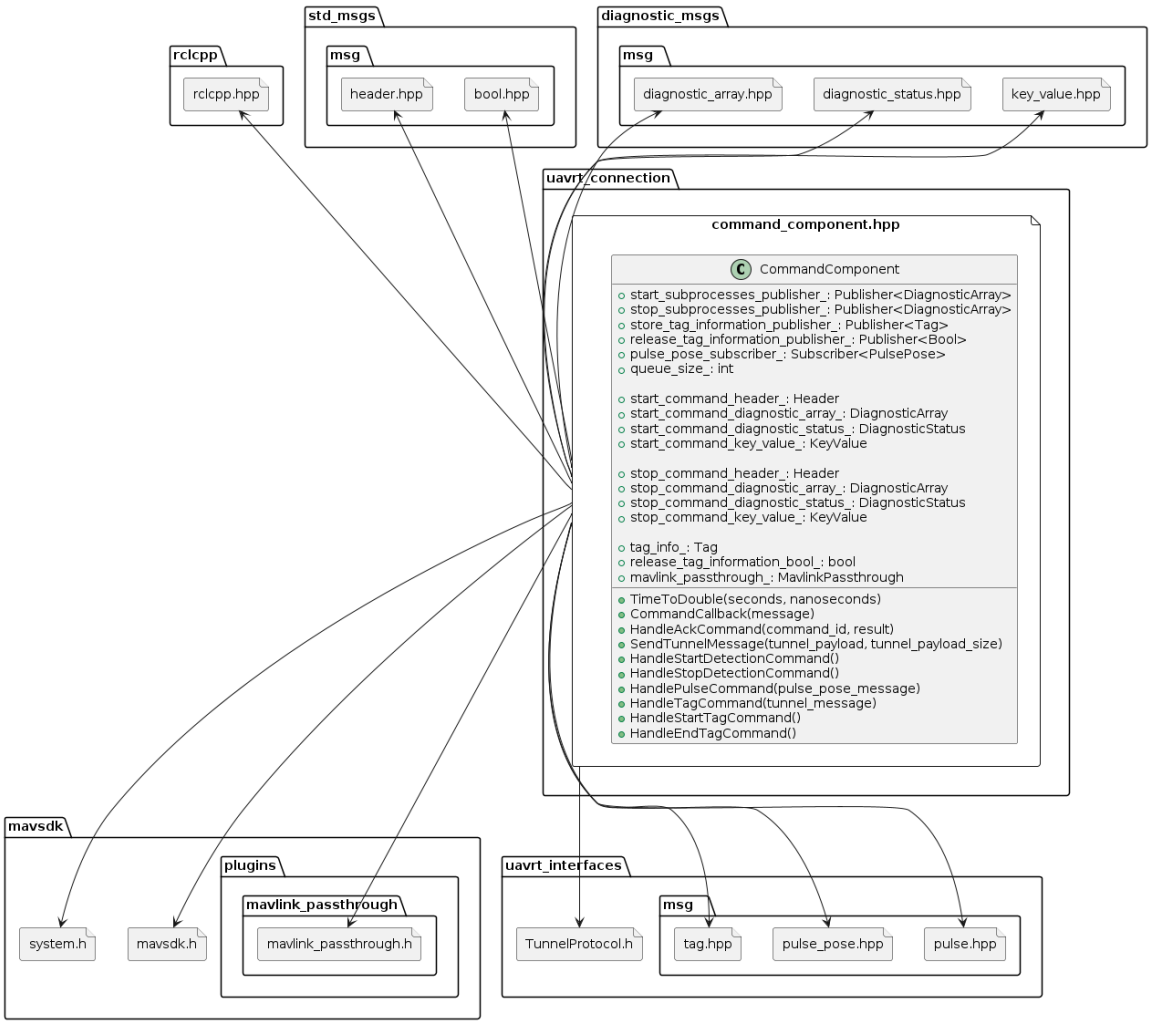

Figure 3.23: A class diagram illustrating the command_component.hpp file, its imports, and the variables and functions declared in the CommandComponent class.

The command_component.hpp file resides in the uavrt_connection package. The primary function of this file is to provide a class declaration for the CommandComponent class, as well as defining additional variables used in the command_component.cpp file. Header files from ROS 2, MAVSDK, and Boost libraries, as well the uavrt_interfaces package, are imported in and the command_component.hpp file. The structure of the telemetry_component.hpp file is described in Figure 3.23. These imports are described

in Table 3.12 and Table 3.13.

Imported library or header	Provided functionality
rclcpp/rclcpp.hpp	rclcpp provides the canonical C++ API for interacting with ROS 2 [87].
std_msgs/msg/ bool.hpp	Standard metadata for higher-level stamped data types. This is generally used to communicate timestamped data in a particular coordinate frame [81].
diagnostic_msgs/msg/ diagnostic_array.hpp	This message is used to send diagnostic information about the state of the robot [83].
diagnostic_msgs/msg/ diagnostic_status.hpp	This message holds the status of an individual component of the system [84].
diagnostic_msgs/msg/ key_value.hpp	Key to access a value that is tracked over time [85].

Table 3.12: Imported ROS 2 and MAVSDK library header files and uavrt_interfaces package header files used in the command_component.hpp file.

mavsdk/mavsdk.h	This is the header file for the main class of MAVSDK (a MAVLink API Library). It is used to discover vehicles and manage active connections [41].
mavsdk/system.h	This class represents a system, made up of one or more components (e.g. autopilot, cameras, servos, gimbals, etc) [42].
mavsdk/plugins/ mavlink_passthrough/ mavlink_passthrough.h	This plugin allows you to send and receive MAVLink messages [40].
uavrt_interfaces/msg/ pulse.hpp	This file provides the structure and data types for the custom "pulse" message.
uavrt_interfaces/msg/ pulse_pose.hpp	This file provides the structure and data types for the custom "pulse_pose" message.
uavrt_interfaces/msg/ tag.hpp	This file provides the structure and data types for the custom "tag" message.
uavrt_interfaces/ TunnelProtocol.h	This file provides the structs necessary to construct Tunnel Protocol messages.

Table 3.13: Imported ROS 2 and MAVSDK library header files and uavrt_interfaces package header files used in the command_component.hpp file (cont).

3.3.5 *command_component.cpp*

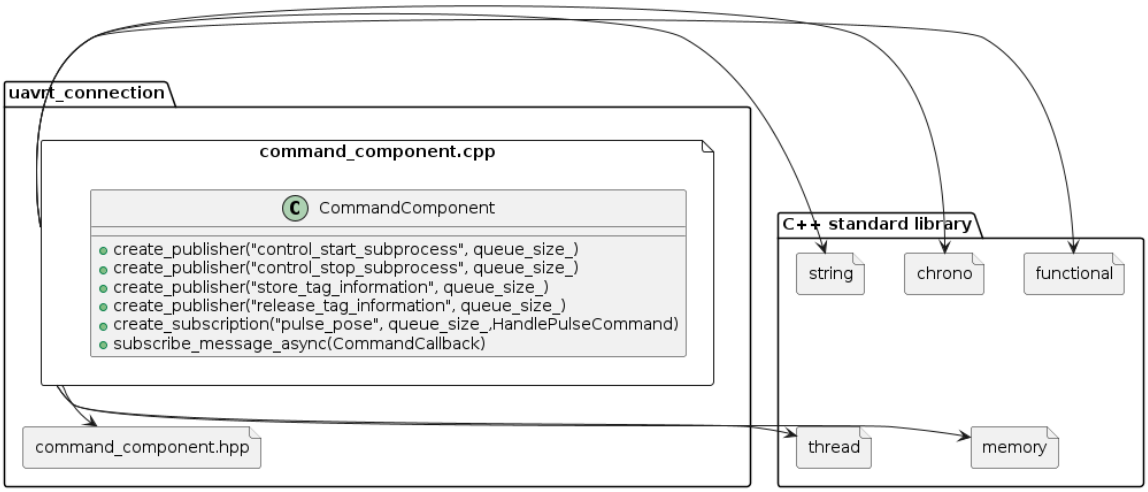

Figure 3.24: A class diagram illustrating the command_component.cpp file, its imports, and the variables and functions declared in the CommandComponent class.

The command_component.cpp file resides in the uavrt_connection package. Figure 3.24 represents the structure of the command_component.cpp file. The primary function of this file is to provide the CommandComponent class. The CommandComponent class manages incoming and outgoing messages between custom QGroundControl and the UAV-RT software system. This includes receiving start and stop commands and tag information from custom QGroundControl, as well as sending detected pulse and associated interpolated position and quaternion data to custom QGroundControl.

The command_component.cpp file begins by importing the necessary C++ standard library header files and the command_component.hpp header file. These imports are described in Table 3.14.

Imported library or header	Provided functionality
functional	This header is part of the function objects library and provides the standard hash function. [16].
memory	This header defines general utilities to manage dynamic memory [17].
string	This header is part of the strings library and defines a container for storing a series of characters [18].
thread	This header is part of the thread support library and defines support for threads, atomic operations, mutual exclusion, condition variables, and futures [19].
chrono	The chrono library defines three main types as well as utility functions and common typedefs: clocks, time points, and durations [13].
uavrt_connection/ command _component.hpp	The header file for the command_component.cpp file.

Table 3.14: Imported C++ standard library header files and other header files used in the command_component.cpp file.

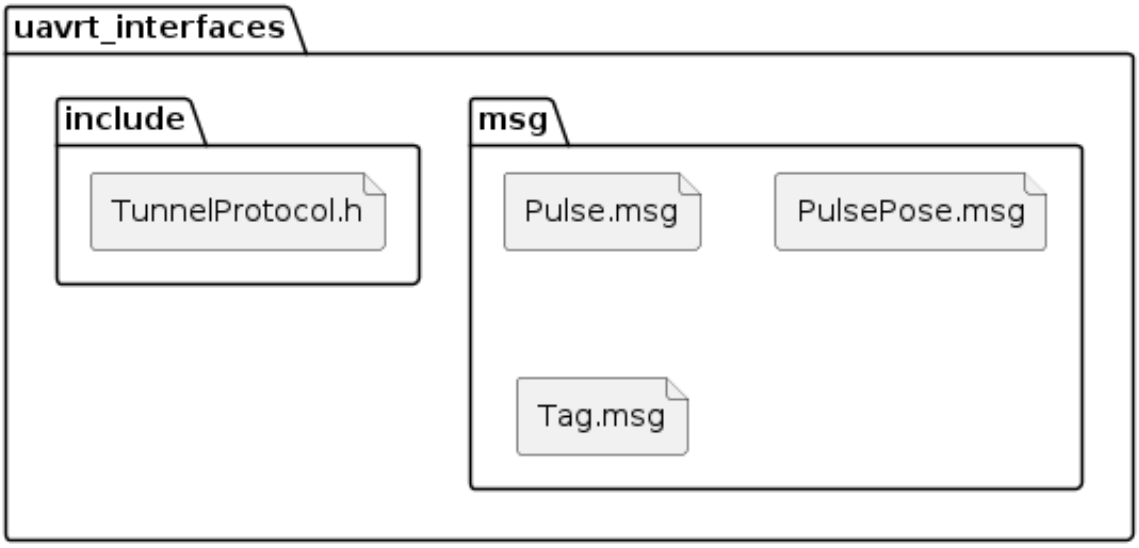

Figure 3.25: A class diagram depicting the files in the uavrt_interfaces package.

Figure 3.25 represents the directory and file hierarchy for the uavrt_interfaces package.

3.4.1 TunnelProtocol.h

The TunnelProtocol.h file resides in the uavrt_interfaces package. The primary function of this file is to provide the constants and structs necessary for transmitting and receiving MAVLink Tunnel Protocol messages. An overview of the Tunnel Protocol was provided in Subsection 2.6.1, along with the details of the data that was packed in Tunnel Protocol messages and the constants defined in TunnelProtocol.h.

3.4.2 PulsePose.msg, Pulse.msg, and Tag.msg

The PulsePose.h, Pulse.h, and Tag.h files resides in the uavrt_interfaces package. The primary function of these files is to provide the definitions for the PulsePose, Pulse, and Tag custom messages. For an overview of ROS 2 custom messages, the

custom messages used within the UAV-RT project, and the data that is packaged in the PulsePose, Pulse, and Tag custom message, refer to Subsection 2.2.3.

Chapter 4

SETUP, INSTALLATION, CONFIGURATION, AND DEMONSTRATION

4.1 Chapter overview

Chapter 4 serves as a comprehensive guide for installing, configuring, and utilizing the UAV-RT system. It outlines the essential hardware components required, provides a detailed, step-by-step guide for preparing the companion computer, and explains the installation of necessary software dependencies. Chapter 4 also covers the installation procedures for custom QGroundControl and the PX4-Gazebo headless simulator, as well as the process of connecting and configuring the PX4 flight controller hardware to the companion computer. Additionally, it instructs on setting the tag information in custom QGroundControl, walks the reader through the initiation of custom QGroundControl, starting the UAV-RT software package on the companion computer, and demonstrates the combined use of both components for data collection and analysis.

4.2 Setup

This section provides an overview of the hardware components necessary for using the UAV-RT package. The required equipment is detailed in Table 4.1 and Table 4.2. These tables outline both the equipment types needed and the specific equipment used during the development and testing phases of the UAV-RT package.

The UAV-RT package can be used with or without an unmanned aerial vehicle (UAV). Pictures are provided for both configurations: Figure 4.1 shows the UAV-RT hardware configuration without the vehicle, while Figure 4.2 shows the system

configuration with the UAV included.

For comprehensive details regarding the UAV, including its assembly and calibration procedures, refer to the 'The Vehicle' section on the Dynamic and Active Systems Lab (DASL) UAV-RT Northern Arizona University website [107]. This thesis does not cover the vehicle construction process nor the creation of the protective box that sits on top of the UAV. The equipment listed in Table 4.1 and Table 4.2 would be required, whether the configuration involves the UAV or not. It is worth noting Bryce Fennell's dedication to maintaining the UAV and designing a new protective box during this iteration of the UAV-RT project.

Type of equipment	Specific equipment
Companion computer	The companion computer used for this project was the UDOO X86 II Ultra.
Ground control station (GCS)	The GCS used for this project was the 2019 Dell Latitude Rugged 5424 Laptop. Any mid-range laptop will due. The Dell Latitude Rugged series is recommended if the GCS will be exposed to harsh weather.
VHF (very high frequency) directional antenna	The VHF antenna used for this project was the RA-23K VHF Antenna. A RW-2 Coxial Cable will be also be required. One end should be a BNC male connection, while the other end should be a BNC female connection.
Software defined radio (SDR)	The SDR used for this project was the Airspy Mini. In testing, the Airspy Mini produced less noise over the Airspy R2 and it has a longer range than the Airspy HF+.
Telemetry radio	The telemetry radio used for this project was the mRo SiK 915 MHz Telemetry Radio. The air/ground bundle is recommended, as it will come with both a version of the small and large mRo SiK Telemetry Radio. The small version should be used within the case of the vehicle, while the large version is attached to the GCS.

Table 4.1: The necessary hardware components in order to the use the UAV-RT software package.

Flight controller	The flight controller used for this project was the 3DR Pixhawk 1 Flight Controller, which has since been discontinued. A replacement for the 3DR Pixhawk 1 Flight Controller would be the mRo Pixhawk Flight Controller (Pixhawk 1).
GPS (Global Positioning System) module and embedded digital magnetometer	The GPS module and embedded digital magnetometer used for this project was the mRo GPS u-Blox Neo-M8N IST8308.
Wildlife tracking transmitter (tag)	The wildlife tracking transmitter used for this project was the Holohil BD-2 Wildlife Tracking Transmitter. Both transmitters weighing 1.0g and 1.5g were used during testing and development.
Field monitor	The field monitor used for this project was the UDOO 7 inch HDMI/USB Display/Touch monitor. An HDMI cable and a portable battery are needed to connect the field monitor to the UDOO X86 II HDMI port.
Wireless keyboard	The wireless keyboard used for this project was the Logitech Illuminated Keyboard K830. The Logitech K400 Plus Wireless Touch Keyboard can also be used.
Additional cables	Additional required cables include 2 6-Pins DF13 to JST-GH cable, a u.FL to RP-SMA adapter, a micro USB cable, 2 3dBi 915 MHz antennas, and a FTDI 5V VCC-3.3V I/O cable.

Table 4.2: The necessary hardware components in order to the use the UAV-RT software package (cont).

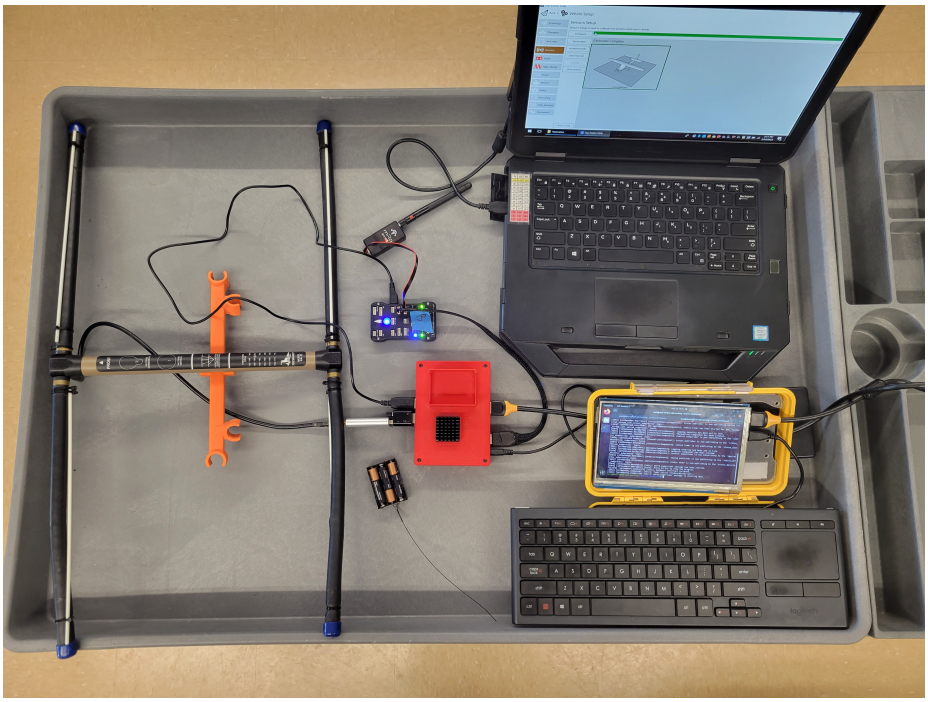

Figure 4.1: UAV-RT hardware (without the UAV).

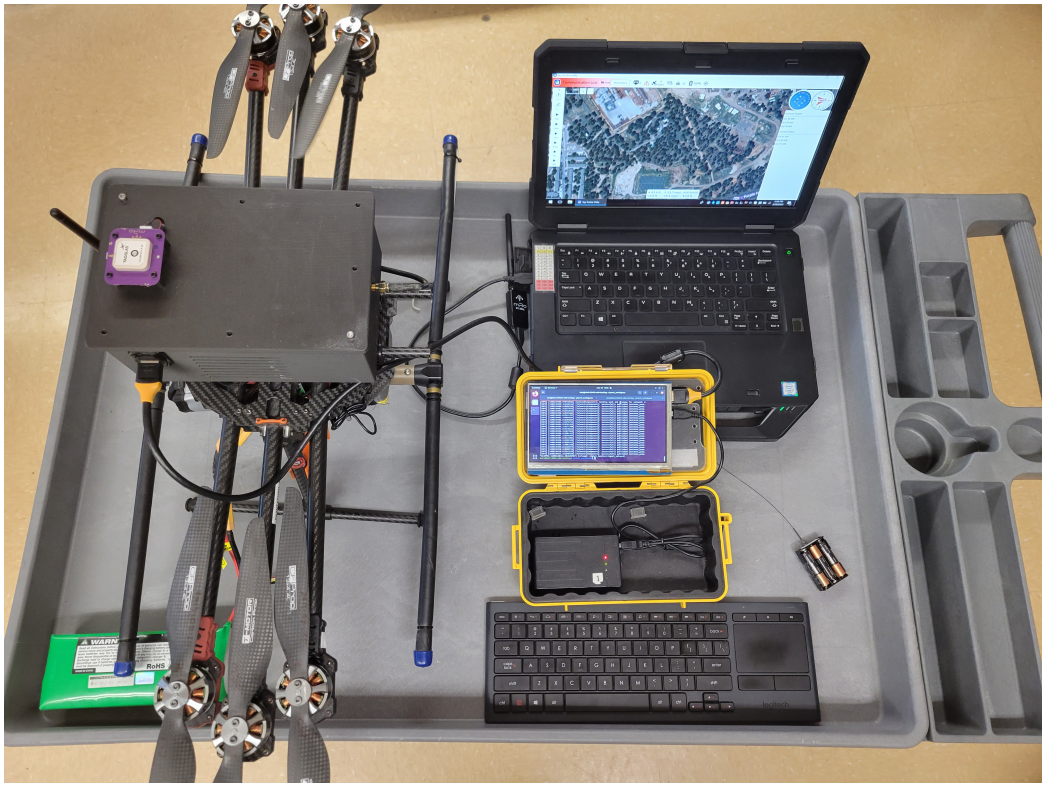

Figure 4.2: UAV-RT hardware.

4.3 Installation

This section provides the steps necessary to installing the software components of the UAV-RT package. An internet connection will be required for the installation steps, but it will not be required during the demonstration steps.

4.3.1 Preparing the companion computer

The first step is installing an operating system (OS) on the companion computer. This step will require a flash drive (12GB or above is recommended), as well as the Ubuntu 20.04.6 LTS (Focal Fossa) desktop image. Ubuntu 20.04.6 was used during testing and development of the UAV-RT package, and so it is the recommended version of Ubuntu. The official Ubuntu documentation provides details regarding the installation of Ubuntu 20.04.6 [108]. This includes creating a bootable USB drive, booting from created USB drive and following the installation setup wizard, creating a login account, and updating Ubuntu 20.04.6 once the installation is complete.

4.3.2 Installing software dependencies on the companion computer

This subsection provides a step-by-step guide on how to install the necessary software dependencies to use the UAV-RT software system. Ubuntu 20.04.6 LTS (Focal Fossa) and the standard Debian/Ubuntu "apt" package manager were used for the installation process.

Python 3

```
sudo apt update && sudo apt upgrade
sudo apt install python3.8
python --version
```

Cmake 3.16.3+

```
sudo apt update && sudo apt upgrade
sudo apt install cmake
cmake --version
```

Git

```
sudo apt update && sudo apt upgrade
sudo apt install git
git --version
```

GNU Compiler Collection (GCC) 6.3+

```
sudo apt update && sudo apt upgrade
sudo apt install build-essential
sudo apt install manpages-dev
gcc --version
```

Netcat

```
sudo apt update && sudo apt upgrade
sudo apt install netcat
dpkg -L netcat
```

Boost

```
sudo apt update && sudo apt upgrade
sudo apt install libboost-all-dev
```

scipy

```
sudo apt update && sudo apt upgrade
sudo apt install python3-pip
pip3 install scipy
```

libusb and pkg-config

```
sudo apt update && sudo apt upgrade
sudo apt install libusb-1.0-0-dev
sudo apt install pkg-config
```

airspyone_host

This is installation will provide the airspy library, as well as the airspy_rx tool. The installation instructions for the airspy library can be found in the readme of the airpyone_host repository [1].

csdr

```
sudo apt update && sudo apt upgrade
sudo apt install libfftw3-dev
wget https://github.com/ha7ilm/csdr/archive/master.zip
unzip master.zip
cd csdr-master
make
sudo make install
sudo ldconfig
```

ROS 2

The UAV-RT software system supports the Galactic Geochelone distribution of ROS 2 [70]. The instructions for installing ROS 2 Galactic Geochelone can in the ROS 2 Galactic Geochelone documentation [72].

Follow from the beginning of the instructions to the end, as it can be difficult to troubleshoot errors later on unless ROS 2 is correctly installed. Run the examples that are listed at the bottom of the installation instructions to ensure that ROS 2 is correctly installed on your machine [71].

Ros2dep

ROS 2 packages are built on frequently updated Ubuntu systems. It is always recommended that you ensure your system is up to date before installing new packages.

```
sudo apt update && sudo apt upgrade
sudo apt install python3-rosdep2
```

You will then need to update rosdep.

```
rosdep update
```

colcon

colcon is an iteration on the ROS build tools catkin_make, catkin_make_isolated, catkin_tools and ament_tools.

```
sudo apt update && sudo apt upgrade
sudo apt install python3-colcon-common-extensions
```

MAVLink and MAVSDK C++

This codebase supports MAVLink V2 and uses MAVSDK C++ as the interface to the MAVLink protocol.

```
cd ~
wget https://github.com/mavlink/MAVSDK/releases/download/v1.4.6/
    libmavsdk-dev_1.4.6_ubuntu20.04_amd64.deb
sudo dpkg -i libmavsdk-dev_1.4.6_ubuntu20.04_amd64.deb
```

uavrt_source

A functional ROS 2 workspace is required for installing the uavrt_source and other UAV-RT ROS 2 packages. The installation instructions for the uavrt_source and other UAV-RT ROS 2 package will fail unless the previous dependencies have been met and a ROS 2 workspace has been created.

Other UAV-RT ROS 2 packages will follow a similar installation procedure.

Within a terminal window, run the following commands:

```
source /opt/ros/galactic/setup.bash
mkdir -p ~/uavrt_workspace/
cd ~/uavrt_workspace/
git clone https://github.com/dynamic-and-active-systems-lab/uavrt_source/
```

"All required rosdeps installed successfully" should be returned after the following command:

```
rosdep install -i --from-path uavrt_source --rosdistro galactic -y
```

There should be a "build", "install", "log", and "uavrt_source" directory in the root of the workspace (`~/uavrt\_workspace`) after the following command:

```
source /opt/ros/galactic/setup.bash
colcon build
. install/local_setup.bash
```

If these commands did not fail, then the installation of the uavrt_source package should be complete.

uavrt_supervisor

```
source /opt/ros/galactic/setup.bash
cd ~/uavrt_workspace/uavrt_source/
git clone https://github.com/dynamic-and-active-systems-lab/uavrt_supervisor/
cd ~/uavrt_workspace/
rosdep install -i --from-path uavrt_source --rosdistro galactic -y
```

The following command will only build out the uavrt_supervisor package. This is done to isolate errors or warnings:

```
colcon build --packages-select uavrt_supervisor
source /opt/ros/galactic/setup.bash
. install/local_setup.bash
```

uavrt_connection

```
source /opt/ros/galactic/setup.bash
cd ~/uavrt_workspace/uavrt_source/
git clone https://github.com/dynamic-and-active-systems-lab/uavrt_connection/
cd ~/uavrt_workspace/
rosdep install -i --from-path uavrt_source --rosdistro galactic -y
```

The following command will only build out the uavrt_connection package. This is done to isolate errors or warnings:

```
colcon build --packages-select uavrt_connection
source /opt/ros/galactic/setup.bash
. install/local_setup.bash
```

uavrt_interfaces

```
source /opt/ros/galactic/setup.bash
cd ~/uavrt_workspace/uavrt_source/
https://github.com/dynamic-and-active-systems-lab/uavrt_interfaces
    /tree/d05ffe1d87382fe1aff44449917311d985a8ffcf
cd ~/uavrt_workspace/
rosdep install -i --from-path uavrt_source --rosdistro galactic -y
```

The following command will only build out the uavrt_interfaces package. This is done to isolate errors or warnings:

```
colcon build --packages-select uavrt_interfaces
source /opt/ros/galactic/setup.bash
. install/local_setup.bash
```

uavrt_detection and channelizer

Similar to the uavrt_interfaces package, the uavrt_detection and channelizer packages have received updates that introduced additional features. Additionally, this package would need to be generated using MATLAB Coder and the original MATLAB code.

To simplify the installation process for all the necessary packages in this iteration of the UAV-RT project, the C/C++ versions of the uavrt_detection and channelizer packages have been included in the current version of uavrt_source. Note that these compiled packages do not represent the most recent versions but align with the development stage described in this document.

In future iterations of the UAV-RT project, these packages will no longer be included within the uavrt_source package. Instead, the uavrt_detection and channelizer packages would be installed in a manner similar to the installation process for the UAV-RT ROS 2 packages.

Final installation step

Once all of the necessary UAV-RT packages have been cloned into uavrt_source, run the following commands again within a terminal window:

```
cd ~/uavrt_workspace/
source /opt/ros/galactic/setup.bash
```

"All required rosdeps installed successfully" should be returned after the following command:

```
rosdep install -i --from-path uavrt_source --rosdistro galactic -y
```

After this command:

```
colcon build
source /opt/ros/galactic/setup.bash
. install/local_setup.bash
```

You will see something similar to:

```
Starting >>> uavrt\_PACKAGE
...
Finished <<< uavrt\_PACKAGE [Time(s)]
...
Summary: 1 package finished [Time(s)]
```

If these commands did not fail, then the installation of the UAV-RT codebase should be complete on the companion computer.

4.3.3 Installing custom QGroundControl on the ground control station

The custom QGroundControl version used for this iteration of the project underwent testing on both Linux and Windows platforms. Custom QGroundControl was used on Linux for benchtop testing, while real-world flight testing was conducted with custom QGroundControl running on Windows.

Note that the custom QGroundControl version used during the development and testing of the UAV-RT project is not available for download from the QGroundControl documentation site. The UAV-RT project is currently under development, and the custom QGroundControl version is continually being updated. The specific versions of custom QGroundControl used for this part of the project are hosted in the DASL Google Drive. These versions can be downloaded by using the following links:

- The Linux version can be downloaded here: `https://drive.google.com/file/d/1W83ulHcgtthF56lxluJhkliFAa3LElA4/view?usp=sharing`
- The Windows version can be downloaded here: `https://drive.google.com/file/d/1pGoLHbPW3AV9kW3leFIKKEKDZhOe9TLS/view?usp=sharing`

Follow the steps in Subsection 4.5.1 on to how to initiate and use custom QGroundControl.

Note that in future iterations of the UAV-RT project, the plan is to download the custom QGroundControl versions from the DASL website or Github, and not the DASL Google Drive.

4.3.4 Installing PX4-Gazebo headless simulator for bench top testing

Bench top testing was completed during the testing and development of the UAV-RT package. This provided a controlled environment, in which software could be tested and configured. The PX4-Gazebo simulator was used as part of the bench top testing environment. This simulator will act as a simulation of the PX4 flight controller and can be used in tantum with the UAV-RT package and custom QGroundControl. This part of the installation step is optional and is not required to fully utilize the UAV-RT package.

In order to use the PX4-Gazebo simulator, Docker will need to be installed on the target computer. Refer to the Docker installation on Ubuntu documentation on how to do so [21]. Afterwards, the PX4-Gazebo Docker image will need to be downloaded. Referring to the PX4-Gazebo simulator documentation [91], the required command to download and run the PX4-Gazebo Docker image is as follows: `sudo docker run --rm -it jonasvautherin/px4-gazebo-headless:1.13.2`.

A successful run of the PX4-Gazebo simulator should result in Figure 4.3 and Figure 4.4. The version of PX4-Gazebo simulator might be slightly different from the version found in Figure 4.3. Referring to Figure 4.21, you would use the following command in order to use the UAV-RT package with the PX4-Gazebo simulator: `ros2 run uavrt_connection main 1`. The remaining steps as listed in Section 4.5 still apply.

```
dasl@dasl-UBUNTU-WORKSTATION:~$ sudo docker run --rm -it jonasvautherin/px4-gazebo-headless:1.12.1
[  0%] Built target ver_gen
[  0%] Generating ../../logs
[  1%] Built target parameters_xml
[  2%] Built target uorb_headers
[  2%] Built target logs_symlink
[  3%] Built target drivers_board
[  3%] Built target git_ecl
[  3%] Built target mixer_gen
[  3%] Built target mixer_gen_6dof
[  4%] Built target output_limit
[  5%] Built target rc
[  6%] Built target version
[  6%] Built target git_gps_devices
[  6%] Built target component_information_header
[  6%] Built target git_mavlink_v2
[  6%] Built target git_gazebo
[  6%] Built target component_general_json
[  6%] Built target tinybson
[  6%] Built target perf
[  7%] Built target work_queue
[  7%] Built target ecl_airdata
[  8%] Built target ecl_geo_lookup
[  8%] Built target ecl_geo
[  8%] Built target MixerBase
[  8%] Performing build step for 'sitl_gazebo'
[  9%] Built target romfs_gen_files_target
```

Figure 4.3: Starting the PX4-Gazebo simulator within a terminal window.

```
INFO  [ecl/EKF] GPS checks passed
INFO  [ecl/EKF] 4228000: EKF aligned, (baro hgt, IMU buf: 12, OBS buf: 9)
INFO  [ecl/EKF] 4232000: EKF aligned, (baro hgt, IMU buf: 12, OBS buf: 9)
INFO  [ecl/EKF] 4440000: EKF aligned, (baro hgt, IMU buf: 12, OBS buf: 9)
INFO  [ecl/EKF] 4444000: EKF aligned, (baro hgt, IMU buf: 12, OBS buf: 9)
INFO  [ecl/EKF] 4444000: EKF aligned, (baro hgt, IMU buf: 12, OBS buf: 9)
INFO  [ecl/EKF] 4512000: EKF aligned, (baro hgt, IMU buf: 12, OBS buf: 9)
INFO  [ecl/EKF] reset position to GPS
INFO  [ecl/EKF] reset velocity to GPS
INFO  [ecl/EKF] starting GPS fusion
INFO  [ecl/EKF] reset position to GPS
INFO  [ecl/EKF] INFO  [ecl/EKF] reset velocity to GPS
reset position to GPS
INFO  [ecl/EKF] reset velocity to GPS
INFO  INFO  [ecl/EKF] INFO  [ecl/EKF] reset position to GPS
INFO  [ecl/EKF] reset velocity to GPS
INFO  [ecl/EKF] starting GPS fusion
starting GPS fusion[ecl/EKF] starting GPS fusion

INFO  [ecl/EKF] reset position to GPS
INFO  [ecl/EKF] reset velocity to GPS
INFO  [ecl/EKF] starting GPS fusion
INFO  [ecl/EKF] reset position to GPS
INFO  [ecl/EKF] reset velocity to GPS
INFO  [ecl/EKF] starting GPS fusion
INFO  [tone_alarm] home set
INFO  [tone_alarm] notify negative
```

Figure 4.4: What the PX4-Gazebo simulator looks like while it is running a terminal window.

4.4 Configuration

This section provides the steps necessary for configuring the hardware and software components of the UAV-RT system. This includes the process of connecting

the PX4 flight controller to the companion computer, configuring the PX4 flight controller using custom QGroundControl, and setting the tag information within custom QGroundControl.

4.4.1 Connecting and configuring the PX4 flight controller

In order to properly use the PX4 flight controller with the companion computer chosen for the UAV-RT systen (the UDOO X86 II Ultra), it is necessary that a connection is made between one of the telemetry ports on the PX4 flight controller and a serial or USB port on the companion computer. Since the companion computer uses Intel Braswell x86 Processor pinout headers, which only is 1.8V compliant while the PX4 flight controller is 3.3V compliant, the serial option is not available.

A 6-Pin DF13 cable and a FTDI 5V VCC-3.3V I/O cable will need to be cut, soldered, and connected together for the connection between the PX4 flight controller and the USB port on the companion computer. One end of the DF13 cable will plug into the telemetry port on the PX4 flight controller, but other end of the cable will not be needed. Make a cut down the middle of the DF13 cable. The serial connection on the FTDI cable will also not be required. A cut can be made at any part of the cable, so long as the serial connection is cut off and the shielding on the cable can be trimmed back. The DF13 and FTDI cables can now be connected. Refer to the Table 4.2 as a reference to the required wiring configuration. Note: The VCC wires on DF13 and FTDI cables are not required. The ends of the VCC wires can be cut and covered with heat-shrink wrap or electrical tape.

FTDI Cable	6-Pin DF13 Cable
Ground - Black - 1	Ground - Black - 6
CTS - Brown - 2	CTS - Black - 4
VCC - Red - 3	VCC - Red - 1
TXD - Orange - 4	RXD - Black - 3
RXD - Yellow - 5	TXD - Black - 2
RTS - Green - 6	RTS - Black - 5

Table 4.3: Wiring configuration for connecting a 6-Pin DF13 cable and a FTDI 5V VCC-3.3V I/O cable.

Figure 4.5 is a close-up picture of what the connection between the DF13 and FTDI cables look like, while Figure 4.6 is a picture that includes additional components within the frame. Note: Both pictures were taken during the development and testing phase of the UAV-RT package. The connection was originally made using a breadboard as an intermediary between the 6-Pin DF13 and FTDI 5V VCC-3.3V I/O cables. The cables have since been soldered together and cover with heat-shrink wire wrap. Dr. Michael Shafer and Byrce Fennell helped immensely with cutting, soldering, and connecting together the two cables.

Once the DF13 and FTDI cables have been connected together, the PX4 can be connected to the companion computer. This part of the process assumes that you can run custom QGroundControl on the companion computer. Subsection 4.5.1 reviews the steps for running custom QGroundControl on the ground control station. The steps would be the same on the companion computer.

Connect the DF13 end of the cable into a telemetry port on the PX4 flight controller and the FTDI end of the cable into a USB port on the companion computer. Either telemetry port will work. Once the PX4 flight controller is connected to the

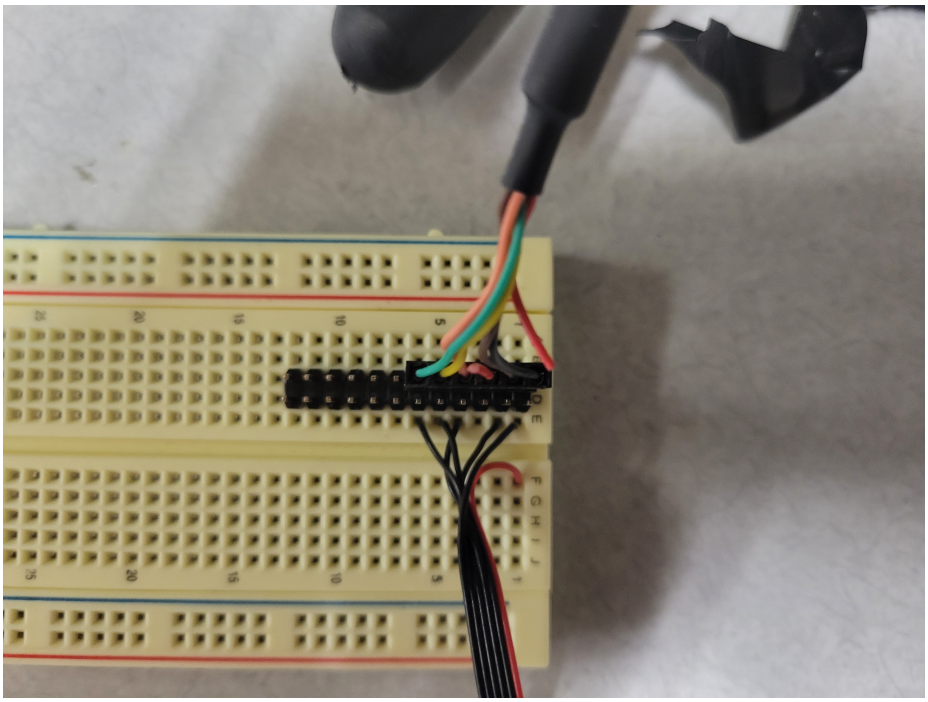

Figure 4.5: Close-up picture of what the connection between the 6-Pin DF13 and FTDI 5V VCC-3.3V I/O cables look like.

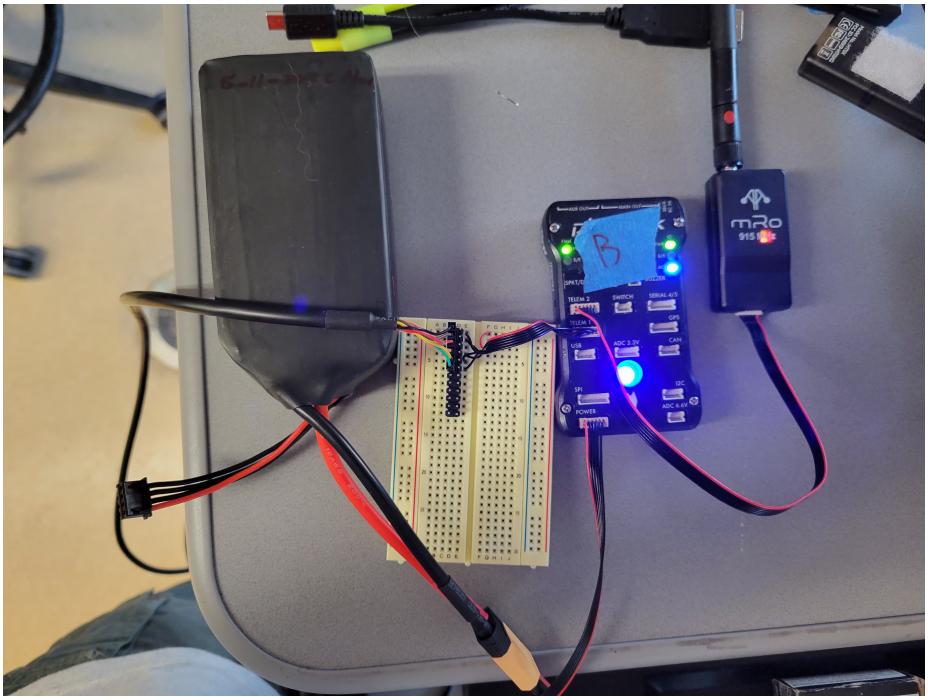

Figure 4.6: Connection between the 6-Pin DF13 and FTDI 5V VCC-3.3V I/O cables, along with additional components.

companion computer and supplied with power, open up custom QGroundControl on the companion computer. Click the "Q" logo in top left corner of the application window, and click "Vehicle Setup". Prior to proceeding, ensure that PX4 flight controller has been updated to the correct version. The version number is listed in the "Summary" section on the left-hand side. The PX4 flight controller can be updated by following the steps in the "Firmware" section on the left-hand side, or by following the PX4 firmware documentation [90].

After updating the PX4 flight controller if necessary, click "Parameters" on the left-hand side. Either type "Serial" in the search bar or scroll down until you find the "Serial" section. On the right-hand side, there should be three entries, with one listed as "SER_TEL2_BAUD" and set to "Auto". Click on "Auto" and select "57600 8N1" from the drop down. Refer to Figure 4.7 for an example of what the completed process looks like. From there, either type "MAVLink" in the search bar or scroll up until you find the "MAVLink" section. Several settings will need to be edited. Refer to Figure 4.8 for the required settings and options to chose within the drop down menus. After that is completed, close down the custom QGroundControl application and unplug the PX4 flight controller from the companion computer.

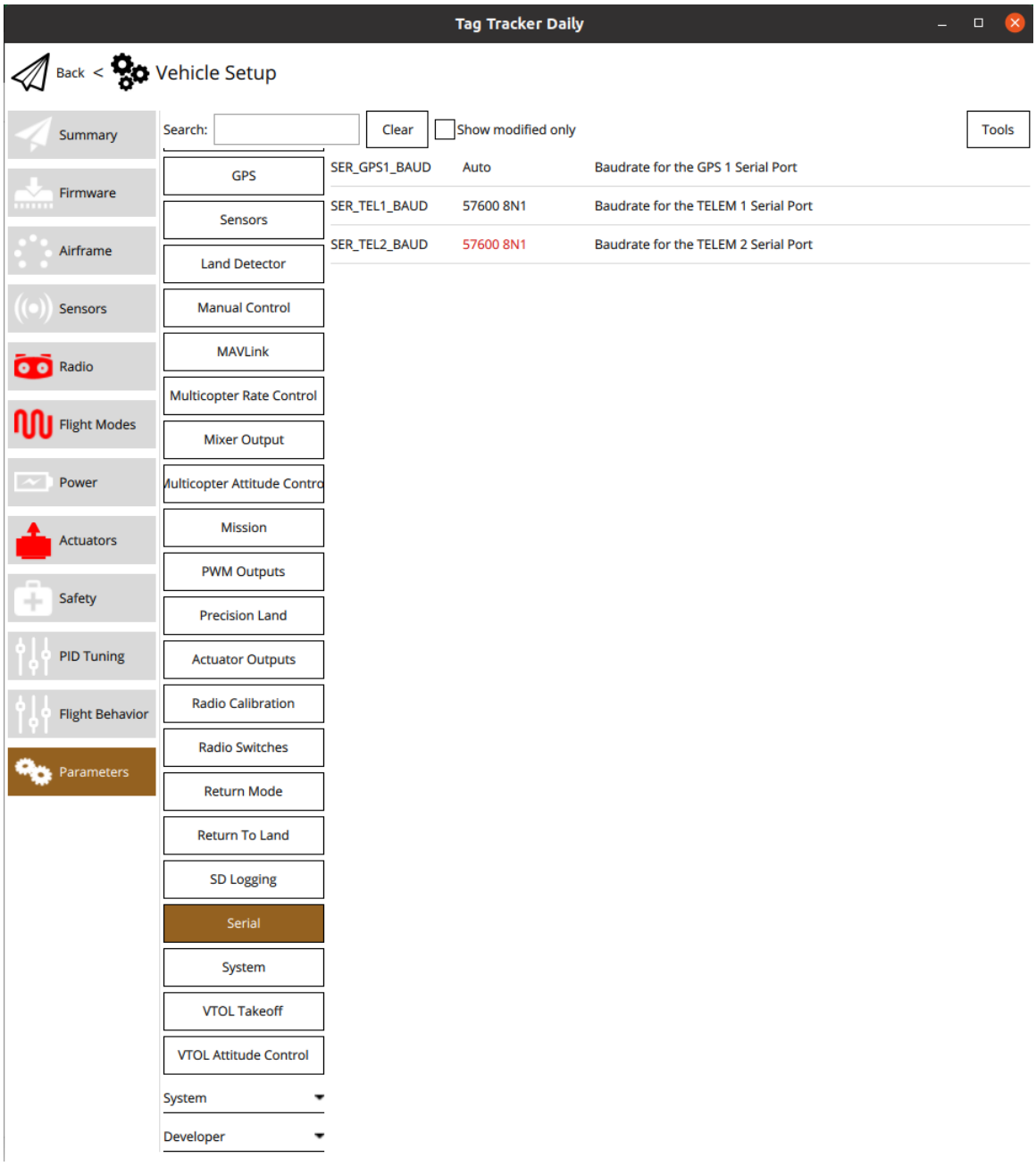

Figure 4.7: Editing the "Serial" parameters on the PX4 flight controller.

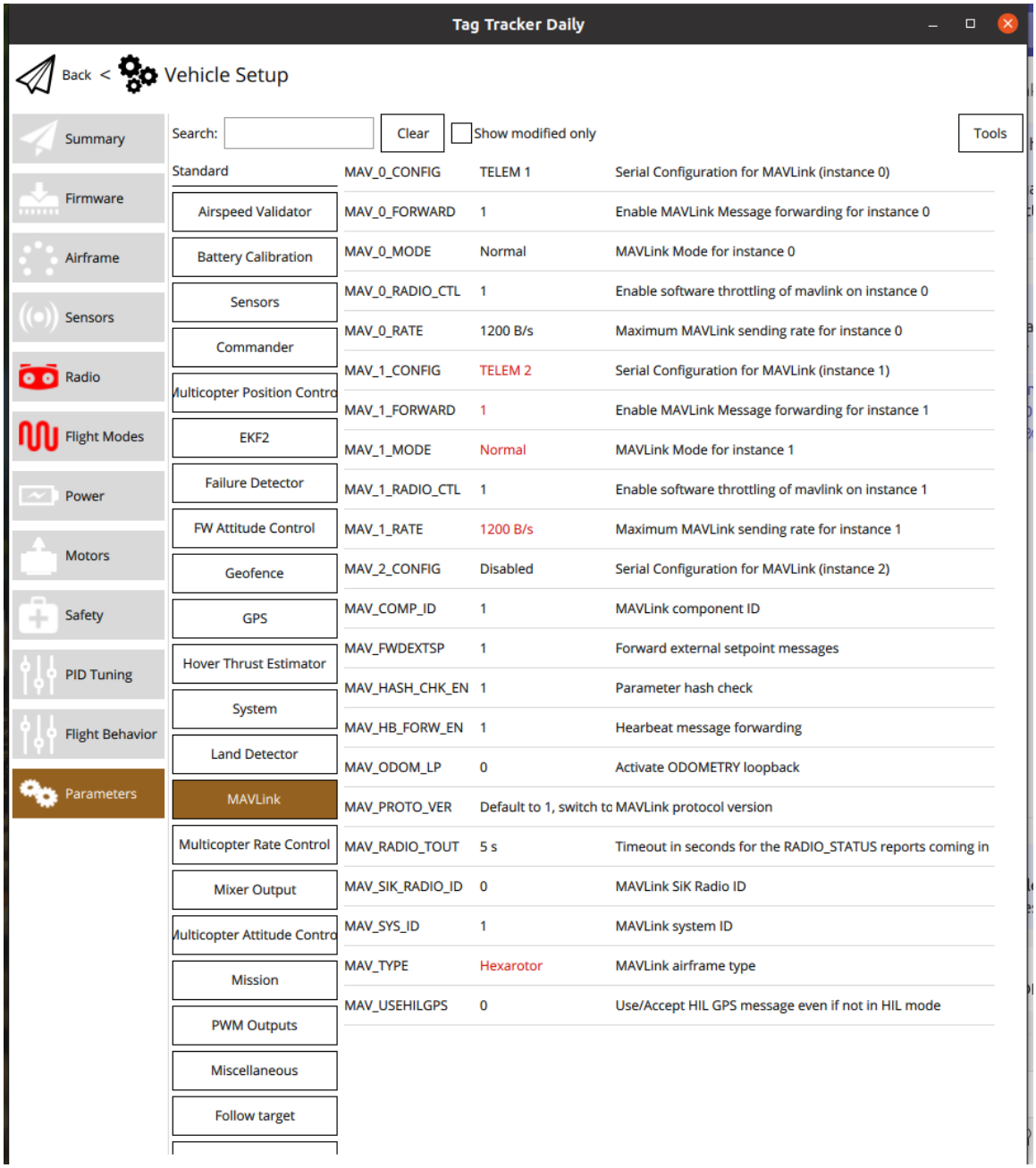

Figure 4.8: Editing the "MAVLink" parameters on the PX4 flight controller.

4.4.2 Set the tag information within custom QGroundControl

Custom QGroundControl is used to transmit tag information to the uavrt_connection package, which sends the tag information to the uavrt_supervisor package. In order

to do so, the tag information must be set within custom QGroundControl on the ground control station. This process assumes that custom QGroundControl has been installed on the ground control station.

Navigate to the "Documents" section within the file explorer on the ground control station. In that directory, there should be a folder labelled "Tag Tracker Daily". Navigate to the "Parameters" directory and open up the "TagInfo.txt" document. The values for the tag information should be entered in the text file in following order: `id, freq_hz, ip_msecs, pulse_width_msecs, ip_uncertainty_msecs, ip_jitter_msecs, k, false_alarm`. Refer to Subsection 2.2.3 for more information on what each of these parameters means. Figure 4.9 is an example of what a correct entry to the "TagInfo.txt" looks like. Once the tag information has been entered in the appropriate format, save and close the "TagInfo.txt" file. Custom QGroundControl should now be ready to send tag information to the uavrt_connection package.

```
1  # id, freq_hz, ip_msecs, pulse_width_msecs, ip_uncertainty_msecs, ip_jitter_msecs, k, false_alarm
2  2, 148710000, 1254, 20, 60, 10, 3, 0.005
```

Figure 4.9: Setting tag information within the "TagInfo.txt" document.

4.5 Demonstration

In this section, a step-by-step process will be provided for initiating and terminating custom QGroundControl on the ground station, initiating and terminating the UAV-RT software package on the companion computer, as well as retrieving the flight data from the companion computer. The demonstration does not require the vehicle.

The order of initiation for the different components is flexible, but it is crucial to ensure that all required hardware is connected before proceeding. For the sake of simplicity in this demonstration, initiate custom QGroundControl first and allow it to remain running while starting the uavrt_connection and uavrt_supervisor packages. Once the UAV-RT software packages are running, the "Tag", "Start", and "Stop"

commands can be used.

With the current iteration of the UAV-RT software package, the user will be required to initiate and terminate the UAV-RT software packages on the companion computer. This will require a keyboard, mouse, and monitor to be connected to the computer. During flight, this hardware will not need to be connected to the companion computer.

The sensors for the flight controller as well as the GPS module will need to be calibrated prior to a flight. The steps for calibrating these pieces of equipment can be reviewed within the official QGroundControl documentation [98].

4.5.1 Initiating custom QGroundControl on the ground control station

1. Ensure execute permissions are allowed for the custom QGroundControl executable on a Linux machine. This can be accessed by right-clicking the executable and selecting "Properties" from the drop-down menu. (Figure 4.10)

2. Open a terminal window within the same directory as the QGroundControl executable. Type the following into the terminal window: `./TagTracker.AppImage`. (Figure 4.11)

3. The custom QGroundControl executable should now be running. (Figure 4.12)

4. If custom QGroundControl is being started on a Windows machine, first, it needs to be installed. Then, the application can be started from the start menu or desktop by double-clicking the custom QGroundControl shortcut. (Figure 4.13)

5. You will see the same layout for custom QGroundControl, regardless of whether you are running it on Linux or Windows. Leave custom QGroundControl run-

ning in the background. (Figure 4.14)

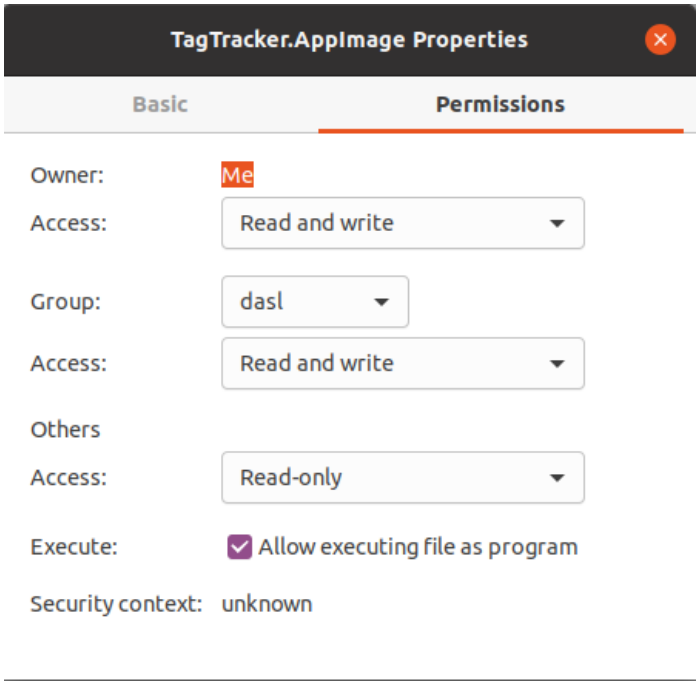

Figure 4.10: Setting execute permissions for the custom QGroundControl executable on a Linux machine.

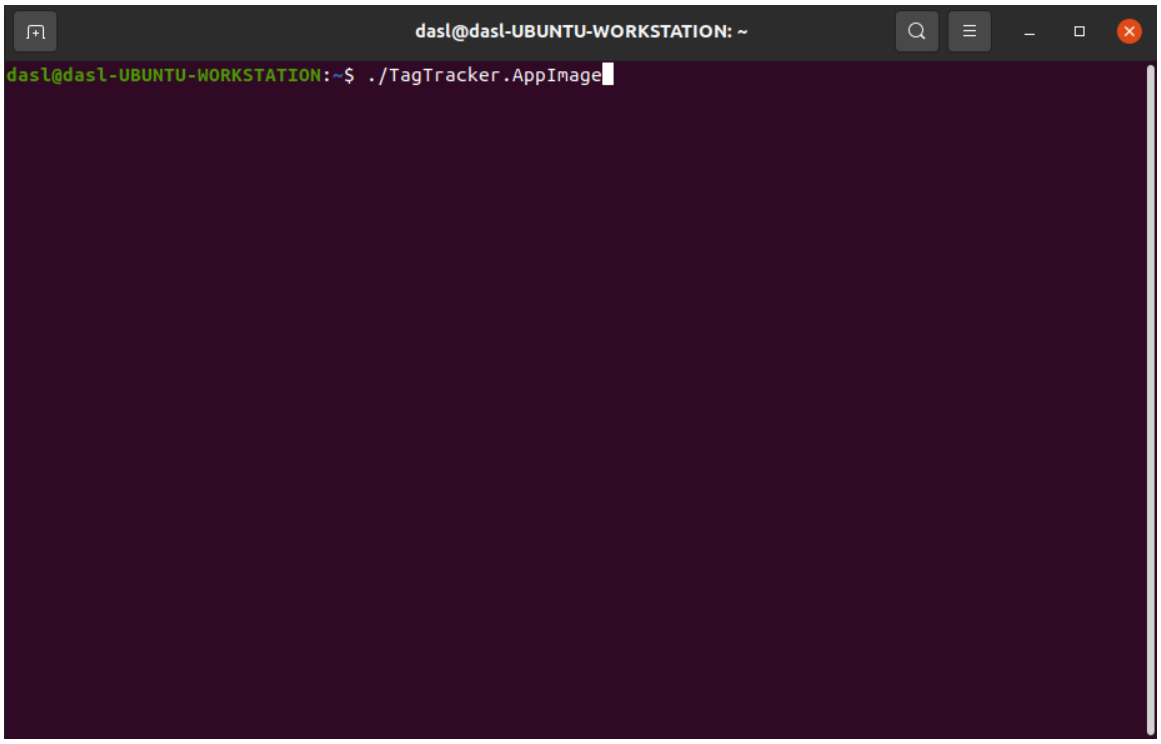

Figure 4.11: Opening a terminal window and launching QGroundControl on Linux.

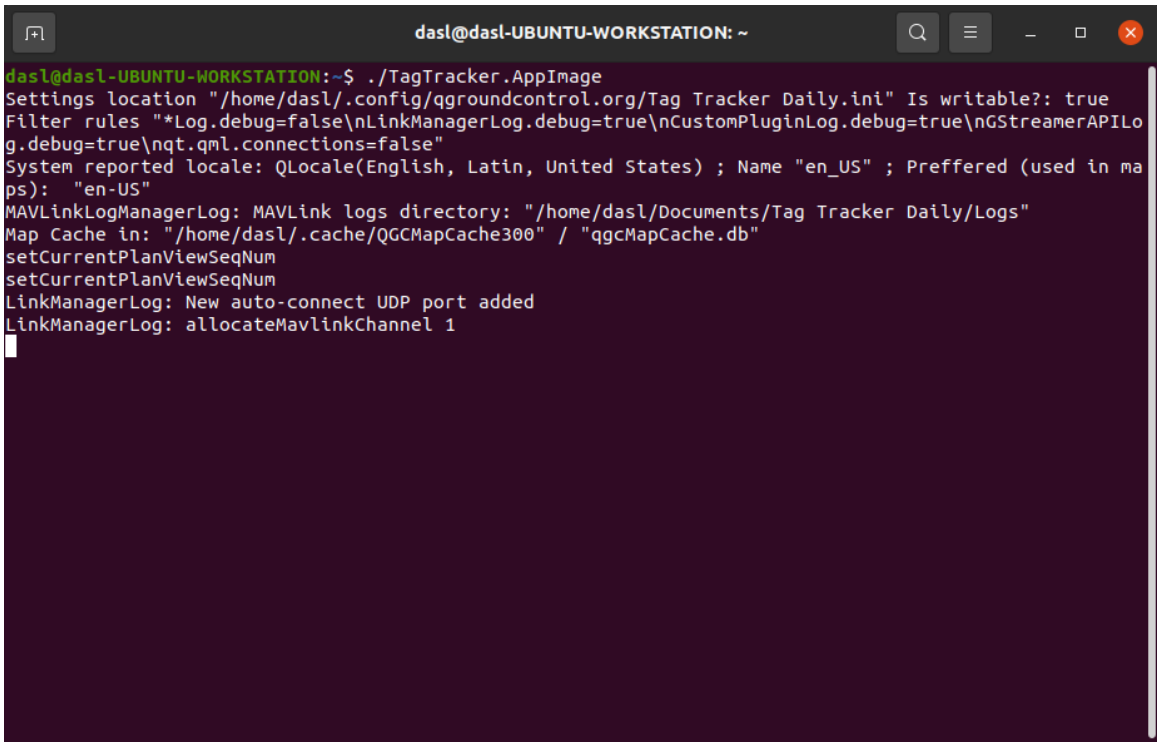

Figure 4.12: Successful running of the custom QGroundControl executable.

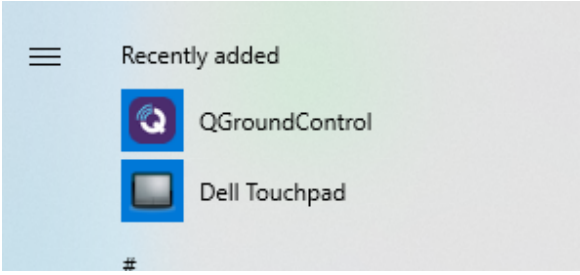

Figure 4.13: Starting custom QGroundControl on a Windows machine by installation and double-clicking the shortcut.

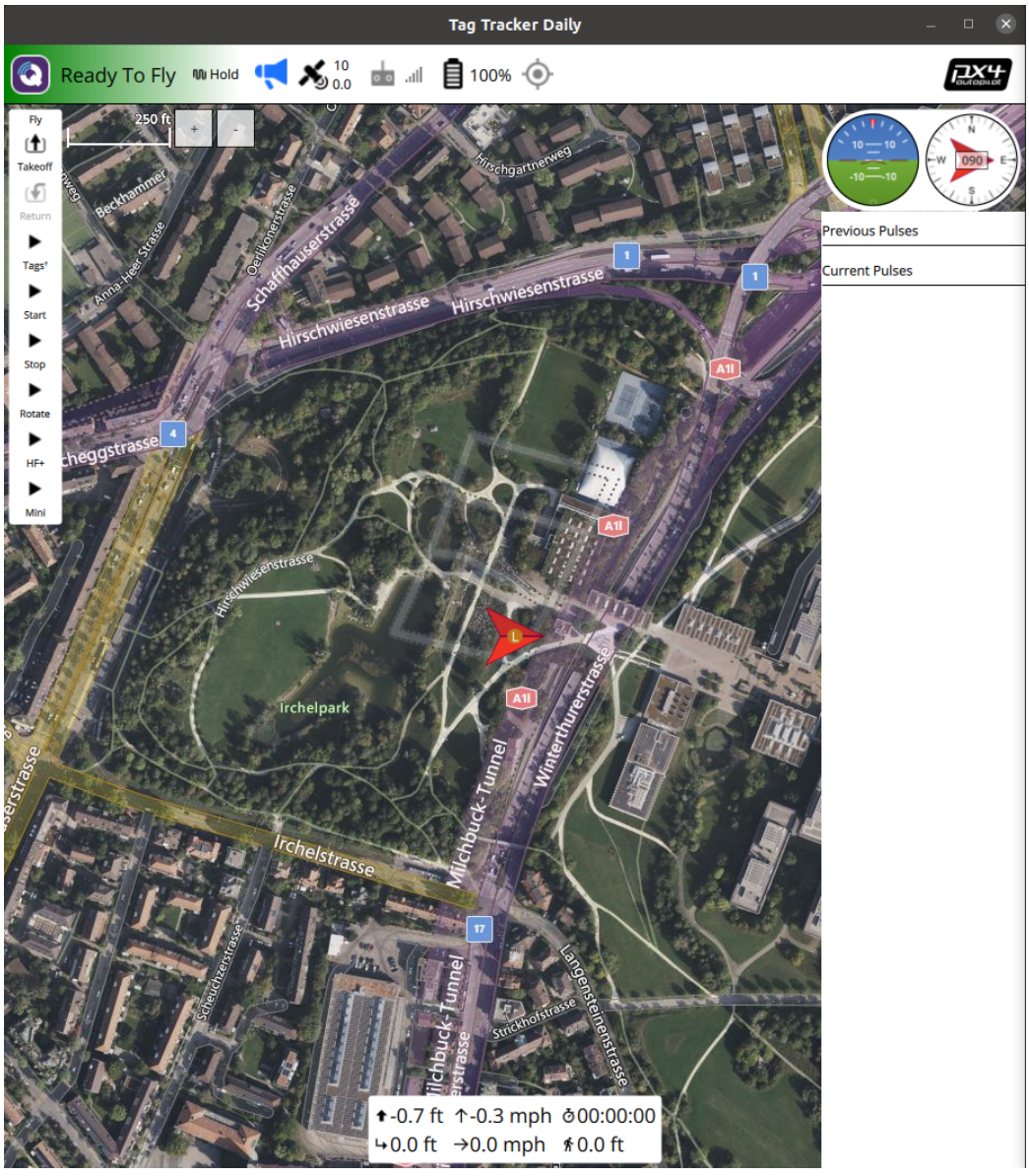

Figure 4.14: Leave custom QGroundControl running in the background.

4.5.2 Initiating the UAV-RT software package on the companion computer

1. On the home screen of the companion computer, open up the home directory. (Figure 4.15)

2. Open up the uavrt_workspace directory. (Figure 4.16)

3. Within the uavrt_workspace directory, right click and select "Open in Terminal" from the drop down menu. (Figure 4.17)

4. A new terminal window should open within the uavrt_workspace directory. Click the "Open a New Tab" button at the top left corner of the terminal window. (Figure 4.18)

5. A new tab should open with the terminal window. The new tab should also be in the uavrt_workspace directory. In the left tab, you will need to source the ROS 2 installation. Type the following into the terminal window: `source /opt/ros/galactic/setup.bash`. (Figure 4.19)

6. While still in the left tab, you will need to source the packages within the uavrt_workspace directory. Type the following into the terminal window: `. install/setup.bash`. (Figure 4.20)

7. Initiate the uavrt_connection package by typing the following: `ros2 run uavrt_connection main 0`. Note: The numerical value at the end of the command should be "1" if the package is being used with the Gazebo SITL instead. (Figure 4.21)

8. The uavrt_connection package should now be running. (Figure 4.22)

9. In the left tab, you will also need to source the ROS 2 installation. Type the following into the terminal window: `source /opt/ros/galactic/ setup.bash`. (Figure 4.23)

10. Source the packages within the uavrt_workspace directory. Type the following into the terminal window: `. install/setup.bash`. (Figure 4.24)

11. Initiate the uavrt_supervisor package by typing the following: `ros2 run uavrt_supervisor`. The uavrt_supervisor package should now be running. (Figure 4.25)

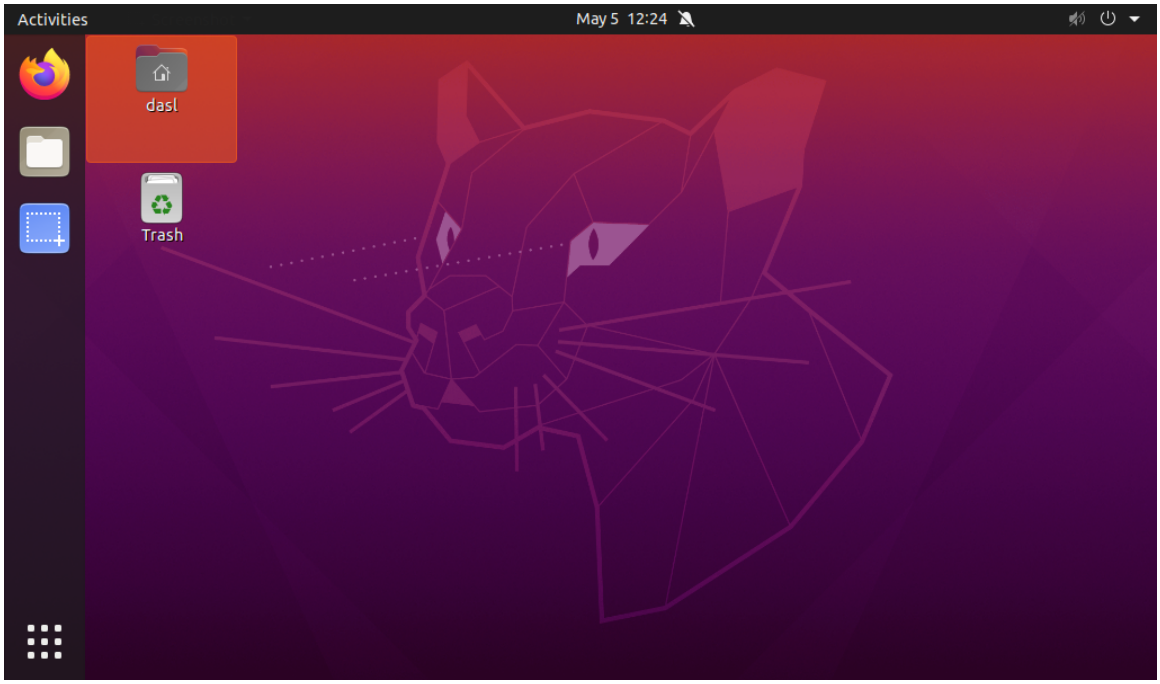

Figure 4.15: Open the home directory on the companion computer.

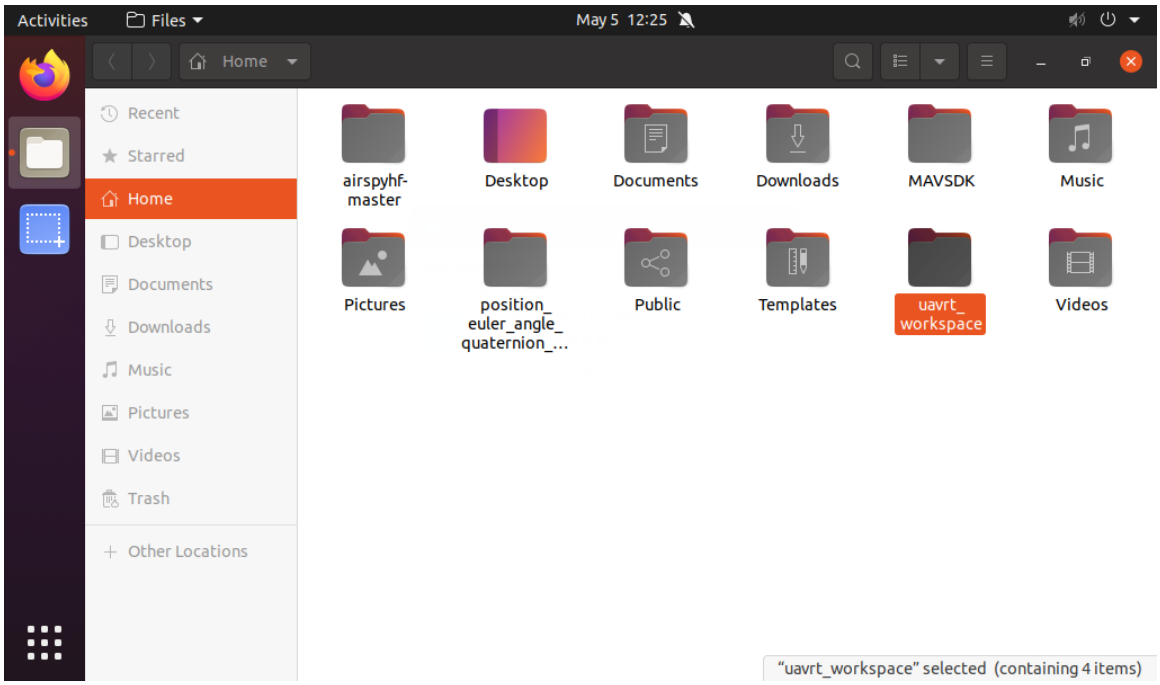

Figure 4.16: Navigate to the uavrt_workspace directory.

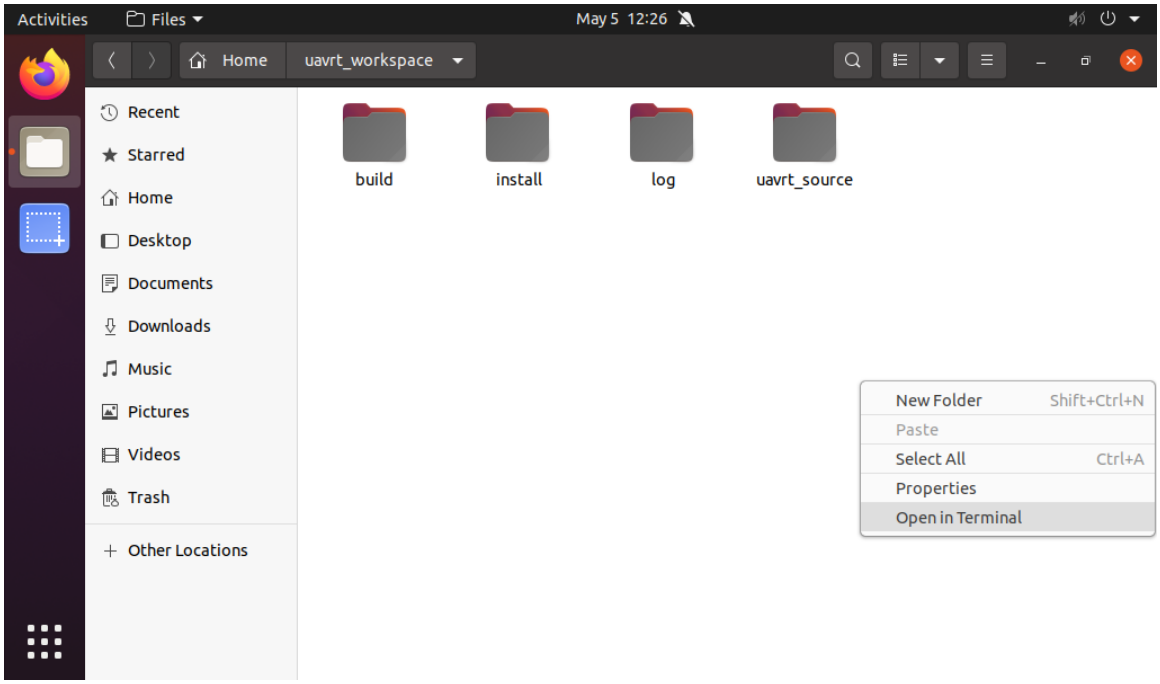

Figure 4.17: Open a terminal window in the uavrt_workspace directory.

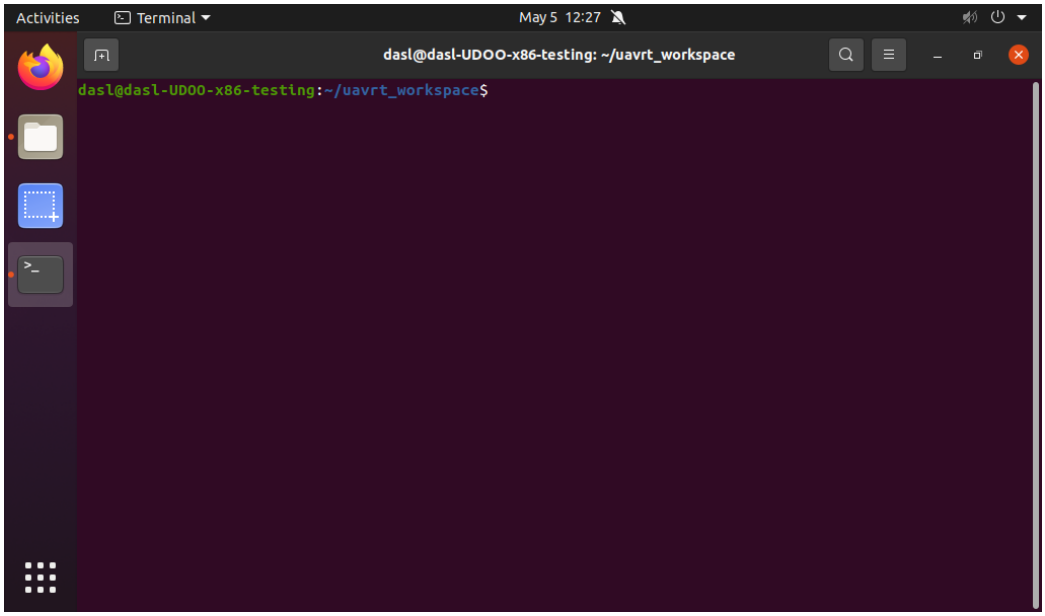

Figure 4.18: Open a new terminal tab in the uavrt_workspace directory.

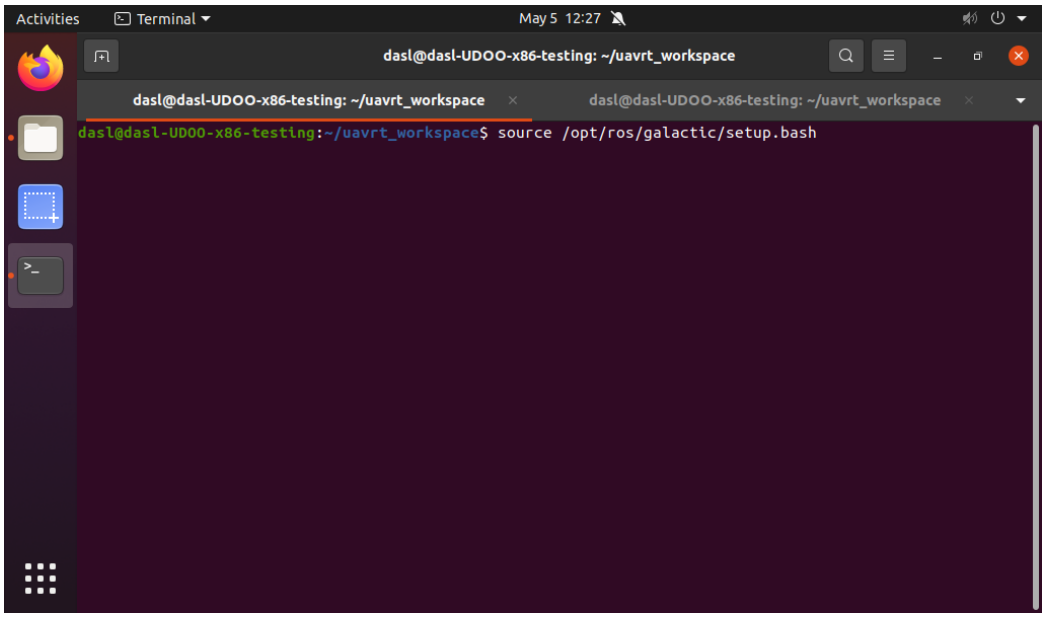

Figure 4.19: Source ROS 2 installation in the left terminal tab.

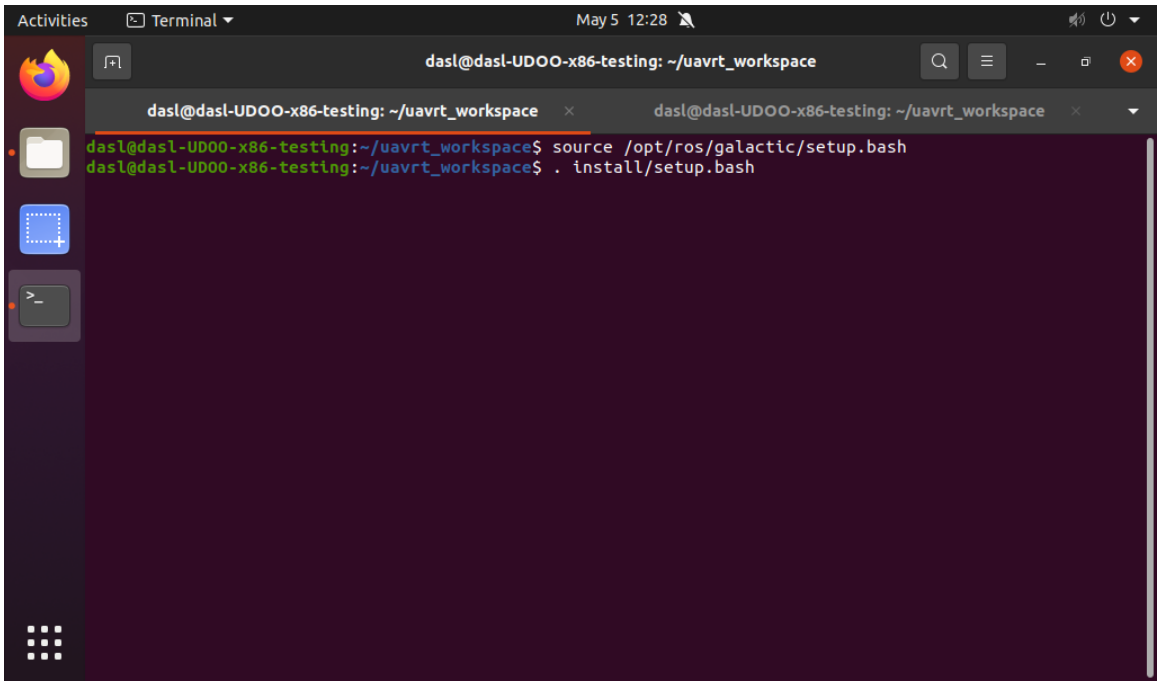

Figure 4.20: Source packages in the uavrt_workspace directory in the left terminal tab.

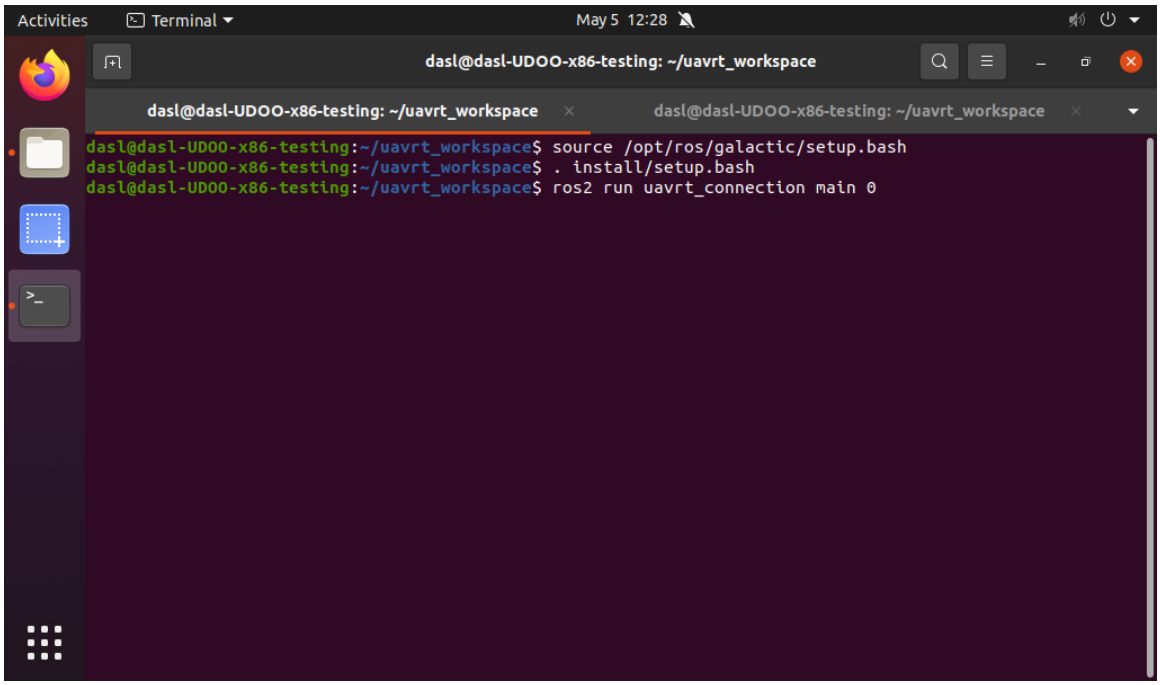

Figure 4.21: Initiate the uavrt_connection package.

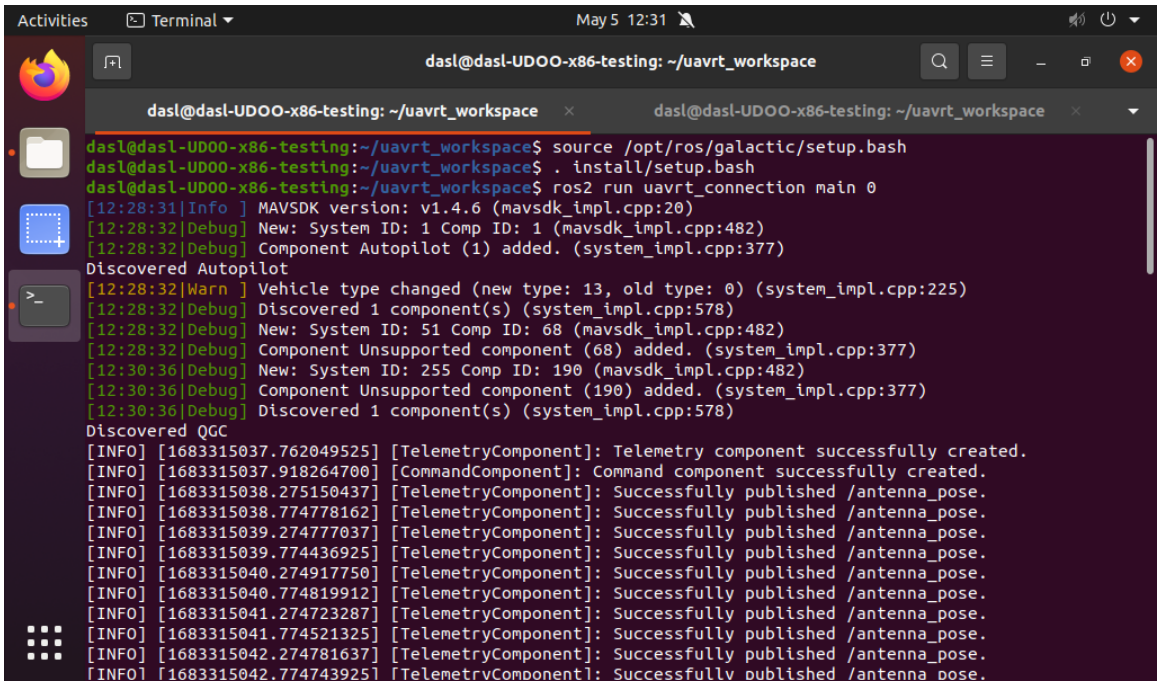

Figure 4.22: Confirm the uavrt_connection package is running.

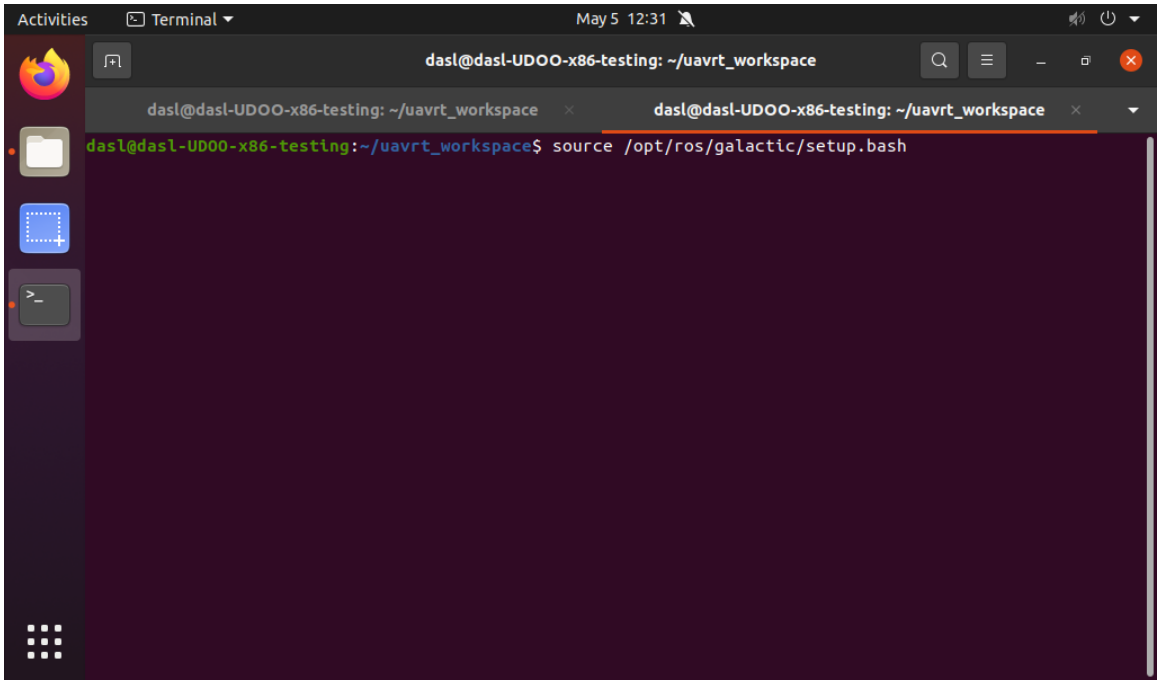

Figure 4.23: Source ROS 2 installation in the left terminal tab.

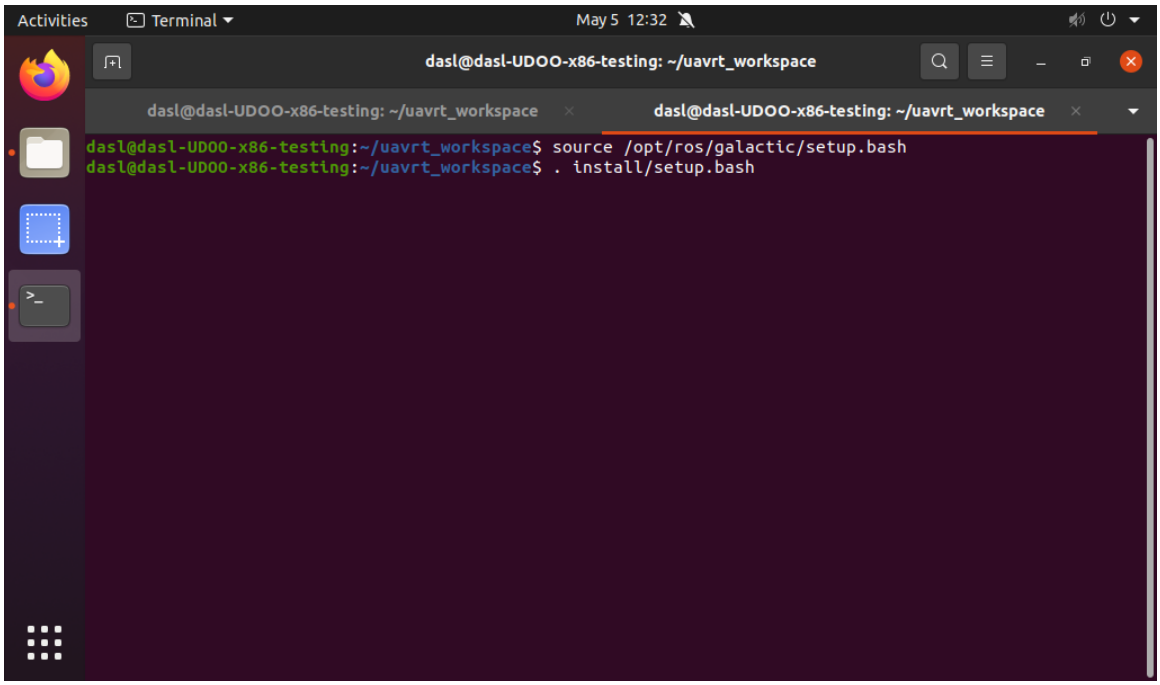

Figure 4.24: Source packages in the uavrt_workspace directory in the left terminal tab.

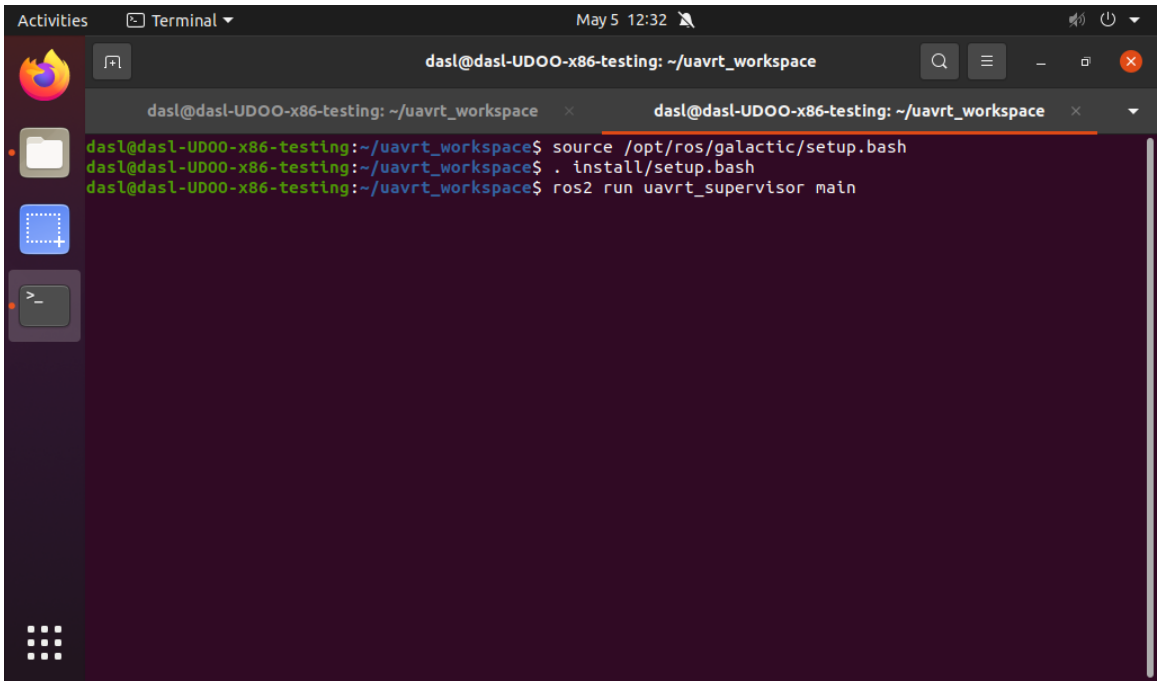

Figure 4.25: Initiate the uavrt_supervisor package.

4.5.3 Using custom QGroundControl and the UAV-RT software package

1. Switch back to the custom QGroundControl executable. In the top left, click the "Tags" button. This will send the tag information stored on the GCS to the UAV-RT software system running on the companion computer. Upon doing so, the following output should be displayed by uavrt_supervisor running in the right tab of the terminal window. Afterwards, click the "Start" button within the custom QGroundControl executable. (Figure 4.26)

2. If the error message "No such process" is displayed in the uavrt_supervisor terminal window, it means that the uavrt_supervisor package was unable to startup the airspy_rx software. Click the "Stop" button within the custom QGroundControl executable, shut down the uavrt_supervisor process by pressing Ctrl+C on the keyboard attached to the companion computer, then unplug the Airspy SDR from the companion computer and plug it back in. Start the uavrt_supervisor process by pressing the up arrow, and pressing enter. Send the tag information from custom QGroundControl using the "Tag" button and press the "Start" button. (Figure 4.27)

3. If there were no error messages displayed in the uavrt_supervisor terminal window, then the uavrt_supervisor process is successfully processing data. (Figure 4.28)

4. The uavrt_detection process will have been started by the uavrt_supervisor process. The information from the uavrt_detection process will be displayed with same terminal window as the uavrt_supervisor process. In the output for the uavrt_detection process, there will be lines that say "Current Mode: " and followed by a S(canning), C(onfirmed), or T(racking). If the tag information is

correct, the uavrt_detection process should be displaying "Current Mode: T" messages. (Figure 4.29)

5. Example of what "Confirmed mode" looks like. (Figure 4.30)

6. The uavrt_connection package will also display log messages that can be used to confirm pulse data is being transmitted. (Figure 4.31)

7. Tracked messages will show on the right-hand side of custom QGroundControl. Remove the keyboard, mouse, and field monitor from the companion computer. The UAV-RT software package will stay running on the companion computer so long as it has power. Tracked messages will appear on the right-hand side of the custom QGroundControl executable. (Figure 4.32)

8. Once the data collection is complete, connect the keyboard, mouse, and monitor to the companion computer. Click the "Stop" button in the custom QGround-Control executable. This will shut down all of the child processes started by the uavrt_supervisor package. (Figure 4.33)

9. In the terminal window for the uavrt_supervisor process, press Ctrl+C on your keyboard. A message should be displayed stating that the uavrt_supervisor is shutting down. Do the same for the uavrt_connection process in the other terminal window. Press the X at the top right of the custom QGroundControl executable's window to shut the executable down. (Figure 4.34)

10. The flight data from the recent flight can be found in the "log" directory within the uavrt_source directory. (Figure 4.35)

11. Flight directory that corresponds to the recent flight. (Figure 4.36)

12. A log directory is created for each of subprocess types and contains logs files specific to the the subprocess type. The "detector_log" directory will also contain the configuration files that were necessary to start up individual uavrt_detection processes. (Figure 4.35)

13. A "tag_id_" directory is made for each of the tags that were tracked. (Figure 4.36)

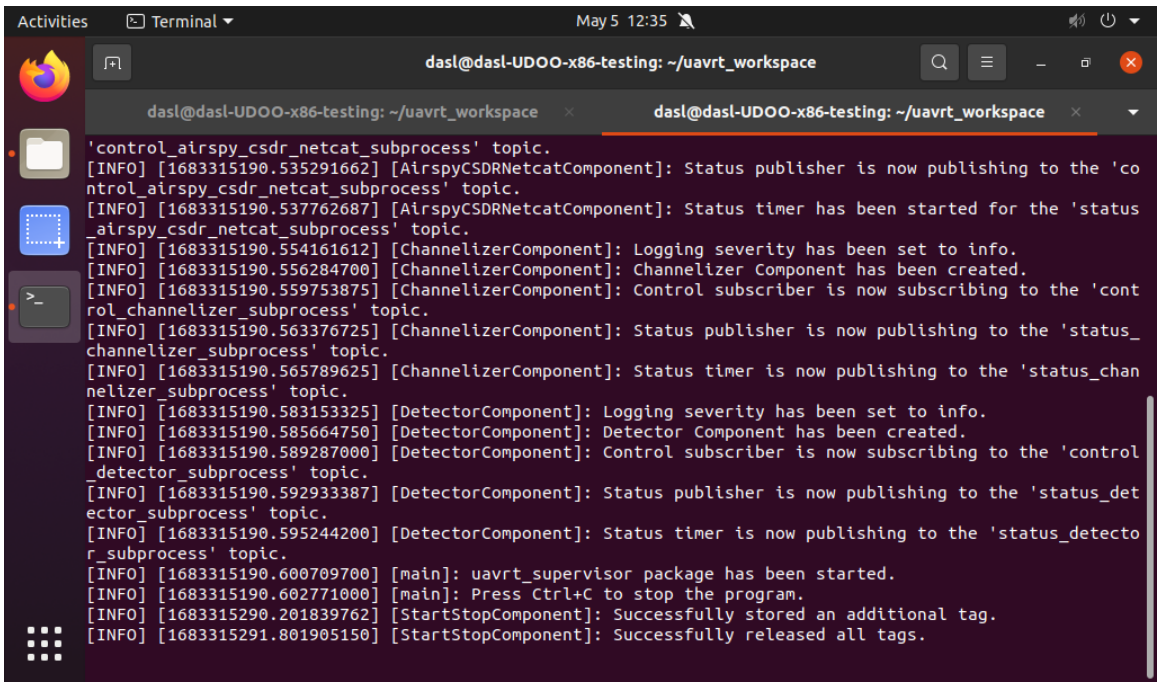

Figure 4.26: Tag information sent to the running uavrt_supervisor package.

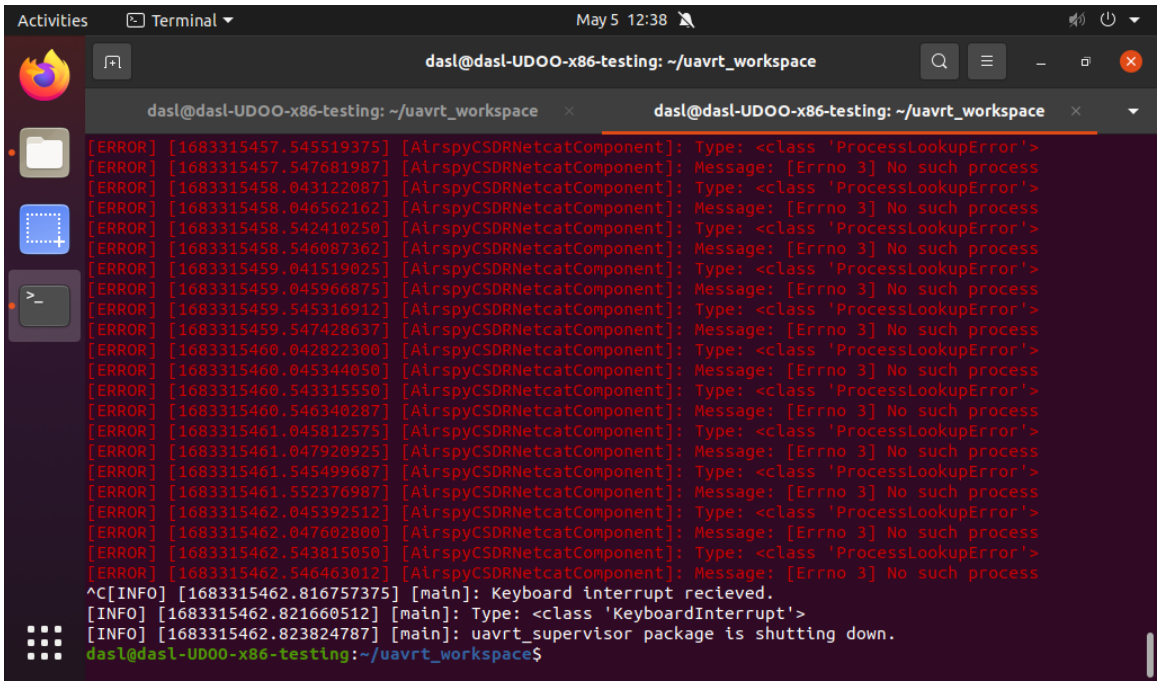

Figure 4.27: "No such process" error is displayed in the uavrt_supervisor terminal window.

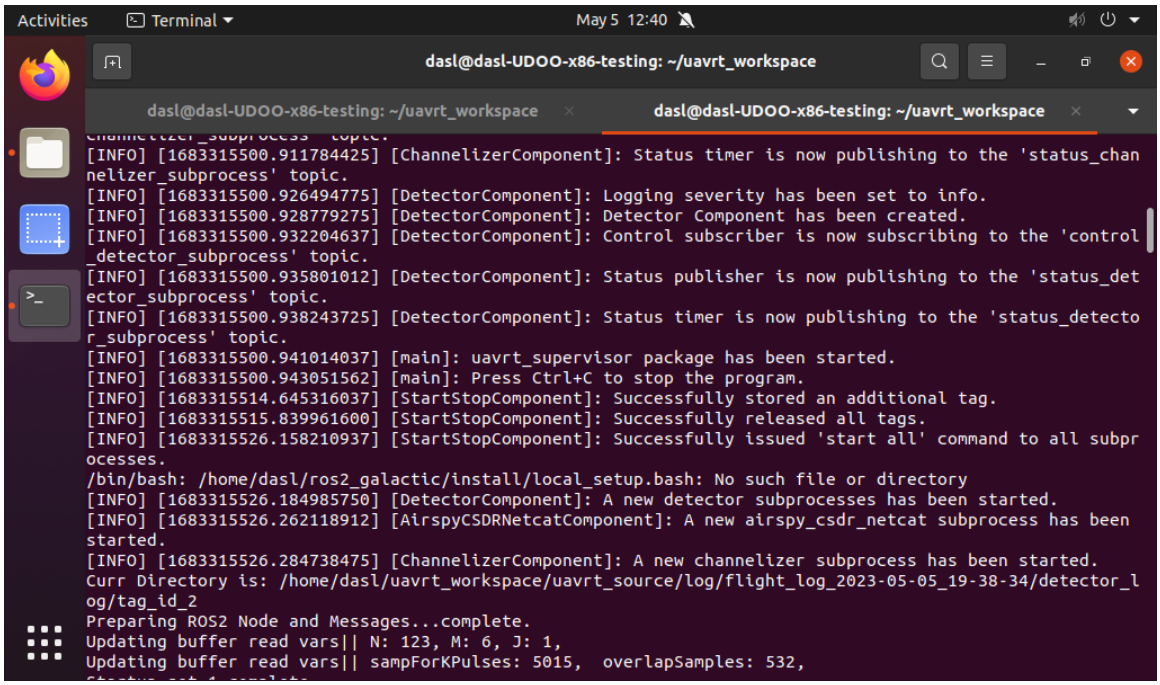

Figure 4.28: The uavrt_supervisor process is successfully processing data.

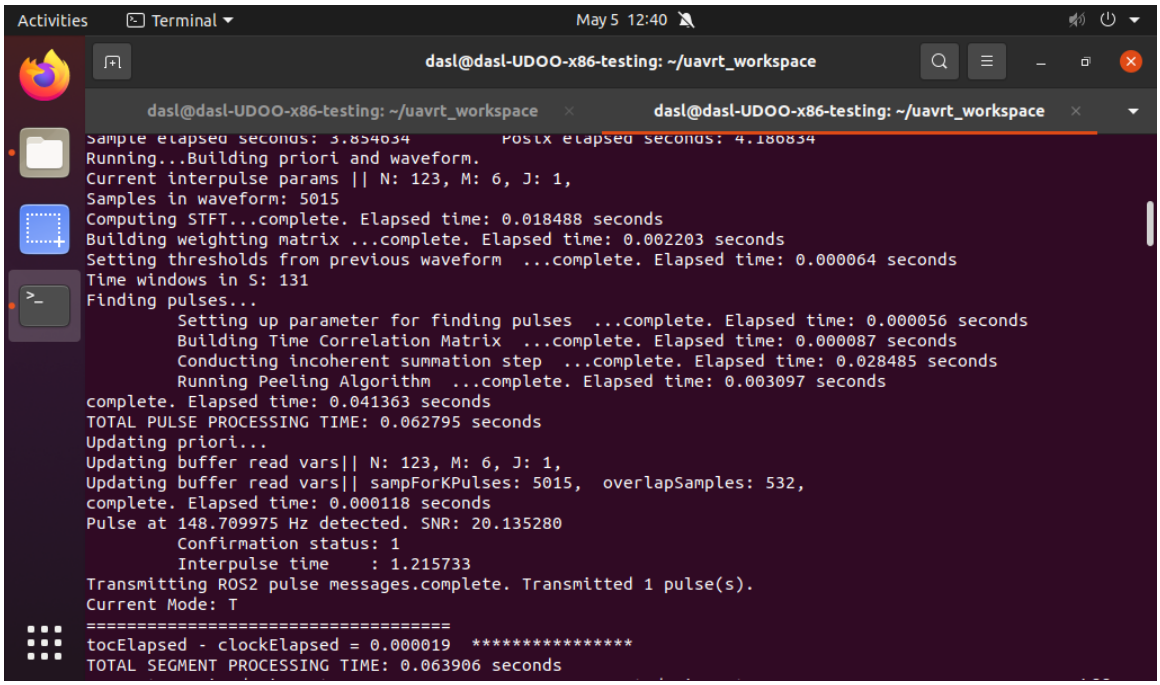

Figure 4.29: Example of what "Tracking mode" looks like.

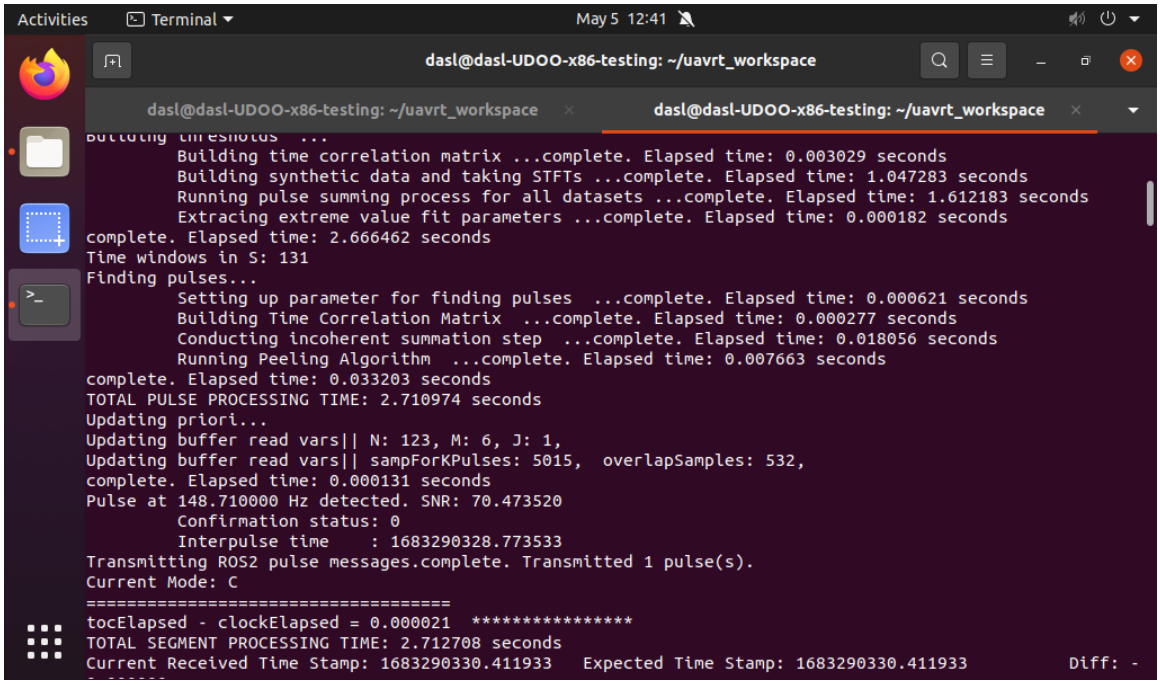

Figure 4.30: Example of what "Confirmed mode" looks like.

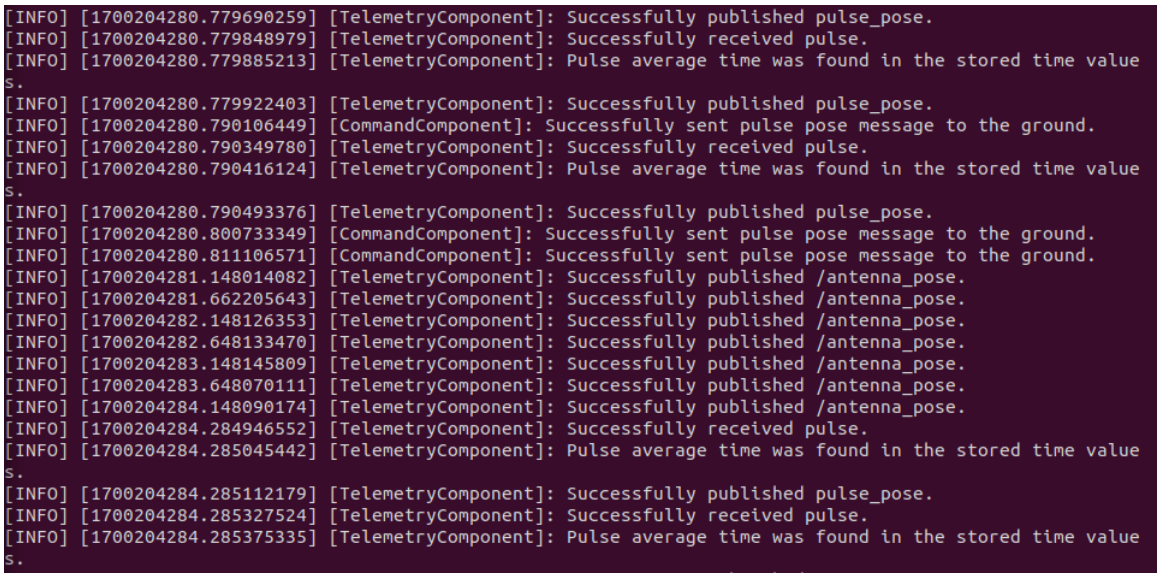

Figure 4.31: Log messages confirming pulse data transmission.

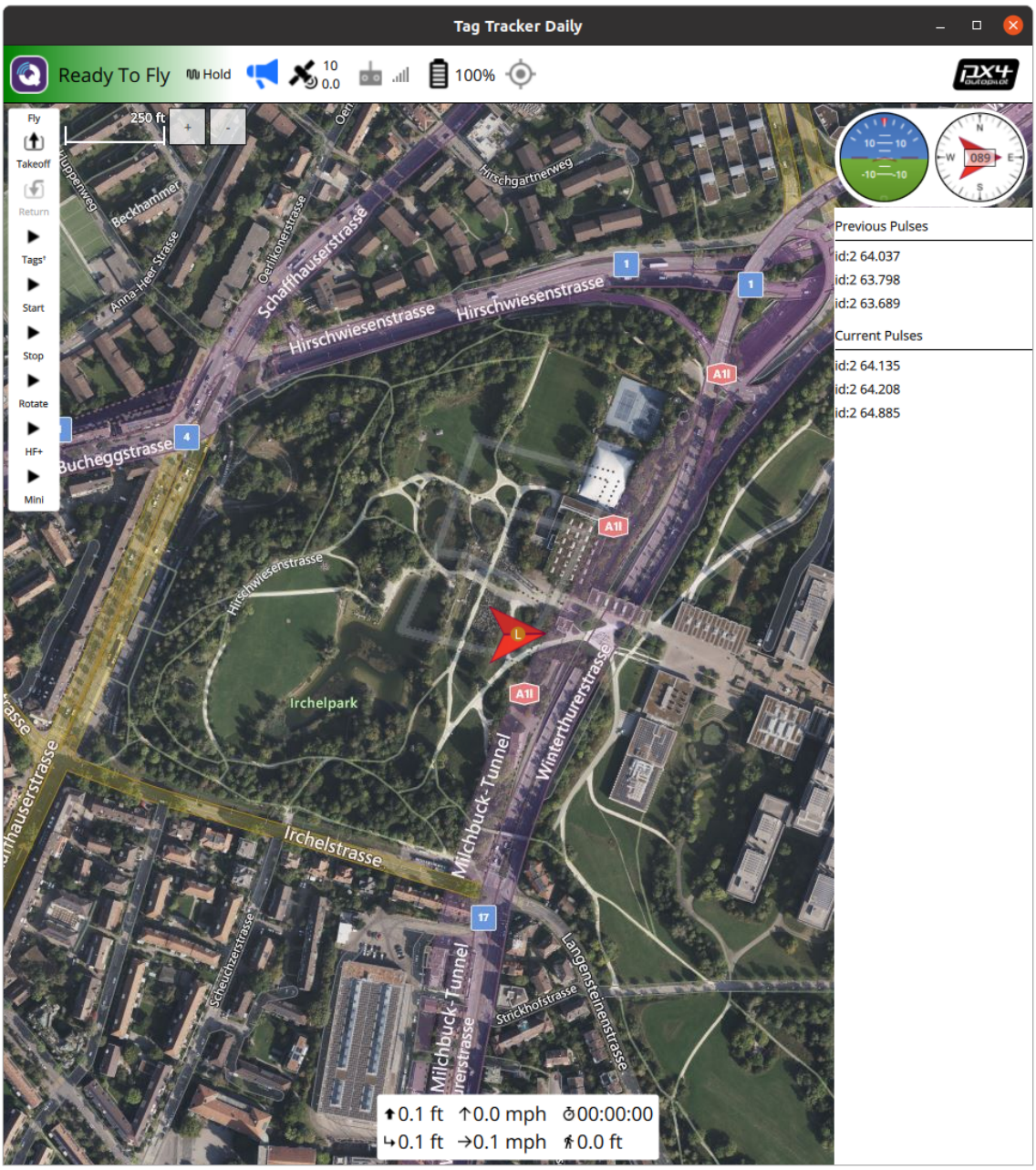

Figure 4.32: Tracked messages will show on the right-hand side of custom QGroundControl.

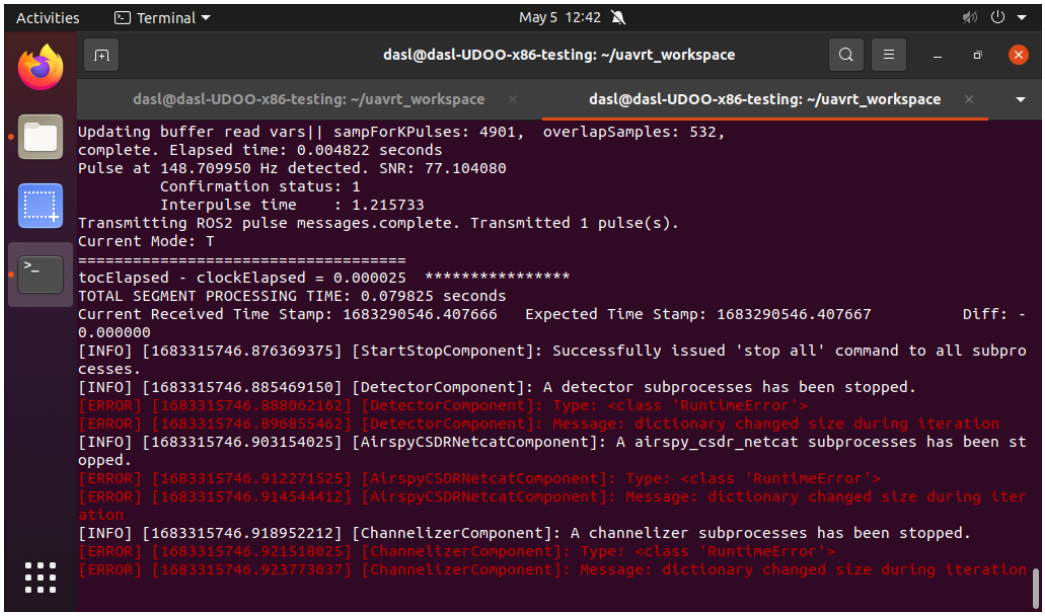

Figure 4.33: Stop all processes and shut down custom QGroundControl.

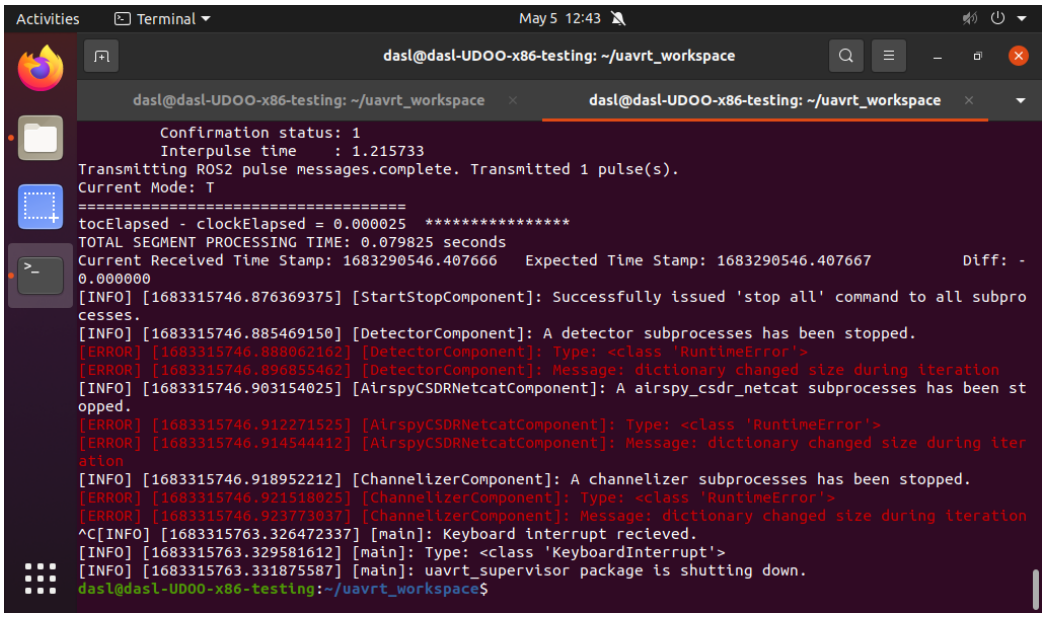

Figure 4.34: Shutdown uavrt_supervisor and uavrt_connection processes.

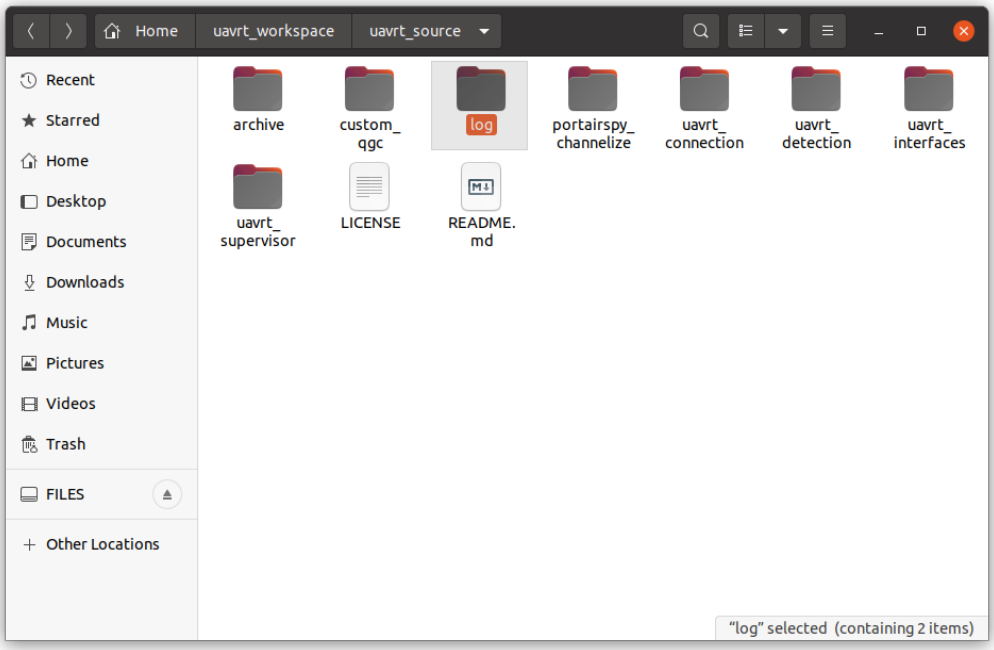

Figure 4.35: Access flight data in the "log" directory.

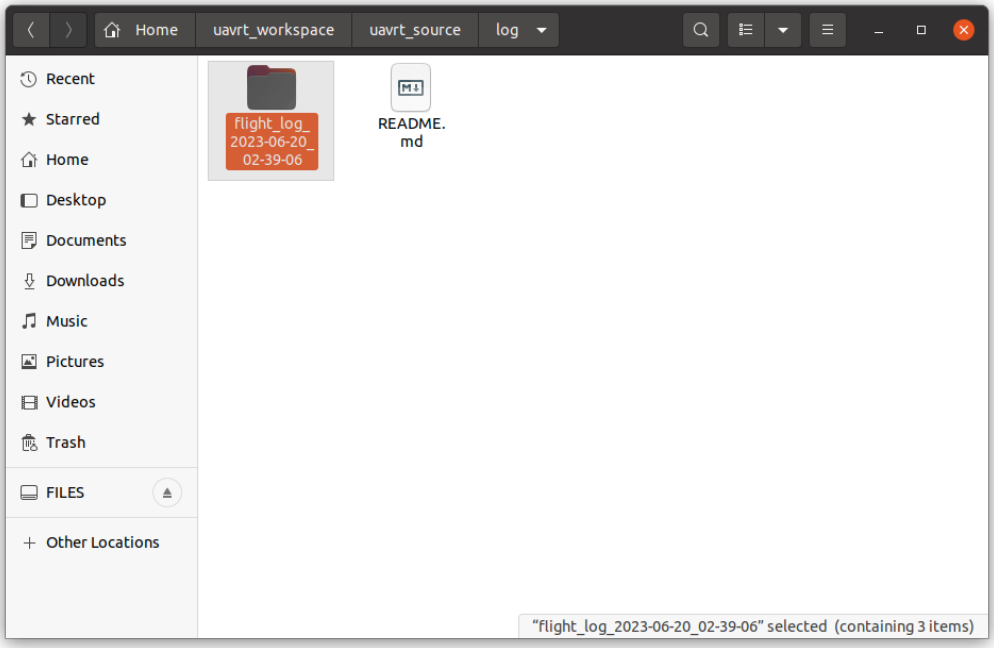

Figure 4.36: Flight directory that corresponds to the recent flight.

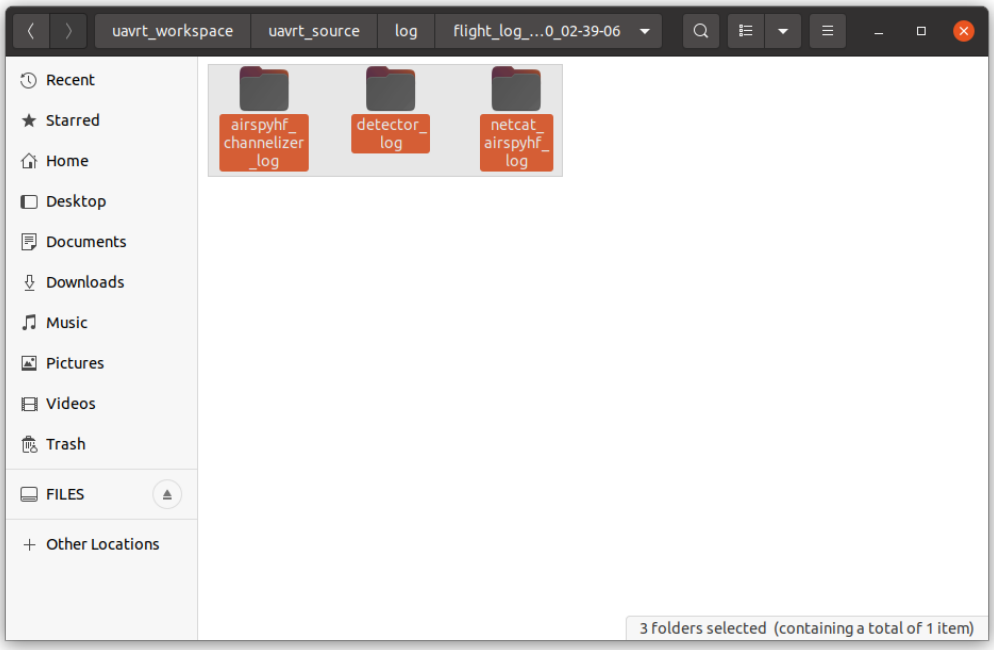

Figure 4.37: Log directory for subprocesses with uavrt_detection process configurations.

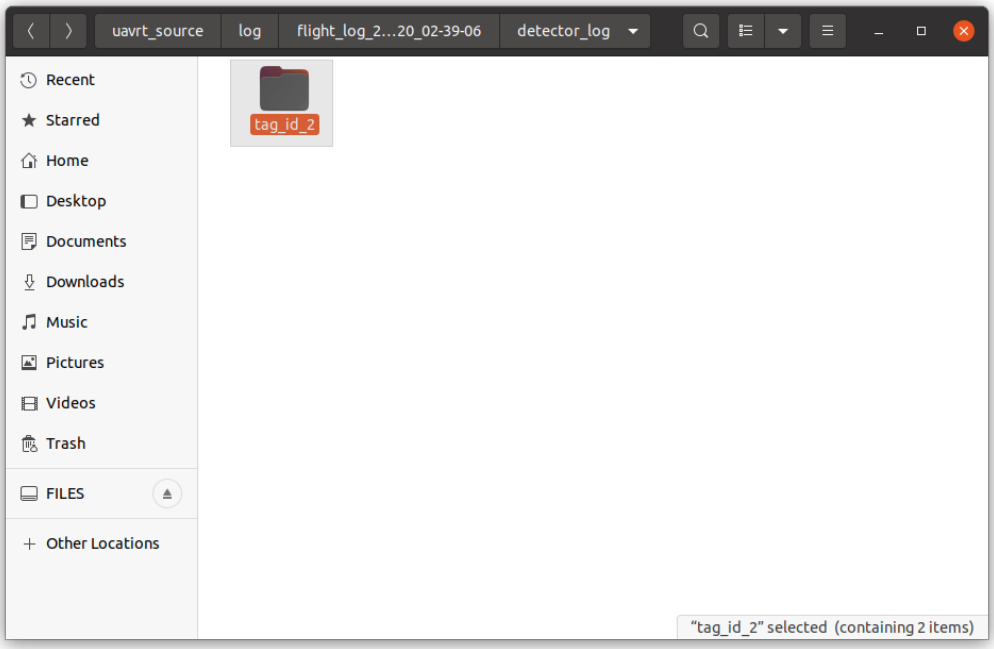

Figure 4.38: Individual directory for each tracked tag.

4.5.4 Executing individual processes for certain software components

The main focus of Section 4.5 was to demonstrate how to operate the UAV-RT software system as a whole. In the case of development and testing, it is beneficial to outline how processes would be started for the channelizer and uavrt_detection packages in their own terminal windows. This subsection also includes starting csdr, airspy_rx, and Netcat tools in one terminal window. The processes should be started in the order presented below.

Below is an example of how a channelizer processes is started with a sampling rate of 375,000 and decimation rate of 100:

```
cd ~/uavrt_workspace/uavrt_source/portairspy_channelize
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/dasl/uavrt_workspace/
    uavrt_source/portairspy_channelize"
./airspy_channelize 375000 100
```

A terminal window is required for running the csdr, airspy_rx, and Netcat tools, as these tools are required to pipe VHF data into the channelizer process. Below is an example of how a complete csdr, Netcat, and airspy_rx command call:

```
/usr/local/bin/airspy\_rx -f 148.710 -r - -p 0 -a 3000000 -t 0 -d |
    csdr fir\_decimate\_cc 8 0.05 HAMMING |
        netcat -u localhost 10000
```

A directory that contains "config" and "output" directories is necessary for starting a uavrt_detection process. The "config" directory contains a "detectorConfig.txt" file. Refer to [102] on the information and steps necessary to create a "detectorConfig.txt" file. Below is an example of how a uavrt_detection process is started:

```
cd ~/uavrt_workspace
source /opt/ros/galactic/setup.bash
. install/setup.bash
cd ~/example_detector
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/dasl/uavrt_workspace/
    uavrt_source/portairspy_channelize"
ros2 run uavrt_detection uavrt_detection
```

Chapter 5

TESTS AND RESULTS

5.1 Chapter overview

Chapter 5 provides a comprehensive evaluation of the UAV-RT system through systematic testing and analysis. Four tests were conducted: a set of performance benchmarks measuring resource allocation, a packet verification test analyzing reliability, a Tunnel Protocol message capacity test, and a field test analyzing the accuracy of gathered data. The presented conclusions offer insight on the system's capabilities and limitations, and identifies areas for improvement.

5.2 CPU and memory usage test

The goal of this test was conduct a set of benchmarks to evaluate the CPU and memory usage of the uavrt_connection, uavrt_connection, uavrt_detection, and channelizer packages. This test also assessed the CPU and memory usage of the airspy_rx, csdr, and, Netcat subprocesses. This test was conducted using two different computer configurations, with each computer representing varying levels of performance capabilities relative to one another. The specifications of the computers used for this test are listed in Table 5.1. The Ubuntu 20.04.6 LTS (Focal Fossa) operating system was used on each of the machines. The instructions listed in Section 4.3 explain how to prepare a machine to run and install the UAV-RT software system that was required for this test.

Computer model	CPU	SMT capable	Storage	GPU	RAM
UDOO X86 II Ultra	4 Core 4 threads Intel Pentium N3710 @ 2.56GHz	No	32GB eMMC storage	Intel HD Graphics 405	8GB DDR3L Dual Channel
Dell Precision T1700	4 Core 8 threads Intel Xeon E3-1240 v3 @ 3.40GHz	Yes	500GB hard disk drive	NVIDIA Quadro K600	16GB non-ECC DDR3 memory

Table 5.1: Computer specifications.

The Linux "top" utility [30] and Linux "pgrep" utility [31] were used to determine the amount of CPU and memory usage of a process. Below is the complete top and grep command call that was used for this test:

```
top -p $(pgrep -d',' -f "airspy_channelize|uavrt_detection|ros2|main|
    airspy_rx|csdr|netcat")
```

Figure 5.1 is an example of the output from the "top" and "pgrep" utilities while the UAV-RT software system was running on the Dell Precision T1700. Figure 5.2 is an example of the output from these utilities when the UAV-RT software system was running on the UDOO X86 II Ultra.

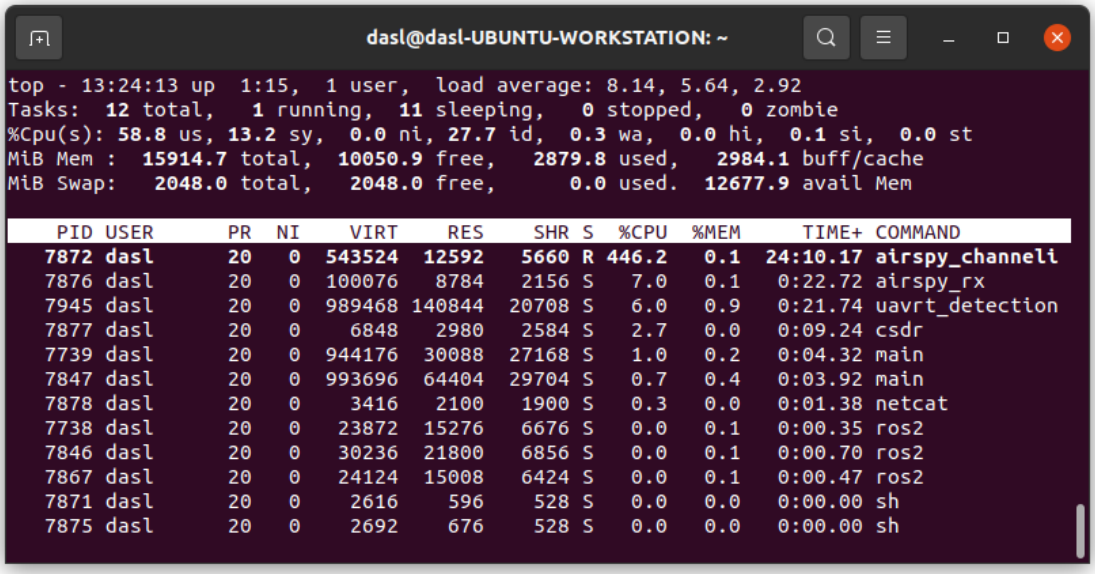

Figure 5.1: UAV-RT software system running on the Dell Precision T1700.

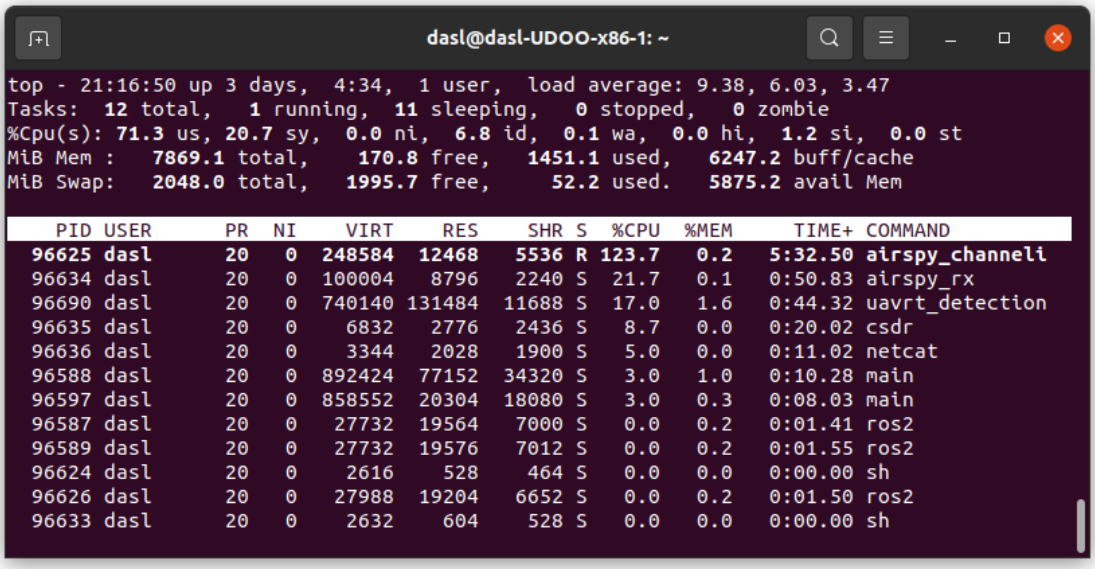

Figure 5.2: UAV-RT software system running on the UDOO X86 II Ultra.

The "%MEM" column denotes the percentage of physical RAM used by a process, indicating its share of the system's memory.

The "%CPU" column in the "top" command represents the percentage of CPU

utilization for a process, with the default display reflecting the usage relative to a single CPU core. On systems with multiple cores, the percentage can exceed 100%, indicating combined utilization across all cores.

The total possible %CPU usage can increase if simultaneous multi-threading (SMT), also known as Hyper-Threading in Intel processors, is enabled [27]. As noted by Hennessy and Patterson, SMT increases the usage of the functional units in a processor by using thread-level parallelism to hide long-latency events [26]. While SMT can improve overall system throughput by enabling better utilization of the processor's resources, it does not guarantee a linear increase in CPU availability, especially when it comes to inter-process communication (IPC).

There are multiple reasons why the additional %CPU due to SMT does not equate to a straightforward doubling of processing power:

- Resource sharing: SMT enables multiple threads to share the resources of a single physical core. However, these resources, such as the instruction pipeline, execution units, and cache, are finite. When two threads are executing simultaneously, they may compete for these resources, leading to a situation where both threads cannot achieve their maximum performance independently.

- Cache sharing: SMT allows threads to share the same cache. While this can improve overall throughput, it can also lead to cache contention, where one thread's data might overwrite another thread's data in the cache. This can result in increased cache misses and impact performance.

- Instruction mix: The effectiveness of SMT depends on the nature of the workload. If the workload involves a mix of instructions that can be executed simultaneously, SMT can provide substantial benefits. However, for certain types of instructions or workloads with dependencies, the simultaneous execution may

not be fully exploited, limiting the gain in CPU availability.

- Thread synchronization overhead: In scenarios where threads need to synchronize their execution or communicate frequently, there can be overhead associated with managing this synchronization. This overhead might offset some of the benefits gained from SMT, especially if threads spend significant time waiting for synchronization.

- System factors: The effectiveness of SMT is also dependent on factors such as memory, I/O, and the overhead of the system (such as power supply) [26]. The effectiveness of SMT will be reduced if these factors are limited in availability to the threads.

Testing procedure

To conduct this test, the UAV-RT software system was installed and configured on each of the machines listed in Table 5.1, and then started using the steps found in Section 4.5. Each computer represents its own benchmark. A tag was being actively tracked for both benchmarks. Each benchmark ran for a total of 30 minutes, and CPU and memory usage readings were captured at 1 minute, 5 minutes, and 10 minutes, 15 minutes, 20 minutes, 25 minutes, and 30 minutes for each of the processes associated with the UAV-RT software system. Note that the results for the channelizer process were scaled down in order to properly compare the results of UAV-RT software system.

The results of the UAV-RT software system benchmark for the UDOO X86 II Ultra are displayed in Table 5.2 and Table 5.3. The summary statistics for the data displayed in Table 5.2 are shown in Table 5.4, and the resulting histogram of the data in Table 5.2 is shown in Figure 5.2.

The results of the UAV-RT software system benchmark for the Dell Precision

T1700 are displayed in Table 5.5 and Table 5.6. The summary statistics for the data displayed in Table 5.5 are shown in Table 5.7, and the resulting histogram of the data in Table 5.5 is shown in Figure 5.4.

Process type	%CPU-1 min.	%CPU-5 min.	%CPU-10 min.	%CPU-15 min.	%CPU-20 min.	%CPU-25 min.	%CPU-30 min.
channelizer	170.1	167.1	162.0	154.2	166.3	158.8	158.3
airspy_rx	22.9	22.3	21.3	22.6	21.7	21.6	22.3
uavrt_ detection	16.9	16.7	19.6	16.3	17.6	17.3	15.0
csdr	9.0	8.7	9.7	9.3	9.0	8.6	9.3
uavrt_ connection	3.3	3.3	3.3	2.8	3.0	3.1	3.7
uavrt_ supervisor	3.3	2.7	3.0	3.0	3.3	2.7	2.7
netcat	5.3	5.7	4.7	5.3	5.0	5.0	5.0

Table 5.2: CPU usage of the UAV-RT system running on the UDOO X86 II Ultra.

Process type	%MEM
channelizer	0.2
airspy_rx	0.1
uavrt_detection	1.6
csdr	0.0
uavrt_connection	1.0
uavrt_supervisor	0.3
netcat	0.0

Table 5.3: Memory usage of the UAV-RT system running on the UDOO X86 II Ultra.

Process type	Mean (%)	Standard Deviation (%)	Median (%)	Minimum (%)	Maximum (%)	Range (%)
channelizer	162.42	5.37	162.0	154.2	170.1	15.9
airspy_rx	22.07	0.68	22.3	21.3	22.9	1.6
uavrt_ detection	17.09	1.26	17.3	15.0	19.6	4.6
csdr	9.07	0.48	9.0	8.6	9.7	1.1
uavrt_ connection	3.17	0.28	3.3	2.8	3.7	0.9
uavrt_ supervisor	2.93	0.26	3.0	2.7	3.3	0.6
netcat	5.10	0.39	5.0	4.7	5.7	1.0

Table 5.4: Summary statistics of CPU usage for each process type in the UAV-RT system on UDOO X86 II Ultra.

Process type	%CPU-1 min.	%CPU-5 min.	%CPU-10 min.	%CPU-15 min.	%CPU-20 min.	%CPU-25 min.	%CPU-30 min.
channelizer	473.8	471.3	481.0	486.0	464.1	487.4	482.0
airspy_rx	7.3	7.6	7.3	7.7	7.6	7.0	7.3
uavrt_ detection	6.7	7.0	6.7	7.0	7.7	6.0	6.3
csdr	3.7	3.0	3.3	3.0	2.7	3.0	3.0
uavrt_ connection	2.0	1.3	2.0	2.3	2.0	2.0	2.3
uavrt_ supervisor	1.3	1.7	1.3	1.3	2.0	1.3	1.3
netcat	0.7	0.3	0.3	0.7	0.3	0.3	0.3

Table 5.5: CPU usage of the UAV-RT system running on the Dell Precision T1700.

Process type	%MEM
channelizer	0.1
airspy_rx	0.1
uavrt_detection	0.9
csdr	0.0
uavrt_connection	0.2
uavrt_supervisor	0.4
netcat	0.0

Table 5.6: Memory usage of the UAV-RT system running on the Dell Precision T1700.

Process type	Mean (%)	Standard Deviation (%)	Median (%)	Minimum (%)	Maximum (%)	Range (%)
channelizer	476.21	8.12	482.0	464.1	487.4	23.3
airspy_rx	7.23	0.29	7.3	7.0	7.7	0.7
uavrt_ detection	6.57	0.46	6.7	6.0	7.7	1.7
csdr	3.0	0.35	3.0	2.7	3.7	1.0
uavrt_ connection	2.0	0.44	2.0	1.3	2.3	0.7
uavrt_ supervisor	1.6	0.35	1.3	1.3	2.0	0.7
netcat	0.43	0.17	0.3	0.3	0.7	0.4

Table 5.7: Summary statistics of the data in Table 5.5.

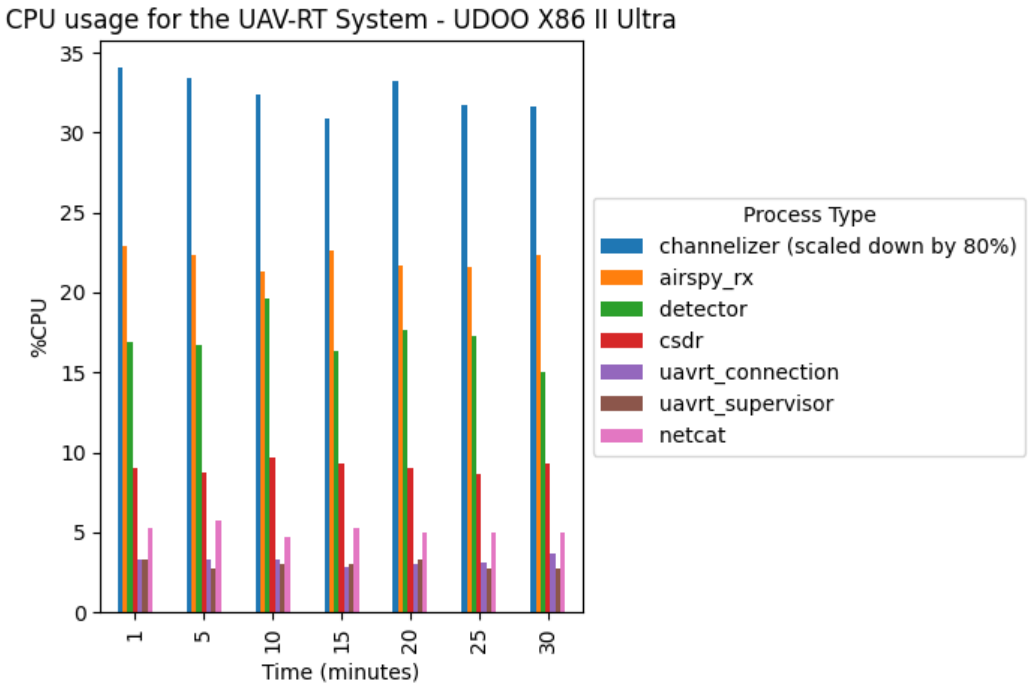

Figure 5.3: Histogram of the data in Table 5.2.

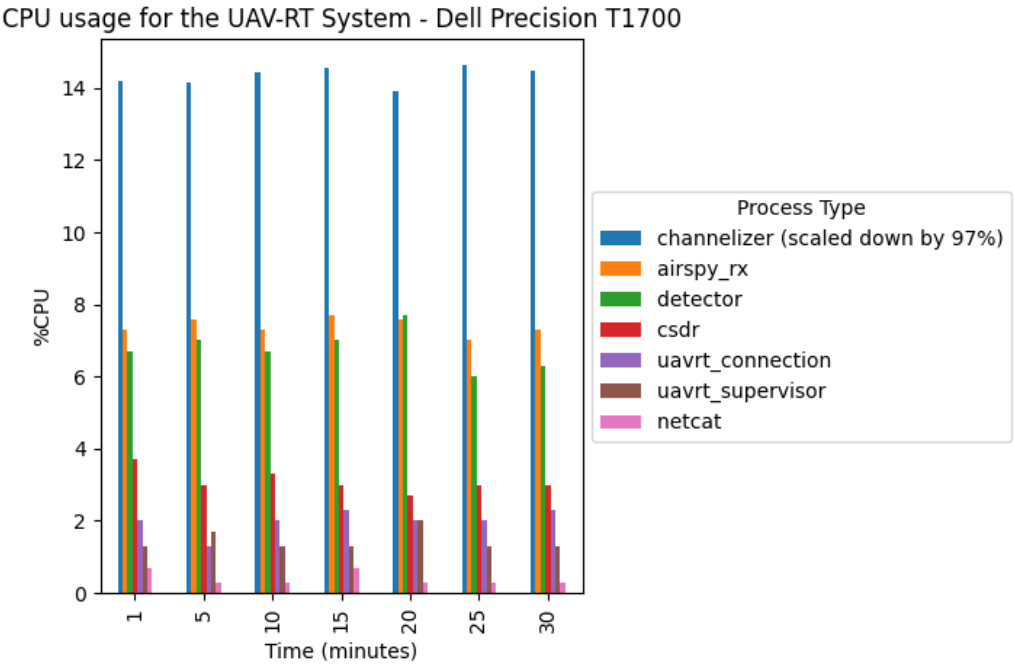

Figure 5.4: Histogram of the data in Table 5.5.

Discussion

Based on the information in Table 5.4, the following conclusions can be made:

- **channelizer:**
  - The mean CPU usage for a channelizer process is 162.42%, with a standard deviation of 5.37%.
  - The median is 162.0%, and the range is 15.9%.
  - The process demonstrates relatively consistent CPU usage, as suggested by the small standard deviation.
- **airspy_rx:**
  - The mean CPU usage for an airspy_rx process is 22.07%, with a small standard deviation of 0.68%.
  - The data indicates stable performance, as the range is only 1.6%, and the values are close to the mean.
- **uavrt_detection:**
  - The mean CPU usage for a uavrt_detection process is 17.09%, with a standard deviation of 1.26%.
  - The data suggests moderately consistent CPU usage, with a range of 4.6%.
- **csdr:**
  - The mean CPU usage for a csdr process is 9.07%, and the standard deviation is 0.48%.
  - The low standard deviation and small range (1.1%) suggest relatively stable CPU usage for this process.

- **uavrt_connection:**
  - The mean CPU usage for a uavrt_connection process is 3.17%, with a standard deviation of 0.28%.
  - The data indicates consistent and relatively low CPU usage, as reflected by the small standard deviation and range.
- **uavrt_supervisor:**
  - The mean CPU usage for a uavrt_supervisor process is 2.93%, and the standard deviation is 0.26%.
  - The low standard deviation and range (0.6%) suggest stable and low CPU usage for this process.
- **netcat:**
  - The mean CPU usage for a netcat process is 5.10%, with a standard deviation of 0.39%.
  - The data indicates moderate and stable CPU usage for the netcat process.

Based on the information in Table 5.7, the following conclusions can be made:

- **channelizer:**
  - The mean CPU usage for a channelizer process is 476.21%, with a standard deviation of 8.12%.
  - The median is 482.0%, and the range is 23.3%.
  - The process demonstrates relatively consistent CPU usage, as suggested by the small standard deviation.

- **airspy_rx:**
  - The mean CPU usage for an airspy_rx process is 7.23%, with a small standard deviation of 0.29%.
  - The data indicates stable performance, as the range is only 0.7%, and the values are close to the mean.
- **uavrt_detection:**
  - The mean CPU usage for a uavrt_detection process is 6.57%, with a standard deviation of 0.46%.
  - The data suggests a moderately consistent CPU usage, with a range of 1.7%.
- **csdr:**
  - The mean CPU usage for a csdr process is 3.0%, and the standard deviation is 0.35%.
  - The low standard deviation and small range (1.0%) suggest relatively stable CPU usage for this process.
- **uavrt_connection:**
  - The mean CPU usage for a uavrt_connection process is 2.0%, with a standard deviation of 0.44%.
  - The data indicates consistent and relatively low CPU usage, as reflected by the small standard deviation and range.
- **uavrt_supervisor:**

- The mean CPU usage for a uavrt_supervisor process is 1.6%, and the standard deviation is 0.35%.
- The low standard deviation and range (0.7%) suggest stable and low CPU usage for this process.

- **netcat:**
  - The mean CPU usage for a netcat process is 0.43%, with a standard deviation of 0.17%.
  - The data indicates low and stable CPU usage for a netcat process.

Table 5.3 and Table 5.6 show that the memory usage of each process started by the UAV-RT software system remained constant on both computers.

Figure 5.3 and Figure 5.4 show that the channelizer process makes up the majority of the CPU usage associated with the UAV-RT software system when one of each process is running.

The UDOO X86 II Ultra has 4 physical cores without SMT capabilities. As such, the total %CPU for the UDOO X86 II Ultra is 400%, as each core contributes 100%. Totalling the mean values for each of the processes in Table 5.4 results in a %CPU of 222.85%. If each uavrt_detection package process takes up 17.09 %CPU on average, then the theoretical maximum number of uavrt_detection packages that could be started and run on the UDOO X86 II Ultra is 10. However, the actual amount is lower due to other processes running on the computer.

The same extrapolation approach cannot be applied to the Dell Precision T1700. It has 4 physical cores with SMT enabled. With SMT enabled, the total %CPU available could see an increase of 0% to 100%. As such, the total %CPU available to the Dell Precision T1700 could be between 400% and 800%. For the reasons

discussed at the beginning of Section 5.2, extrapolating a fixed percentage increase in %CPU availability oversimplifies the complexities of a uavrt_detection package process and the impact of SMT on the different types of instructions and resource usage in uavrt_detection package processes.

5.3 Packet verification test

The goal of this test was to confirm the proper reception of all transmitted detected pulse data packets to custom QGroundControl. This verification process used the PX4-Gazebo headless simulator detailed in Subsection 4.3.4. The testing was carried out on the Dell Precision T1700 machine specified in Table 5.1. Refer to 4.5 for information on operating the UAV-RT system.

The test procedure involved verifying packets that had been received by examining the terminal output of custom QGroundControl and the contents of the "pulse_pose_log.txt" file. The CustomPluginLog within custom QGroundControl needed to be toggled on to view the detected pulse data packets received by custom QGroundControl. The location where this setting could be toggled for custom QGroundControl is illustrated in Figure 5.5. The appearance of the terminal output is depicted in Figure 5.6.

The "pulse_pose_log.txt" file was generated by the uavrt_connection package. The process of generating the "pulse_pose_log.txt" file and understanding its fields was explained in Subsection 3.3.3. The formatting of the "pulse_pose_log.txt" file was detailed in Figure 5.7.

The tag information for the tracked tag in this test is entered in the "TagInfo.txt" document for custom QGroundControl use. See Subsection 4.4.2 for instructions on how to enter tag information into this file. Refer to the Table 5.8 for details on the tag information used in this test. See Table 2.14 for details regarding each parameter.

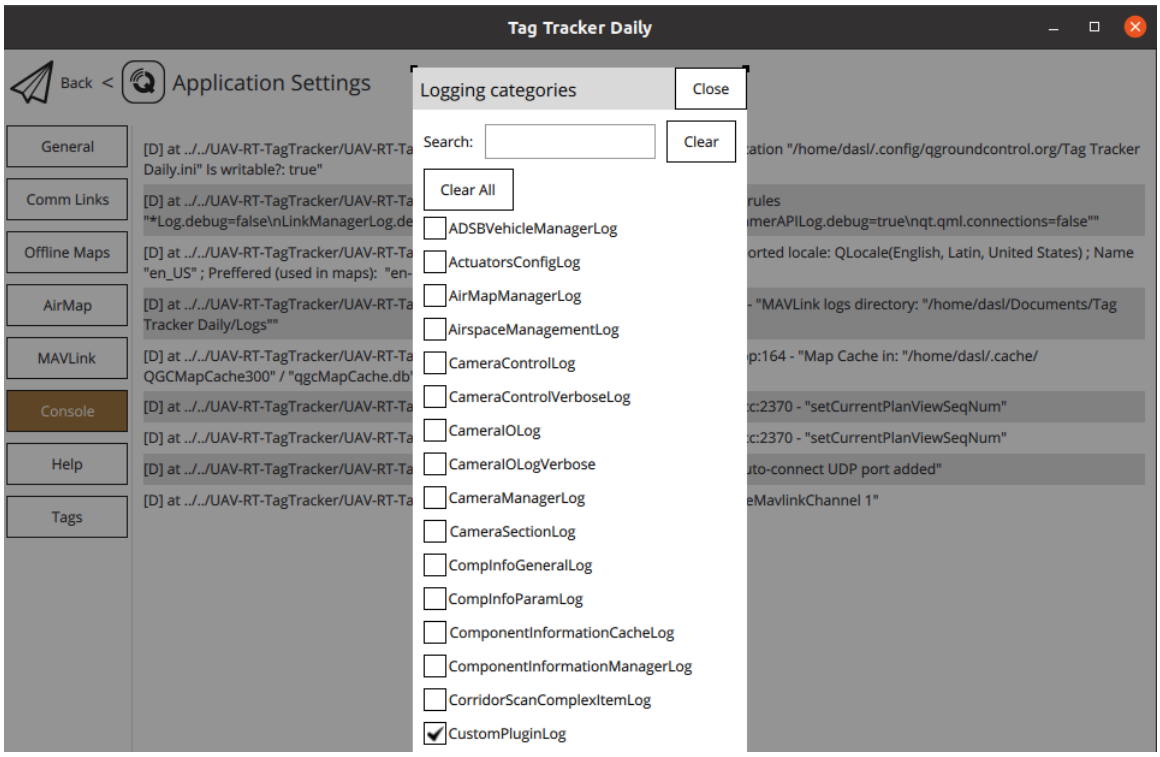

Figure 5.5: CustomPluginLog setting in custom QGroundControl.

```
dasl@dasl-UBUNTU-WORKSTATION:~$ ./TagTracker.AppImage
Settings location "/home/dasl/.config/qgroundcontrol.org/Tag Tracker Daily.ini" Is writable?: true
Filter rules "*Log.debug=false\nLinkManagerLog.debug=true\nCustomPluginLog.debug=true\nGStreamerAPILog.de
bug=true\nqt.qml.connections=false"
System reported locale: QLocale(English, Latin, United States) ; Name "en_US" ; Preffered (used in maps):
  "en-US"
MAVLinkLogManagerLog: MAVLink logs directory: "/home/dasl/Documents/Tag Tracker Daily/Logs"
Map Cache in: "/home/dasl/.cache/QGCMapCache300" / "qgcMapCache.db"
setCurrentPlanViewSeqNum
setCurrentPlanViewSeqNum
LinkManagerLog: New auto-connect UDP port added
LinkManagerLog: allocateMavlinkChannel 1
Adding target QHostAddress("172.17.0.2") 18570
setCurrentPlanViewSeqNum
setCurrentPlanViewSeqNum
ParameterManagerLog: Attemping load from cache
ParameterManagerLog: Parameters loaded from cache /home/dasl/.config/qgroundcontrol.org/ParamCache/1_1.v2
setCurrentPlanViewSeqNum
setCurrentPlanViewSeqNum
CustomPluginLog: Tunnel command ack received - command:result "start tags" 1
CustomPluginLog: Tunnel command ack received - command:result "tag" 1
CustomPluginLog: Tunnel command ack received - command:result "end tags" 1
CustomPluginLog: Tunnel command ack received - command:result "start detection" 1
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 2 148 1700205395.35 5.81 1
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 2 148 1700205396.58 -8.99 1
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 2 148 1700205419.23 69.65 1
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 2 148 1700205420.49 69.58 1
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 2 148 1700205421.74 70.22 1
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 2 148 1700205423.00 70.30 1
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 2 148 1700205424.26 71.18 1
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 2 148 1700205425.51 70.81 1
```

Figure 5.6: Custom QGroundControl terminal output.

```
2, 148.71, 1700205419, 230494976, 1700205419, 250494957, 1700205420, 434494972, 1700205420, 4294967295,
69.6484, 0.219541, 1, 69.8253, 1, 1, 47.3978, 8.54561, 488.077, 0.00189066, -0.00109068, 0.71339,
0.700764
2, 148.71, 1700205420, 487028360, 1700205420, 507028341, 1700205421, 691028357, 1700205421, 4294967295,
69.5808, 0.216148, 2, 69.8253, 1, 1, 47.3978, 8.54561, 488.071, 0.0019522, -0.0011528, 0.713467, 0.700685
2, 148.71, 1700205421, 743561745, 1700205421, 763561726, 1700205422, 947561741, 1700205423, 67561626,
70.218, 0.250311, 3, 69.8253, 1, 1, 47.3978, 8.54561, 488.049, 0.00199207, -0.00128593, 0.713615,
0.700534
2, 148.71, 1700205422, 999828339, 1700205423, 19828320, 1700205424, 203828335, 1700205424, 323828220,
70.3041, 0.293186, 1, 70.7781, 1, 1, 47.3978, 8.54561, 488.032, 0.00204829, -0.00132115, 0.713694,
0.700453
2, 148.71, 1700205424, 256361723, 1700205424, 276361704, 1700205425, 460361719, 1700205425, 4294967295,
71.179, 0.358614, 2, 70.7781, 1, 1, 47.3978, 8.54561, 488.011, 0.00209553, -0.00132549, 0.713854, 0.70029
2, 148.71, 1700205425, 512894869, 1700205425, 532894850, 1700205426, 716894865, 1700205426, 4294967295,
70.8072, 0.329191, 3, 70.7781, 1, 1, 47.3978, 8.54561, 487.995, 0.00217034, -0.00144991, 0.713951,
0.700191
2, 148.71, 1700205426, 769161701, 1700205426, 789161682, 1700205427, 973161697, 1700205428, 93161583,
70.844, 0.329893, 1, 70.8175, 1, 1, 47.3978, 8.54561, 487.988, 0.00208936, -0.00163105, 0.714158, 0.69998
2, 148.71, 1700205428, 25695086, 1700205428, 45695066, 1700205429, 229695082, 1700205429, 349694967,
70.7422, 0.32225, 2, 70.8175, 1, 1, 47.3978, 8.54561, 487.98, 0.00204972, -0.00162538, 0.714365, 0.699769
2, 148.71, 1700205429, 282228470, 1700205429, 302228451, 1700205430, 486228466, 1700205430, 4294967295,
70.8654, 0.331523, 3, 70.8175, 1, 1, 47.3978, 8.54561, 487.989, 0.00200457, -0.00165884, 0.714541,
0.699589
```

Figure 5.7: Formatting and contents of the "pulse_pose_log.txt" file.

Tag ID	Frequency (Hz)	Interpulse duration (ms)	Pulse width (ms)	Interpulse uncer-tainty (ms)	Interpulse jitter (ms)	Number of inte-grations	False alarm proba-bility
2	148710000	1254	20	60	10	3	0.005

Table 5.8: Tag information used for the packet verification test.

Testing procedure

Terminal windows were opened to initiate processes for the following software components: uavrt_connection package, uavrt_supervisor package, PX4-Gazebo headless simulator, and custom QGroundControl. The Linux "script" utility [28] was launched in each terminal window before executing the processes to capture their output. The processes were executed in no particular order.

After the processes were started, a "Tag" command was sent from custom QGround-

Control to the uavrt_connection package, which was subsequently forwarded to the uavrt_supervisor package. It was confirmed that the "Tag" command had been received by the uavrt_connection and uavrt_supervisor packages by reviewing the terminal output for each of the processes. A "Start" command was then sent from custom QGroundControl to the uavrt_connection and uavrt_supervisor packages. This command was validated by examining the terminal output of the UAV-RT packages.

The active tracking of the tag was confirmed by reviewing the terminal output from custom QGroundControl and the outputs from the uavrt_connection and uavrt_supervisor packages. The testing setup was allowed to run uninterrupted for 30 minutes. After the 30-minute duration, a "Stop" command was sent from custom QGroundControl to the uavrt_connection and uavrt_supervisor packages. Once the "Stop" command had been confirmed, all processes, including the script utility processes, were stopped. Using the script utility, the contents of each terminal window were saved to text files in the home directory.

A Python script was then used to process and compare the contents of two files that were generated during the test: the "pulse_pose_log.txt" and the output of custom QGroundControl, saved as "custom_qgc_output_packet_verification_test.txt." The script begins by reading the contents of "pulse_pose_log.txt" and "custom_qgc_output_packet_verification_test.txt." It specifically processes the Unix time values from columns 2 (seconds) and 3 (nanoseconds) in "pulse_pose_log.txt." The values in column 3 are rounded up to eight digits, and these results are then added to the corresponding entry in column 2. This processed data is stored in an array called "pulse_pose_log_time." The script then extracts the Unix time values from column 5 (seconds and nanoseconds) in the "custom_qgc_output_packet_verification_test.txt" file and creates an array named "custom_qgc_output_time." The script then performs a comparison between the two arrays. It identifies and prints any time values

from "custom_qgc_output_time" that are not present in "pulse_pose_log_time," with a specified tolerance of .07 nanoseconds.

Using this Python script, four detected pulse data packets were identified as not received by custom QGroundControl during this test. Refer to 5.9 for the detected pulse data packets that were not received by custom QGroundControl.

Dropped packet	Unix time	GMT
1	1700205609	12:20:09 MST GMT -7
2	1700205681	12:21:21 MST GMT -7
3	1700206726	12:38:46 MST GMT -7
4	1700207037	12:43:57 MST GMT -7

Table 5.9: Dropped packets during the packet verification test.

Discussion

It is difficult to know exactly what caused these detected pulse packets to be dropped. This test was conducted using the PX4-Gazebo headless simulator in lieu of a physical Pixhawk flight controller. Telemetry radios were not required for this test. This narrows down the list of components where packets could have been dropped. Since a connection to custom QGroundControl and a Pixhawk flight controller is required in order to exchange Tunnel Protocol messages between the UAV-RT software packages and custom QGroundControl, it is difficult to further reduce the list of active components for this test.

It is worth noting that the rate at which detected pulse packets are being sent from the uavrt_connection package to custom QGroundControl is already being throttled within the uavrt_connection package. The initial packet verification test was run with a 10 milliseconds delay before a detected pulse packets was forwarded to the

Pixhawk flight controller to be sent to the ground. An additional packet verification test was conducted with a 20 milliseconds delay. Packet loss did not occur during this additional packet verification test.

Field testing that incorporates a physical Pixhawk flight controller and a pair of telemetry radios should also be completed in order to determine what effect interference and distance has on the packet transmission and receiving.

5.4 Tunnel Protocol message capacity test

The goal of this test was to provide insight on the data transmission bottleneck noted in Section 5.3 and to determine the amount of data that can be transmitted using Tunnel Protocol messages before data packets begin to drop. This testing process used the PX4-Gazebo headless simulator detailed in Subsection 4.3.4. The testing was carried out on the Dell Precision T1700 machine specified in Table 5.1.

The test procedure involved verifying packets that had been received by examining the terminal output of custom QGroundControl and the contents of the terminal output of a custom script. The CustomPluginLog within custom QGroundControl needed to be toggled on to view the detected pulse data packets received by custom QGroundControl. The location where this setting could be toggled for custom QGroundControl is illustrated in Figure 5.5. The appearance of the terminal output is depicted in Figure 5.6.

The custom script used for this test uses code from the main.py file described in Subsection 3.2.1 to establish a UDP connection with the PX4-Gazebo headless simulator and custom QGroundControl. The PulseInfo_t and HeaderInfo_t structs described in Subsection 2.6.1 are used to encapsulate the data that is sent to custom QGroundControl. Code from the command_component.cpp file described in Subsection 3.3.5 is used to send the encapsulated data from the custom script to custom

QGroundControl. A for-loop is used to instantiate a Tunnel Protocol message, fill the data fields of the message, and then send the message to custom QGroundControl. The custom script prints a confirmation message to the terminal each time a Tunnel Protocol message is sent to custom QGroundControl.

In order to control the Tunnel Protocol message transmission rate between the custom script and custom QGroundControl, a delay was used in the body of the for-loop of the custom script. This delay was measured in milliseconds. The rate at which packets were transmitted from the custom script to custom QGroundControl was measured in packets per millisecond. The delay was facilitated by sleeping the thread using the C++ Standard Library thread class [19].

Testing procedure

PulseInfo_t structs are instantiated and filled with the data from the first packet that was dropped during the test described in Section 5.3. Refer to Table 5.9 for the detected pulse data packets that were not received by custom QGroundControl during the test described in Section 5.3. This kept the amount of data encapsulated in the PulseInfo_t struct and packaged in the Tunnel Protocol messages at a constant size of 128 bytes. Note that the "tag_id" data field of the PulseInfo_t struct was set equal to the current iteration of the for-loop used in the custom script. This value was used to assign a packet number to each of the packets set to custom QGroundControl, and can be viewed in the terminal output of custom QGroundControl. The "command" data field within the HeaderInfo_t struct was set 7 to indicate to custom QGroundControl that the message is a detected pulse packet. Refer to Table 2.16 for information regarding the constants that are used in Tunnel Protocol messages.

An initial test was conducted where 10 detected pulse data packets were sent to custom QGroundControl. A delay was not used for this test. Of the 10 detected pulse

data packets that were sent, only the first packet was received by custom QGroundControl. Refer to Figure 5.8 for the terminal output from custom QGroundControl. Decreasing the number of detected pulses packets that were sent to custom QGroundControl did not increase the number of detected pulse packets that were received by custom QGroundControl.

```
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 0 148 1700205609.00 70.46 1
```

Figure 5.8: Detected pulse data packets received by custom QGroundControl when 10 packets are sent with no delay.

Additional tests beyond the initial test incorporate the delay referred to at the beginning of Section 5.4. When a delay of 1 millisecond was used and 10 detected pulse packets were sent to custom QGroundControl, 4 of the 10 packets were received by custom QGroundControl. Refer to Figure 5.9 for the terminal output from custom QGroundControl. The number of detected pulses packets that were sent to custom QGroundControl was reduced from 10 to 5 packets while the delay remained at 1 millisecond. With a delay of 1 millisecond, 2 of the 5 detected pulse packets were received by custom QGroundControl. Refer to Figure 5.10 for the terminal output from custom QGroundControl.

```
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 0 148 1700205609.00 70.46 1
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 2 148 1700205609.00 70.46 1
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 6 148 1700205609.00 70.46 1
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 9 148 1700205609.00 70.46 1
```

Figure 5.9: Detected pulse data packets received by custom QGroundControl when 10 packets are sent with a 1 millisecond delay.

```
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 0 148 1700205609.00 70.46 1
CustomPluginLog: PULSE tag_id:frequency_hz:time:snr:confirmed 1 148 1700205609.00 70.46 1
```

Figure 5.10: Detected pulse data packets received by custom QGroundControl when 5 packets are sent with a 1 millisecond delay.

Additional tests were run where the delay was increased from 1 millisecond to 10 milliseconds and the total number of detected pulse packets that were sent to

custom QGroundControl remained at 10. Refer to Table 5.10 for the results of these additional tests.

Delay (ms)	Total number of packets sent	Total number of packets received	Percentage of packets that were dropped
0	10	1	90%
1	10	4	60%
2	10	6	40%
3	10	8	20%
4	10	10	0%
5	10	10	0%
6	10	10	0%
7	10	10	0%
8	10	10	0%
9	10	10	0%
10	10	10	0%

Table 5.10: Percentage of dropped packets when the total number of packets was equal to 10.

Additional tests were run where the delay was increased from 1 millisecond to 10 milliseconds and the total number of detected pulse packets that were sent to custom QGroundControl remained at 100. Refer to Table 5.11 for the results of these additional tests.

Delay (ms)	Total number of packets sent	Total number of packets received	Percentage of packets that were dropped
0	100	1	99%
1	100	20	80%
2	100	53	47%
3	100	99	1%
4	100	100	0%
5	100	100	0%
6	100	100	0%
7	100	100	0%
8	100	100	0%
9	100	100	0%
10	100	100	0%

Table 5.11: Percentage of dropped packets when the total number of packets was equal to 100.

Additional tests were run where the delay was increased from 1 millisecond to 10 milliseconds and the total number of detected pulse packets that were sent to custom QGroundControl remained at 1000. Refer to Table 5.12 for the results of these additional tests.

Delay (ms)	Total number of packets sent	Total number of packets received	Percentage of packets that were dropped
0	1000	3	99.7%
1	1000	200	80%
2	1000	519	48.1%
3	1000	774	22.6%
4	1000	996	0.4%
5	1000	1000	0%
6	1000	1000	0%
7	1000	1000	0%
8	1000	1000	0%
9	1000	1000	0%
10	1000	1000	0%

Table 5.12: Percentage of dropped packets when the total number of packets was equal to 1000.

A final set of tests were run where the delay was increased from 1 millisecond to 10 milliseconds and the total number of detected pulse packets that were sent to custom QGroundControl remained at 1500. Refer to Table 5.13 for the results of these additional tests.

Delay (ms)	Total number of packets sent	Total number of packets received	Percentage of packets that were dropped
0	1500	4	99.73%
1	1500	309	79.4%
2	1500	781	47.93%
3	1500	1160	22.7%
4	1500	1492	0.53%
5	1500	1500	0%
6	1500	1500	0%
7	1500	1500	0%
8	1500	1500	0%
9	1500	1500	0%
10	1500	1500	0%

Table 5.13: Percentage of dropped packets when the total number of packets was equal to 1500.

Figure 5.11 shows the plotted results from Table 5.10, Table 5.11, Table 5.12, and Table 5.13.

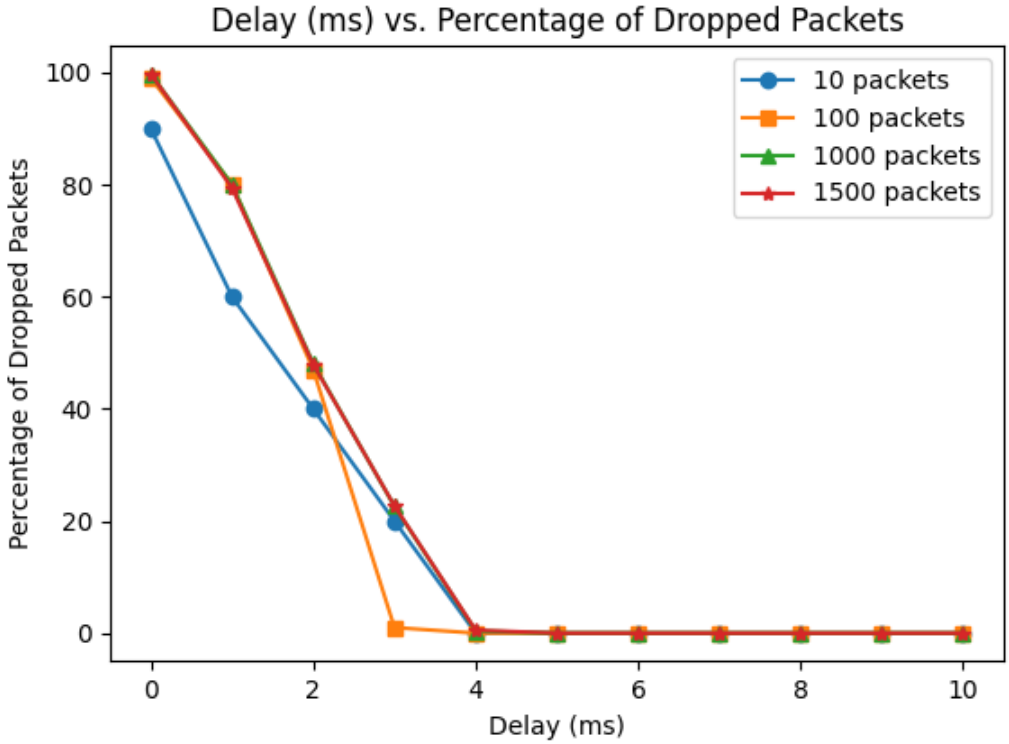

Figure 5.11: Plotted results for the data in Table 5.10, Table 5.11, Table 5.12, and Table 5.13.

Discussion

The data in Table 5.10, Table 5.11, Table 5.12, and Table 5.13 shows that, as the delay increases, the percentage of dropped packets decreases. The results plotted in Figure 5.11 indicate that packet losses begin to occur at Tunnel Protocol message transmission rates exceeding approximately 375 packets per millisecond.

It is important to note that the detected pulse data packets were instantiated and sent in rapid succession for the tests conducted in Section 5.4. In practice, detected pulse data packets would be sent from the UAV-RT software packages to custom QGroundControl at a slower rate. This is due to the amount of time signal processing takes within the channelizer and uavrt_detection package processes. The time that is required for the pulse detection algorithm to run in the uavrt_detection

package processes is also a factor.

A delay of 10 milliseconds was used and a total of 1486 detected pulse packets were sent during the test conducted in Section 5.3. With a delay of 10 milliseconds and a total of 1486 detected pulse packets, there should not have been dropped packets for the test conducted in Section 5.3. With this information and the results in Section 5.4, it is not clear where detected pulse data packets are being dropped when transmitted from the UAV-RT software packages to custom QGroundControl.

Referring to both Section 5.3 and Section 5.4, an answer as to where packets are being dropped could be found by editing the Pixhawk firmware so that packet information is printed out to the terminal when packets are transmitted from the UAV-RT software packages to custom QGroundControl. The tests described in Section 5.4 and Section 5.3 could be conducted again using the edited Pixhawk firmware. This would provide additional insight into how Tunnel Protocol messages are being transmitted using the Pixhawk firmware and whether the Pixhawk component of the UAV-RT system is responsible for packets being dropped.

5.5 Rotation test

The goal of this test was to ensure the accuracy of the telemetry data gathered with the UAV-RT software system. The results were collected with the UAV-RT system stationary on a cart. To ensure an accurate testing setup, the UAV was included in the testing procedure.

Testing procedure

The test took place on April 6th, 2023, from 17:46:19 MST GMT -7 to 17:47:35 MST GMT -7. The test site was located at E Pine Knoll Dr, Flagstaff, AZ 86001, in a section of the Northern Arizona University South Recreation Fields. The latitude and

longitude of the UAV-RT system was 35°10'29.5"N and 111°39'26.5"W, respectively, while the latitude and longitude of the tag was 35°10'29.5"N and 111°39'26.5"W, respectively. Figure 5.12 displays an image of the UAV-RT system at the testing location, while Figure 5.13 displays an image of the tag at the testing location. Figure 5.14 displays an image of the distance between the UAV-RT system and tag. Figure 5.15 provides an GPS map view of the rotation test location, where the red circle represents the location of the tag and the blue circle represents the location of the UAV-RT system.

Figure 5.12: UAV-RT system used for the rotation test.

Figure 5.13: Tag used for the rotation test.

Figure 5.14: Distance between components.

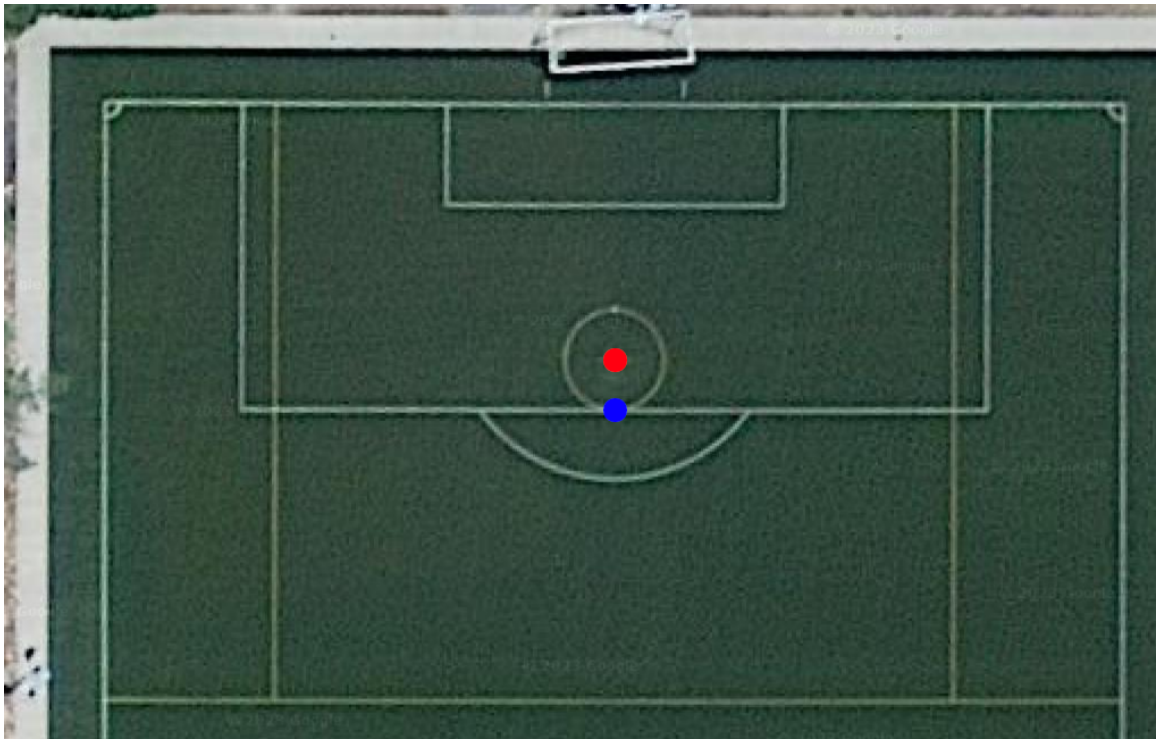

Figure 5.15: GPS map view of the rotation test location.

The UAV-RT system initially pointed towards the tag at a magnetic heading of 351° North. The sensors for the flight controller and the GPS module were calibrated using custom QGroundControl running on the GCS. Subsequently, the uavrt_supervisor and uavrt_connection packages were started on the companion computer. The tag information was transmitted from custom QGroundControl to the UAV-RT software system, with the action confirmed by reviewing the terminal output of the uavrt_supervisor package. A "Start" command was then transmitted from custom QGroundControl to the UAV-RT software system, which was again confirmed through terminal output checks. It was also confirmed that the tag was being properly detected by a uavrt_detection package subprocess.

Once the system was confirmed to be functioning properly, the field monitor was disconnected from the UAV-RT system hardware, and the cart holding the UAV-RT system was rotated for 1 minute. After completing a full 360° rotation, the cart was set

down, and a "Stop" command was transmitted from custom QGroundControl to the UAV-RT software system. This action was confirmed by reconnecting the field monitor and reviewing the terminal output of the uavrt_supervisor and uavrt_connection packages.

The "pulse_pose_log.txt" file was then moved from the following directory on the companion computer to an external flash drive:

```
/home/dasl/uavrt_workspace/uavrt_source/log/
    flight_log_2023-04-06_10-46-01/detector_log/tag_id_2/output
```

The contents of the "pulse_pose_log.txt" file were processed and plotted using a Python script. Figure 5.16 depicts a plot of the yaw values in degrees versus the time values in seconds, with the red line representing the magnetic heading of the tag when facing towards its location. Figure 5.17 shows a plot of the yaw values in degrees versus the SNR values (dimensionless).

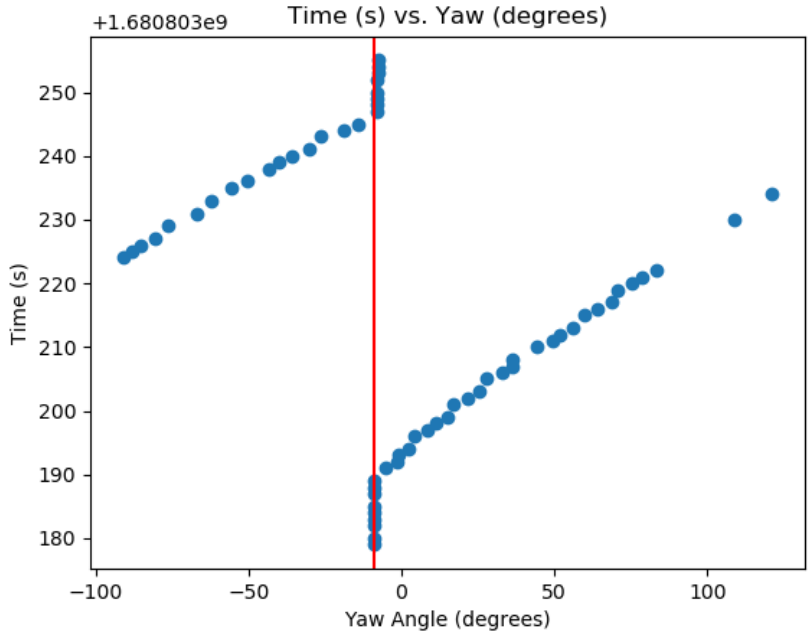

Figure 5.16: 2D plot of SNR vs. Yaw (degrees).

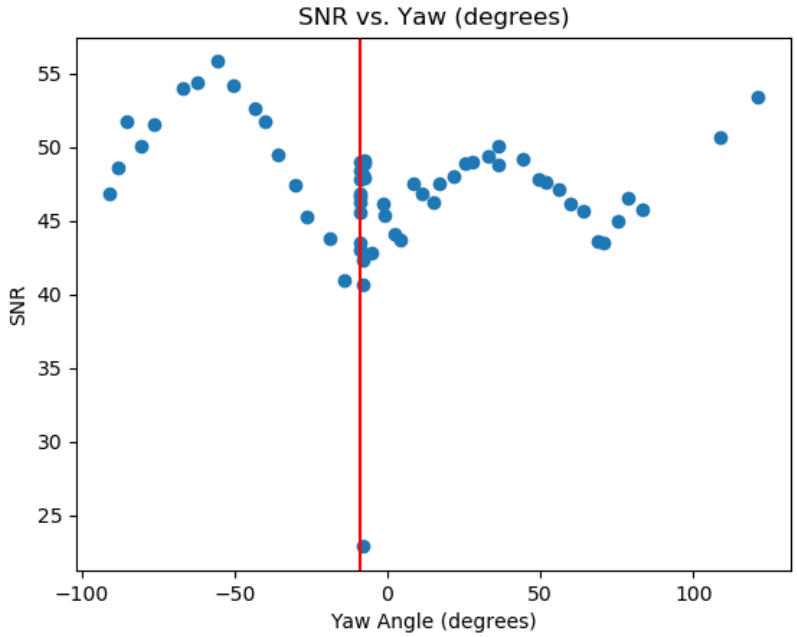

Figure 5.17: 2D plot of Time (s) vs. Yaw (degrees).

Discussion

Figure 5.16 successfully illustrates the rotation of the UAV-RT system during a one-minute interval. The Yaw values align with the initial magnetic heading of the tag, subsequently increasing and decreasing as the UAV-RT system rotates, and ultimately returning to match the magnetic heading as the test concludes.

Figure 5.17 does not exhibit the expected correlation. The highest SNR values were expected to align with the magnetic heading of the tag, as the signal strength would be greatest when the UAV-RT system faced towards the tag, due to the gain pattern of the attached VHF directional antenna. This gain pattern is illustrated by Figure 5.17. The second maximum should have corresponded to the UAV-RT system facing the exact opposite direction of the tag. Although the correct pattern is visible in Figure 5.17, it is shifted by approximately 50°. Additional testing was undertaken to verify that this discrepancy was not due to the incorrect conversion of quaternion

data to Euler angles. Further testing was also conducted to confirm that the issue did not stem from incorrectly plotting the data. Both rounds of testing affirmed the accuracy of data conversion and plotting.

A handheld radio attached to a monopole antenna was used to confirm signal reflection was occurring at the testing site. It is possible that the unexpected results are due to signal reflection caused by nearby buildings. Additional field testing, conducted at a greater distance away from buildings, would be necessary to validate the accuracy of the UAV-RT software system. This additional field testing, including the use of the UAV and the collection of results while not on the ground, should eliminate potential interference encountered at ground level. Conducting field tests with the UAV in flight would provide a more accurate representation of how the UAV-RT system is intended to be used.

It should be noted that the handheld radio also showed increased signal strength when the VHF antenna was perpendicular to the ground at this testing site. Future ground field tests should take this result into account.

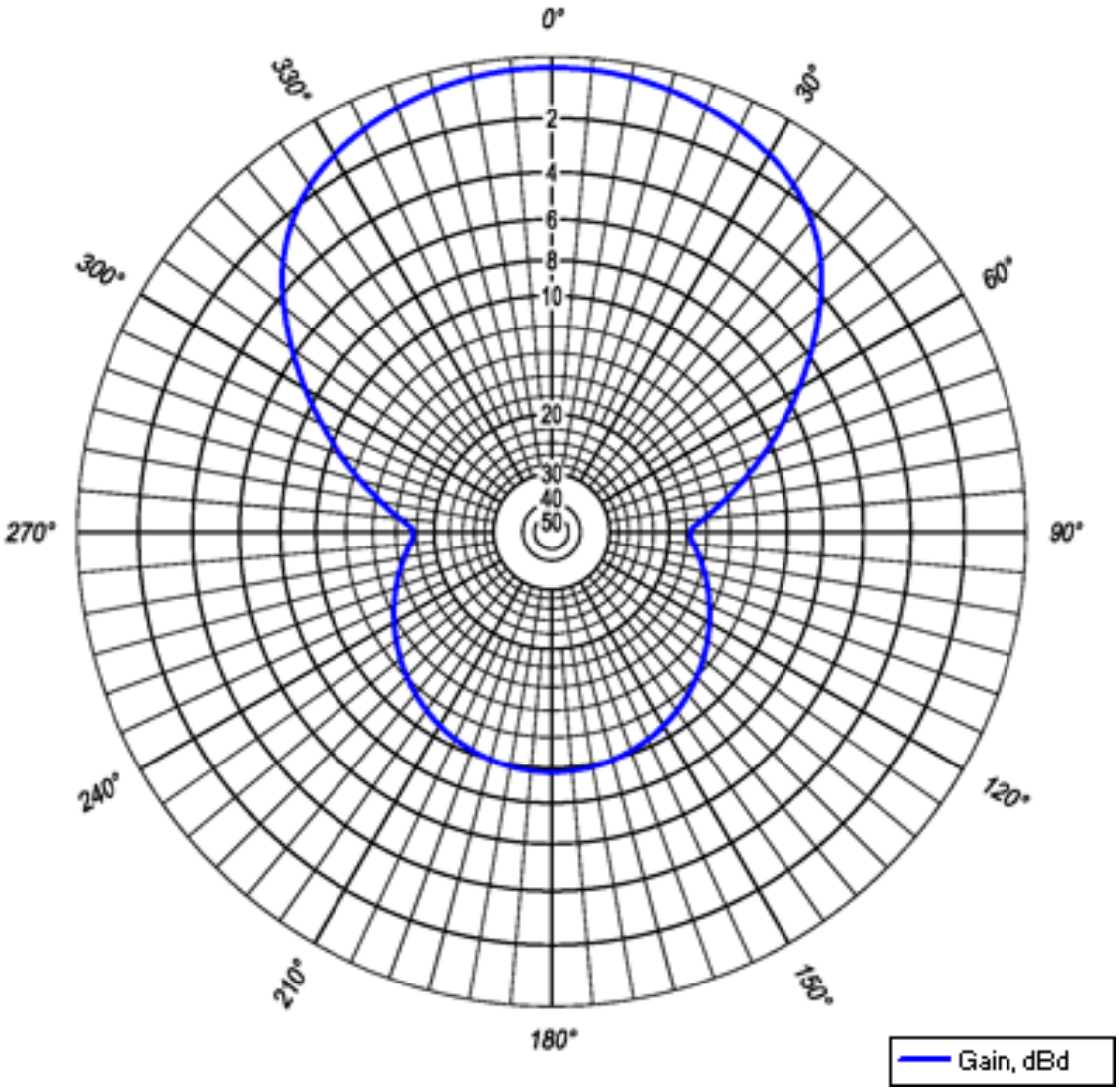

Figure 5.18: Gain pattern of the RA-23K VHF Directional Antenna [106].

Chapter 6

CONCLUSION

6.1 Summary

This thesis has presented a distributed software architecture designed to handle message transmission and reception, subprocess management, and the processing of pulse and telemetry data for an unmanned aerial vehicle. The architecture leverages ROS 2 for interprocess communication, MATLAB Coder for code generation within certain packages, Custom QGroundControl, developed by Don Gagne, for enhanced ground control station functionality, MAVLink as a communication protocol, the C++ MAVSDK library for interfacing with the flight controller, the Airspy library and airspy_rx tool for collecting IQ data, and csdr and Netcat for signal processing and data transmission.

The Unmanned Aerial Vehicle - Radio Telemetry software packages were implemented to form the architecture's backbone, ensuring efficient real-time data management, processing, and transmission. The UAV-RT software package described are as follows:

- The uavrt_supervisor package serves as the central hub, orchestrating data flow, subprocess control, and communication while generating essential logging and configuration files.

- The uavrt_connection package manages communication with the PX4 flight controller and Custom QGroundControl, ensuring accurate telemetry information through interpolation.

- The uavrt_interfaces package defines essential message structures.
- The channelizer, implemented in MATLAB and developed by Dr. Paul Flikkema and Dr. Michael Shafer, reduces sample rates for efficient data management.
- The uavrt_detection package, rooted in the research of Dr. Paul Flikkema and Dr. Michael Shafer and developed by Dr. Michael Shafer, serves as the core component for real-time pulse detection.

The project involved the use of custom software tools, adaptation of existing protocols, and the development of software packages tailored to the specific requirements of the system. It demonstrated the ability to process real-time data from wildlife tags, enabling the tracking of animal movements. This work represents a step forward in the development of technology for wildlife monitoring and environmental research, showcasing the feasibility and advantages of UAV-based wildlife tracking.

6.2 Future work

This section outlines potential future tasks and developments within the scope of this iteration of the project.

File transfer protocol (FTP)

To access logs and flight data stored on the companion computer, users need to insert a flash drive into the companion computer and then transfer the files from the companion computer's disk to the flash drive. The instructions for this process can be found in Section 4.5.

An enhanced approach for data transfer involves integrating MAVLink FTP [37] via MAVSDK [39]. This would enable specific files to be downloaded to the GCS using the telemetry radio. This method does not require an active internet connection, but

it may not be suitable for transferring large files due to the limited bandwidth of the telemetry radio utilized in this project.

Account for multirate tags

The uavrt_supervisor package operates under the assumption that each tag corresponds to a single rate for an animal. This assumption does not hold true when tracking with an animal that frequently switches between resting and traveling states. In that scenario, the animal would need to be tracked at two different rates.

Forwarding status messages to custom QGroundControl

The uavrt_supervisor and uavrt_connection packages utilize the ROS 2 logging module in Python and the C++ library to log status messages to both a terminal window and text files. These messages are also published on the ROS 2 network. However, there's currently no subscription mechanism in place to transmit these published messages to custom QGroundControl for real-time user visibility. The status messages serve the purpose of conveying information about both errors that have occurred and messages that inform the user of successful actions.

Prevent state lock

After launching the uavrt_supervisor and uavrt_connection packages on the companion computer and starting custom QGroundControl on the GCS, it becomes possible for the user to issue "Start" and "Stop" commands from custom QGroundControl to the UAV-RT packages before sending the tag information. This action can lead to the uavrt_supervisor and uavrt_connection packages getting stuck in an inoperative state, necessitating a reset for both packages. To prevent this, "locks" should be implemented within both packages to ensure that the state of each package can

only transition when the appropriate criteria are met. For example, in the event that the user issues a "Start" command before they have issued a "Tag" command, the uavrt_connection package should verify that a lock variable has been set to "True". This lock variable would signify that the "Tag" message has already been issued and the tag information has been processed. If the lock variable is "False," then the uavrt_connection package would log a warning and send a status message to custom QGroundControl.

Automatically starting the UAV-RT system

The Linux "crontab" utility [29] can be used to automate the startup of the uavrt_supervisor and uavrt_connection packages upon the companion computer's startup. Crontab allows the user to schedule script execution at specific times or in response to particular events. By creating a shell script and configuring crontab, the user can initiate the uavrt_supervisor and uavrt_connection packages when they log into their account on the companion computer. This ensures that these packages are launched without manual intervention.

Video demonstration

A video demonstrating the process that was explained in Section 4.5 would be beneficial for users that want to have a recorded visual aid and auditory step-by-step guidance to follow.

Automatic installation script

Simplifying the installation of dependencies needed to run the UAV-RT software system on the companion computer can be achieved by using a tool like GNU Automake [25]. The individual UAV-RT packages can be installed using a bash script, as GNU

Automake does not support package cloning. This bash script can also take care of installing, configuring, and building the ROS 2 workspace, reducing the timed needed for troubleshooting.

Saving output from subprocesses

The logging structure for the UAV-RT project has been implemented, but the logs from channelizer and airspy_csdr_netcat subprocesses are not being saved. The information that is output by these subprocesses can be useful for debugging purposes.

Testing with a Raspberry Pi

The uavrt_supervisor and uavrt_connection packages were initially developed and tested on the UDOO X86 II Ultra platform. Thanks to the contributions of Don Gagne, it is theoretically possible to deploy these packages on a Raspberry Pi. However, further development and testing are required to verify and ensure compatibility with the Raspberry Pi platform.

BIBLIOGRAPHY

[1] *airspyone_host*. https://github.com/airspy/airspyone_host. Accessed: September 2023.

[2] *airspy_rx.c*. https://github.com/airspy/airspyone_host/blob/master/airspy-tools/src/airspy_rx.c, 2018. Accessed: October 2023.

[3] Matthew B. Amato-Yarbrough. *UAV-RT Connection package*. https://github.com/dynamic-and-active-systems-lab/uavrt_connection, 2023. Accessed: October 2023.

[4] Matthew B. Amato-Yarbrough. *UAV-RT Interfaces package*. https://github.com/dynamic-and-active-systems-lab/uavrt_interfaces, 2023. Accessed: October 2023.

[5] Matthew B. Amato-Yarbrough. *UAV-RT Source directory*. https://github.com/dynamic-and-active-systems-lab/uavrt_source/tree/main, 2023. Accessed: October 2023.

[6] Matthew B. Amato-Yarbrough. *UAV-RT Supervisor package*. https://github.com/dynamic-and-active-systems-lab/uavrt_supervisor, 2023. Accessed: October 2023.

[7] *boost/qvm/quat.hpp*. https://www.boost.org/doc/libs/1_65_1/libs/qvm/doc/boost_qvm_quat_hpp.html. Accessed: September 2023.

[8] *boost/qvm/quat_operations.hpp*. https://www.boost.org/doc/libs/1_66_0/libs/qvm/doc/boost_qvm_quat_operations_hpp.html. Accessed: September 2023.

[9] Oliver Cliff, Robert Fitch, Salah Sukkarieh, Debbie Saunders, and Robert Heinsohn. Online localization of radio-tagged wildlife with an autonomous aerial robot system. In *Robotics Science and Systems Conference 2015, ed.*, pages 1–9, 07 2015.

[10] Thomas R. Consi, Joseph R. Patzer, Brady Moe, Samuel A. Bingham, and Kristopher Rockey. An unmanned aerial vehicle for localization of radio-tagged sturgeon: Design and first test results. In *OCEANS 2015 - MTS/IEEE Washington*, pages 1–10, 2015.

[11] *Standard library header vector*. https://en.cppreference.com/w/cpp/header/tuple, 2022. Accessed: September 2023.

[12] *Standard library header algorithm*. https://en.cppreference.com/w/cpp/header/algorithm, 2023. Accessed: September 2023.

[13] *Standard library header chrono*. https://en.cppreference.com/w/cpp/header/chrono, 2023. Accessed: August 2023.

[14] *Standard library header cmath*. https://en.cppreference.com/w/cpp/header/cmath, 2023. Accessed: September 2023.

[15] *Standard library header fstream*. https://en.cppreference.com/w/cpp/header/fstream, 2023. Accessed: September 2023.

[16] *Standard library header functional*. https://en.cppreference.com/w/cpp/header/functional, 2023. Accessed: September 2023.

[17] *Standard library header memory*. https://en.cppreference.com/w/cpp/header/memory, 2023. Accessed: August 2023.

[18] *Standard library header string*. https://en.cppreference.com/w/cpp/header/string, 2023. Accessed: September 2023.

[19] *Standard library header thread*. https://en.cppreference.com/w/cpp/header/thread, 2023. Accessed: September 2023.

[20] *Standard library header vector*. https://en.cppreference.com/w/cpp/header/vector, 2023. Accessed: September 2023.

[21] *Install Docker Engine on Ubuntu*. https://docs.docker.com/engine/install/ubuntu/. Accessed: August 2023.

[22] *FCC 97-114; Document Number: 97-11584*. https://transition.fcc.gov/Bureaus/Engineering_Technology/Orders/1997/fcc97114.txt, 1997. Accessed: November 2023.

[23] Don Gagne. *Custom QGroundControl*. https://github.com/DonLakeFlyer/UAV-RT-TagTracker, 2023. Accessed: October 2023.

[24] *Netcat*. https://netcat.sourceforge.net/, 2004. Accessed: October 2023.

[25] *Automake*. https://www.gnu.org/software/automake/, 2022. Accessed: October 2023.

[26] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, 6th edition, 2017.

[27] *Determining when to use simultaneous multithreading*. https://www.ibm.com/docs/en/i/7.5?topic=performance-determining-when-use-simultaneous-multithreading, 2023. Accessed: November 2023.

[28] Michael Kerrisk. *script - Linux manual page*. https://man7.org/linux/man-pages/man1/script.1.html. Accessed: November 2023.

[29] Michael Kerrisk. *crontab - Linux manual page*. https://man7.org/linux/man-pages/man5/crontab.5.html, 2012. Accessed: October 2023.

[30] Michael Kerrisk. *top — Linux manual page*. https://man7.org/linux/man-pages/man1/top.1.html, 2023. Accessed: November 2023.

[31] *pgrep — Linux manual page*. https://man7.org/linux/man-pages/man1/pgrep.1.html, 2023. Accessed: November 2023.

[32] *MATLAB Coder*. https://www.mathworks.com/products/matlab-coder.html. Accessed: October 2023.

[33] *dsp.Channelizer*. https://www.mathworks.com/help/dsp/ref/dsp.channelizer-system-object.html, 2023. Accessed: October 2023.

[34] *DEBUG_FLOAT_ARRAY*. https://mavlink.io/en/messages/common.html#DEBUG_FLOAT_ARRAY. Accessed: October 2023.

[35] *MAVLink Documentation*. https://mavlink.io/en/. Accessed: October 2023.

[36] *Tunnel Protocol*. https://mavlink.io/en/services/tunnel.html. Accessed: September 2023.

[37] *File Transfer Protocol (FTP)*. https://mavlink.io/en/services/ftp.html, 2023. Accessed: October 2023.

[38] *MAVSDK C++ Library*. https://mavsdk.mavlink.io/main/en/cpp/, 2023. Accessed: October 2023.

[39] *mavsdk::Ftp Class Reference*. https://mavsdk.mavlink.io/main/en/cpp/api_reference/classmavsdk_1_1_ftp.html, 2023. Accessed: October 2023.

[40] *mavsdk::MavlinkPassthrough Class Reference*. https://mavsdk.mavlink.io/main/en/cpp/api_reference/classmavsdk_1_1_mavlink_passthrough.html, 2023. Accessed: September 2023.

[41] *mavsdk::Mavsdk Class Reference*. https://mavsdk.mavlink.io/main/en/cpp/api_reference/classmavsdk_1_1_mavsdk.html, 2023. Accessed: August 2023.

[42] *mavsdk::System Class Reference*. https://mavsdk.mavlink.io/main/en/cpp/api_reference/classmavsdk_1_1_system.html, 2023. Accessed: August 2023.

[43] *mavsdk::Telemetry Class Reference*. https://mavsdk.mavlink.io/main/en/cpp/api_reference/classmavsdk_1_1_telemetry.html, 2023. Accessed: September 2023.

[44] *Position Struct Reference*. https://mavsdk.mavlink.io/main/en/cpp/api_reference/structmavsdk_1_1_telemetry_1_1_position.html#structmavsdk_1_1_telemetry_1_1_position_1, 2023. Accessed: October 2023.

[45] *Quaternion Struct Reference*. https://github.com/airspy/airspyone_host/blob/master/airspy-tools/src/airspy_rx.c, 2023. Accessed: October 2023.

[46] Hoa Van Nguyen, Michael Chesser, Lian Pin Koh, S. Hamid Rezatofighi, and Damith C. Ranasinghe. Trackerbots: Autonomous unmanned aerial vehicle for real-time localization and tracking of multiple radio-tagged animals. *Journal of Field Robotics*, 36(3):617–635, Jan 2019.

[47] *NumPy*. https://pypi.org/project/numpy/, 2023. Accessed: September 2023.

[48] *About Composition*. https://docs.ros.org/en/galactic/Concepts/About-Composition.html. Accessed: November 2023.

[49] *About logging and logger configuration*. https://docs.ros.org/en/galactic/Concepts/About-Logging.html. Accessed: November 2023.

[50] *About ROS 2 client libraries*. https://docs.ros.org/en/galactic/Concepts/About-ROS-2-Client-Libraries.html. Accessed: November 2023.

[51] *About ROS 2 interfaces*. https://docs.ros.org/en/galactic/Concepts/About-ROS-Interfaces.html. Accessed: November 2023.

[52] *Creating custom msg and srv files*. https://docs.ros.org/en/galactic/Tutorials/Beginner-Client-Libraries/Custom-ROS2-Interfaces.html. Accessed: November 2023.

[53] *Executors*. https://docs.ros.org/en/galactic/Concepts/About-Executors.html. Accessed: November 2023.

[54] *Overview and usage of RQt*. https://docs.ros.org/en/galactic/Concepts/About-RQt.html. Accessed: November 2023.

[55] *ROS 2 Documentation*. https://docs.ros.org/en/galactic/index.html. Accessed: November 2023.

[56] *Understanding ROS 2 Topics*. https://docs.ros.org/en/galactic/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html. Accessed: November 2023.

[57] *nodelet*. https://wiki.ros.org/nodelet, 2017. Accessed: November 2023.

[58] *Nodes*. https://wiki.ros.org/Nodes, 2018. Accessed: November 2023.

[59] *builtin_interfaces/msg/Time Message*. https://docs.ros2.org/galactic/api/builtin_interfaces/msg/Time.html, 2021. Accessed: November 2023.

[60] *diagnostic_msgs/msg/DiagnosticArray Message*. https://docs.ros2.org/galactic/api/diagnostic_msgs/msg/DiagnosticArray.html, 2021. Accessed: November 2023.

[61] *diagnostic_msgs/msg/DiagnosticStatus Message*. https://docs.ros2.org/galactic/api/diagnostic_msgs/msg/DiagnosticStatus.html, 2021. Accessed: November 2023.

[62] *diagnostic_msgs/msg/KeyValue Message*. https://docs.ros2.org/galactic/api/diagnostic_msgs/msg/KeyValue.html, 2021. Accessed: November 2023.

[63] *geometry_msgs/msg/Pose Message*. https://docs.ros2.org/galactic/api/geometry_msgs/msg/Pose.html, 2021. Accessed: November 2023.

[64] *geometry_msgs/msg/PoseStamped Message*. https://docs.ros2.org/galactic/api/geometry_msgs/msg/PoseStamped.html, 2021. Accessed: November 2023.

[65] *geometry_msgs/msg/Quaternion Message*. https://docs.ros2.org/galactic/api/geometry_msgs/msg/Point.html, 2021. Accessed: November 2023.

[66] *geometry_msgs/msg/Quaternion Message*. https://docs.ros2.org/galactic/api/geometry_msgs/msg/Quaternion.html, 2021. Accessed: November 2023.

[67] *std_msgs/msg/Bool Message*. https://docs.ros2.org/galactic/api/std_msgs/msg/Bool.html, 2021. Accessed: November 2023.

[68] *std_msgs/msg/Header Message*. https://docs.ros2.org/galactic/api/std_msgs/msg/Header.html, 2021. Accessed: November 2023.

[69] *Execution and Callbacks*. https://docs.ros2.org/galactic/api/rclpy/api/execution_and_callbacks.html. Accessed: September 2023.

[70] *Galactic Geochelone*. https://docs.ros.org/en/galactic/Releases/Release-Galactic-Geochelone.html. Accessed: September 2023.

[71] *Galactic Geochelone Examples*. https://docs.ros.org/en/galactic/Installation/Ubuntu-Install-Debians.html#id7. Accessed: September 2023.

[72] *Galactic Geochelone Installation*. https://docs.ros.org/en/galactic/Installation/Ubuntu-Install-Debians.html. Accessed: September 2023.

[73] *Logging*. https://docs.ros2.org/galactic/api/rclpy/api/logging.html. Accessed: September 2023.

[74] *Node*. https://docs.ros2.org/galactic/api/rclpy/api/node.html. Accessed: September 2023.

[75] *Timer*. https://docs.ros2.org/galactic/api/rclpy/api/timers.html. Accessed: September 2023.

[76] *Topics*. https://docs.ros2.org/galactic/api/rclpy/api/topics.html. Accessed: September 2023.

[77] *geometry_msgs/msg/Point Message*. https://docs.ros2.org/latest/api/geometry_msgs/msg/Point.html, 2020. Accessed: September 2023.

[78] *geometry_msgs/msg/Pose Message*. https://docs.ros2.org/latest/api/geometry_msgs/msg/Pose.html, 2020. Accessed: September 2023.

[79] *geometry_msgs/msg/Pose Stamped Message*. https://docs.ros2.org/latest/api/geometry_msgs/msg/PointStamped.html, 2020. Accessed: September 2023.

[80] *geometry_msgs/msg/Quaternion Message*. https://docs.ros2.org/latest/api/geometry_msgs/msg/Quaternion.html, 2020. Accessed: September 2023.

[81] *std_msgs/msg/Bool.msg*. https://docs.ros2.org/latest/api/std_msgs/msg/Bool.html, 2020. Accessed: September 2023.

[82] *std_msgs/msg/Header Message*. https://docs.ros2.org/latest/api/std_msgs/msg/Header.html, 2020. Accessed: September 2023.

[83] *diagnostic_msgs/DiagnosticArray.msg*. http://docs.ros.org/en/latest/api/diagnostic_msgs/html/msg/DiagnosticArray.html, 2022. Accessed: September 2023.

[84] *diagnostic_msgs/DiagnosticStatus.msg*. http://docs.ros.org/en/latest/api/diagnostic_msgs/html/msg/DiagnosticStatus.html, 2022. Accessed: September 2023.

[85] *diagnostic_msgs/KeyValue.msg*. http://docs.ros.org/en/latest/api/diagnostic_msgs/html/msg/KeyValue.html, 2022. Accessed: September 2023.

[86] *diagnostic_msgs*. http://wiki.ros.org/diagnostic_msgs, 2023. Accessed: August 2023.

[87] *rclcpp.hpp File Reference*. https://docs.ros2.org/latest/api/rclcpp/rclcpp_8hpp.html, 2023. Accessed: August 2023.

[88] *rclpy*. https://docs.ros2.org/latest/api/rclpy/index.html, 2023. Accessed: August 2023.

[89] *3DR Pixhawk 1 Flight Controller (Discontinued)*. https://docs.px4.io/main/en/flight_controller/pixhawk.html, 2023. Accessed: November 2023.

[90] *Loading Firmware*. https://docs.px4.io/main/en/config/firmware.html, 2023. Accessed: September 2023.

[91] *PX4-Gazebo simulator headless*. https://github.com/JonasVautherin/px4-gazebo-headless, 2023. Accessed: August 2023.

[92] *pathlib — Object-oriented filesystem paths*. https://docs.python.org/3/library/pathlib.html, 2023. Accessed: September 2023.

[93] *time — Time access and conversions*. https://docs.python.org/3/library/time.html, 2023. Accessed: September 2023.

[94] *os — Miscellaneous operating system interfaces*. https://docs.python.org/3/library/os.html, 2023. Accessed: August 2023.

[95] *signal — Set handlers for asynchronous events*. https://docs.python.org/3/library/signal.html, 2023. Accessed: August 2023.

[96] *subprocess — Subprocess management*. https://docs.python.org/3/library/subprocess.html, 2023. Accessed: August 2023.

[97] *QGroundControl*. http://qgroundcontrol.com/. Accessed: October 2023.

[98] *Sensor Setup (PX4)*. https://docs.qgroundcontrol.com/master/en/SetupView/sensors_px4.html, 2023. Accessed: September 2023.

[99] András Retzler. *csdr*. https://github.com/ha7ilm/csdr, 2019. Accessed: October 2023.

[100] András Retzler. *fir_decimate_cc*. https://github.com/ha7ilm/csdr#fir_decimate_cc, 2019. Accessed: October 2023.

[101] Gilberto Antonio Marcon dos Santos, Zachary Barnes, Eric Lo, Bryan Ritoper, Lauren Nishizaki, Xavier Tejeda, Alex Ke, Han Lin, Curt Schurgers, Albert Lin, and Ryan Kastner. Small unmanned aerial vehicle system for wildlife radio collar tracking. In *2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems*, pages 761–766, 2014.

[102] Michael W. Shafer. *UAV-RT Detection package*. https://github.com/dynamic-and-active-systems-lab/airspyhf_channelize, 2023. Accessed: October 2023.

[103] Michael W. Shafer and Paul G. Flikkema. *Channelizer Package*. https://github.com/dynamic-and-active-systems-lab/uavrt_detection, 2023. Accessed: October 2023.

[104] Michael W. Shafer and Paul G. Flikkema. Tracking small wildlife with minimal-complexity radio frequency transmitters: Near-optimal detection. *IEEE Access*, 11:40029–40037, 2023.

[105] Michael W. Shafer, Gabriel Vega, Kellan Rothfus, and Paul Flikkema. Uav wildlife radiotelemetry: System and methods of localization. *Methods in Ecology and Evolution*, 10(10):1783–1795, 2019.

[106] *Gain pattern - RA-23K VHF Directional Antenna*. https://www.telonics.com/images/products/antennas/gain_ra23.png. Accessed: November 2023.

[107] *The Vehicle*. https://uavrt.nau.edu/index.php/docs/vehicle/, 2019. Accessed: August 2023.

[108] *Install Ubuntu desktop*. https://ubuntu.com/tutorials/install-ubuntu-desktop#1-overview. Accessed: August 2023.

[109] Kurt VonEhr, Seth Hilaski, Bruce E. Dunne, and Jeffrey Ward. Software defined radio for direction-finding in uav wildlife tracking. In *2016 IEEE International Conference on Electro Information Technology (EIT)*, pages 0464–0469, 2016.