Name:	**Timothy Molloy**
Supervisor:	**Dr Luis Mejias Alvarez**
Project Title:	**Autonomous Helicopter Navigation System – Ground Control Station and Flight Control Subsystems**
Year:	**2010**
Volume:	**1 of 1**

This project report was submitted as part of the requirements for the award of the

BACHELOR OF ENGINEERING (Aerospace Avionics)

in the

SCHOOL OF ENGINEERING SYSTEMS
FACULTY OF BUILT ENVIRONMENT AND ENGINEERING

at the

QUEENSLAND UNIVERSITY OF TECHNOLOGY
BRISBANE, AUSTRALIA

Signature of Candidate

……………………………………………………

“ ‘Reeling and Writhing, of course, to begin with,’ the Mock Turtle replied; ‘and then the difference branches of Arithmetic— Ambition, Distraction, Uglification, and Derision.’ ”

- Lewis Carrol

Alice’s Adventures in Wonderland

Executive Summary

The 2010 Autonomous Helicopter Navigation System (AHNS) was an undergraduate fourth year thesis project at the Queensland University of Technology (QUT). It constituted the development of a quadrotor system capable of autonomous control, navigation and localisation within a GPS denied environment. A systems engineering methodology was applied to divide the system into seven subsystems for allocation to the four project members; Michael Hamilton, Michael Kincel, Liam O'Sullivan and Tim Molloy. Their respective responsibilities were project management; platform and pilot; localisation and state estimation; and ground control station and flight control.

The systems engineering methodology as applied to the ground control station (GCS) and flight control subsystems resulted in the establishment of subsystem designs. The purpose of the GCS was seen to originate from the project's fifth high level objective (HLO). In a systems context it provides an interface for data monitoring and flight control. Three requirements from the customer, treated as baseline, specified WiFi link communications and control of the aircraft flight mode whilst four derived system requirements specified data handling and display. The requirements were used to in design to ensure each GCS component had a specific rationale.

The flight control system's high level objective required the design of an autopilot to enable stability augmented flight and autonomous position hold. This objective was specified in three system requirements that established the design architecture as a cascaded PID design consisting of separate attitude and position controllers. The control inputs of the platform were established, enabling design of three PID attitude control loops. The loop designs and implementations were iteratively explored. A similar method was followed for position control.

Following the design stage, component testing was undertaken. For the GCS this constituted acceptance testing whilst flight control could not be tested without hardware and software system integration. In the subsequent integration and testing stages the altitude controller designs were iterated extensively with the final result being a controllable quadrotor platform. Due to perceived risks to the platform and personnel in the flight-testing of position control, only altitude hold was attempted and completed.

In summary, in the ground control station and flight control subsystems, all but one system requirement was completed. As a testament to methodology and engineering process and tools used, the GCS high level objective was completed whilst the flight control objective was partially completed.

Statement of Authorship

The work contained in this project report has not been previously submitted for a degree or diploma at any other tertiary educational institution. To the best of my knowledge and belief, the project report contains no material previously published or written by another person except where due reference is made.

Signed ……………………..……………………..

Date ……………………..

Acknowledgements

I am grateful to Dr Luis Mejias for offering the AHNS project again in 2010 and letting us explore engineering solutions; even those that had the potential to lead us astray. I would also like to thank all those academics who had the patience during lectures, tutorials and meetings to listen to the latest project ramblings and offer some perspective.

To my aerospace avionics colleagues, perhaps the lab will be a more productive work area now we are finishing! I wish to express my deepest gratitude to the gentlemen of AHNS 2010. Without the three of you, I doubt fourth year would be worth remembering. Together you taught me the value of teamwork but I am most thankful and proud of the friendships we developed.

I wish to finally thank my family for their support and for tolerating my vices and virtues.

Table of Contents

List of Tables

List of Figures

List of Code Segments

Definitions

AHNS	Autonomous Helicopter Navigation System
AVR	A popular series of microcontrollers
CPU	Central Processing Unit
GUI	Graphical User Interface
HLO	High Level Objective
HMI	Human-machine interface
ISP	In system programming – a serial programming standard
ISR	Interrupt Service Routine
OCR	Output Compare Register (as in an AVR microcontroller)
PC	Personal computer
PID	Proportional, Integral, Derivative, as in the PID controller
QUT	Queensland University of Technology
RC	Radio Control
SVN	Subversion – a version control system
SSH	Secure Shell
UAV	Uninhabited Aerial Vehicle
UDP	User Datagram Protocol
USART	Universal Synchronous/Asynchronous Receiver/Transmitter

Chapter 1 Project Introduction

The Autonomous Helicopter Navigation System (AHNS) project is an undergraduate fourth year thesis project at the Queensland University of Technology (QUT). It constitutes the development of a helicopter system capable of autonomous control, navigation and localisation within a GPS denied environment. In 2010, four students undertook it as a systems engineering project in partial completion of the Bachelor of Engineering (Aerospace Avionics). The four students were Michael Hamilton, Michael Kincel, Liam O'Sullivan and the author (Tim Molloy). Using systems engineering nomenclature, the system comprised of seven subsystems, which were distributed as roles to each member as shown in Figure 1.1.

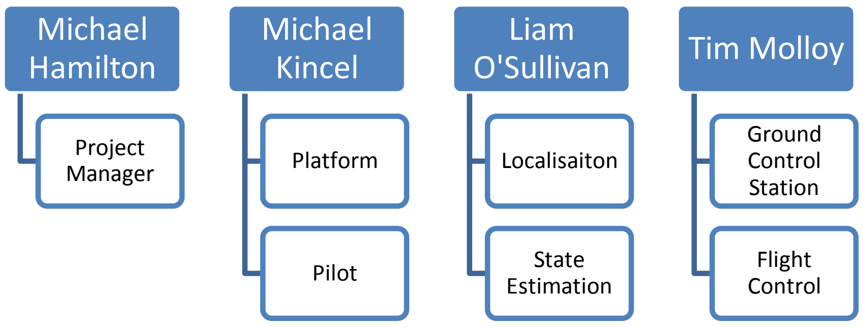


Figure 1.1 - Group Members and Roles

1.1 Background

The development of indoor autonomous helicopter platforms, specifically quadrotor platforms, is an active area of research in the fields of robotics and aerospace. Research interest in their use as autonomous agents is driving industry interest in their capabilities and applications [1, 2, 3, 4]. Quadrotor platforms have found application in visual servoing developments [5], non-linear control [2, 6], modelling for control [2], swarming [1] and simultaneous localisation and mapping [4].

By considering the autonomous platforms in a more general sense as uninhabited aerial vehicles (UAVs) their applications become less abstract. Industry applications include infrastructure inspection, agriculture and search and rescue. AHNS therefore reflects the current research interests of the Australian Research Centre for Aerospace Automation (ARCAA). It also provides the opportunity for undergraduates to apply and develop the skills of systems engineering and systems design whilst contributing to an ongoing project focused on avionics development.

1.2 Autonomous Helicopter Navigation System 2010

1.2.1 Aims and Objectives

The aims and objectives of AHNS for 2010 are summarised in six high level objectives. These were developed and agreed upon in consolation with the project supervisor (or customer). Particular objectives of note include the development of control and state estimation without reliance on previous work; development of a ground control station that aims to combine the functionality of previous AHNS Human-Machine Interfaces and autopilot ground stations; the freedom to choose a quadrotor platform instead of a conventional RC helicopter; and the use of WiFi communications.

1.2.1.1 HLO-1 Platform

A platform should be developed and maintained to facilitate flight and on board hardware integration.

1.2.1.2 HLO-2 Localisation

The system should be capable of determining its position with the aid of image processing within an indoor environment to an appropriate time resolution.

1.2.1.3 HLO-3 State Estimation

A method of estimating the states of the helicopter system should be designed and implemented. The resolution of the estimations should facilitate their employment in the control system design.

1.2.1.4 HLO-4 Autonomous Hovering Flight

An autopilot system should be developed to enable sustained indoor autonomous hovering flight. The control system should be designed to enable future ingress and egress manoeuvre to longitudinal and hovering flight.

1.2.1.5 HLO-5 Ground Control Station

A ground control station (GCS) that supports appropriate command and system setting inputs and data display and logging should be developed. The design should be derived from previous AHNS developments and enable future ground station developments.

1.2.1.6 HLO-6 Communications

The communications system should enable transfer of control, state and localisation data to the ground control station. It should provide with a flexible wireless data link available on consumer-electronic devices.

1.2.2 System Architecture

The system architecture can be considered a graphic depiction of the project aims. The electronic hardware architecture of Figure 1.2 demonstrates the accommodation of a variety of sensors. For state estimation, the sensors include the 6 degree-of-freedom inertial measurement unit (IMU) and the magnetic compass. The aim to control position leads to localisation or position sensors such as Vicon, the Camera and an Ultrasonic altimeter. Finally, to control the platform with state estimation the four quadrotor electronic speed controllers, the mode control unit, the radio control receiver, the flight computer and the ground control station are all required.

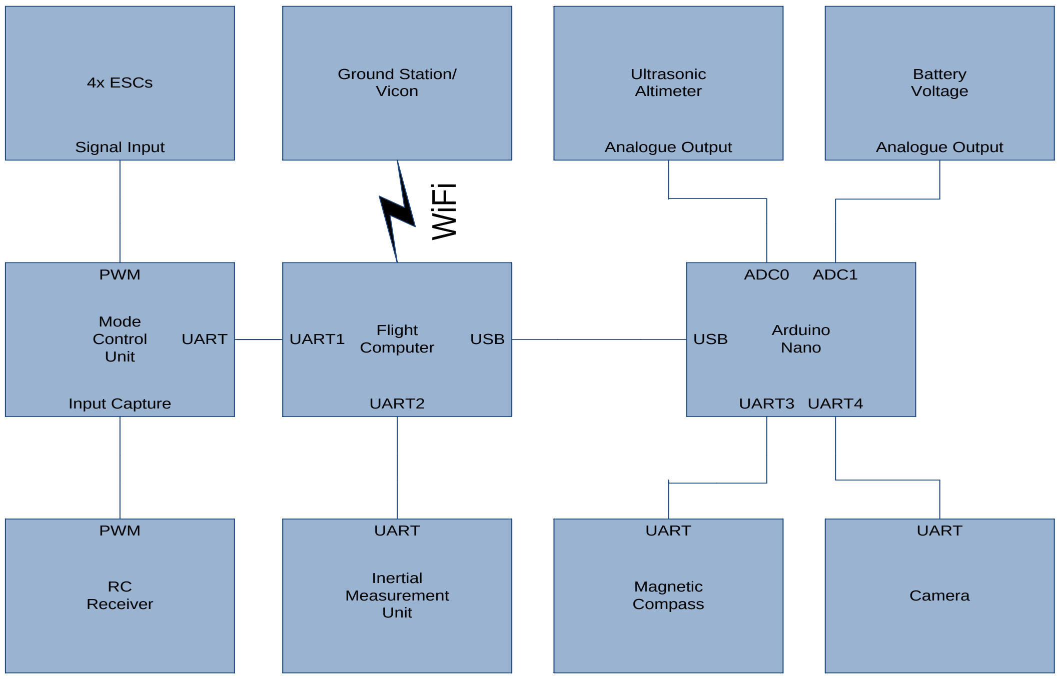


Figure 1.2–Hardware System Architecture

As the project aimed to develop the on board avionics from an array of hardware components the software architecture was designed in modules to accommodate state estimation, localisation and control. Figure 1.3 thus compliments Figure 1.2 in describing the AHNS 2010 overall system architecture.

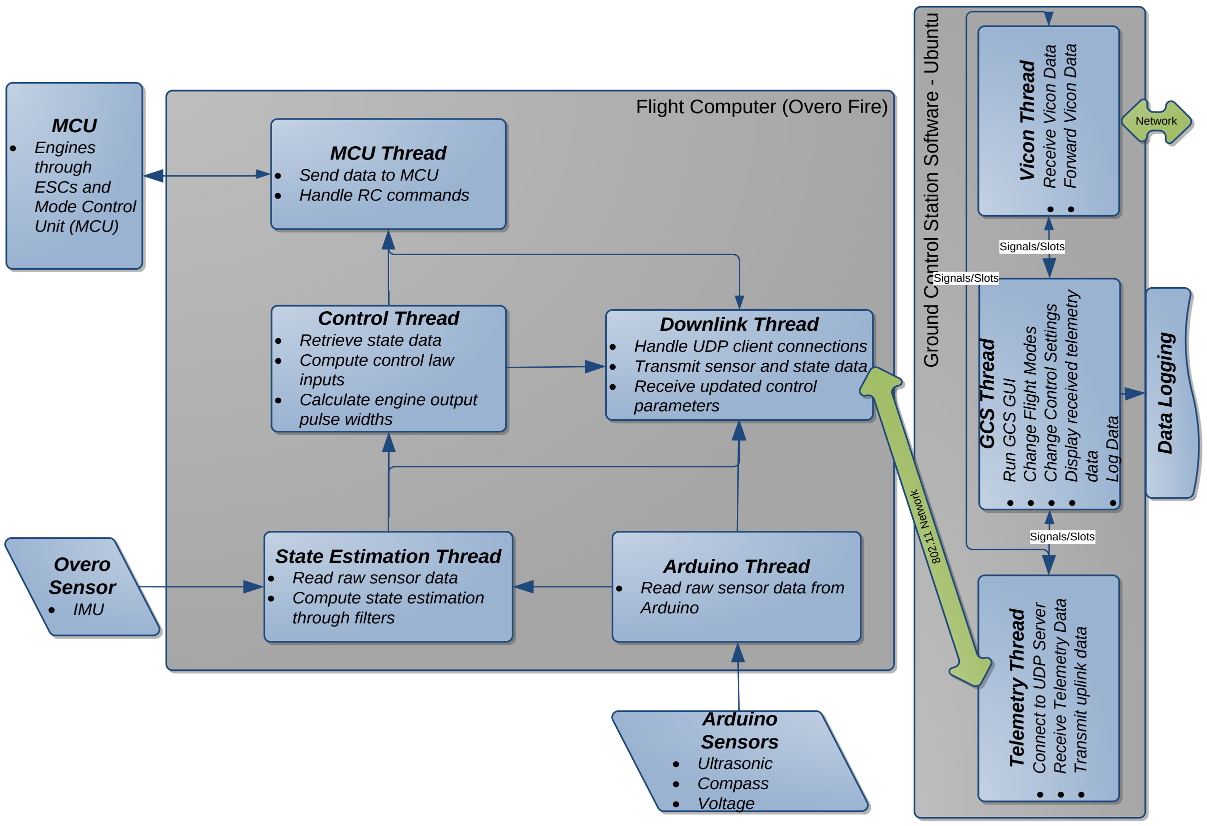


Figure 1.3 - Combined GCS, Flight Computer and MCU Software Architecture

1.3 Thesis Aim and Outline

This thesis presents the systems engineering based processes followed by the author and the AHNS team in order to ensure system development and delivery on the high level objectives. Importantly it also presents the results of this process, including what was developed, what was tested, what worked, what did not work and what was learnt. The thesis does not aim to provide a holistic appreciation of the AHNS project; it describes the project as it related to the author's subsystems of ground control station and flight control.

Chapter Two describes the project management processes followed to ensure project delivery. It discusses the systems engineering approaches used for AHNS project management, subsystem division and work break down.

Chapter Three describes the subsystem design and development stages of the Ground Control Station and Flight Control subsystems. It includes both preliminary designs and final designs with accompanying discussions on the iterations required during the project.

Chapter Four is important in judging the capabilities of the developed system. Through acceptance testing and system integration, the subsystems of AHNS are shaped into the final project deliverables.

Chapter Five completes the formal systems engineering treatment of the project by presenting a conformance matrix for the Ground Control Station and Flight Control subsystems.

Chapter Six presents the project conclusions and project recommendations as they related to the problems of ground control station and flight control development. Technical and generic lessons learnt are collated to reflect the author's views on how the project was executed.

Chapter 2 Project Management

2.1 Systems Engineering Methodology

The engineering methodology followed at all stages of project development was of utmost importance to the AHNS team. At project inception, the project tasks were grouped into seven subsystems, which could then be assigned to individual project members. The author was assigned the GCS and Flight Control subsystems. Following this the five stages of system engineering shown in Figure 2.1 were applied recursively at the system and subsystem levels.

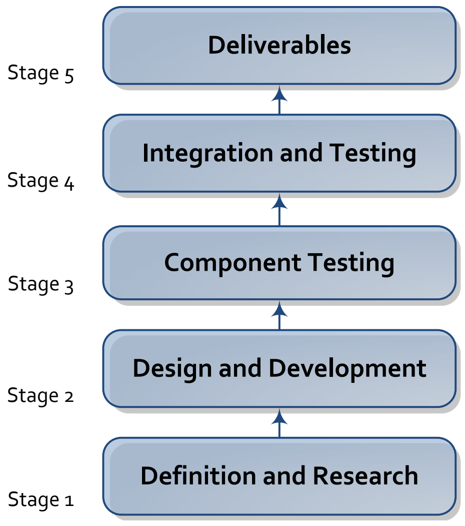


Figure 2.1 - Five Stages of Systems Engineering Project

Stage 1 dictated that system requirements (SRs) be defined first for each subsystem to ensure the validity of the deliverables completed and to establish agreeable acceptance tests for the final deliverables. These system requirements were made traceable to the GCS and Flight Control high level objectives. The stage also involved trade study completion.

It should be noted that in previous AHNS projects only a fraction of the aims established at this stage were completed. Within QUT avionics, following the systems engineering approach to projects often results in this level of project completion. In itself this is not the fault of system engineering, but the over estimation of available time and underestimation of risk on the part of project groups. To reduce the amount of risk-exposed work, AHNS 2010 took considerable time to identify that work which was critically important to completing the HLO’s and compared this to the work left uncompleted by previous projects. A key outcome was the adoption of several baseline requirements. Baseline requirements enabled a component of the system design and development to be specified almost in its entirety so that less schedule risk was present as later design iteration could be avoided.

In Stage 2, design and development was undertaken at the subsystem level. Where interfaces were required, the standards were agreed upon between subsystems. The subsystems were therefore never considered purely in isolation. Instead the focus was as much on interface design as it was on internal operation. The role of project members therefore involved communication of interfaces as much as subsystem development.

Following design and development it was desirable to test the component or subsystem without also testing its interface. That is, isolated tests on the subsystem were carried out where possible to ensure valid design and implementation to the relevant system requirements. In this manner the doctrine of success refinement could be applied to the subsystem to ensure correct subsystem design.

Integration and testing represented the major milestone in the project timeline but it is misleading to consider each subsystem decoupled through their initial design and implementation stages. Indeed, it was particularly difficult to separate the requirement for the GCS to provide network functionality from the communications subsystem and its need to log and display all state data from the state subsystem. Likewise, the control could not be implemented, designed or even tested in isolation. More often than not only after this stage could design iteration take place.

The final stage of delivery was essentially a formal process of documentation and system acceptance and conformance testing. In practise the iterations at the third and fourth stages introduced considerable schedule pressure and it was therefore necessary to advance to the delivery stage prior to complete system integration and testing. The effect was the need to review the achievable system requirements in light of the technical and schedule pressures.

At each stage of project progress the engineering methodology remained focused on the delivery of the subsystem aims and thus system aims against identified project milestones.

2.2 Work Packages

A work breakdown structure consisting of a number of work packages per subsystem outlined all the tasks to be completed to achieve the high level objectives and system requirements. The work packages for the GCS and flight control subsystems are categorised in Figure 2.2 by the stage of the systems engineering process they occurred in. The following sections give brief descriptions of the tasks the author was required to complete in each.

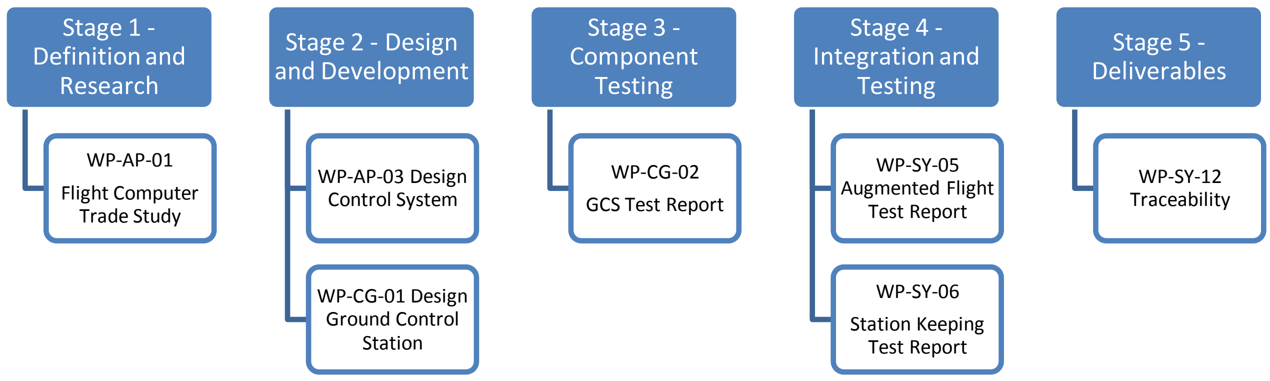

Stage 1 - Definition and Research
WP-AP-01 Flight Computer Trade Study
Stage 2 - Design and Development
WP-AP-03 Design Control System
WP-CG-01 Design Ground Control Station
Stage 3 - Component Testing
WP-CG-02 GCS Test Report
Stage 4 - Integration and Testing
WP-SY-05 Augmented Flight Test Report
WP-SY-06 Station Keeping Test Report
Stage 5 - Deliverables
WP-SY-12 Traceability

Figure 2.2–GCS and Flight Control Project Stages and Work Packages

2.2.1 WP-AP-01 Flight Computer Trade Study

Research the information that is required to achieve the System Requirements for the flight computer. Produce a Trade study from this research, and recommend the best option for the project.

2.2.2 WP-AP-03 Design Control System

Design and develop the control system to keep the platform stable while in flight. Complete a design document outlining the control system developed.

2.2.3 WP-CG-01 Design Ground Control Station

Design and develop the ground control station to receive and log data sent by the platform. This GCS should also be able to send pre-designed commands to the helicopter.

2.2.4 WP-CG-02 GCS Test Report

Test the ground control stations ability to display and log transmitted data from the platform during a flight test. Also test the GCS ability to transmit commands to the platform.

2.2.5 WP-SY-05 Augmented Flight Test Report

Test the completed platform for its performance of augmented flight.

2.2.6 WP-SY-06 Station Keeping Test Report

Test the completed platform for its performance of station keeping flight.

2.2.7 WP-SY-12 Traceability

Using a Traceability matrix, provide evidence that the System Requirements were met.

2.3 Source Code Management

The author was one of two project members expected to manage the creation, version control and backup of project source code. In a project where up to four people required access to these materials, it was impractical to rely on manual means. Where possible, to reduce scheduling and technical risk, software development tools were used.

2.3.1 Version Control

Google Code, a free service for open source projects was used to provide Subversion server hosting (SVN). Subversion provided the project members with the ability to check out resources to any computer, commit changes directly to the server, manage code and documentation online, tag versions of interest and collaborate on wiki spaces. It was also an effective solution to automate backup. Two Google Code projects were setup as part of the project, the first for mainstream project development, ahns10 [7], and the second for network interfacing, heliconnect10 [8].

2.3.2 Development Tools

Although their usefulness is often debated, an integrated development environment (IDE) and cross-platform AVR programmer were used rather than rely on the GNU toolchain or the Windows based AVRStudio respectively. Doxygen was also used as a tool for code documentation rather than rely on a manual process and to encourage code commenting.

2.3.2.1 Qt Creator Integrated Development Environment

QtCreator was used in the generation and management of all GCS code [9]. It provided many tools invaluable for learning the Qt framework including integrated help documentation, examples and a comprehensive graphical user interface (GUI) layout editor. It should be noted that the build process could be carried out manually using `qmake` and `make` if necessary thus use of QtCreator did not lock the author into using it. Inclusion of Doxygen and SVN interfacing as features further aided the development of the GCS GUI. Using QtCreator was a key factor in reducing schedule and technical pressures during GCS development.

2.3.2.2 AVRDUDE AVR Programmer

Development of the Mode Control Unit (MCU) software was undertaken under Linux in the C programming language. This required the use of third-party open source programs to compile and flash the AVR chip as described in [10]. The C library used was avr-libc and the program used to flash the device was AVRDude. Both resources were sourced without modification from the Ubuntu software repository.

2.3.2.3 Doxygen

Doxygen is a source code documentation tool used by AHNS to generate coding reports on the ground control station and mode control unit [11]. The contents of the documentation relate to function and variable descriptions, code operation comments and implementation level considerations. The author attempted to include enough detail to make the code comprehendible within the limited time available during the development stages of the MCU and GCS software.

Chapter 3 Subsystem Development

This chapter describes the design and development stages of the systems engineering process followed for the ground control station and flight control subsystems.

3.1 Ground Control Station

The GCS provides one of the two interfaces between system operators and the quadrotor controller (the second interface being the pilot's radio control link).The primary use of the GCS is therefore to present flight data from onboard sensors for monitoring and for mission control. The systems engineering stages of design, implementation and testing were followed during its development.

3.1.1 System Requirements

Delivery of a GCS was flagged in initial customer discussions and finalised in the fifth project high level objective. From the high level objective several system requirements were created.

[HLO-5] Ground Control Station established that a ground control station that supports appropriate command and system setting inputs and data display and logging should be developed. The design should be derived from previous AHNS developments and enable future ground station developments.

The system requirements shown in Table 3.1 are divided into baseline and derived requirements. Baseline requirements were specified by the customer and from previous AHNS experience whilst derived requirements were developed after a period of preliminary systems design.

The requirements were written to avoid subjectivity in judging if they had been met however the difficulty in testing a GUI based software system was still a concern. Concern came from the inability to use automated testing tools due to limited development time. This method of testing could have ensured stable code operation by profiling resource usage, checking for improper threading and suggesting areas of optimisation or code review. The other concern with GUI testing is the subjective nature of operator expectations. To alleviate the concerns it was necessary to define the methods that would be used for acceptance testing with the customer at the same stage as system requirement definition. The acceptance tests are listed in the table of system requirements and described in [12] and Table 3.2.

Table 3.1 - GCS System Requirements

Code	System Requirement	Acceptance Test
SR-B-02	The GCS shall enable autopilot flight mode switching between manual, stability augmented flight, and autonomous station keeping.	AT-02
SR-B-08	The autopilot system gain and reference parameters shall be updatable in flight using an 802.11g WLAN uplink from the GCS.	AT-08
SR-B-09	The airborne system shall transmit telemetry data including state data to the GCS using 802.11g WLAN.	AT-09
SR-D-07	The airborne system shall transmit all actuator inputs, including radio control inputs, to the GCS.	AT-17
SR-D-08	The GCS shall log all telemetry and uplink data communications.	AT-18
SR-D-09	Aircraft state data and control inputs received shall be displayable on the GCS along with appropriate time references.	AT-19
SR-D-10	The GCS shall provide display of avionics system health monitoring including telemetry, uplink, radio control link and battery level status read-outs.	AT-20

Table 3.2 - GCS System Requirement Acceptance Tests

Code	Means	Test Criteria
AT-02	Inspection	The GCS will switch between the three modes while the platform in on the ground and while in the air. The operator will ensure that the onboard processor has received the commands and activate the corresponding mode.
AT-08	Inspection	System gain and reference parameters will be updated from the GCS to the onboard processor. The operator will inspect the platform in-flight to ensure that the uploaded data has been modified.
AT-09	Testing Log Data	Logged telemetry data, including state data, will be inspected to ensure that the onboard processor is sending the correct information, and that is being received by the GCS.
AT-17	Testing Log Data	After a flight test while the platform is transmitting information to the GCS, the log data will be analysed to ensure that all actuator inputs are received.
AT-18	Testing Log Data	After a flight test while the platform is transmitting information to the GCS, the log data will be analysed to ensure that the telemetry and uplink data communications are received.
AT-19	Inspection	During the flight test, the transmitted aircraft state data and control inputs will be inspected on the GSC for accuracy.
AT-20	Inspection	During the flight test, the transmitted avionics system health monitoring including telemetry, uplink, radio control link and battery level status will be inspected on the GSC for accuracy.

3.1.2 Design and Development

AHNS 2009 demonstrated the development of both a Human Machine Interface (HMI) and autopilot GUI. It was noted that the HMI was important in providing real time data visualisation whilst the autopilot GUI was needed to update and run the flight control. Splitting control and data display obviously added additional and unnecessary interfaces. To meet the 2010 HLO and system requirements it was decided to merge the functionality of both 2009 GUIs into a single GCS.

AHNS 2009 left a considerable amount of legacy C++ OpenGL and Qt code. Learning OpenGL fell outside the scope of AHNS 2010 but where possible code was reused with minimal modification. The legacy Qt threading and network UDP code was judged unsuitable after testing and code inspection. The addition of WiFi dictated a GCS capable of multiple simultaneous threads, rather than sequential execution. It was also noted that UDP does not require application level packet synchronisation; a timestamp sufficies in detecting delayed and duplicate packets.

3.1.2.1 Thread Architecture

The GCS architecture is based on the philosophy that time-critical or background data processing should be performed in a separate thread to that running the GUI. In Qt this was suggested to be the case as both the GUI and network objects require event loop processing for user events and datagram receive events. Being able to create multiple event loops using threading removes the possibility of the GUI becoming slow and unresponsive.

Figure 3.1 shows the initial architecture of the GCS based on two threads. The telemetry thread was originally envisaged to receive, transmit, log and display all airborne system data via a UDP WiFi communication link. Following initial implementation however the existence of the GUI objects in a separate thread meant the telemetry thread could realistically only be run in the background to check for data that had been received or required transmission. The GCS thread therefore handles user input events and data display and logging.

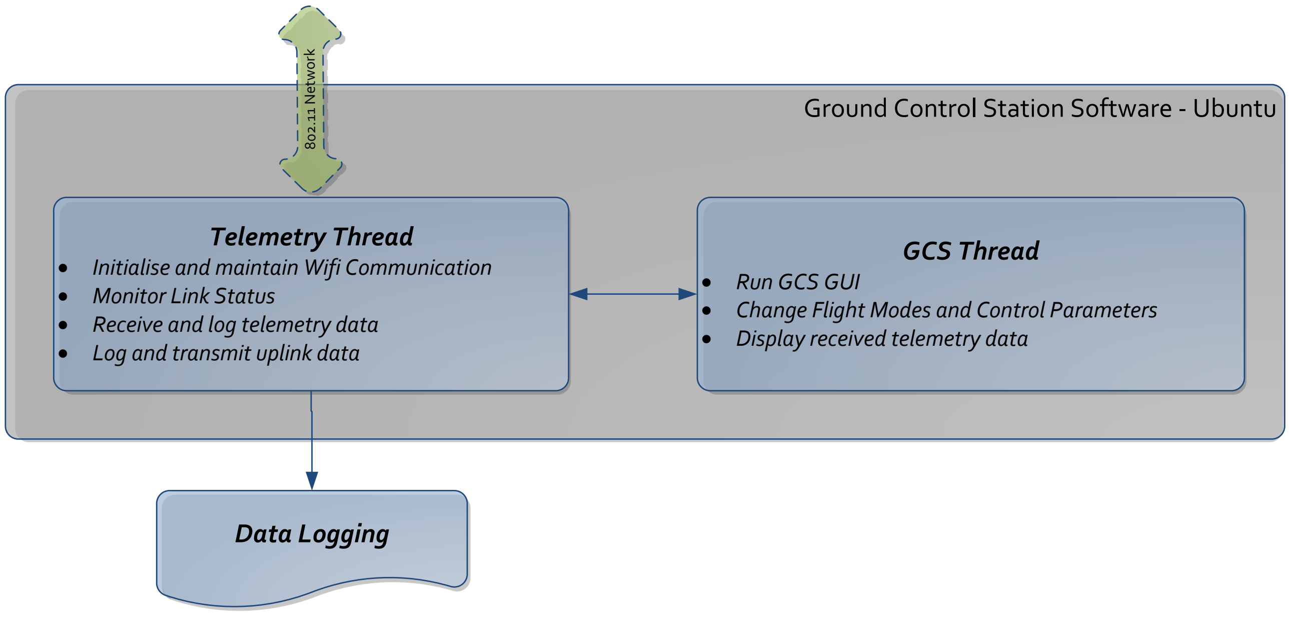

802.11 Network
Ground Control Station Software - Ubuntu
Telemetry Thread
Initialise and maintain Wifi Communication
Monitor Link Status
Receive and log telemetry data
Log and transmit uplink data
GCS Thread
Run GCS GUI
Change Flight Modes and Control Parameters
Display received telemetry data
Data Logging

Figure 3.1–Initial GCS Architecture

The final GCS architecture of Figure 3.2 reflects several more iterations in implementation and testing. Three threads were deemed necessary to enable the GCS to display, access and forward data from onboard and Vicon sensors. The Vicon thread is necessary to use the proprietary C++ shared library to access object position and pose information without locking the GUI. Vicon data is accessed through the GCS rather than the flight computer because the onboard software is not coded in C++. The computer running the GCS is also often more powerful and can be connected to the server using Ethernet rather than WiFi when latency is a concern.

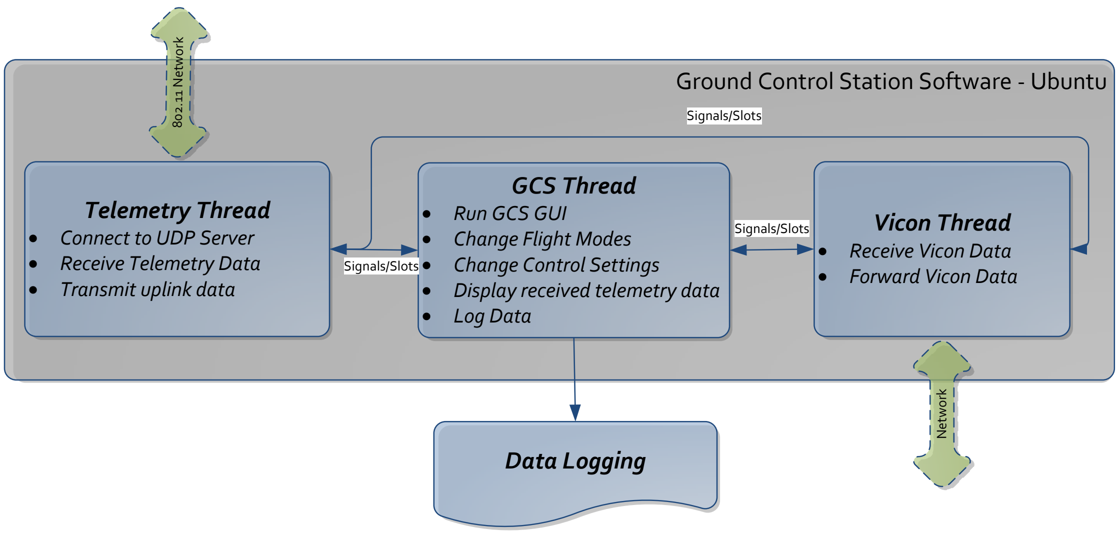


Figure 3.2 - Final GCS Architecture

3.1.2.2 GUI Design

The use of the GCS in every flight test necessitates efficient GUI layout. Layout design was judged an individual's preference and would change depending on the stage of flight. For example once telemetry was engaged its settings would not need to be modified. The approach of designing a static GUI layout in 2009 led to a considerable amount of time being dedicated to designing an operator efficient layout. A static layout also means GUI layout modifications require code changes, a condition that increases the risk of untested code being used after system integration. GUI design was therefore not based on layout but on making design choices to enable layout customisation.

With knowledge of the Qt GUI framework the means of giving the user control of the GUI layout is to create dockable, atomic GUI units termed *widgets*. Each widget can be dragged and docked in different locations inside or outside the main window. The atomic nature of the widgets enables code reuse and unit testing; both considered industry best practise in software design. The purpose and design of each individual widget is associated with a specific system requirement.

3.1.2.2.1 Artificial Horizon

The artificial horizon was designed in 2009 for the now defunct HMI GUI. Its aim was to provide an OpenGL visualisation of the current aircraft altitude and pitch and roll attitude. After testing it was found that altitudes over 10 meters could not be displayed. The redesign abstracts it as a standalone widget and disables the altitude display. Its use continues as it contributes to the array of state data displayed in fulfilment of SR-D-09.

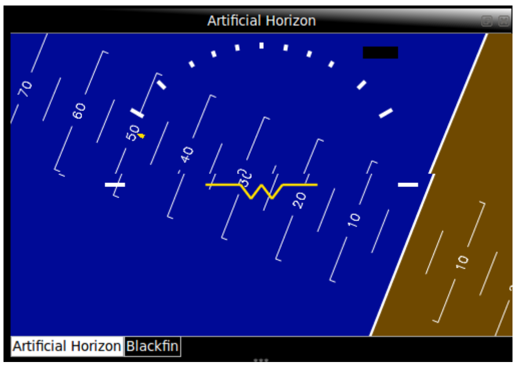
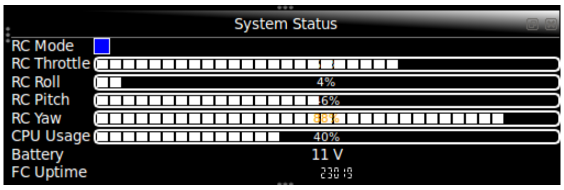


Figure 3.3 – OpenGL Artificial Horizon Widget

3.1.2.2.2 System Status

The system status widget is the primary method of displaying onboard system health and status information as required by SR-D-10 and SR-D-07. It is designed to display the RC link status, battery level and commanded RC inputs. The commanded RC pulse information is displayed in the form of percentage of range 1000 to 2000 micro-seconds.

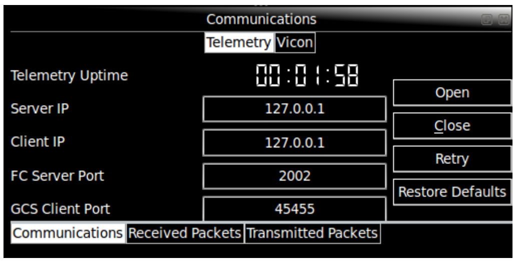

System Status

RC Mode 🟦
RC Throttle
RC Roll 4%
RC Pitch 46%
RC Yaw 88%
CPU Usage 40%
Battery 11 V
FC Uptime 230:19

Figure 3.4 - System Status Widget

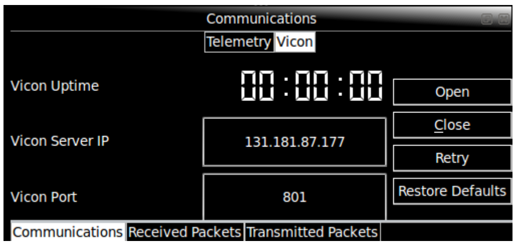
3.1.2.2.3 Communications

The communications widget is designed to enable configuration of the network connections to both the flight computer and Vicon. Settings for each connection include server IP addresses and port. Buttons provide thread start, stop and restart control whilst a timer for each thread monitors uptime and connection status. This enables fulfilment of SR-B-08 and SR-B-09.



Telemetry Uptime 00:01:58
Server IP 127.0.0.1
Client IP 127.0.0.1
FC Server Port 2002
GCS Client Port 45455

Figure 3.5 - Communications Widget, Telemetry Tab

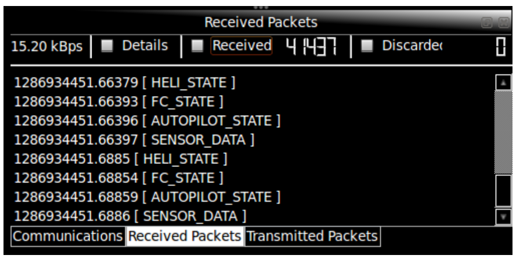


Vicon Uptime 00:00:00
Vicon Server IP 131.181.87.177
Vicon Port 801

Figure 3.6 - Communications Widget, Vicon Tab

3.1.2.2.4 Received Data Console

The received data console is designed to provide information on the packets being received from the flight computer. It displays timestamps and details of the packets received. Summary information provided includes packet rate and packet discard count. The function to enable all packets to be shown enables completion of SR-D-08 whilst the time reference display of packets meets SR-D-09.

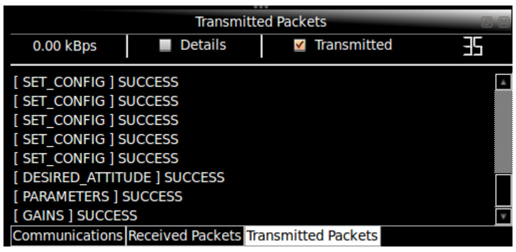


```
1286934451.66379 [ HELI_STATE ]
1286934451.66393 [ FC_STATE ]
1286934451.66396 [ AUTOPILOT_STATE ]
1286934451.66397 [ SENSOR_DATA ]
1286934451.6885 [ HELI_STATE ]
1286934451.68854 [ FC_STATE ]
1286934451.68859 [ AUTOPILOT_STATE ]
1286934451.6886 [ SENSOR_DATA ]
```

Figure 3.7 - Received Console Widget

3.1.2.2.5 Transmitted Data Console

The transmitted data console is the complement to the received data console. It displays the status of all transmitted data packets in fulfilment of SR-D-08. It includes a transmission count and rate.



```
[ SET_CONFIG ] SUCCESS
[ SET_CONFIG ] SUCCESS
[ SET_CONFIG ] SUCCESS
[ SET_CONFIG ] SUCCESS
[ SET_CONFIG ] SUCCESS
[ DESIRED_ATTITUDE ] SUCCESS
[ PARAMETERS ] SUCCESS
[ GAINS ] SUCCESS
```

Figure 3.8 - Transmitted Console Widget

3.1.2.2.6 Data Plotter

The data plotting widget was included in the GCS design based on feedback from AHNS 2009. It provides real-time data plotting of data from the sensors, autopilot, flight computer and Vicon to fulfil SR-D-09. The widget does not manage its own data, instead it accesses a centralised, shared array of data. This enables multiple data plotters to run with minimal memory overhead.

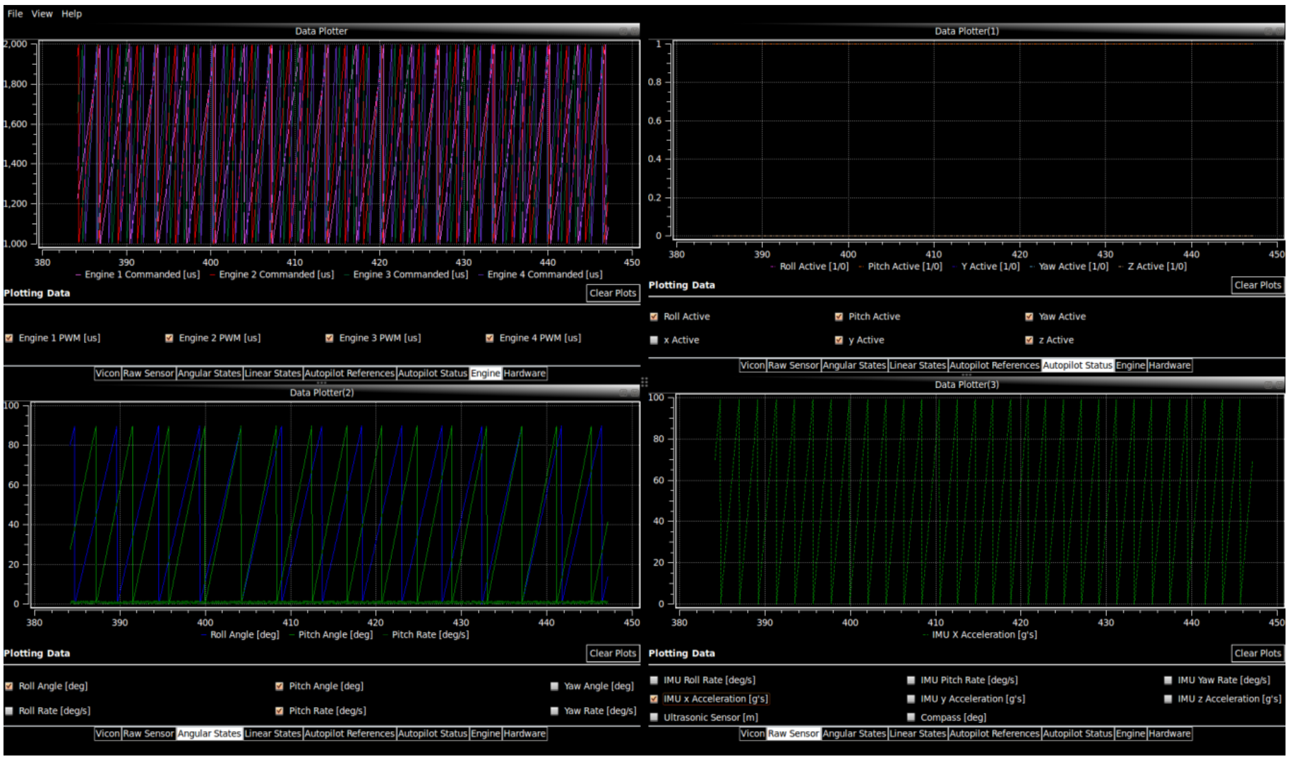


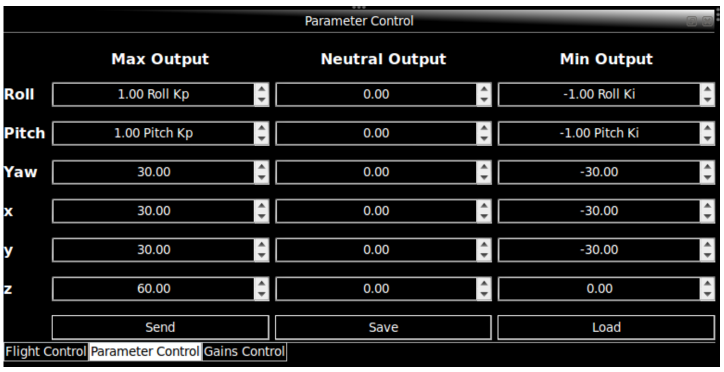
Figure 3.9 - Data Plotter Widgets

3.1.2.2.7 Blackfin Camera Feed

The Blackfin Camera widget was as part of the State Estimation subsystem. It demonstrates the ability for multiple team members to contribute widgets without concern for other aspects of the GCS. It enables connection and configuration of the camera and OpenGL display of a live video feed through the camera's 802.11 WiFi interface.

3.1.2.2.8 Control Parameters

The purpose of the parameter control widget is to enable the operator to set, save and load the trims and bounds on all active control loops without modifying the onboard flight computer code. This directly aims to meet SR-B-08. During some control loop implementation stages it is nonsensical to bound the control loop outputs, in these cases the parameters can be used to transmit other required parameters to the flight computer.

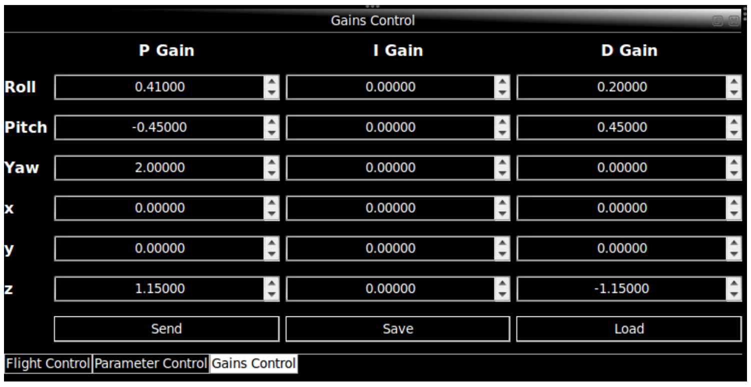


	Max Output	Neutral Output	Min Output
Roll	1.00 Roll Kp	0.00	-1.00 Roll Ki
Pitch	1.00 Pitch Kp	0.00	-1.00 Pitch Ki
Yaw	30.00	0.00	-30.00
x	30.00	0.00	-30.00
y	30.00	0.00	-30.00
z	60.00	0.00	0.00

Figure 3.10 - Parameter Control Widget

3.1.2.2.9 Control Gains

SR-B-08 requires the ability to modify gains from the GCS. The control gains widget is designed to provide the GCS operator with a means of setting, saving and loading the PID control gains on all airborne control loops.

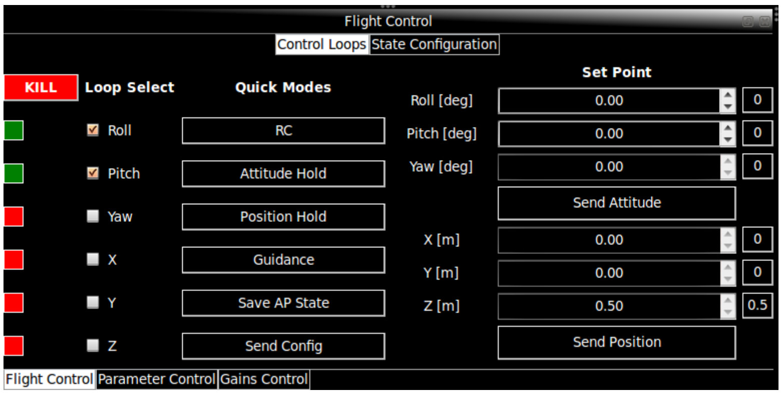


	P Gain	I Gain	D Gain
Roll	0.41000	0.00000	0.20000
Pitch	-0.45000	0.00000	0.45000
Yaw	2.00000	0.00000	0.00000
x	0.00000	0.00000	0.00000
y	0.00000	0.00000	0.00000
z	1.15000	0.00000	-1.15000

Figure 3.11 - Gains Control Widget

3.1.2.2.10 Flight Control

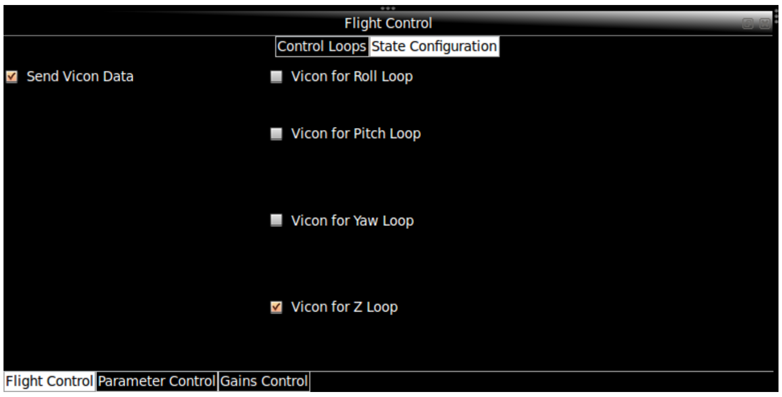
The flight control widget is designed to meet SR-B-02; to enable the operator to set and control the active control loops and report current settings. The risk of uncontrolled flight is also countered through inclusion of a manual '*KILL*' button to force the MCU and all engines into failsafe. The inclusion of Vicon in the state estimation system required that there be a means of online switching between Vicon and onboard sensors.



🟩 ☑ Roll
🟩 ☑ Pitch
🟥 ☐ Yaw
🟥 ☐ X
🟥 ☐ Y
🟥 ☐ Z

Roll [deg] 0.00 0
Pitch [deg] 0.00 0
Yaw [deg] 0.00 0
X [m] 0.00 0
Y [m] 0.00 0
Z [m] 0.50 0.5

Figure 3.12 - Flight Control Widget, Control Tab



☑ Send Vicon Data
☐ Vicon for Roll Loop
☐ Vicon for Pitch Loop
☐ Vicon for Yaw Loop
☑ Vicon for Z Loop

Figure 3.13 - Flight Control Widget, State Tab

3.1.3 Implementation

The use of threads, dockable widgets, UDP network code and OpenGL led to considerable challenges in C++ Qt development. To ensure the final implementation was such that the system requirements were met several iterations of design and implementation were required.

3.1.3.1 Main Window Implementation

The GCS is based on the gcsMainWindow class derived from the QMainWindow class. This class runs the main event loop called from the main function of the GCS process. The class acts as the parent of all the graphical widgets and non-graphic objects used in the GCS. All inter-thread and inter-object signal/slot connections are made within this parent class. Figure 3.14 is a diagram of the members of the class, including the object variable names and classes. The window provides a stylised visual canvas but once each widget is placed in QDockWidget objects they can be moved freely on screen.

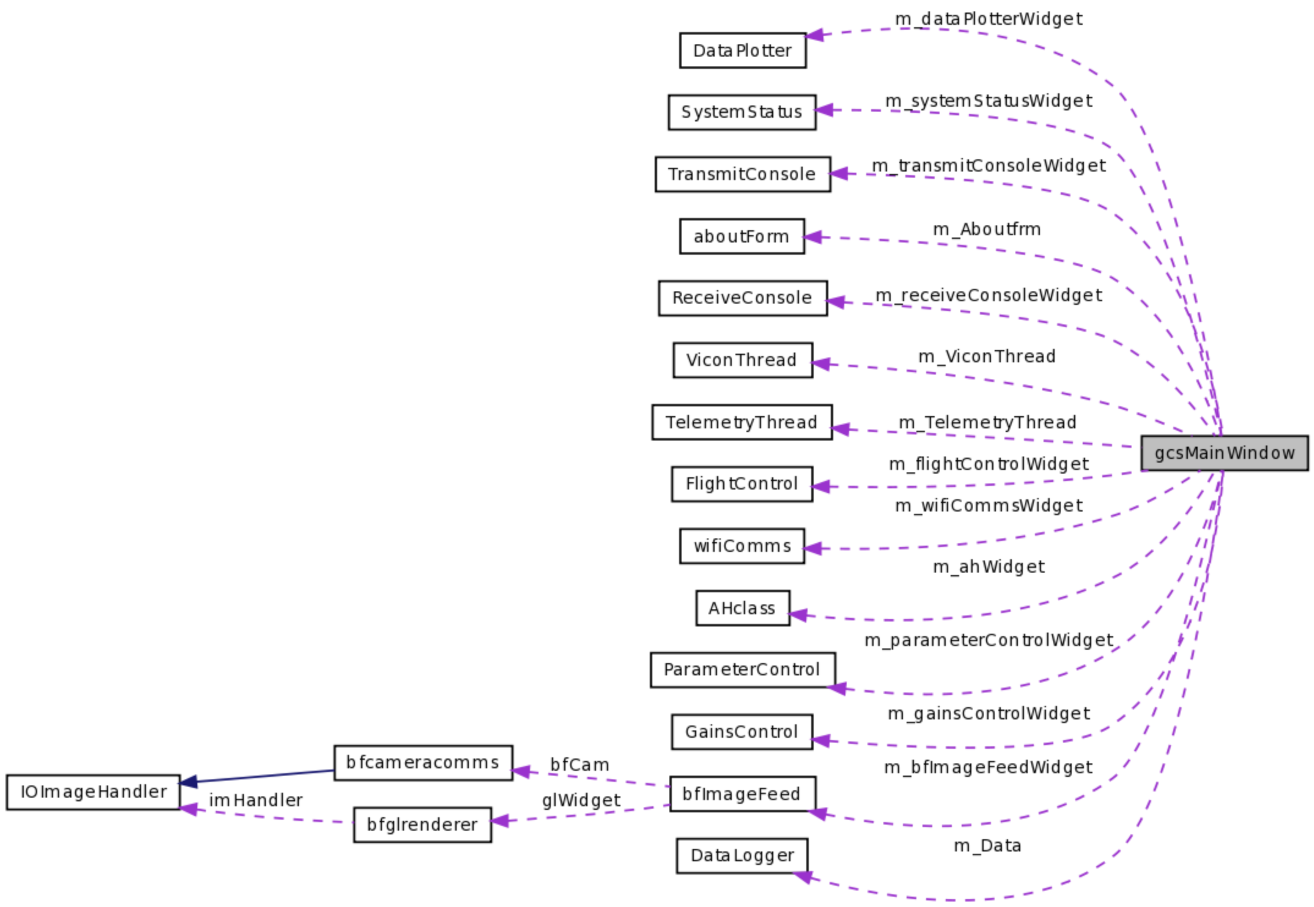


Figure 3.14 - Member Objects of the gcsMainWindow Class

3.1.3.2 Thread Implementation

The threads implemented in the GCS are derived objects of the class QThread as shown in Figure 3.15. Not shown is the parent process, referred hereafter to as the GCS thread. The telemetry and Vicon threads are required to provide a continuous supply of information that the GCS thread is required to display and log. The use of QThreads enabled use of a Qt signal/slot mechanism to provide non-blocking, cross thread resource sharing. A mutex or semaphore based system was not deemed appropriate as the GUI would be unresponsive as a telemetry or Vicon data transmit or receive operation was completed.

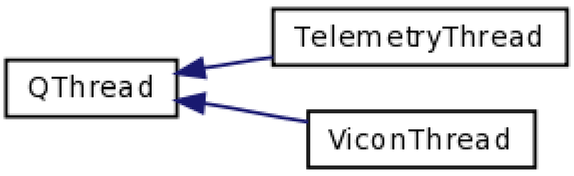


Figure 3.15 - GCS Thread Inheritance

The disadvantage of the signal slot mechanism is obviously the need for a pass by value function call. Pass-by-value signals are only used for cross-thread signals and with simple structure data types to reduce overhead. In the GCS thread a centralised data storage object was also created to given all widgets centralised access to the cross-thread telemetry and Vicon data without multiple slot function calls (Figure 3.16).

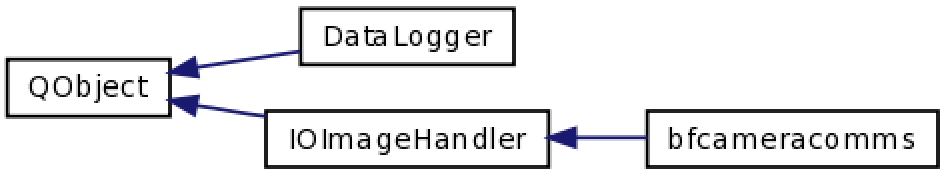


Figure 3.16 - Non-GUI Widget QObjects

The QThreads are started and stopped through emission of *ConnectionStart*, *ConnectionClose* or *ConnectionRetry* signals from the corresponding buttons in the GCS Communications widget. The signals are handled by slots in the gcsMainWindow. The slots pass the server and client information required for network connection to the thread object constructors. This implementation enables both the main window and the communications widget to monitor connection status by simply checking the thread's existence.

3.1.3.2.1 Telemetry Thread Implementation

Besides the initial creation of the QUdpSocket, the thread's execution is entirely event driven. That is, the thread's operation is limited to running its event loop which operates on a first-in-first-out queue of signals. This is avoids manual polling of the socket for received data as the *readyRead* signal is automatically added to the event loop queue. When executed the received signal calls the *DataPending* slot to extract the packet type and data. The slot then emits a signal that contains the received data to the GCS thread for data logging and display.

Data transmission is also achieved using the cross-thread Qt signal/slot system. Data structures that require transmission are emitted with signals from widgets in the GCS thread. The signals are connected to unique slots in the telemetry thread which function to pack the data into raw byte form within QByteArray objects. The raw data is then packed with a message header into a datagram to be transmitted using *writeDatagram* method of the QUdpSocket. The packet format is shown in Figure 3.17 with command types specified in the common files from heliconnect10 [8].

Header		***Message***
Time Stamp (32bit timeval)	Command Type (uint32_t)	Data Bytes (Packed struct)

Figure 3.17 - UDP Datagram Format

UDP is not connection oriented but the airborne and GCS telemetry code does have application level checks for reliable data transmission. To initially confirm the GCS has communication with the server, the socket sends a packet with the type COMMAND_OPEN and no data. If communication is established the server will respond in kind with a message of type COMMAND_ACK and data of COMMAND_OPEN. The telemetry thread queues timer events that periodically check for replies and resend the packet. After not connecting for a period time the thread notifies the GCS thread of its failure to connect and terminates. For critical autopilot packets the same send, acknowledge and retry implementation is followed. Instead of terminating the thread on failure however a failed message is displayed in the transmit console GUI widget and the thread continues normal operation. Vicon data packets requiring transmission are not implemented with any form of reliability above standard UDP due to their high update rate.

3.1.3.2.2 Vicon Thread Implementation

The Vicon thread does not rely on Qt network objects and it is not event driven. Proprietary code is used to create a client object using the server network address provided and create a connection. If connection is successful, the thread will run until stopped. Conversely if the client fails to connect the thread is terminated and the gcsMainWindow object is notified. During operation the Vicon server is queried and data is received using vendor functions. The queries are time intensive and if threading is not implemented correctly these cause the GCS thread to become unresponsive. Once a query yields valid Vicon state data these are emitted to the data logger and GCS thread objects using Qt signal/slots.

3.1.3.3 Widget Implementation

Implementation of the individual widgets is a relatively straight forward task of coding GUI object events. Besides the obvious button click events, other objects such as the text edit boxes in the ParameterControl and GainsControl classes trigger signal emits. The challenge for all widgets is placing them in individual QDockWidget objects to enable rearrangement. To do this every widget needs to inherent from QWidget or QGLWidgetas shown in Figure 3.18 and Figure

3.19. The objects are also created as members of the gcsMainWindow class as shown in Figure 3.14. Finally, QDockWidget objects are created and added to the dockable areas of the main window. In the case of the DataPlotter widgets, where multiple instances of the class can exist, multiple QDockWidgets are required.

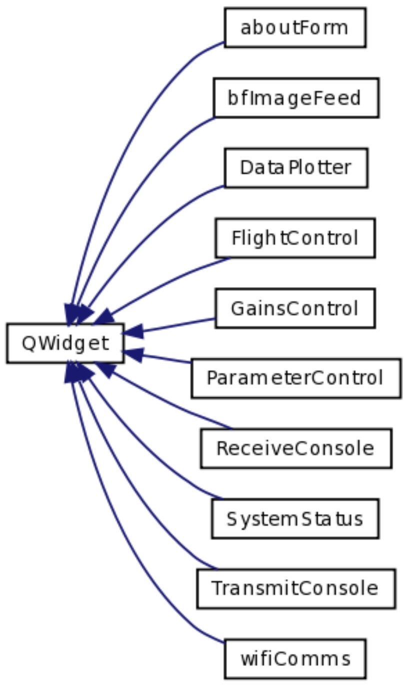


Figure 3.18 - Inheritance of Widgets from QWidget

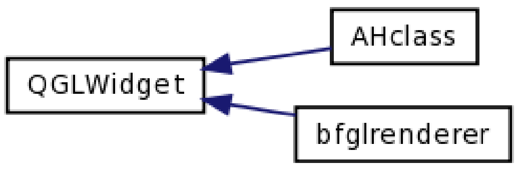


Figure 3.19 - Inheritance from QGLWidget

To achieve multiple DataPlotter objects the gcsMainWindow class includes two QLinkedList objects. One is used to track the DataPlotter widgets and the other their QDockWidget objects. Tracking the objects is important to ensure dynamic creation and deletion of the objects. The use of the third party, open-source graphing widgets from Qwt [13] posses no hurdles as data is accessed from the centralised DataLogger object. Note that creating new DataPlotter widgets does not lead to data duplication.

3.1.3.4 Iteration and Tools in Implementation

The design of the GCS is such that if implemented correctly all the HLOs and SRs should be met. During software integration testing however code inefficiencies and poor performance not concerning the SRs were noticed. This included increased processor load due to excessive number-to-string conversions in the transmitted and received consoles and redrawing OpenGL scenes and replotting data with each received packet.

The iterative nature of systems engineering when applied to software development meant new implementation methods were sought. To overcome the number-to-string conversions options are included to show only that data that is necessary. Likewise update performance of the Artificial Horizon and plotting widgets is controlled by a 25Hz timer in the gcsMainWindow window. The GCS event loop therefore can reduce the update rate to significantly less than the telemetry rate.

Systems integration testing has also resulted in code reimplementation. The System Status widget is an example of flight test results determining code level changes. Pulses original displayed in the widget were the individual engine commands and the RC Link was set by a throttle failsafe value. After an incident involving a flipped quadrotor it was deemed necessary for the GCS operator to have oversight of the RC pilot's commanded inputs and control mode. The GUI design remained essentially unchanged with the exception of the revised data to better meet the SRs of data display.

The iterative software implementation process required the use of many tools including the Qt Creator for code development and Qt documentation, Doxygen for code documentation and review, SVN for version control and tcpdump for network debugging. The process yielded a stable GCS under Ubuntu with OpenGL, Vicon and Qwt code contributing to instabilities under Mac OSX.

3.2 Flight Control

AHNS by definition requires a flight control system to manoeuvre the aircraft during flight. Developing an attitude control system to enable flight is a challenge with astatically unstable and non-linear quadrotor platform. Taking this further in an attempt to enable localised autonomous flight further increases the system complexity and risks. This section describes the design process followed for the flight control subsystem and the final designs created and implemented. It documents the quadrotor dynamics used during controller design, attitude controller revisions and the position controller.

3.2.1 System Requirements

The development of a flight control system was explicitly requested by the customer and captured in the high level objectives in HLO-4 Autonomous Hovering Flight. The objective was to develop an autopilot to enable sustained indoor autonomous hovering flight. For future compatibility it was also to be designed to enable future ingress and egress manoeuvre to longitudinal and hovering flight.

System requirements for the flight control system are summarised in Table 3.3. SR-B-10 originated from discussion with the customer and from recommendations of previous AHNS groups. To meet this requirement the final controller design was to be tested using AT-10 in an inspection of the control methodology. SR-D-03 and SR-D-04 are concerned with the ability of the final controller design to control attitude and position respectively; both with minimum RC pilot input. Subjective testing of these requirements is avoided by applying specific acceptance testing standards from Table 3.4.

Table 3.3 - Flight Control System Requirements

Code	System Requirement	Acceptance Test
SR-B-10	The autopilot control methodology shall be based on cascaded PID control loops.	AT-10
SR-D-03	The autopilot shall provide stability augmented flight.	AT-13
SR-D-04	The autopilot shall provide autonomous station keeping capability within a 1 meter cubed volume of a desired position.	AT-14

Table 3.4 - Flight Control Acceptance Tests

Code	Means	Test Criteria
AT-10	Inspection	The control implementation will be reviewed to ensure PID control is implemented and includes saturation, rate limiters, anti-windup considerations as required.
AT-13	Inspection	The platform will receive movement commands to move in a direction and speed. The platform must move as desired while in stable flight.
AT-14	Testing Log Data	The platform will receive a command to station keep at a fixed co-ordinate for one minute. The telemetry data received at the GCS will be analysed to ensure that it did not move outside a 1 meter cubed volume of the desired position.

3.2.2 Design and Development

3.2.2.1 Quadrotor System Control Inputs

The quadrotor configuration used by AHNS is that used in [3, 14] and shown in Figure 3.20[14]. The configuration is such that all engines produce upwards thrust with the pair1 and 3 rotating counter to the pair 2 and 4. Motion is commanded by exploiting the torque and thrust forces produced by each engine on the quadrotor.

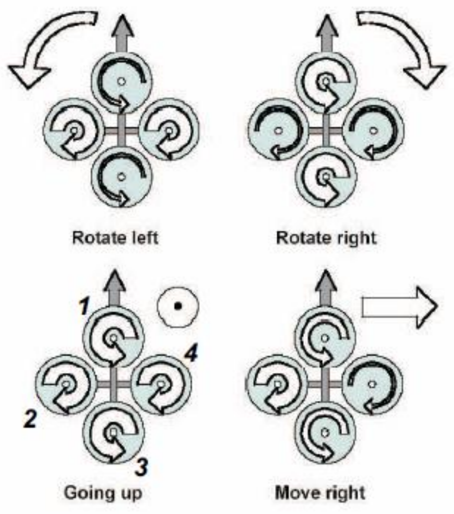


Figure 3.20 - Quadrotor motor control input effects [14]

A process of control abstraction described in [3, 14]s is used to determine the required individual engine commands. This is necessary to implement any attitude and position control in hardware as the engine speed controllers are driven by pulse width modulated (PWM) signals. To accelerate the quadrotor vertically the thrust force of each engine is increasing or decreasing. This obviously requires the respective change of motor speeds. The control input force is thus

$$U_1 = T_{1c} + T_{2c} + T_{3c} + T_{4c}$$

To vary roll or pitch, asymmetric engine thrust is produced between engines 1 and 3 or 2 and 4. The control inputs are therefore

$$U_2 = -T_{2r} + T_{4r}$$
$$U_3 = T_{1p} - T_{3p}$$

The effect of these attitude controls on position should be noted as pitch and roll variations also lead to longitudinal and lateral motion respectively [3, 14]. The final control input available with this engine configuration is yaw. Yaw is controlled by exploiting the opposing torque of the two engine pairs, 1 and 3 and 2 and 4 [3]. Thus decreasing the speed of one pair will cause the aircraft to yaw in the direction of their propeller rotations. The control inputs force is therefore

$$U_4 = T_{2y} + T_{4y} - \left(T_{1y} + T_{3y}\right)$$

Although these effects are well understood and the rigid body equations of motion are provided along with MATLAB/Simulink simulation means in [3, 14]it is still not possible to use this information to gain tune. The design stage of the controller is therefore based around determining the required states for control and the PID design variations to be implemented and tested. Provisions have been included in the design of the software and hardware systems to log all state and control data for facilitation of linear system model development.

To determine the individual engine thrust and thus speeds required due to a combination of collective, roll, pitch and yaw commands (U_1, U_2, U_3 *and* U_4) the following mixing matrix is applicable:

$$\begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & -1 \\ 1 & -1 & 0 & 1 \\ 1 & 0 & -1 & -1 \\ 1 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix}$$

The attitude controller is therefore based on three attitude control PID loops; roll, pitch and yaw. For position control the z or attitude input, U_1 has a dedicated control loop. Body-frame x and y coordinate control loops have been previously implemented as providing set points to the pitch and roll attitude control[3, 14]. This approach is therefore followed as it confirms the need for a cascade position control design. For cascade position control it is therefore imperative that a suitable attitude controller is designed, implemented and tested.

3.2.2.2 Attitude Control Design

In stability augmentation mode the attitude controller is to maintain the quadrotor at a level attitude, without considering the altitude of the aircraft or its position. Attitude controller design was therefore based entirely on controlling the roll, pitch and yaw dynamics of the quadrotor with the possibility of RC signals being used to correct drift. An iterative design, implementation and testing process was used for each proposed PID control loop architecture.

3.2.2.2.1 Initial Design – Static Angle Set Point Control

The initial design of the attitude control was based on control of the filtered angles and angular rates provided by the state estimation subsystem. The roll, pitch and yaw control loop architecture was identical and is shown in Figure 3.21. The setpoint or reference angle was designed to be static and changeable only through the GCS. This reduced the role of the RC pilot to controlling the throttle, as roll, pitch and yaw control outputs (U_2, U_3, U_4) where autopilot controlled. At the implementation stage the possibility of superposition using weighted addition in the mode control unit (MCU) was added for augmented (or blue) mode.

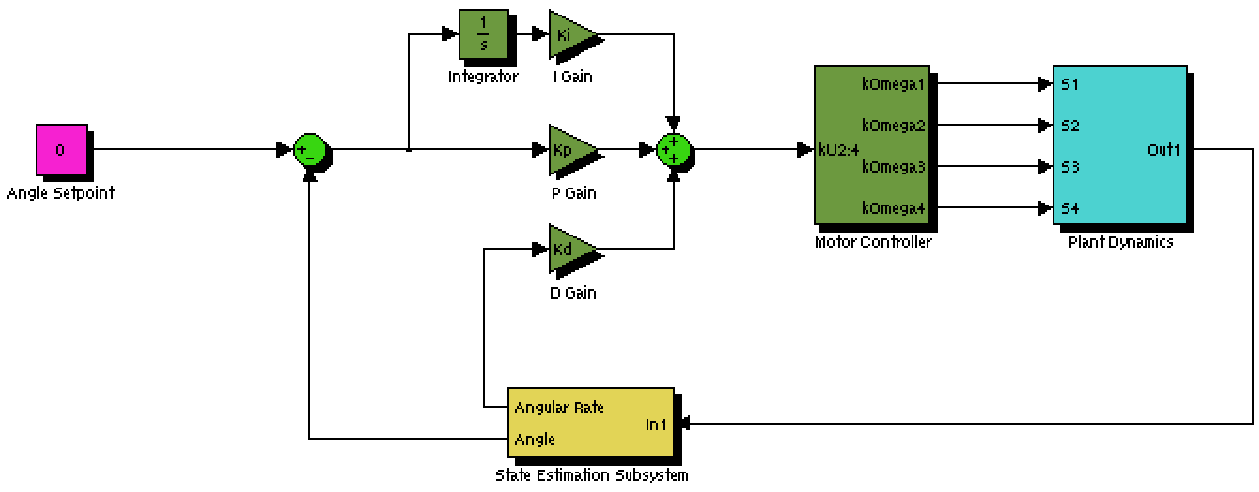


Figure 3.21 - Attitude Angle Static Control Loop

Each control loop was tested in a SIMULINK rigid body simulator before implementation. This suggested the quadrotor would remain at a stable attitude provided three gains (proportional, integral and derivative) and four loop parameters were provided (maximum output, minimum output, neutral or trim output and integrator anti-wind up limit). After implementation the roll and pitch control loops were tested in a test rig, on a bungee and unrestricted. Yaw on the other hand was left untested due to the lack of a final yaw angle state estimation solution.

The design had its gains tunned initially in the test rig using angle information from the on board state estimation and IMU. The test concluded that the rig's dynamics had a profound impact on the attitude stability. Nevertheless, Figure 3.22 illustrates that the controller performed better when control disturbances in the form of changing setpoints were introduced. Unrestricted testing of the angle controllers was used to confirm this observation.

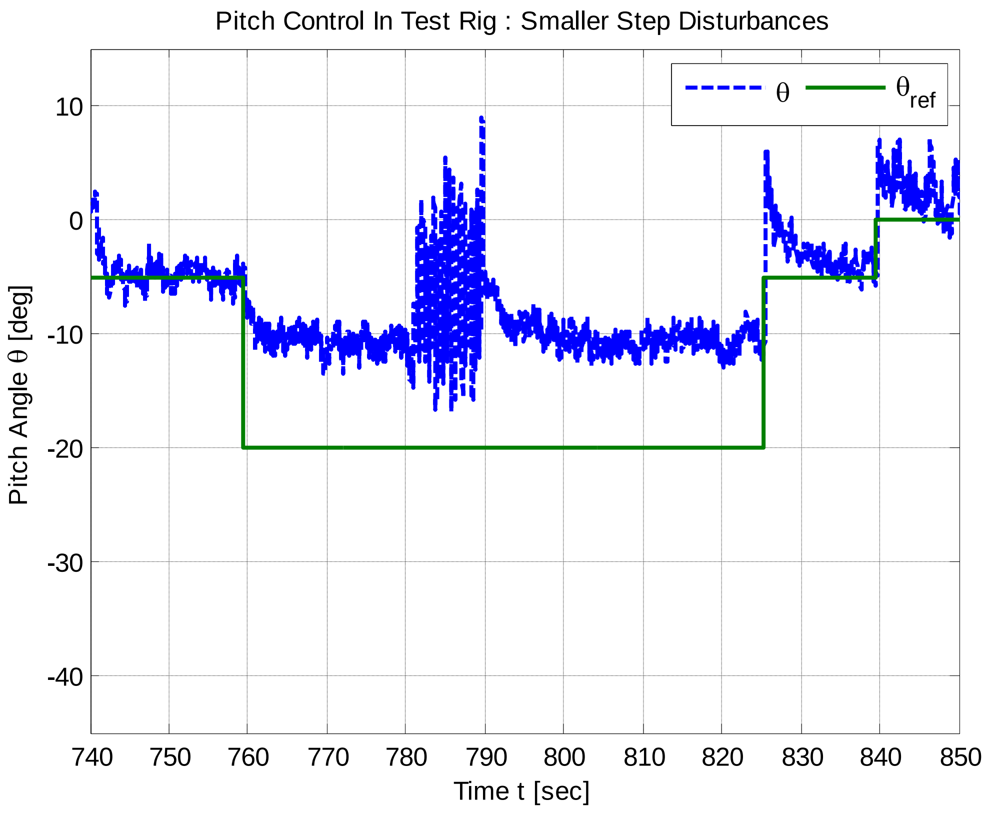


Figure 3.22 - Static Angle Control in Test Rig Small Step Disturbances

During a series of bungee and unrestricted controller tests it became apparent that the controller gains could not be tuned. The process for unrestricted tunning was to trim the quadrotor on the ground, then give the autopilot control over the roll and pitch whilst the pilot controlled yaw and throttle. This process never gave convincing results as the RC pilot was unable to correct for position drift and disturbances when near the ground without overriding the controller. A controller design where the pilot was able to correct for the sources of drift with modification of the control loop setpoints was therefore required; removal of the sources of drift such as state estimation and initial disturbances being other, but least practical, solution.

3.2.2.2.2 Revised Design – Dynamic RC Angle Set Point Control

In the case of static setpoints, the GCS operator took on the role of emulating an outer-loop position controller in updating the setpoints to attempt to correct for position drift and initial disturbances. The dynamic setpoint angle controller design however moved this responsibility to the RC pilot and the RC transmitter. As Figure 3.23 shows the roll or pitch RC channel provides an adjustment to the setpoint of their respective attitude control loops. In this way the duty cycle of the RC pulse, 1000-2000 micro-seconds was mapped to an angle value of $\pm 15°$.Note that assigning a yaw RC pulse a heading angle of$\pm 180°$ or $0°$ *to* $360°$ did not make sense and thus the yaw angle setpoint remained static.

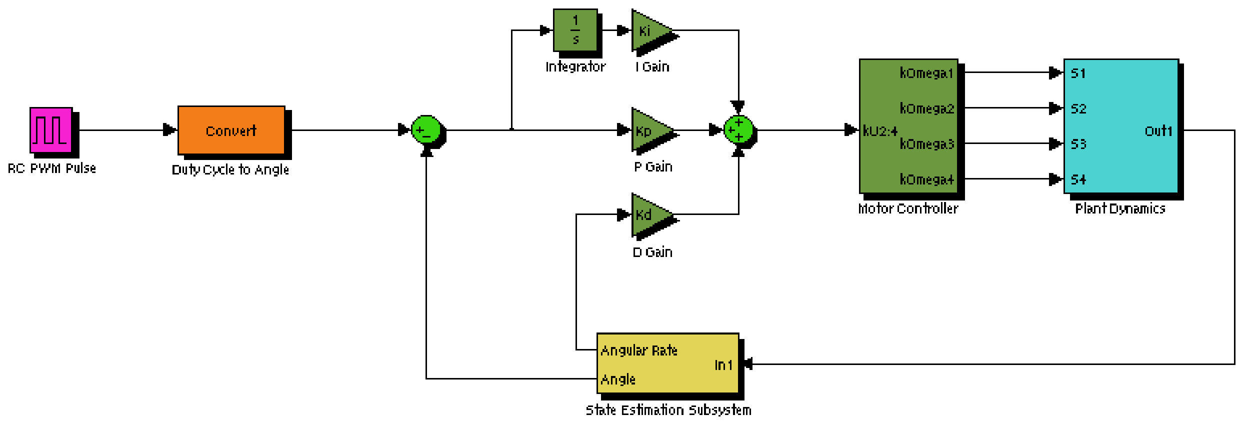


Figure 3.23 - Attitude Angle Dynamic Control Loop

In terms of implementation, this design was considerably more complex as the mode control unit was required to capture and forward the pulse duty cycle values via USART connection to the flight computer. Unrestricted flight testing was however more promising with the pilot having the ability to overcome initial disturbances provided the scaling range of angles was broad enough. Indeed the scaling of the pulse to an arbitrary range $\pm 15°$ was difficult to justify given movement in the centre of gravity or simply battery could overcome the effectiveness of the controller and RC pilot. Nevertheless, two short flights were achieved with roll and pitch angle plots shown in Figure 3.24 and Figure 3.25.

The RC pilot did not attempt to provide significant angular corrections in flight as modification of the angular setpoints by as little as 5 degrees resulted in divergent system behaviour. To counter divergent oscillations proportional gain was decreased and derivative gains was increased. These adjustments did provide some improvement, particularly derivative gain. The oscillations were however impossible for the pilot to correct due to a lack effective yaw control causing the quadrotor to continuously rotate around its centre of mass.

During the final flight tests the range of appropriate proportional gains became extremely limited, with the derivative gain term beginning to dominate. Although state estimation was providing sound results, the delay in estimation and likelihood of estimate overshoots suggested that an attitude control solution independent of the angle estimation be sought.

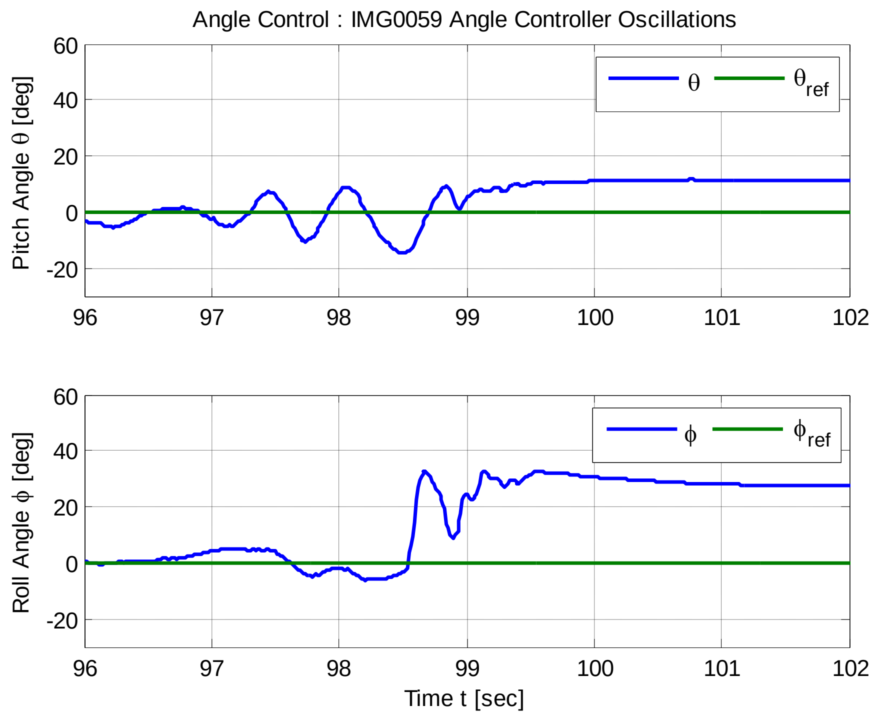


Figure 3.24 - Dynamic Angle Control Oscillations

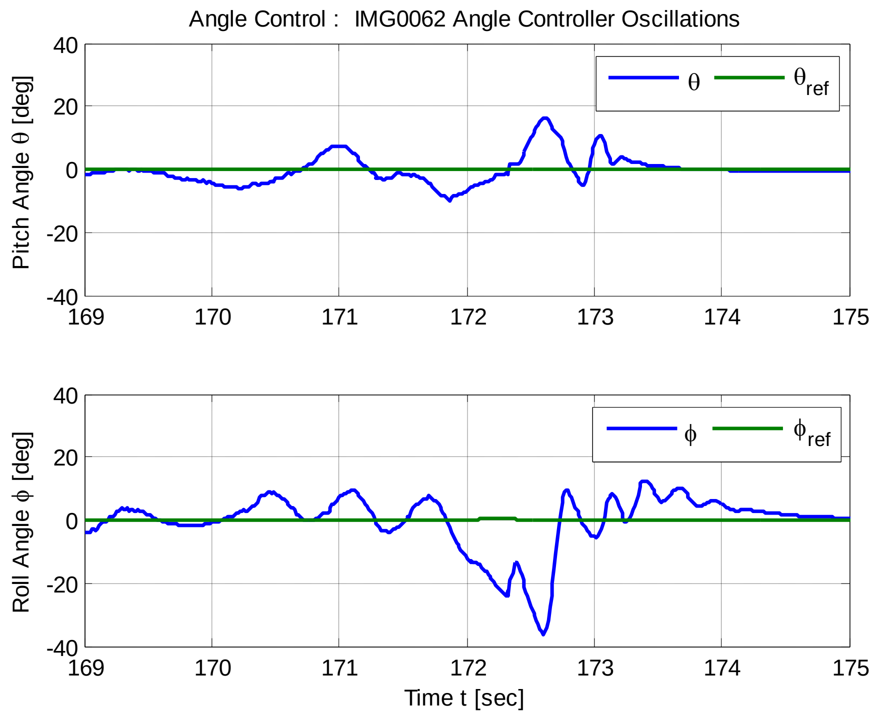


Figure 3.25 - Dynamic Angle Control Oscillations during Yawing Flight

3.2.2.2.3 Final Design – Dynamic RC Angle Rate Set Point Control

The final attitude control design reflects the lessons learnt from the previous two controller design and testing iterations. To minimise the risk of the design not providing platform stabilisation other methods such as AeroQuad [15] and a three axis RC feedback gyro [16] were reviewed. Successful RC gyro flight reinforced the understanding that angular rates, even raw or lowpass filtered IMU rates, can form the basis of quadrotor attitude control.

The control loop architecture for the rate driven attitude controller is shown in Figure 3.26. The loop design is generic and thus can be used for roll, pitch and yaw rates without modification for modulus control of heading error. There are also fewer parameters and gains to select with the maximum and minimum output parameters rendered redundant by engine saturations enforced in the MCU. The concept of a dynamic setpoint is retained but arbitrary scaling is avoided by performing loop calculations in micro-seconds high of a PWM signal. The pulse time is in the range 1000-2000 micro-seconds with 1500 micro-seconds corresponding to a centred RC transmitter and rate of $0^\circ/s$. RC Transmitter pulses are therefore no longer scaled, instead the IMU rate or state estimation rate is mapped from the limits of the IMU gyros, $\pm 300^\circ/s$, to the limits of the PWM signal.

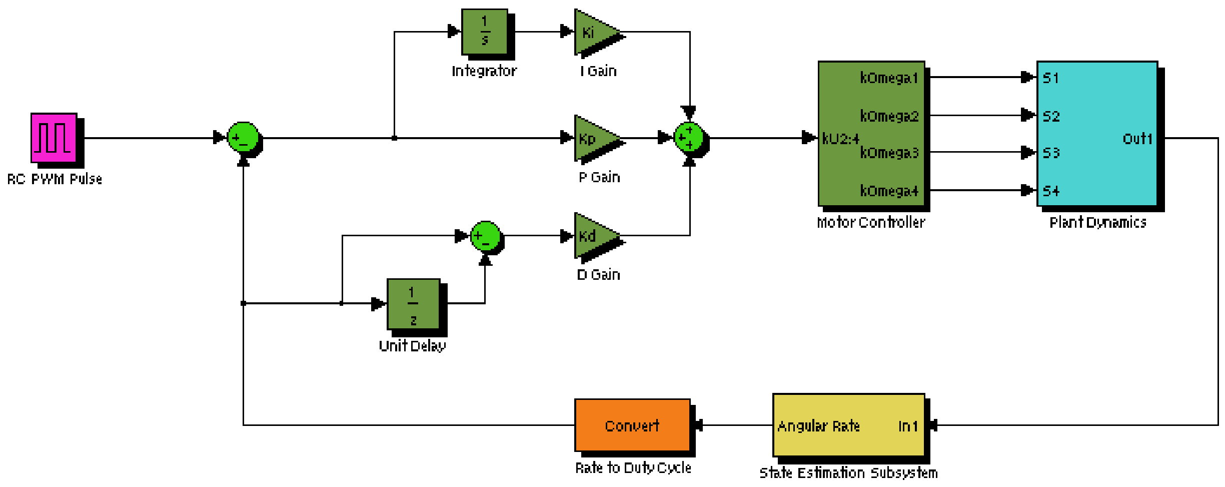


Figure 3.26 - Final Attitude Control Loop

The design does however share the implementation complexity of flight critical MCU and flight computer USART communications. It also has the possibility of too much RC pilot control, for this reason an RC control factor is introduced to avoid full RC deflection generating $\pm 300^\circ/s$ control.It should also be noted that controllers based on rates and the difference in rates have been developed to control angles but these still retain a core architecture based on rates.

3.2.2.3 Position Control Design

The autopilot control was described in SR-B-10 as taking the form of a cascaded proportional, integral and derivative controller. That is, the autopilot design consists of cascaded position and attitude PID control loops as shown in Figure 3.27. In operation, the autonomous station keeping mode position control uses information from the localisation subsystem to drive the attitude control loops.

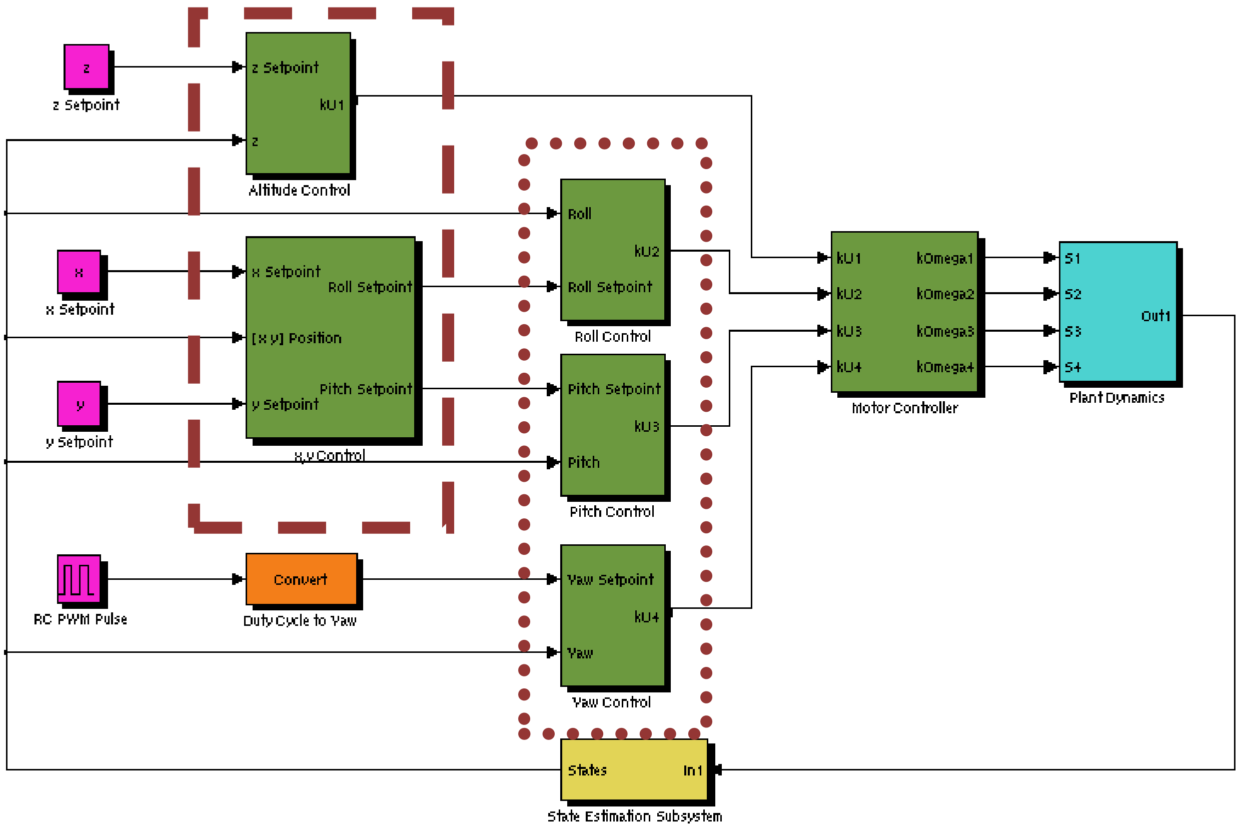


Figure 3.27 - Cascaded Position Controller Design

Figure 3.27shows that the interface between the attitude control blocks (dotted) and the position control blocks (dashed) are is the roll, pitch and yaw control setpoints. The attitude controller design this was originally designed for was one using on angles as setpoints. Use of it with an angular rate based attitude controller without modification is therefore not possible. The position controllers are required to monitor the angles to ensure they not exceed stable limits. Certainly the position controllers are also required to ensure the angular rates are at least not kept constant as this would signify continuous rotation around a quadrotor axis.

An implementation of the x,y position controller based on position error body frame rotations and PID generated pitch and roll angle commands was undertaken however it was abandoned when it became apparent testing the system safely was not possible. Risks to personnel, equipment and the quadrotor itself were difficult to mitigate to levels that justified attempting autonomous position.

Altitude controller design is unlike the other position control loops in that the output can be provided directly to a quadrotor control input. In the designed, implemented and tested altitude controller the throttle command output is generated from the PID control loop of Figure 3.28. The altitude control design has the standard PID gains and an integral anti-wind up limit parameter. As the output is directly linked to engine control however the MCU is entrusted with determining control saturation.

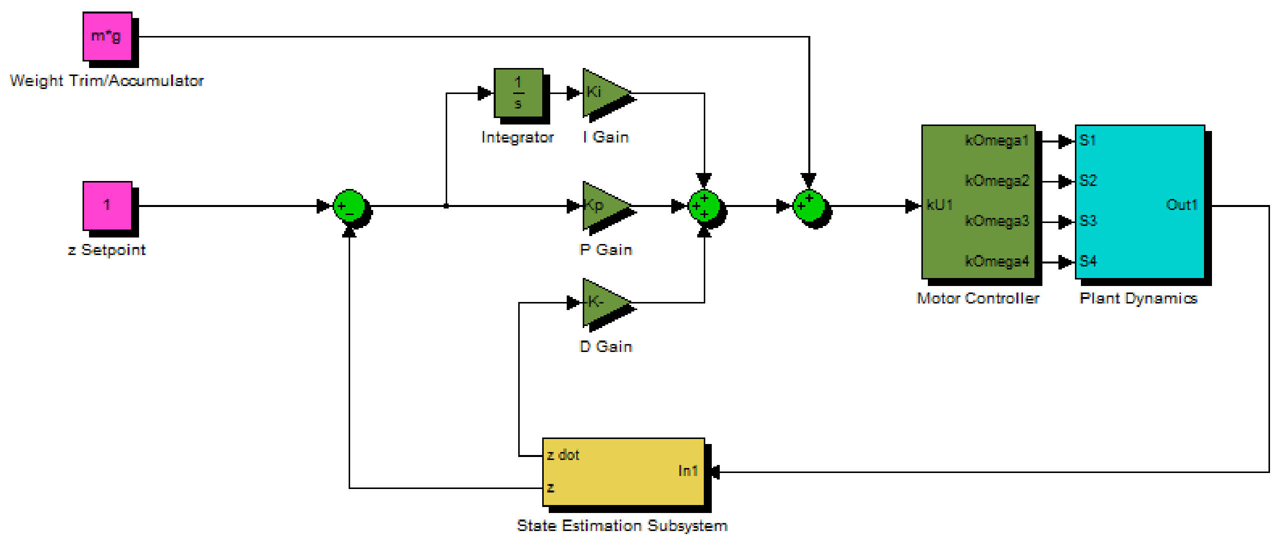


Figure 3.28 - Altitude Control Loop

The remaining parameter of the altitude controller is the neutral setpoint which provides the thrust required to support the quadrotor when the error of the control loop is zero.

3.2.2.4 Mode Control Unit

The mode control unit is the interface between the RC equipment, the flight computer control code and the engine speed controllers. An Atmega 328 microprocessor was chosen to enable hardware PWM generation for ESC control and pulse capture using hardware interrupts. Given the limited practical interfaces with the Overo Fire the MCU was also required to use USART communications to send the captured RC pulse widths to the flight computer and to receive the commands generated by the airborne control code. Figure 3.29 is a flowchart of the MCU operation.

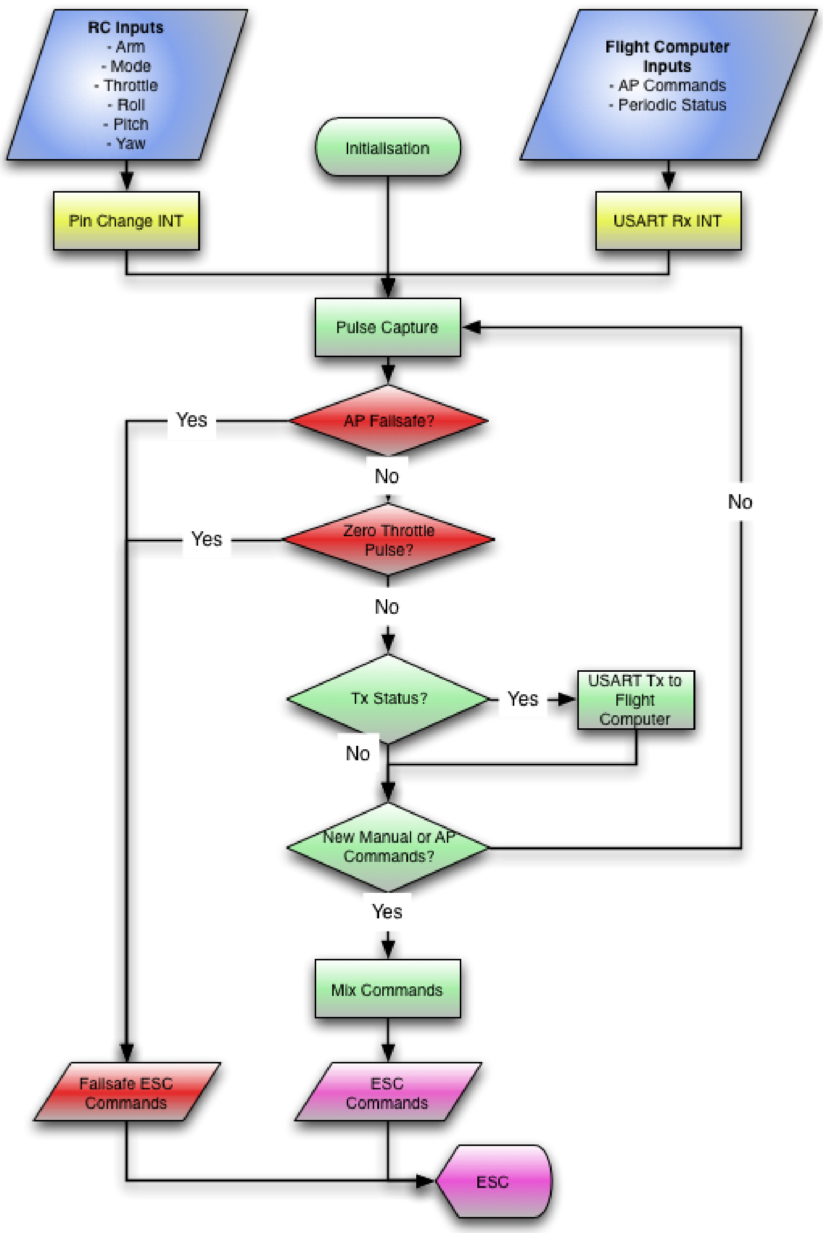


Figure 3.29 - Mode Control Unit Flow Chart

3.2.2.4.1 Pulse Capture

The six channels of the RC equipment each output PWM pulses with a frequency of 50Hz and widths of 1000-2000 micro-seconds. To capture the pulse widths the mode control unit uses the pin change interrupts and the 8-bit Timer2.

Timer 2 has been configured for 125 kHz operation and thus has a resolution of 8 micro-seconds. This was done by selecting a counter-clock prescaler of 64 and a clock prescaler of 2 to reduce the hardware oscillator from 16MHz to 8MHz for the whole chip then to 125kHz for

Timer2. Timer overflows are counted using the overflow interrupt service request (ISR). Information from Timer2 is therefore used to record system time in micro-seconds where

$$micro() = \frac{64}{8} \times \left(Timer2 + (OverflowCount \times 256)\right)$$

This is used to calculate the time between pin changes and thus the pulse widths as shown in the pin change interrupt service routine below.

```
/**
 * @brief Interrupt called due to changes on any of the channel ports
 *
 * Determines those channels changed, if they were rising or falling
 * and logs the time that these changed.
 */
ISR(PCINT1_vect)
{
  static uint8_t previousPINC = 0;
  uint8_t changedPINC = 0;
  uint8_t i = 0;
  uint16_t tempPulse = 0;

  // Time Changed
  uint32_t timeChanged = micro();

  // Flag the Changes
  changedPINC = PINC ^ previousPINC;

  // Determine Falling or Rising Edge
  for (i = 0; i < NUM_CHANNELS; ++i)
  {
    if (BRS(changedPINC,i))
    {
      if (BRS(previousPINC,i)) // fallen
      {
        if(inputChannel[i].isHigh) // got edge
        {
          newRC = 1;
          inputChannel[i].isHigh = 0;
          tempPulse = timeChanged - inputChannel[i].startTime;
          if ((tempPulse > PC_PWM_MIN) && (tempPulse < PC_PWM_MAX)) // update only
with valid pulse
          {
            inputChannel[i].measuredPulseWidth = tempPulse;
          }
        }
      }
      else // risen
      {
        inputChannel[i].isHigh = 1;
        inputChannel[i].startTime = timeChanged;
      }
    }
  }
  // Update Previous
  previousPINC = PINC;

}
```

Code Segment 3-1 – Pulse Capture ISR

3.2.2.4.2 USART Communications and Flight Computer Interface

The MCU USART is required to forward the commanded RC channels to the flight computer for the purpose of flight control. It is also used to receive the required throttle, roll, pitch and yaw control loop outputs. For forwarding commands the MCU Serial library of the airborne software is used by the flight computer to send a three character header representing a request for the periodic MCU data. The USART ISR flags this request and the MCU responds by sending the flight computer the mode the MCU is currently in (Manual, Augmented or Autopilot based on the combination of RC transmitter switch position) and the RC commands as captured from the receiver. When the MCU is receiving control commands it accepts a sequence of nine bytes; the first three being the message header; the following five being the autopilot mode (from the GCS operator) and the control loop throttle, roll pitch and yaw commands; and the remaining byte representing the end of command packet.

To achieve data receipt and transmission the USART is configured at a baud rate of 57600 (with a baud error of 3.5%). It was found that higher baud rates, although necessary for higher update rates, could not be used with the a 16MHz hardware crystal due to baud errors between the flight computer and the MCU. The baud rate error is reduced using two stop bits and the avoidance of parity.

Note that the MCU has commands from the flight computer and the RC receiver. The MCU mode is used to select which commands are sent to the engines as this enables RC pilot override. The MCU also applies the mixing matrix outlined previously if the attitude control is being performed without the hardware gyro; otherwise the RC pilot inputs will be passed through to the correct gyro channels.

3.2.2.4.3 PWM Generation

Each of the four quadrotor ESCs are controlled by pulse width modulated signals of pulse widths 1000-2000μs. The frequency of these pulses in an RC system is often 50Hz however for quadrotor control it was desired to have a frequency as close to 400 Hz as possible. The two remaining 8-bit timers on the Atmega 328 were used to generate the four separate output pulses required for each of the four outputs.

To avoid software processing to generate the pulses, the timers are configured in phase correct pulse generation mode. The choice of prescaler, N, was set at 64 to enable a frequency of $245.1\ Hz$.

$$f_{PWM} = \frac{f_{CLK}}{N \times 510} = \frac{8 \times 10^6}{64 \times 510} = 245.1\ Hz$$

The pulse widths, t_h, are then controlled by modifying the four output compare register values (OCR0A, OCR0B, OCR1A, OCR1B) using the equation below

$$OCRxn = \frac{t_h}{2 \times \frac{N}{f_{CLK}}}$$

For example, the minimum and maximum pulse widths of 1000-2000μs correspond to register values of 62.5 and 125 respectively. Using the hardware generation scheme avoids processing time however it does the choice of prescaler and hardware crystal leads to a PWM resolution limit of 16μs with an 8-bit timer without using input capture. Initial testing did not suggest this limitation was a concern for quadrotor control however its effects did become apparent.

3.2.2.4.4 Mode Indicator

The mode control unit uses the two, two-way switches of the Dx6i RC transmitter to switch between the pure manual (red LED), augmented (blue LED) or autopilot (green LED) modes of operation. In this way the RC pilot is ensured final control over the flight mode of the platform without regard for the flight computer USART connection. The active control loop selections of the GCS do create different combinations of the passed through RC commands. For example if the roll and pitch control loops are not engaged on the GCS then the RC pilot will still have control over them regardless of the MCU mode. In this respect the MCU mode is essentially a means of giving the RC pilot’s consent to the flight control.

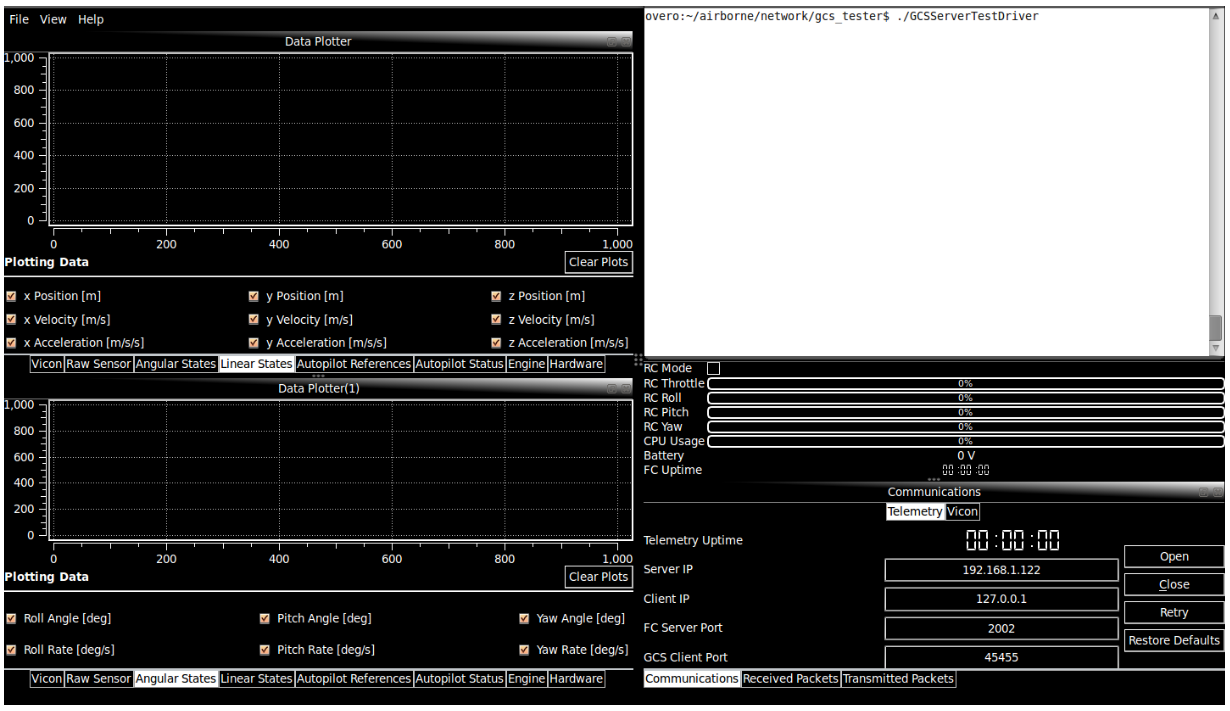
Chapter 4 Testing and Integration

This chapter describes the testing and integration stage of the systems engineering process as applied to the GCS and flight control subsystems. The presentation of this stage is restricted by the iterative design process and the linear thesis structure therefore results of this stage already applied in the subsystem design process are not rediscussed and the focus is on acceptance testing.

4.1 GCS Testing

The testing of the GCS revolved around GCS data transmission and receipt without the need for the entire airborne software system. Instead data was transmitted and received between the GCS and a test application on the flight computer using the airborne UDP network library.

It is important to consider the conditions under which the GCS screenshots in this section were taken and what these represent. Figure 4.1 shows the GCS running on the local Ubuntu computer with a terminal showing the secure shell (SSH) connection to the GCS test application about to be executed. Note the selection of the Overo's IP address and server port in the GCS Communications widget.UDP data transmission was initiated when the test application was started and the GCS "Start" button in the communications widget was pressed.



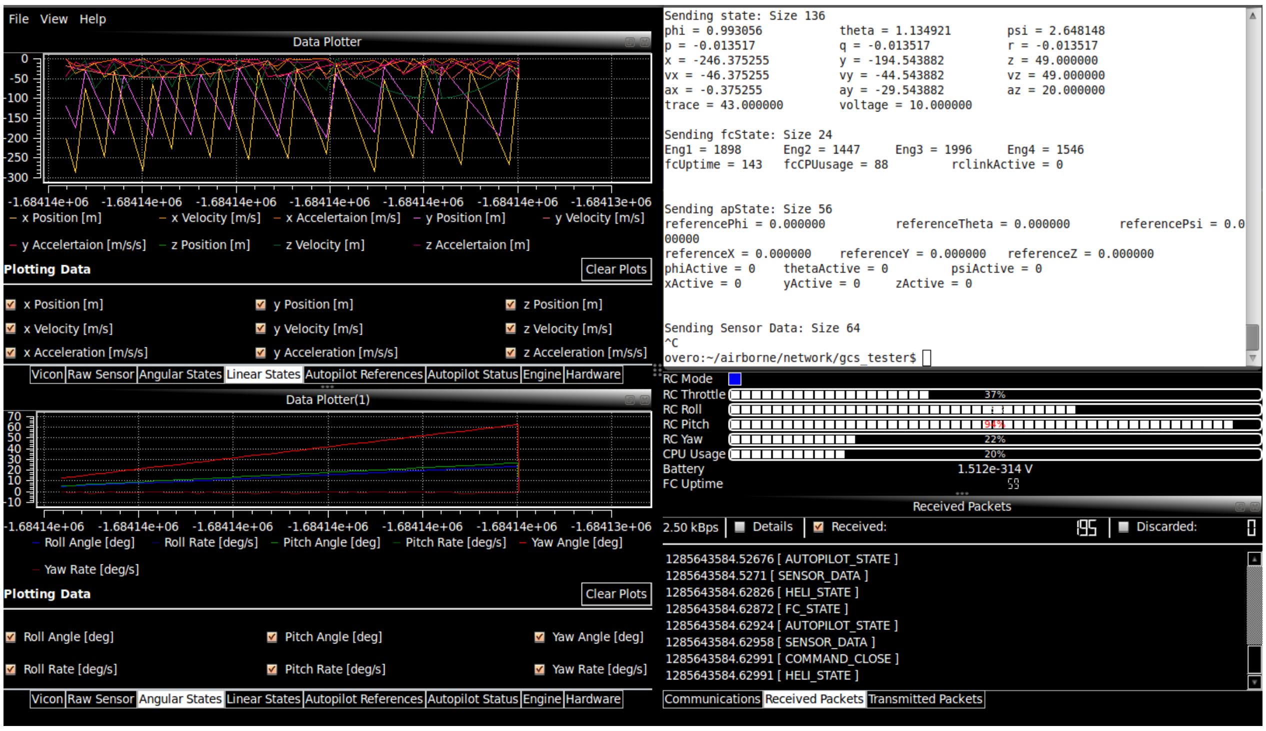
```
overo:~/airborne/network/gcs_tester$ ./GCSServerTestDriver
```

RC Mode
RC Throttle 0%
RC Roll 0%
RC Pitch 0%
RC Yaw 0%
CPU Usage 0%
Battery 0 V
FC Uptime 00:00:00

Telemetry Uptime 00:00:00
Server IP 192.168.1.122
Client IP 127.0.0.1
FC Server Port 2002
GCS Client Port 45455

Figure 4.1 - Remote SSH Connection to Overo and GCS Read for Connection

Figure 4.2 is a screenshot comparing the GCS and GCS test application outputs for linear and angular state data. The received console was also updating in time to display the type and number of packets received. There were no discarded packets and the data can be seen to match that in the SSH connection terminal.



```
Sending state: Size 136
phi = 0.993056          theta = 1.134921        psi = 2.648148
p = -0.013517           q = -0.013517           r = -0.013517
x = -246.375255         y = -194.543882         z = 49.000000
vx = -46.375255         vy = -44.543882         vz = 49.000000
ax = -0.375255          ay = -29.543882         az = 20.000000
trace = 43.000000       voltage = 10.000000

Sending fcState: Size 24
Eng1 = 1898      Eng2 = 1447     Eng3 = 1996     Eng4 = 1546
fcUptime = 143   fcCPUusage = 88         rclinkActive = 0


Sending apState: Size 56
referencePhi = 0.000000          referenceTheta = 0.000000       referencePsi = 0.0
00000
referenceX = 0.000000    referenceY = 0.000000   referenceZ = 0.000000
phiActive = 0    thetaActive = 0         psiActive = 0
xActive = 0      yActive = 0     zActive = 0


Sending Sensor Data: Size 64
^C
overo:~/airborne/network/gcs_tester$
```

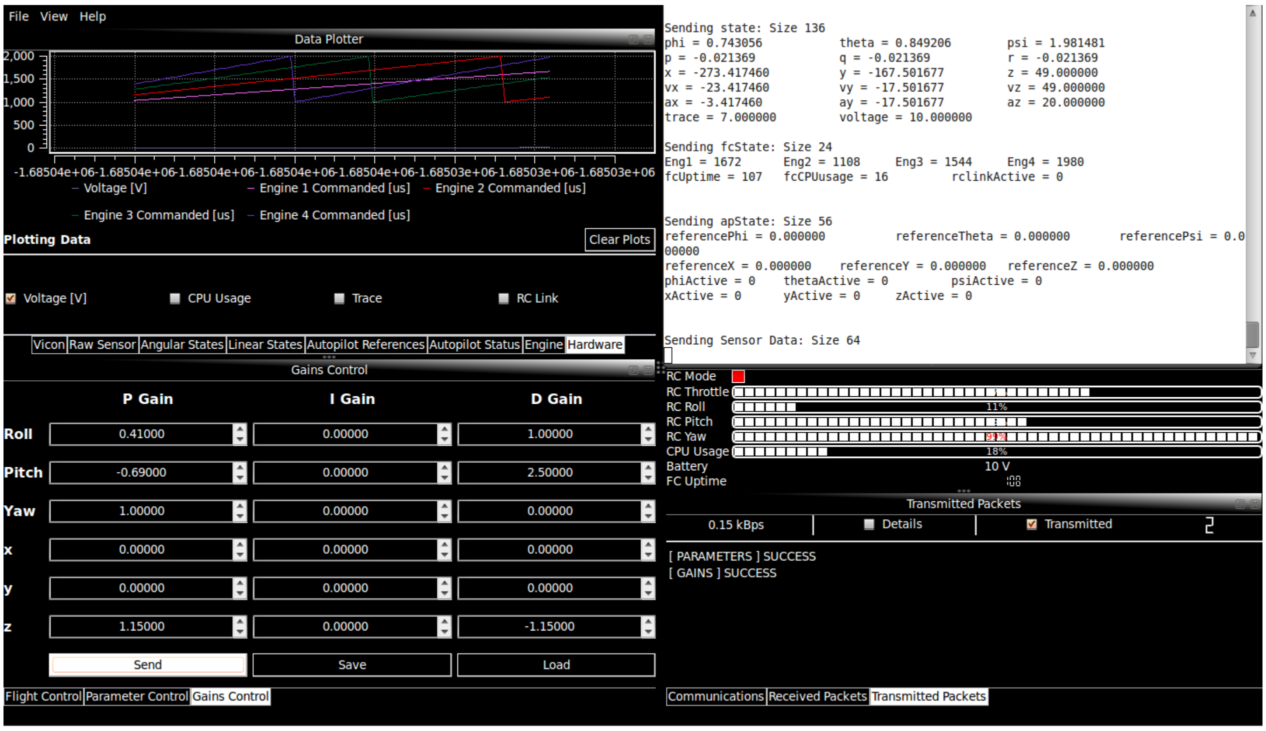
RC Mode 🟦
RC Throttle 37%
RC Roll
RC Pitch 94%
RC Yaw 22%
CPU Usage 20%
Battery 1.512e-314 V
FC Uptime 53

2.50 kBps | Details | ✓ Received: 195 | Discarded: 0

```
1285643584.52676 [ AUTOPILOT_STATE ]
1285643584.5271 [ SENSOR_DATA ]
1285643584.62826 [ HELI_STATE ]
1285643584.62872 [ FC_STATE ]
1285643584.62924 [ AUTOPILOT_STATE ]
1285643584.62958 [ SENSOR_DATA ]
1285643584.62991 [ COMMAND_CLOSE ]
1285643584.62991 [ HELI_STATE ]
```

Figure 4.2–Helicopter State Data in the Data Plotter Widgets

Figure 4.3 and Figure 4.4 are similar demonstrations of the GCS's ability to receive autopilot status data and RC commands using UDP. The data in the data plotter, system status and flight control widgets can be seen to agree with that sent from the test application. It is also important to the note the successful transmission messages in the Transmitted Packets terminal. The GCS considered the gains and parameters properly received as the UDP library has implemented acknowledgement messages. Autopilot mode configuration can also be seen to have occurred as the GCS Flight Control widget was displaying active lights on the loops ticked as active. The lights are green based only on the periodic autopilot state UDP packets, which were clearly being received.



```
Sending state: Size 136
phi = 0.743056          theta = 0.849206        psi = 1.981481
p = -0.021369           q = -0.021369           r = -0.021369
x = -273.417460         y = -167.501677         z = 49.000000
vx = -23.417460         vy = -17.501677         vz = 49.000000
ax = -3.417460          ay = -17.501677         az = 20.000000
trace = 7.000000        voltage = 10.000000

Sending fcState: Size 24
Eng1 = 1672     Eng2 = 1108     Eng3 = 1544     Eng4 = 1980
fcUptime = 107  fcCPUusage = 16         rclinkActive = 0


Sending apState: Size 56
referencePhi = 0.000000         referenceTheta = 0.000000       referencePsi = 0.0
00000
referenceX = 0.000000   referenceY = 0.000000   referenceZ = 0.000000
phiActive = 0   thetaActive = 0         psiActive = 0
xActive = 0     yActive = 0     zActive = 0


Sending Sensor Data: Size 64
```

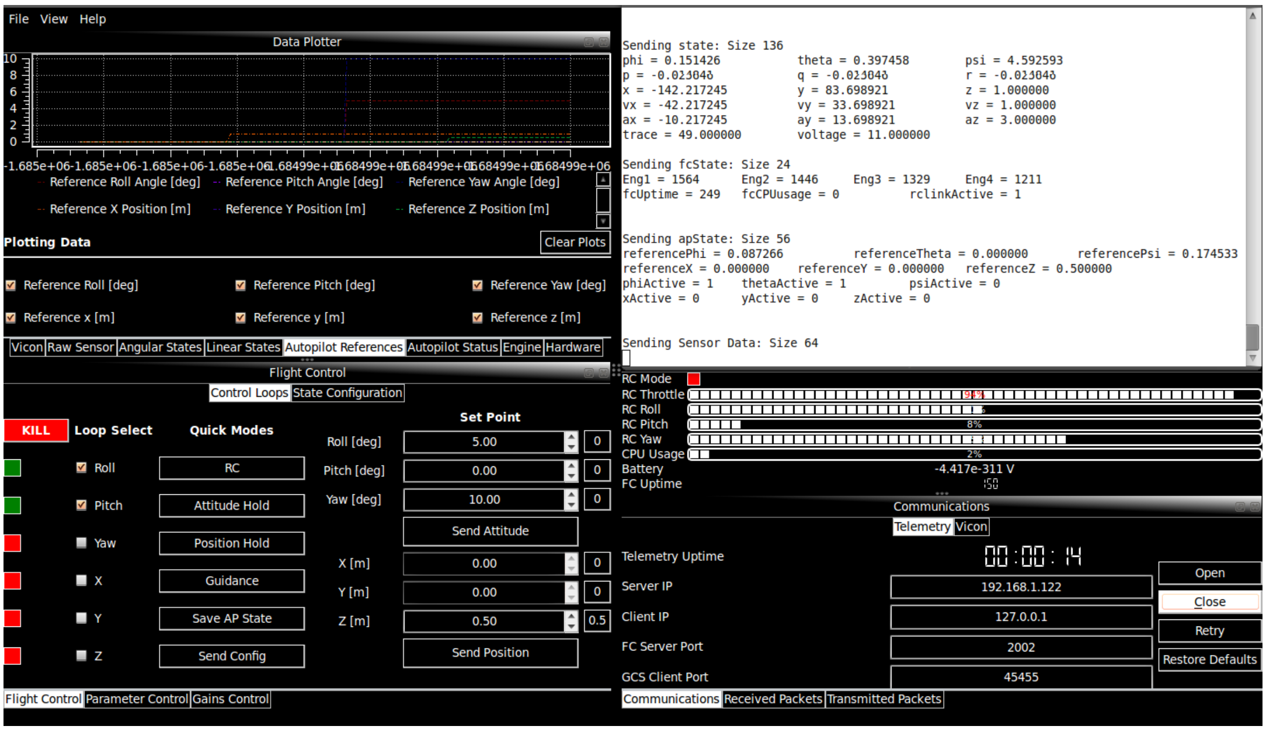
	P Gain	I Gain	D Gain
Roll	0.41000	0.00000	1.00000
Pitch	-0.69000	0.00000	2.50000
Yaw	1.00000	0.00000	0.00000
x	0.00000	0.00000	0.00000
y	0.00000	0.00000	0.00000
z	1.15000	0.00000	-1.15000

RC Roll 11%, RC Yaw 99%, CPU Usage 18%, Battery 10 V, FC Uptime 108

0.15 kBps

```
[ PARAMETERS ] SUCCESS
[ GAINS ] SUCCESS
```

Figure 4.3 - Flight Computer State in Data Plotter and System Status Widget with changed parameter and gains



```
Sending state: Size 136
phi = 0.151426          theta = 0.397458        psi = 4.592593
p = -0.023048           q = -0.023048           r = -0.023048
x = -142.217245         y = 83.698921           z = 1.000000
vx = -42.217245         vy = 33.698921          vz = 1.000000
ax = -10.217245         ay = 13.698921          az = 3.000000
trace = 49.000000       voltage = 11.000000

Sending fcState: Size 24
Eng1 = 1564     Eng2 = 1446     Eng3 = 1329     Eng4 = 1211
fcUptime = 249  fcCPUusage = 0          rclinkActive = 1


Sending apState: Size 56
referencePhi = 0.087266         referenceTheta = 0.000000       referencePsi = 0.174533
referenceX = 0.000000   referenceY = 0.000000   referenceZ = 0.500000
phiActive = 1   thetaActive = 1         psiActive = 0
xActive = 0     yActive = 0     zActive = 0


Sending Sensor Data: Size 64
```

Loop Select		Set Point		
🟩	✓ Roll	Roll [deg]	5.00	0
🟩	✓ Pitch	Pitch [deg]	0.00	0
🟥	Yaw	Yaw [deg]	10.00	0
🟥	X	X [m]	0.00	0
🟥	Y	Y [m]	0.00	0
🟥	Z	Z [m]	0.50	0.5

RC Throttle 94%, RC Pitch 8%, CPU Usage 2%, Battery -4.417e-311 V, FC Uptime 150

Telemetry Uptime	00:00:14
Server IP	192.168.1.122
Client IP	127.0.0.1
FC Server Port	2002
GCS Client Port	45455

Figure 4.4 - Flight Control Widget reflecting change of Active Control Loops and set points

Figure 4.5, Figure 4.6 and Figure 4.7 represent the same data that has been previously displayed in the GCS software of Figure 4.2, Figure 4.3 and Figure 4.4 respectively. It has been established that the received data is consistent with that originally sent. The logged data is identical to the received data thus demonstrating the GCS is logging airborne system data.

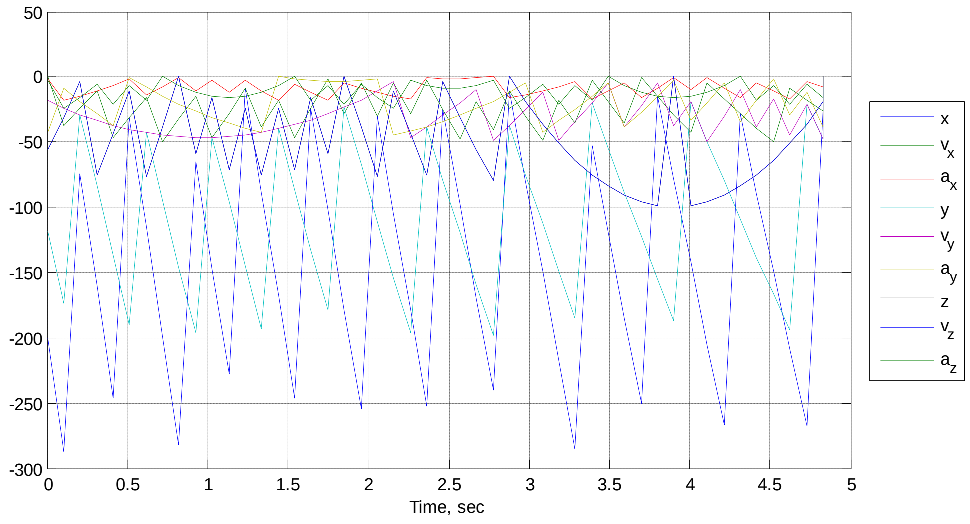


Figure 4.5–MATLAB Plot of GCS Logged Helicopter State Data from Figure 4.2

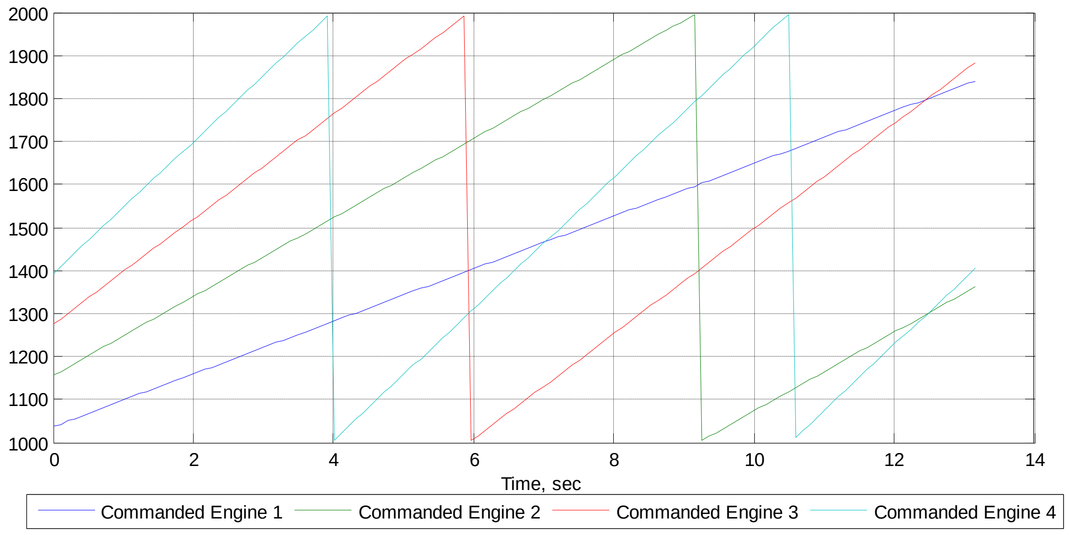


Figure 4.6 - MATLAB Plot of GCS Logged Flight Computer State Data from Figure 4.3

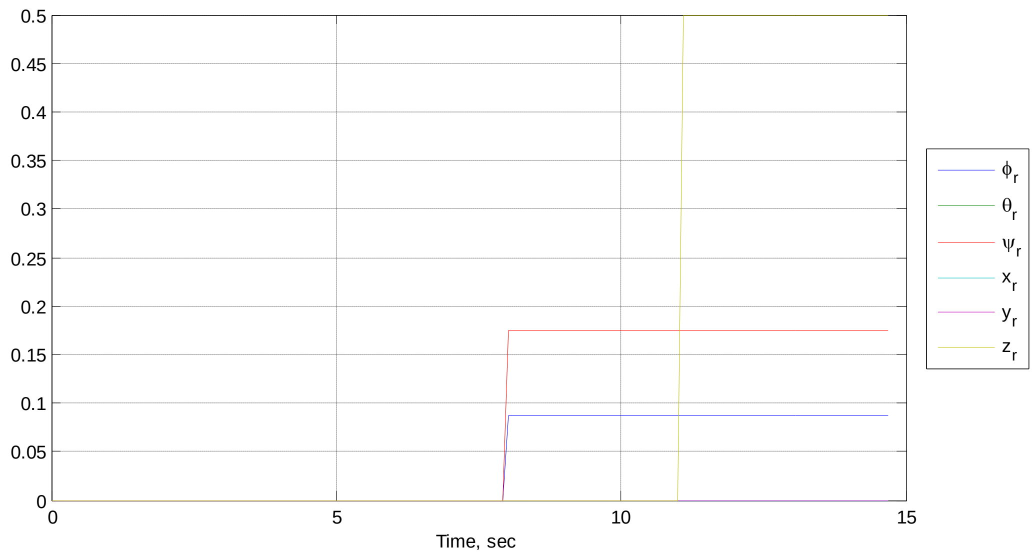


Figure 4.7 - MATLAB Plot of GCS Logged Autopilot States from Figure 4.4

The tests results establish that the GCS is capable of configuring the autopilot and receiving and logging helicopter, autopilot and flight computer state data. Selecting active control loops, updating controller gains and parameters has been demonstrated thus the GCS passedAT-02 and AT-08 and met the requirements SR-B-02 and SR-B-08. Successful display of the system state data fulfilled AT-09, AT-17 and AT-18 thus the GCS met SR-B-09, SR-D-07 and SR-D-08. Inspection of the log data also led to the GCS passing AT-02 and AT-08 and therefore meeting SR-B-02 and SR-B-08.

4.2 Control Update Rate Testing

Testing the AHNS flight control subsystem was a multistage process with measurement of the control update rates forming the first stage in acceptance testing. The control update rate test was performed using flight test data logged from the control and mode control unit loops. The flight computer logs were used to calculate the mean control update rate.

Figure 4.8 is the plot generated from the flight computer control loop update rate log file. It is a representation of the time elapsed between successive control loop updates on the flight computer. The update rate clearly exhibits a large variance suggesting the time between control updates is not fixed with instantaneous update rates dropping below 50 Hz. A mean update rate of 92.7 Hz for the control loop can however be calculated. Given the flight computer relies on the Linux kernel provided by OpenEmbedded it is not possible to force thread execution to 50 Hz as this requires a real-time kernel or a microcontroller with interrupts.

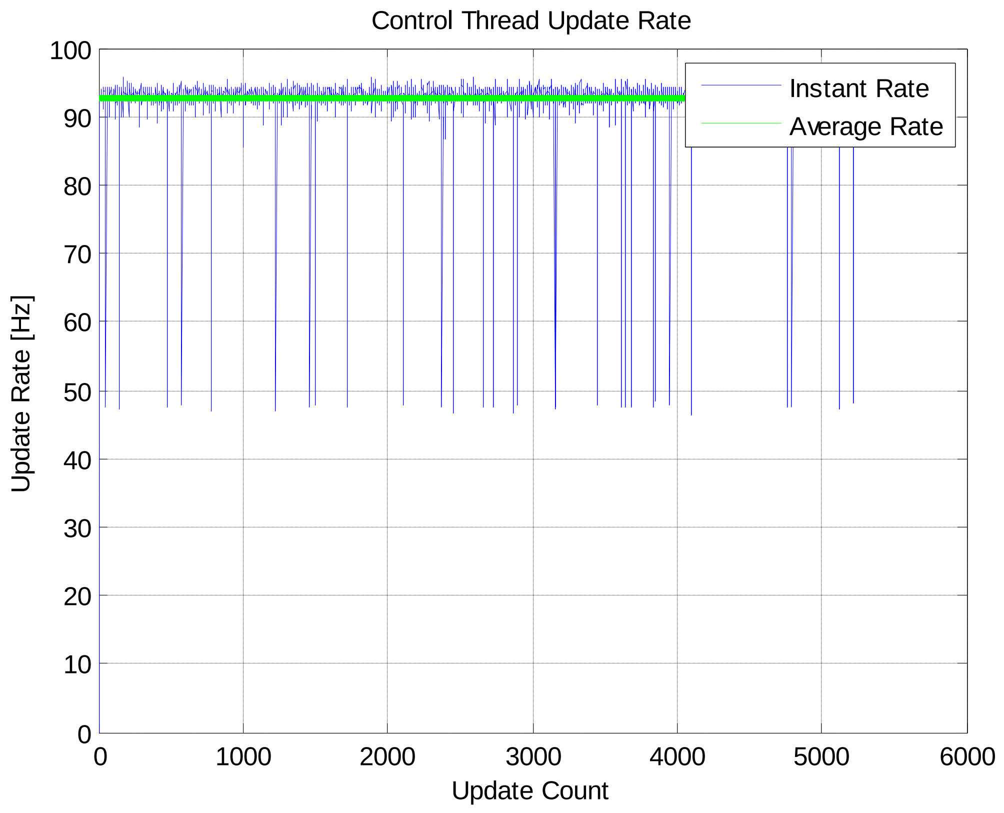


Figure 4.8 - Control Loop Update Rate

The control system update rate is also determined by the MCU thread execution rate. The mode control unit update rate is important, as this is the rate the engine PWM values are updated. Figure 4.9 demonstrates a variance in instantaneous MCU execution rate with a variance towards lower update rates. The mean update rate of 221.26Hz for the MCU does not reduce the update rate of the control system below that of the control thread. The large update variance does suggest the engine command update rates will not be uniform, forming a possible source of dynamic disturbances.

The use of a USART link with 57600 baud rate to the MCU is a factor in the thread update rate. During operation a periodic message request is sent from the flight computer; 3 bytes followed by the 6 bytes of data from the MCU. The updated commands are also sent as 8 bytes of data. The total transmission is therefore 17 bytes or 136 bits which at a baud rate of 57600 limits the maximum MCU hardware update to less than 450 Hz. The choice of baud rate however was limited based on the hardware crystal for the MCU and could not be increased without error introduction.

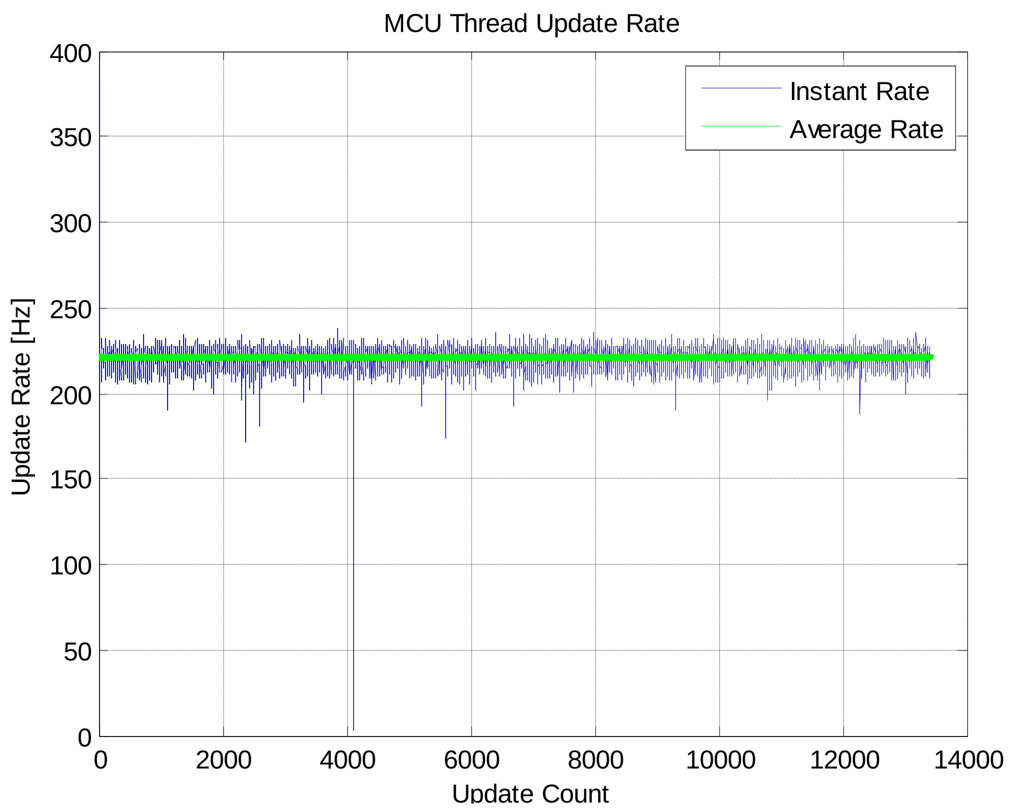


Figure 4.9 - MCU Loop Update Rate

The results of this initial control system test concluded the control and mode control unit threads have mean update rates of 92.7Hz and 221.6Hz respectively. The control update from the flight computer was therefore the slowest of these rates at 92.7 Hz. As this mean was greater than the required 50 Hz in SR-B-03, AT-03 was passed and the requirement met. It should be noted that both update rates were variable and often strayed below these means. A solution to force a constant execution rate however is to use an interrupt based operating system or microprocessor for the flight computer.

4.3 Attitude Hold Flight Testing

Attitude control testing was undertaken in increments with testing on a test rig, bungee rope and eventually unrestricted testing. By the time the final attitude controller based on filtered IMU rates was tested there was little need to calibrate the control directions in the testing apparatus and instead the focus was on unrestricted gain tuning.

The unrestricted attitude or augmented control test procedure consisted of flight computer setup, communication establishment, an engine test, yaw control tuning and finally testing of the roll and pitch control loops. During flight computer setup it was necessary to connect to the Overo using SSH. The IMU could then be initialised as powering it down erased the previous settings. With the Overo and all onboard hardware properly connected, the main application of the flight computer could be executed and the GCS used to connect to it. With telemetry being received, the MCU was initialised by moving all commands to their 1000 micro-second positions; an engine test was then performed to ensure the GCS was receiving and displaying the RC pilot commands.

Tuning the yaw control loop required the GCS operator to engage only the yaw control loop in the Flight Control GCS widget. The pilot therefore remained in control of the throttle, roll and pitch commands when in either the stability augmented (blue) or autopilot (green) MCU modes. Using only the throttle the pilot could raise the platform to a point where it began yawing due to controller differences. The yaw rate initially was quite severe and too fast for RC pilot correction. The yaw proportional control gain was changed in 0.25 increments until the yaw rate was slowed and indeed the yaw motion observed to be simply divergence. Note that derivative and integral gain were not considered and negative proportional gain was observed to lead to a fast, divergent yaw rate.

Tuning the roll and pitch controllers was done by enabling yaw, roll and pitch control loops on the GCS and leaving the RC pilot in command of the throttle. The active control loops still relied on the RC pilot to provide the angular rate setpoints the pilot therefore applied corrective command inputs and the responsiveness of the platform to these were observed. Initial observations involving proportional roll and pitch gain suggested the platform was correcting at a slower than desired rate. Increasing the magnitude of the proportional gain resulted in severe oscillations with limited improvements in response speed. Setting the proportional gain at the maximum for no oscillations and increasing the derivative gains proved to yield controllable roll and pitch. The final set of gains decided on are shown in Table 4.1.

Table 4.1 - PID Gains for Roll, Pitch and Yaw

	Proportional Gain	Integral Gain	Derivative Gain
Roll	0.41	0	1
Pitch	-0.69	0	2.5
Yaw	1	0	0

A measure of stability augmentation performance was also proposed to be the controller delay. This was measured as the time between RC pilot angular rate commands and the plant reaching these rates. Figure 4.10 and Figure 4.11 provide some data on the controller delay. The reference control rates provided by the RC pilot lead the measured rates by between 0.1 and 0.2 seconds. This provided a controllable quadrotor and thus acted as stability augmentation.

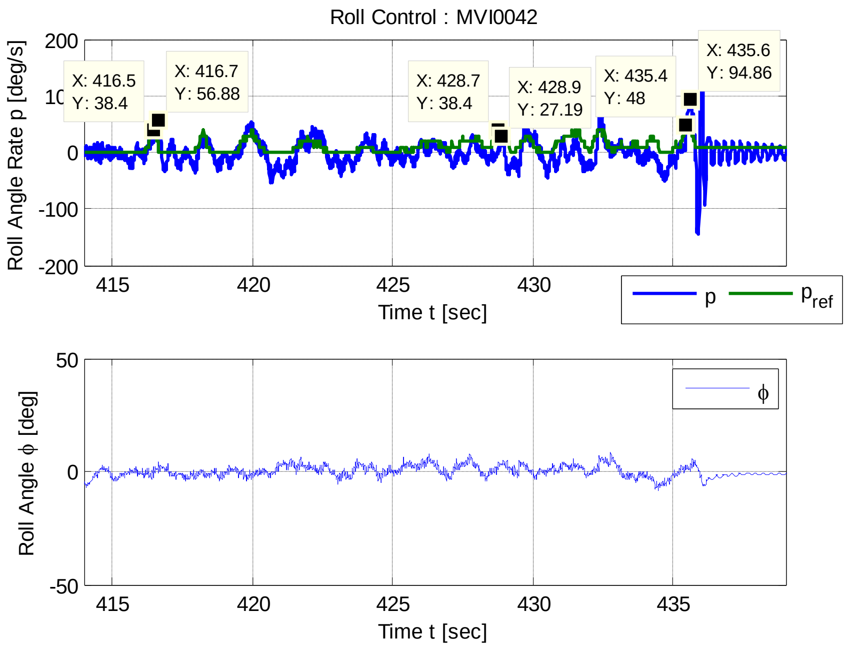


Figure 4.10 - Roll Controller Logs, Kp = 0.41 Kd = 1

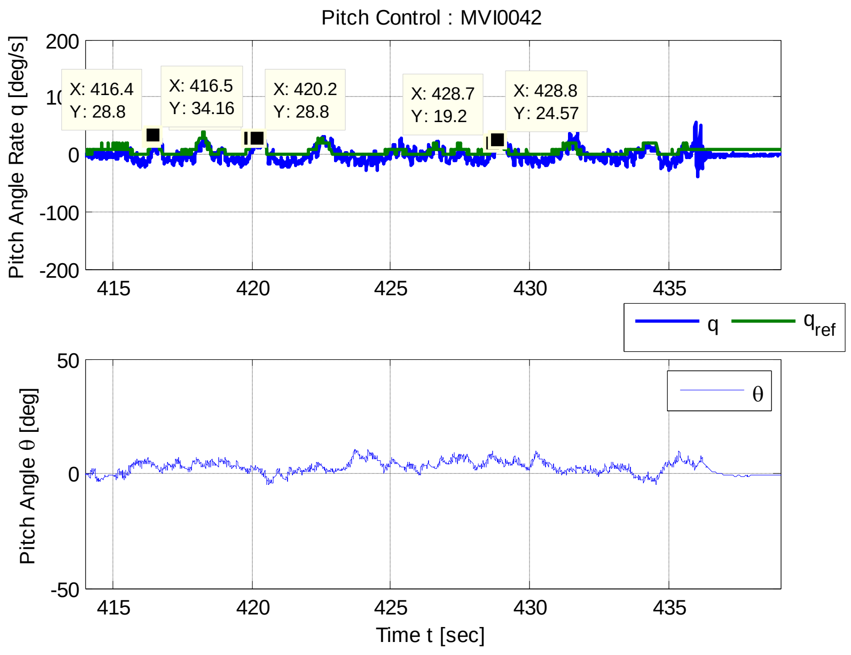


Figure 4.11 - Pitch Controller Logs, Kp = -0.69 Kd = 2.5

The platform oscillates after following the pilot's commands for a period of time. This behaviour is attributable to the mounting of the avionics payload under the quadrotor and thus the movement of the centre of gravity away from the position of the IMU, which is where the angular rates are measured.

4.4 Altitude Hold Flight Testing

To test the position controller, the cascaded attitude control loops needed to provide the RC pilot with a stable controller to enable aggressive corrections in case of mishaps. Testing apparatus had already shown unrestricted platform flight testing was the only possible means of developing useable controller gains. The stability augmentation was achieved however a proven and more thoroughly test attitude control solution was chosen to form the basis of position control testing. Indeed the roll, pitch and yaw attitude controllers (dotted) in Figure 3.27 were substituted for a commercial solution (the three axis RC gyro of Figure 4.12). Note that even with this solution unrestricted position hold in x, y was not attempted due to time constraints and the risk of platform destruction.

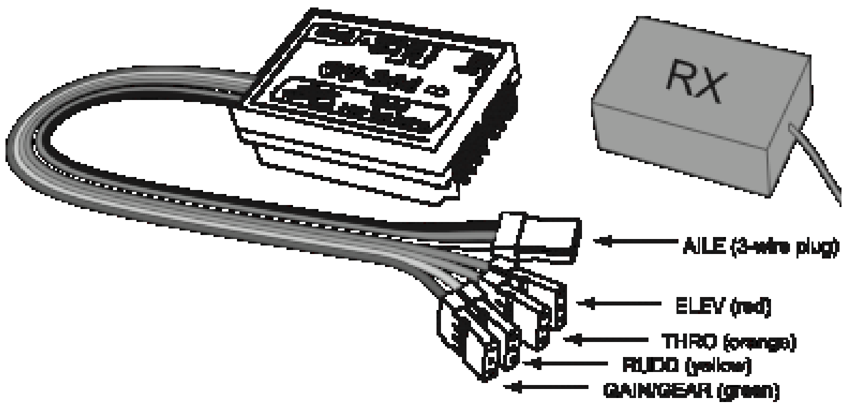


Figure 4.12 - GU-344 3 Axis Gyro [16]

4.4.1 Vicon Altitude Hold

Initial altitude control tests were undertaken using Vicon to measure the platform's altitude and vertical velocity. The flight test procedure was similar to that followed for attitude control testing. The flight computer was engaged and connected to using SSH however the IMU was not configured as the attitude control was handled by the hardware rate gyro. The MCU was instead programmed using an AVR ISP with a version of software that did not provide control mixing. After running the main flight computer the GCS operator was free to connect to the airborne system and then to Vicon. With the Vicon data being forwarded to the flight computer, the operator could then set the altitude control loop to active thus passing the RC roll, pitch and yaw commands directly to the gyro with the altitude throttle commands from the flight control. Engine tests were performed to ensure MCU operation before the pilot engaged autopilot mode giving the flight control complete command over the throttle.

Gain tuning was difficult and at times hazardous as the oscillations and overshoots would take the platform to heights nearing the limits of Vicon coverage. When this occurred the RC pilot was required to "catch" the quadrotor using throttle as there is a discontinuity between autopilot and manual throttle commands. After iterative attempts the proportional gain before the response became unstable was found. The controller's response to a step altitude input is shown in Figure 4.13. The divergent steady state error of this response was originally considered correctable with integral gain however this served to destabilise the response, suggesting an error in implementation or design.

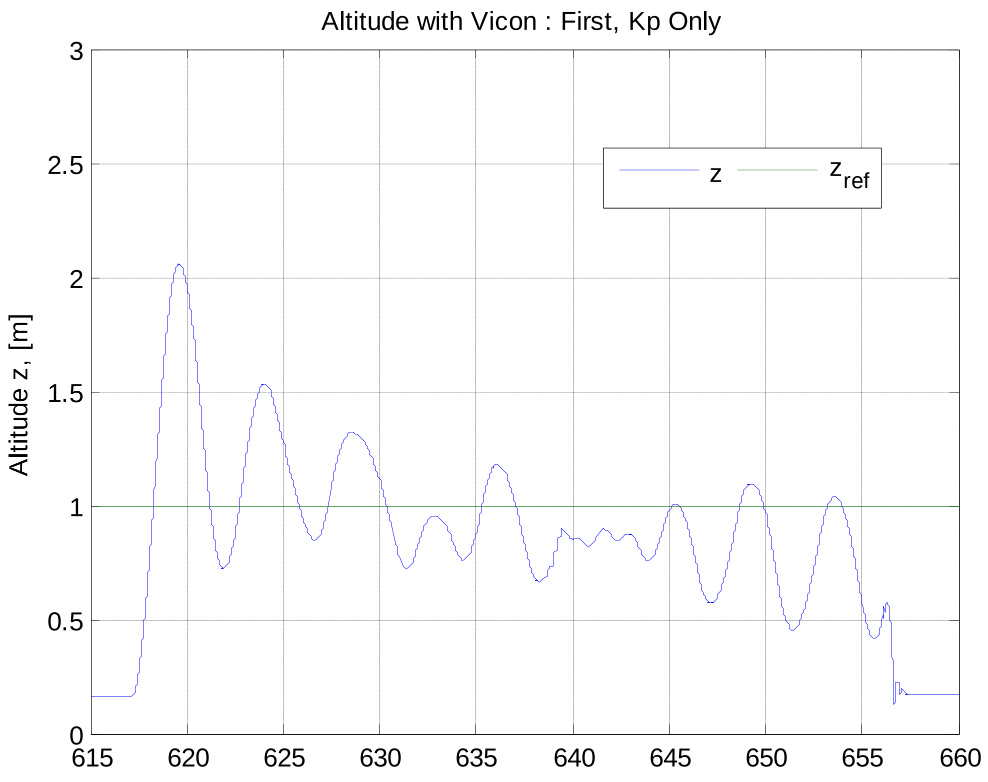


Figure 4.13 - Effect of Dropping Battery Voltage on Attitude Hold with no trim adjustment

After consideration of the controller it was realised that the trim value to counter the platform's weight need to be adjusted as the battery was discharged. Ideally this thrust force is constant however as the battery voltage dropped, so too did the thrust provided by the engine at a fixed PWM value. The result was the slowly lowering altitude hold of Figure 4.13. To test the effect trim increase over time would have, the GCS operator manually increased it in the flight results shown in Figure 4.14.

Sustained altitude hold therefore required the controller to be implemented with autonomous update of this trim. The approach suggested in [17] where an accumulator was used to count the trim up when below height and down when above height was tried and is implemented. The increment amount and how to control ascent rate was however difficult to tune. Configuring these parameters through testing resulted in platform damage, leading to the conclusion that further testing safety measures should be devised before fully autonomous altitude control can be achieved. The final gains and parameters used in Vicon altitude control are listed in Table 4.2.

Table 4.2 - Altitude Control Parameters and Gains

	Proportional Gain	Integral Gain	Derivative Gain	Neutral
Z-axis	1.15	0	-1.15	22.8

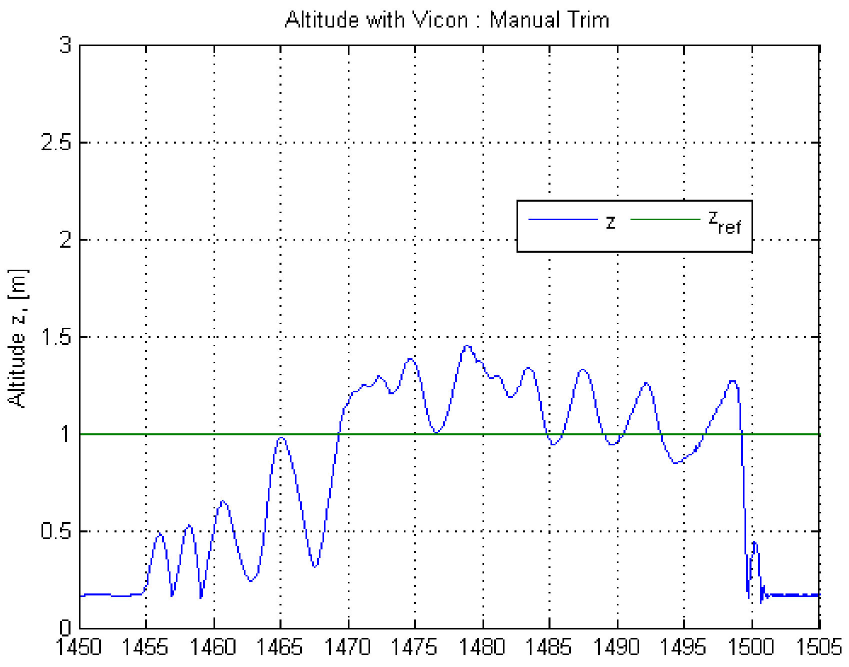


Figure 4.14 - Vicon Altitude Hold with Manual Trim Adjustment

4.4.2 Ultrasonic Altitude Hold

Altitude hold was tested using onboard ultrasonic sensor to measure the vertical position and velocity in an identical manner to when Vicon was used. Due to a more confined test area the altitude setpoint was however reduced. Figure 4.15 shows the response when using the same parameters as Vicon. Although gain tuning was re-performed, the gains were ultimately similar to those previously found.

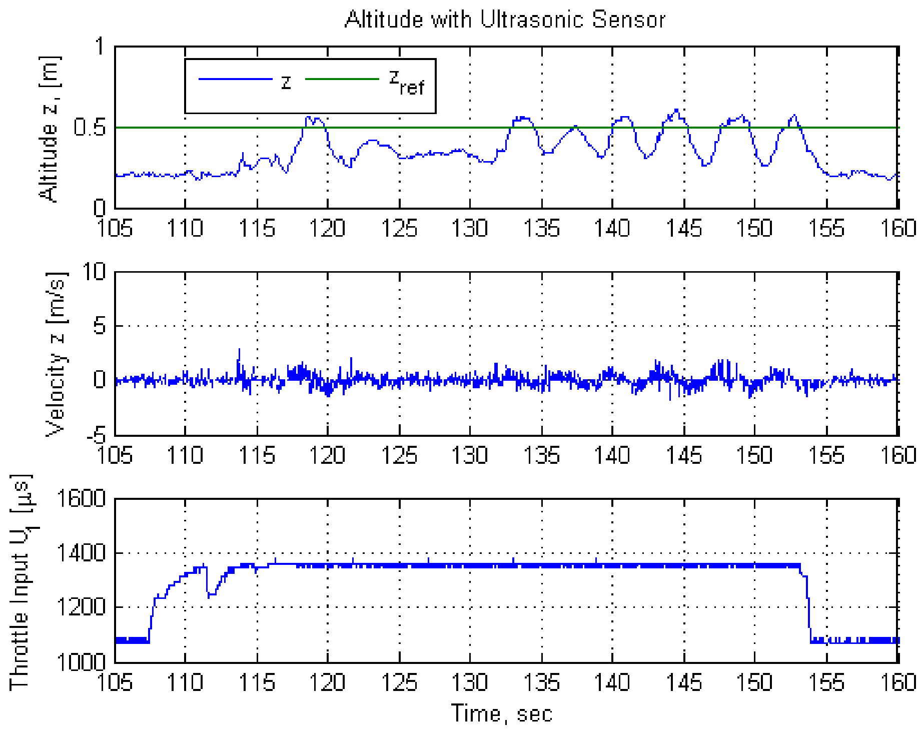


Figure 4.15 - Ultrasonic Altitude Hold with no trimming

Compared to altitude hold with Vicon, the ultrasonic hold is similar in its major features. The long term reducing altitude trend can be observed however as the time of flight is relatively short this is not a primary concern. The oscillations however, which are also present in the Vicon response, suggest that regardless of gain choice the controller implementation cannot control these dynamics. Consideration of the throttle graph demonstrates a possible cause. As explained in the MCU design, choice of hardware crystal and PWM frequency limited the pulse generation resolution to 16μs. The throttle commands therefore can only be changed in steps of this resolution, leading to repeated over- and under-corrections in throttle and thus altitude control.

Control performance although successful in maintaining vertical positions and attitude, is therefore limited by hardware interfacing and testing restrictions.

Chapter 5 Conformance Matrix

Table 5.1 is the conformance matrix for the Ground Control Station and Flight Control subsystems. Each completed requirement was tested against an acceptance test whilst those left not completed were not necessarily tested due to perceived testing risk.

Table 5.1 - GCS and Flight Control Conformance Matrix

Number	Definition	Status	Reference Document
SR-B-02	The GCS shall enable autopilot flight mode switching between manual, stability augmented flight, and autonomous station keeping.	Complete	AHNS-2010-GC-TR-001
SR-B-03	The airborne system shall provide control updates at an average rate of 50Hz.	Complete	AHNS-2010-AP-TR-001
SR-B-08	The autopilot system gain and reference parameters shall be updatable in flight using an 802.11g WLAN uplink from the GCS.	Complete	AHNS-2010-GC-TR-001
SR-B-09	The airborne system shall transmit telemetry data including state data to the GCS using 802.11g WLAN.	Complete	AHNS-2010-AP-TR-002
SR-B-10	The autopilot control methodology shall be based on cascaded PID control loops.	Complete	AHNS-2010-AP-DD-001
SR-D-03	The autopilot shall provide stability augmented flight.	Complete	AHNS-2010-SY-TR-001 AHNS-2010-SY-TR-002
SR-D-04	The autopilot shall provide autonomous station keeping capability within a 1 meter cubed volume of a desired position.	Not Complete	AHNS-2010-SY-TR-003 AHNS-2010-SY-TR-004
SR-D-06	The airborne system shall collect avionics system health monitoring information in the form of radio control link status, flight mode status and battery level.	Complete	AHNS-2010-AP-TR-002
SR-D-07	The airborne system shall collect avionics system health monitoring information in the form of radio control link status, flight mode status and battery level.	Complete	AHNS-2010-AP-TR-002
SR-D-08	The GCS shall log all telemetry and uplink data communications.	Complete	AHNS-2010-GC-TR-001
SR-D-09	The airborne system shall receive and process measurement data from the state estimation and localisation sensors; supporting IMU, Camera, and Ultrasonic sensor.	Complete	AHNS-2010-GC-TR-001
SR-D-10	The GCS shall provide display of avionics system health monitoring including telemetry, uplink, radio control link and battery level status read-outs.	Complete	AHNS-2010-GC-TR-001

Chapter 6 Conclusions and Recommendations

The development of the AHNS ground control station and flight control subsystems has been seen to follow the key stages of the system engineering methodology. The completion of all but one system requirement in the subsystems is a testament to methodology and engineering process and tools used. The system requirements lead directly to the conclusion that the GCS high level objective is passed whilst the flight control objective was only partially completed.

The scope of project completion cannot be judged solely on the high level objectives as all system requirements were attempted up to integration and testing. Indeed the project deliverables include a GCS capable of WiFi data transmission and receipt, data logging, control system management and onboard system configuration and a flight control system capable of RC attitude control and altitude control. Risk, rather than technical challenges prevented the completion of the project.

A number of subsystem specific conclusions and recommendations were developed during the project. These could be considered for the purpose of project continuation or as an indication of the final project state.

6.1 Ground Control Station

- Development of the GCS from scratch required a considerable time investment. The effects of such a time outlay where felt until the end of the project as flight testing time was effectively reduced by three months. Although this time was also used in other subsystems it limited the time available for systems testing and control development; arguably the most important aspects of the project. In 2009 AHNS also developed two GUIs from scratch representing more time lost from the development of an autonomous system. Much of the 2009 code was not used due to a major changing in system architecture; specifically the change in communications and control location. It is recommended that future groups avoid making GCS development a project objective and either modify legacy code (of which there are now two proven, available and modular solutions) or rely on MATLAB or pre-built solutions such as the open source QGroundStation.
- If major pieces of software such as a GCS or network library do need to be developed, documentation and unit testing should be considered part of the development stage rather than extra tasks to be completed. It was realised only after the majority of code was implemented that documentation tools such as Doxygen were capable of generating

high quality code documentation. With respect to unit testing, no time was explicitly allocated to test the implementation or develop methods to test individual aspects of the implementation. Although the GCS is stable, without unit testing there is only user experience to support this claim.

- As the software project became large, it became increasingly difficult to add features to the coupled and often confusing Qt signal/slot architecture. This could have been avoided, particularly in the handling of new UDP messages, by using generic programming and centralising more code.

6.2 Flight Control

- Controller development relied on states provided by the state estimation subsystem during initial testing. Development of both subsystems was rapid therefore it was not always possible to have a coordinated understanding of the system's behaviour. Future efforts should focus on developing tested control and state estimation systems prior to their combined flight testing.
- Initial control design was verified using a rigid body quadrotor simulator. In simulations with ideal engine and propeller dynamics and with perfect knowledge of the states, control was a simple exercise in angle control. Ultimately the results from the simulations proved unusable as the pilot needed to correct aircraft drift and the state estimation was not capable of ideal state determination. It was learnt that controller design based on proven controller designs from practical projects are generally superior to those developed with idealised simulations.
- To avoid platform damage, several testing apparatus were created. The use of these was generally only useful in checking the direction of controller corrections as the platform dynamics are affected or restricted. Gains could only be tuned in practical flight tests.
- Tuning position and altitude control proved to be inherently hazardous to equipment, personnel and the platform itself. With the discrediting of the test rigs and bungee apparatus there was insufficient time to reconsider mitigation methods to tune these controllers, their flight testing was therefore discontinued. Measures such as safety nets, foam padding or isolated testing rooms should be considered to enable safer flight testing.
- The effects pulse capture and pulse generation have on controllability of the system should be examined. The current PWM generation resolution of 16 micro-seconds has been seen to offer imprecise control loop updates. That is, in altitude control essentially the controller is either on or off resulting in considerable oscillations. An improved

method of ESC control should therefore be developed. This involves choosing an MCU crystal that enables hardware generated, high frequency PWM signals with a precise resolution. If this is not possible I2C ESCs should be considered or a software based PWM generation scheme used.

- The update rate of the control is limited by choice of MCU baud rate, which is limited by the choice of pulse width capture resolution which is limited by the choice of hardware crystal. To solve the update rate limitations it is suggest the control is moved to the same microprocessor that handles pulse capture so that RC information is not delayed.

References

[1] Massachusetts Institute of Technology. (2009). Aerospace Controls Laboratory. [Online]. Available: http://acl.mit.edu/publications/.

[2] M. Alpen, K. Frick, and J. Horn, "Nonlinear modeling and position control of an industrial quadrotor with on-board attitude control," in *IEEE International Conference on Control and Automation, 2009*, Christchurch, New Zealand, 2009, pp. 2329-2334.

[3] S. Bouabdallah, P. Murrieri, and R. Siegwart, "Design and Control of an Indoor Micro Quadrotor," in *Advances in unmanned aerial vehicles: state of the art and the road to autonomy*, K. P. Valavanis, Ed. New York: Springer-Verlag, 2007.

[4] M. Achtelika, A. Bachrachb, R. Heb, S. Prenticeb, and N. Royb, "Autonomous navigation and exploration of a quadrotor helicopter in GPS-denied indoor environments," 2008.

[5] E. Altug, J. P. Ostrowski, and R. Mahony, "Control of a quadrotor helicopter using visual feedback," in *Proceedings of IEEE International Conference on Robotics and Automation, 2002.*, Washington DC, 2002, vol. 1, pp. 72-77 vol.71.

[6] I. C. Dikmen, A. Arisoy, and H. Temeltas, "Attitude control of a quadrotor," in *International Conference on Recent Advances in Space Technologies, 2009*, Istanbul, Turkey, 2009, pp. 722-727.

[7] AHNS. (2010). ahns10. [Online]. Available: http://code.google.com/p/ahns10.

[8] T. Molloy. (2010). heliconnect10. [Online]. Available: http://code.google.com/p/heliconnect10/wiki/packetmessages.

[9] Nokia Corporation, "QtCreator Cross-Platform Qt IDE," 2009. Available: http://qt.nokia.com/.

[10] P. Deegan, "Developing for the Atmel AVR microcontroller on Linux," *Linux Journal,* vol. 2005, p. 10, 2005.

[11] D. van Heesch. (2004). Doxygen. [Online]. Available: http://www.doxygen.org.

[12] AHNS-2010-SY-SR-001, " AHNS, System Requirements of," 2010.

[13] U. Rathmann (2010). Qwt - Qt Widgets for Technical Applications. [Online]. Available: http://qwt.sourceforge.net/.

[14] S. Bouabdallah, P. Murrieri, and R. Siegwart, "Design and control of an indoor micro quadrotor," in *Proceedings of IEEE International Conference on Robotics and Automation, 2004*, New Orleans, LA, 2004, vol. 5, pp. 4393-4398.

[15] AeroQuad. (2010). AeroQuad – The Open Source Quadcopter. [Online]. Available: http://aeroquad.com/.

[16] GAUI. (2010). GU-344 Gyro. [Online]. Available: http://eng.gaui.com.tw/d981119/html/shopping_view.asp?sn=1028#

[17] D. Gurdan, J. Stumpf, M. Achtelik, K. Doth, G. Hirzinger, and D. Rus, "Energy-efficient autonomous four-rotor flying robot controlled at 1 khz," in *IEEE International Conference on Robotics and Automation*, Roma, Italy, 2007, pp. 361-366.

List of Appendices

None.

The interested reader should check out the source code and Doxygen files at

http://code.google.com/p/ahns10