Name:	**Liam O'Sullivan**
Supervisor:	**Dr Luis Mejias Alvarez**
Project Title:	**Autonomous Helicopter Navigation System – Flight Computer and State Estimation Subsystem**
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

[illegible]

“When you have eliminated the impossible, whatever remains, however improbable, must be the truth”

- Sherlock Holmes (Conan Doyle)

Executive Summary

The Autonomous Helicopter Navigation System (AHNS) 2010 was a final year project conducted at the Queensland University of Technology (QUT). The aim of this project was to develop hardware and software solutions to automate a small quadrotor platform. This automation would allow the quadrotor to navigate indoors where a GPS signal is not available. The platform would have to rely on a combination of inertial and visual measurements onboard the platform to provide appropriate control updates.

The project consisted of four team members where a systems engineering methodology was applied. Six High Level Objectives (HLOs) were identified for the AHNS 2010 project leading to the development of numerous system requirements for each HLO. The author was primarily responsible for two subsystems related to HLO-4 and HLO-3. The first subsystem was the design, development and testing of a flight computer for the onboard platform (HLO-4). The second subsystem was the design, implementation and verification of the state estimation subsystem (HLO-3).

The flight computer was designed and implemented on the Gumstix Overo Fire, which uses a Linux based operating system. A threading approach for the flight computer was used where 5 threads were written and co-ordinated the flow of sensor data throughout the system. This allowed data from all sensors connected to the flight computer to be measured and processed within the required average update rate of 50 Hz. The processed data could then be used to issue control updates allowing the quadrotor platform to become an autonomous vehicle.

The state estimation subsystem was required for the AHNS project to allow the quadrotor platform to issue accurate control updates. 15 states were identified for the quadrotor and needed to be estimated. The attitude state estimation was calculated onboard the platform through 3 separate Kalman filters. These filters utilised the sensor data provided by the SensorDynamics 6 DOF IMU and OS4000 magnetic compass to estimate the attitude of the quadrotor platform. Position and velocity estimates were provided by the Vicon system as well as an ultrasonic sensor that measured the altitude of the quadrotor. Various other filtering variants were required for these states were the exponentially weighted moving average (EWMA) filter was used the most often.

Flight computer subsystem testing revealed that all sensor information was being collected and processed by the flight computer. This occurred well above the average rate of 50Hz, which was required to meet SR-D-05 and SR-D-06.

Three separate acceptance tests were required to verify the performance of the state estimation subsystem. AT-04 was successfully passed since each attitude estimator onboard the platform was successfully estimating its allocated Euler angle. The accuracy of these estimates was verified using the Vicon system, which validated the Kalman filter statistical parameters used for each estimator. AT-05 and AT-06 were also successfully passed since the Vicon system could estimate the X, Y and Z position of the quadrotor. Passing all three acceptance tests led to the successful completion of SR-B-04, SR-B-05 and SR-B-06.

Overall, all system requirements in the flight computer and state estimation subsystems were met. This achievement provided the AHNS2010 group with a quadrotor platform that could accurately predict its orientation onboard and service all connected sensors above the average 50Hz requirement. It also enables future AHNS groups to continue the advancements in these subsystems leading to a fully autonomous quadrotor solution.

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: [illegible]

Date: 29/10/2010

Acknowledgements

I would like to thank my supervisor Dr Luis Mejias for providing such an interest project to work on. Hopefully our collaboration with quadrotor research will continue well on into the future.

I would also like to thank all three members of the AHNS10 team: Michael Hamilton, Michael Kincel and Timothy Molloy. All three members were extremely dedicated to the project and made the project experience extremely enjoyable. The team would not have succeeded as much as it did without the contribution of these three members.

Finally, I would also like to thank my girlfriend Kirsty Henrichs for providing her loving support during the project duration.

Table of Contents

List of Tables

List of Figures

Definitions

AHNS	Autonomous Helicopter Navigation System
QUT	Queensland University of Technology
HLO	High Level Objective
IMU	Inertial Measurement Unit
IR	Infra Red
DOF	Degree Of Freedom
UART	Universal Asynchronous Receiver Transmitter
SPI	Serial Peripheral Interface
MVRB	Measured Values Register Bank
CRB	Configuration Register Bank
ASCII	American Standard Code for Information Interchange
GCS	Ground Control Station
Wi-Fi	Wireless local area network based on the IEEE 802.11 standards
GPS	Global Positioning Service
Subversion	Software versioning and revision control system
Openembedded	Software framework to create Linux distributions
EWMA	Exponentially Weighted Moving Average

Chapter 1 Project Introduction

1.1 Goals and background

The Autonomous Helicopter Navigation System (AHNS) 2010 was a final year project conducted at the Queensland University of Technology (QUT). The goal of this project was to develop hardware and software to automate a small size electric helicopter for use within an indoor environment. Due to the inherent lack of a GPS signal indoors, a combined estimation approach using inertial measurements and computer vision needed to be developed. The duration of this project spanned over 2 semesters where at the end of the second semester the fully completed system would be delivered to our project supervisor Dr Luis Mejias.

1.2 AHNS 2010 members

The AHNS 2010 project consisted of 4 members. The roles of each of these members in the AHNS 2010 project can be seen in Table 1.1 below.

Table 1.1 - AHNS2010 project members and respective roles

AHNS2010 member	Project role

Michael Hamilton	• Project Manager
	  o Project Management plan
	  o Risk assessment
	  o Budget - money
	• Communication
	  o Wi-Fi interface with GCS and onboard

Michael Kincel	• Pilot
	• Airframe
	  o Landing gear
	  o Avionics payload mounting
	  o Weight budget
	  o Hardware integration
	  o Batteries
	  o Sensor hardware mounting

Timothy Molloy	• Control
	  o Modelling
	  o Simulation
	  o Control software implementation
	• GCS

	◦ Software GUI and onboard/off board implementation
Liam O'Sullivan	• Localisation ◦ Computer Vision, camera selection, integration • State Estimation ◦ IMU software implementation ◦ Sensor software implementation • GCS ◦ Software GUI and onboard/off board implementation

1.3 System architecture

The AHNS 2010 project had the following system architecture (refer to Figure 1.1). From the initial goal of the project, it was decided to change the main platform from a helicopter to a quadrotor-based platform. The reason for this was mainly due to the simplicity of a quadrotor platform compared with a helicopter platform. The inherent simplicity of a quadrotor platform renders it much easier to control and operate. Additionally, quadrotor platforms are easy to repair when damaged which saves both precious time and monetary resources from being wasted.

1.4 Thesis outline

The thesis will be presented in the following structure:

- Chapter 2 discusses the management of the project including the systems engineering approach applied to the project.
- Chapter 3 presents the flight computer subsystem design and development. This will include the relevant sensors and hardware chosen for the project and how the software architecture onboard the platform was structured. The system requirements relevant to the flight computer will also be shown.
- Chapter 4 shows the state estimation subsystem design and development. This will include the states that were estimated and tracked whilst the quadrotor platform was in motion. The individual state estimator design for each of these states will also be presented. The system requirements relevant to the state estimation system will be identified and listed.
- Chapter 5 and 6 lists the testing that was requirement for the flight computer and the state estimation system. The relevant outcomes of this testing will be shown and summarised.
- Chapter 7 contains the conformance matrix for the flight computer and state estimation subsystems with accompanying test report documentation.
- Chapter 8 concludes the thesis with lessons learnt and recommendations for future years.

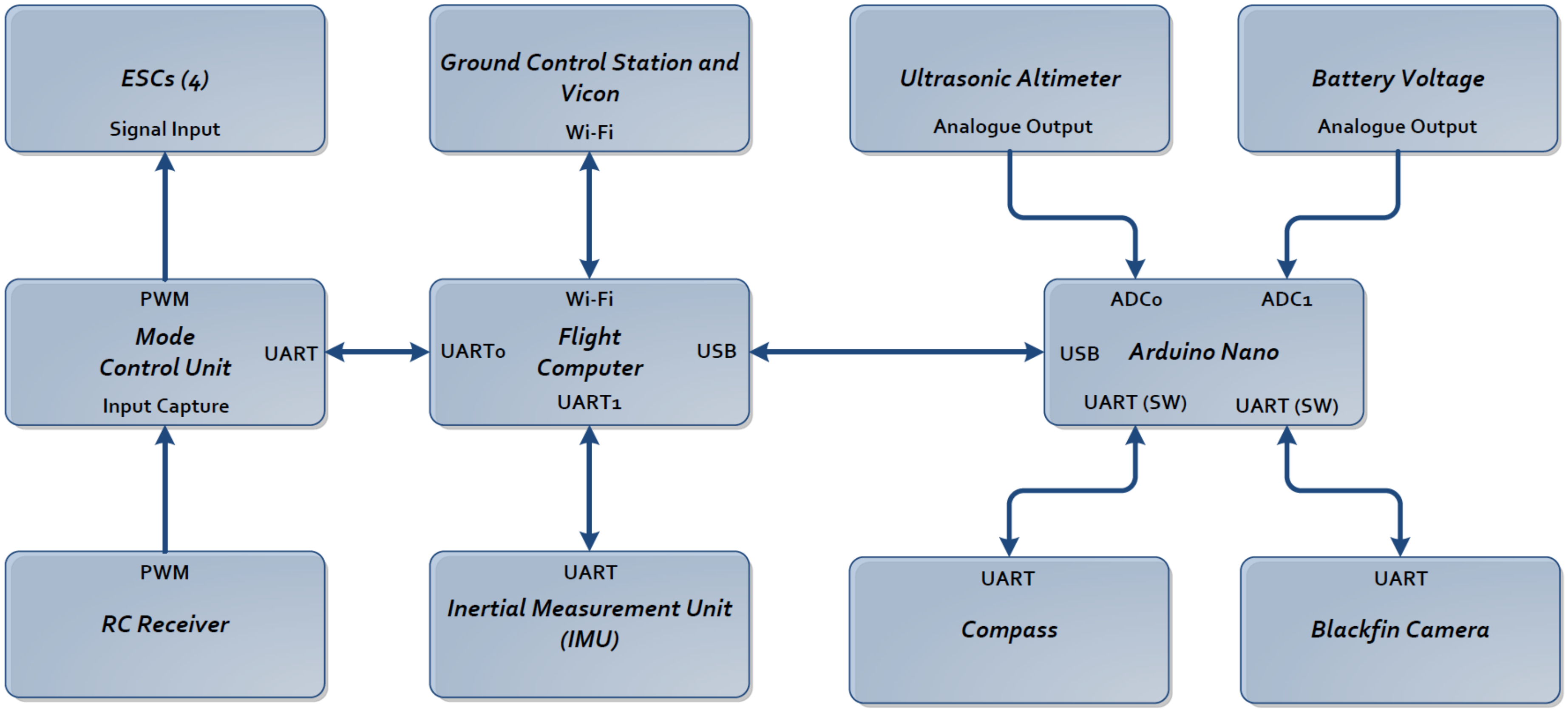


Figure 1.1 - AHNS 2010 system architecture

Chapter 2 Project Management

2.1 Systems engineering

A systems engineering methodology was applied to meet the goals of this project. This means that the following steps or stages were applied to the AHNS2010 project:

- Definition of High Level Objectives (HLOs) and System Requirements (SRs)
- Research into the relevant areas required to meet the requirements
- Design and testing of subsystem solutions
- Integration of the subsystem design into a greater system
- Delivery of the product to the customer

Numerous iterations occurred during these steps, which mirrored the changes in subsystem solution design or requirements. The practical output of this process was the timeline and identification of project roles and work packages. This chapter will present the HLOs developed for the AHNS2010 project and the project roles and work packages that I was allocated. Forthcoming chapters will also detail certain system requirements and subsystems that I was responsible for.

2.2 HLOs

6 High Level Objectives (HLOs) for the AHNS2010 project were identified in collaboration between our supervisor and AHNS2010 members. These HLOs are presented below and are documented in [RD/1].

2.2.1 HLO-1 Platform

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

2.2.2 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.

2.2.3 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.

2.2.4 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.

2.2.5 HLO-5 Ground Control Station

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.

2.2.6 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.

2.3 Project role

Various roles for each team member were allocated after the listed HLO's were identified. My roles for the AHNS2010 project can be summarised in Table 2.1 below.

Table 2.1 - Project roles

Role (HLO)	Detailed Description
Localisation (HLO-2)	• Camera selection • Software to predict platform position (including visual algorithms) • Integration and fusion of positional data for an overall position estimate
State Estimation (HLO-3)	• Implementation of software libraries and routines that can access raw sensor data • Adjustment of the raw sensor data for high quality estimates
Flight Computer (HLO-4)	• Implementation of an onboard software architecture that can process quadrotor inputs and outputs concurrently to achieve mission goals
GCS (HLO-5)	• Development of a software GUI that can transport or display platform information to a platform user

For the purpose of this thesis, my role in the Flight Computer (HLO-4) and the State Estimation (HLO-3) will be fully described and presented. Limited work was required in GCS (HLO-5) since other team members completed this HLO well before schedule. Only the camera was selected for Localisation (HLO-2). Work for HLO-2 failed to proceed during the project duration. This was because by the beginning of the second semester, it became obvious that the development of a stable quadrotor platform would consume the remainder of the project timeline. Thus it was decided to halt all work related to HLO-2 in favour of focusing on problems and issues which existed in other project areas.

2.4 Work packages

The work packages that I was allocated are contained in this section. Each work package also contains a description of the work package and the deliverables that were required to complete the work package.

2.4.1 WP-SE-01: Design State Estimation

Description: Design and develop the state estimation from onboard hardware data to allow the controller to predict future actions. Complete a design document outlining the state estimation developed.

Deliverables: AHNS-2010-SE-DD-001

2.4.2 WP-AP-02: Flight Computer Design Test Report

Description: Test the flight computer threading and verify that each thread is being updated at an average rate of 50Hz. Also, ensure that all sensor data is being retrieved successfully.

Deliverables: AHNS-2010-AP-TR-002

2.4.3 WP-AP-04: Flight Computer Bench Test Report

Description: Test the control system and state estimation on the flight computer, with all external sensors attached. Ensure that the results appear normal before integrating with platform and flight testing.

Deliverables: AHNS-2010-AP-TR-001

2.4.4 WP-SE-02: State Estimation Design Test Report (attitude)

Description: Verify that the state estimation design is working as intended. This test report will focus on the low-level attitude states (Euler rates and Euler angles). Each state estimate needs to be generated at an average rate of 50Hz.

Deliverables: AHNS-2010-SE-TR-001

2.4.5 WP-SE-03: State Estimation Design Test Report (position)

Description: Verify that the state estimation design is working as intended. This test report will focus on the position states of the quadrotor platform using the Vicon system. Each state estimate needs to be generated at an average rate of 50Hz. The Euler angles generated from the Vicon system should also be compared with the low level attitude output from the IMU Euler angle state estimators.

Deliverables: AHNS-2010-SE-TR-002

2.5 Source code and version control

A critical aspect in completing the specified work packages was choosing a location for the project source code and how version control would be implemented. Since multiple users were working on the same source code, a repository for the project needed to be created. The Google code project repository was the service chosen to maintain all AHNS2010 project source code. Google code also provides version control for the project using the 'Subversion' version control system. Thus all project source code was committed to this location and can be accessed from Google code via a Subversion client at '`http://ahns10.googlecode.com/svn/`'.

Chapter 3 Flight Computer

3.1 Introduction

The design and development of the flight computer was a critical aspect of the AHNS project. The desire to receive and process raw sensor data onboard the quadrotor platform created the need for a flight computer system.

3.2 System requirements

A flight computer implementation was required to deliver all airborne HLOs [RD/1] and their relevant system requirements. The system requirements that relate to the flight computer include [RD/2]:

1. SR-D-05: The airborne system shall receive and process measurement data from the state estimation and localisation sensors; supporting IMU, MCU, Ultrasonic, Battery voltage sensor and Magnetic compass devices.
2. SR-D-06: The airborne system shall collect avionics system health monitoring information in the form of radio control link status and battery level.

It was also inferred indirectly from other system requirements that these processes should be updated at an average rate of 50 Hz. From these system requirements, it was clear that new sensor and software architectures would need to be designed and developed to meet the requirements.

3.3 Design and development

The quadrotor platform had the following device or sensor architecture (refer to Figure 3.1). The flight computer main process was run on the Gumstix Overo Fire which had the following devices connected to it:

- IMU
- Arduino
- Compass (connected via the Arudino)
- Ultrasonic sensor (connected via the Arduino)
- Battery voltage sensor (connected via the Arduino)
- Mode control unit

Software libraries were developed for each of the above sensors to collect their respective data inputs. Implementation of these libraries required knowledge of how the sensors sent their data and how it can be obtained. The following sections describe the different sensors that are connected to the flight computer, how they can be accessed, the software interfaces or functions required to collect the data on the Overo Fire and how the flight computer code was structured.

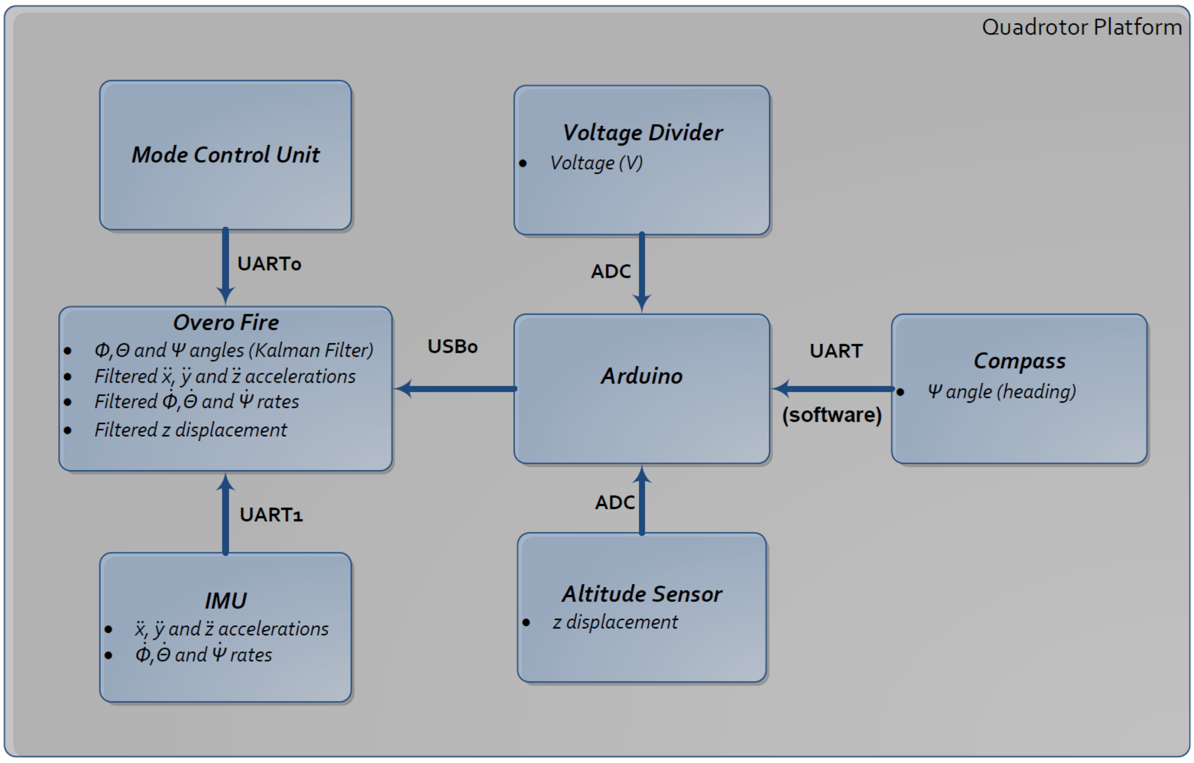


Figure 3.1 - Sensor architecture onboard the quadrotor platform

3.3.1 Inertial Measurement Unit (IMU)

The IMU which was chosen for the AHNS project was the Sensordynamics 6DOF IMU (refer to Figure 3.2). This IMU was inherited from the previous AHNS group following the damage of the previous Analog Devices ADIS16350 IMU.

3.3.2 IMU hardware interface

The Sensordynamics 6DOF IMU is connected to the Overo Fire under the AHNS2010 sensor architecture through UART. Originally, it was designed for the IMU to be connected to the Overo Fire via SPI input as the Sensordynamics IMU datasheet claimed that it could be connected via the SPI protocol. However under investigation it was found that the SPI was not yet implemented on the IMU board. Thus the only option to connect to the IMU from the Overo Fire was through the IMU's UART. The default settings of the IMU's UART were:

- Baud rate: 38400
- Data bits: 8 bits
- Start bits: 1 bit
- Stop bits: 1 bit
- Parity: No Parity

This baud rate can be reconfigured by connecting to the IMU and sending the ASCII string "*CHBR**n***" ending with the return character of '0x0D'. The ***n*** character in the ASCII string denotes what baud rate the IMU will be changed to. The following baud rates are available for the IMU:

- ***n*** = 0 : 19200 baud
- ***n*** = 1 : 38400 baud (default)
- ***n*** = 2 : 57600 baud
- ***n*** = 3 : 115200 baud

The baud rate was reconfigured to use 115200 baud to enable high state estimation update rates. However once the IMU has been powered down, the IMU reverts to its default baud rate of 38400. Thus the IMU must be reconfigured each time it is powered on to deliver data at the highest baud rate of 115200. Functions were implemented in the IMU's software library that enables the IMU 's baud rate to be reconfigured every time the Overo Fire is powered on.

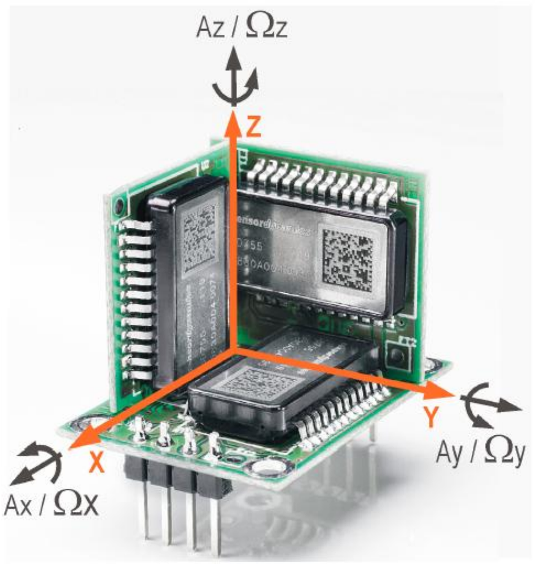


Figure 3.2 - SensorDynamics 6DOF IMU [RD/3]

3.3.3 IMU architecture

The IMU sensor data is collected by the IMU's onboard microcontroller and stored in the IMU's firmware in the Measured Values Register Bank (MVRB). This register bank is composed of 16 registers with 16 bits in each register. The following table describes what sensor data is held in each of these registers (refer to Table 3.1).

Table 3.1 - MVRB contents [RD/3]

Addr	Parameter Name	Description	Register Size / encoding	Reading scale
0x00	MR1 – Rate X	Angular rate along X axis, measure range 1	16 bit / 2's complement	$\pm 100\ °$ 25 Hz bandwidth
0x01	MR1 – Rate Y	Angular rate along Y axis, measure range 1	16 bit / 2's complement	$\pm 100\ °$ 25 Hz bandwidth
0x02	MR1 – Rate Z	Angular rate along Z axis, measure range 1	16 bit / 2's complement	$\pm 100\ °$ 25 Hz bandwidth
0x03	MR1 – Accelerometer X	Acceleration along X axis, measure range 1	16 bit / 2's complement	± 2 g 40 Hz bandwidth
0x04	MR1 – Accelerometer Y	Acceleration along Y axis, measure range 1	16 bit / 2's complement	± 2 g 40 Hz bandwidth
0x05	MR1 – Accelerometer Z	Acceleration along Z axis, measure range 1	16 bit / 2's complement	± 2 g 40 Hz bandwidth
0x06	MR2 – Rate X	Angular rate along X axis, measure range 2	16 bit / 2's complement	$\pm 300\ °$ 75 Hz bandwidth
0x07	MR2 – Rate Y	Angular rate along Y axis, measure range 2	16 bit / 2's complement	$\pm 300\ °$ 75 Hz bandwidth
0x08	MR2 – Rate Z	Angular rate along Z axis, measure range 2	16 bit / 2's complement	$\pm 300\ °$ 75 Hz bandwidth
0x09	MR2 – Accelerometer X	Acceleration along X axis, measure range 2	16 bit / 2's complement	± 5 g 100 Hz bandwidth
0x0A	MR2 – Accelerometer Y	Acceleration along Y axis, measure range 2	16 bit / 2's complement	± 5 g 100 Hz bandwidth
0x0B	MR2 – Accelerometer Z	Acceleration along Z axis, measure range 2	16 bit / 2's complement	± 5 g 100 Hz bandwidth
0x0C	Temperature	Temperature	16 bit, unsigned	$1°\mathrm{C}/\mathrm{LSB}$

				80°C offset
0x0D	Status_L	Status info LSB	16 bit, unsigned	Status
0x0E	Status_H	Status info MSB	16 bit, unsigned	Status
0x0F	reserved	reserved	16 bit, unsigned	NA

The configuration parameters of the IMU are stored in the Configuration Register Bank (CRB). This bank is comprised of another 16 register with 16 bits in each register. The 4 most important registers contained in the CRB can be seen in the table below (refer to Table 3.2). The IMU serial data output is controlled by the CRB registers OutputData_L and OutputData_H. The contents of these registers determine what sensor data will be sent on the serial UART.

Table 3.2 - Important CRB registers [RD/3]

Address	**Category**	**Register Name**	**Default Value**	**Register size / coding**
0x0C	UART manager	OutputData_L	0x41	8 bits, unsigned
0x0D	UART manager	OutputData_H	0x70	8 bits, unsigned
0x0E	FW_reg	Firmware version	0x01	8 bits, unsigned
0x0F	CRC_reg	CRC	0x47	8 bits, unsigned

3.3.4 IMU CRB access

The CRB registers can be read or written through the following two serial commands:

1. **Write CRB**: Enables the selected CRB register to be written depending on the serial command sent. The format of this serial command is ***WCRBH$_1$HH$_2$*** where ***H$_1$*** is a 1 digit hex number representing the CRB register address and ***HH$_2$*** is a 2 digits hex number representing the value to be written. A correct serial command transaction terminates with “OK!” being sent back to the user. If a correct serial command is not sent then the IMU responds with the “Error” string.
2. **Read CRB**: Enables the selected CRB register to be read depending on the serial command sent. The format of this serial command is RCRBH where H is a 1 digit hex number representing the CRB register address. The firmware replies with an ASCII character representing the value of the selected CRB register followed by the return character ‘0x0D’. If an error was mode in the serial command sent to the IMU then the IMU will responds with the “Error” string.

Functions were implemented in the IMU software library that can enable the CRB registers of the IMU to be read or written.

3.3.5 IMU sensor data access

The IMU sensor data can be accessed through 2 different serial modes. Each of these modes can be enabled by sending the IMU the appropriate serial command. These modes and associated serial commands include [RD/3]:

1. **Start Free Running Mode**: Enabled by sending the IMU the ASCII command "*STAFRM*" which is terminated with the return character '0x0D'. While the IMU is in this mode, the UART continuously outputs the selected measured values from the OutputData_L and OutputData_H registers. The output format is ***Va11,Val2,...,ValN,Chk*** terminated with the return character '0x0D' where ***ValX*** is a 4 digit hex number representing the selected MVRB value and ***Chk*** is a 2 digits hex number representing the 8 bits sum of the output string negated.
2. **Start One Shot Mode**. Enabled by sending the IMU the ASCII command "STAOSM" which is terminated with the return character '0x0D'. While the IMU is in this mode, the firmware will reply once with a frame containing the selected measured values from the OutputData_L and OutputData_H registers. The output format is ***Va11,Val2,...,ValN,Chk*** terminated with the return character '0x0D' where ***ValX*** is a 4 digit hex number representing the selected MVRB value and ***Chk*** is a 2 digits hex number representing the 8 bits sum of the output string negated.

Functions were implemented in the IMU software library that can enable the IMU to be accessed in either of these modes.

3.3.6 IMU serial data output

By default, OutputData_L and OutputData_H are set to 0x41 and 0x70. Thus the overall OutputData parameter can be read by joining OutputData_H and Output Data_L together. The default OutputData parameter for the IMU (when powered on) is equal to 0x7041 or in binary 0b0111000001000001. Hence when either 'Start Free Running Mode' or 'Start One Shot Mode' modes are invoked with the default values of OutputData_L and OutputData_H, they will return with the following sensor output format:

| MR1 – Rate X | MR2 – Rate X | Temperature | Status_L | Status_H | Chk |

The low level attitude state estimation process relies on the use of IMU data which supplies high output rates and large reading scales. Thus the IMU serial data output included the following MVRB registers:

- MR2 – Rate X
- MR2 – Rate Y
- MR2 – Rate Z
- MR2 – Accelerometer X
- MR2 – Accelerometer Y
- MR2 – Accelerometer Z
- Temperature
- Status_L
- Status_H
- Chk

To configure the IMU serial data output to send this information, OutputData_L needs to be equal to '0xC0' and OutputData_H needs to be equal to '0x7F'. Thus the OutputData parameter will be equal to 0x7FC0 or in binary 0b0111111111000000.

3.3.7 IMU serial sensor data conversion

The IMU serial data output for the rate and accelerometer values are sent as 4 digit hex numbers in the 2's complement format. Thus once the values have been received they need to converted from a 4 digit hex format into a 2's complement format. This can be achieved by using the `short int` data type and using the `sscanf` function. As an example: if we had a character array called `sResult` which contained the 4 digit hex representation of the 'MR2 – Rate X' value and we wanted to convert this value into its 2's complement format, we could use the following C code:

```
short int rateX = 0;
sscanf(sResult, "%hx", &rateX);
```

This would result in the 2's complement format of the 'MR2 – Rate X' value being stored in `rateX`. The 2's complement value would then need to be scaled according to the following scaling factors or resolutions:

- MR1 Rates – $0.0039\ (°/\mathrm{s})/\mathrm{bit}$
- MR2 Rates – $0.0156\ (°/\mathrm{s})/\mathrm{bit}$
- MR1 Accelerations – $0.002031\ (\mathrm{m}/\mathrm{s}^2)/\mathrm{bit}$

- MR2 Accelerations – $0.004062\ (\mathrm{m/s^2})/\mathrm{bit}$

Continuing the previous example, once the `rateX` value is scaled (using the multiplier of 0.0156 for MR2 rates) and stored in a `double` data type, then the measured 'MR2 – Rate X' value has been successful converted. This converted value can then be passed on to the state estimation algorithms for processing. This sensor data conversion was implemented in the IMU software library and is invoked each time sensor data is read from the IMU.

3.3.8 IMU software library functions

The IMU library functions which have been implemented include:

- openIMUSerial: Open the serial port connected to the IMU from the Overo Fire.
- closeIMUSerial: Close the IMU serial port to the IMU from the Overo Fire.
- setIMUBaudRate: Set the baud rate of the IMU.
- readCRBData: read the CRB register data.
- setCRBData: set the CRB register data.
- getIMUSensorData: Put the IMU into one shot mode and collect one frame of IMU sensor data. This function assumes that the OutputData_L registers and the OutputData_H registers have been configured to be equal to '0xC0 and 0x7F' respectively. This function also converts the sensor data from its 2's complement format to the required `double` data type.
- startFreeRunningMode: Starts the IMU in free running mode.
- stopFreeRunningMode: Stops the IMU from executing free running mode.
- setIMUconfig: Reconfigures the IMU to the appropriate output sentence format and baud rate for the quadrotor platform as specified in section 3.3.6.

The header file `imuserial.h` and the source file `imuserial.c` that implement these functions have been included in Appendix A.

3.4 Magnetic compass

The magnetic compass which was chosen to be integrated on the quadrotor platform was the OS4000 magnetic compass (refer to Figure 3.3). This magnetic compass is tilt compensated which allows accurate readings of heading to be obtained even when the platform is on an angle. Tilt compensated compasses are vastly superior to regular compasses but care must be taken in platform placement due to system vibrations.

3.4.1 Magnetic compass hardware interface

The OS4000 magnetic compass connects to the Arduino via a UART. The default settings for the compass's UART are [RD/4]:

- Baud rate: 19200
- Data bits: 8 bits
- Start bits: 1 bit
- Stop bits: 1 bit
- Parity: No Parity

The baud rate of the compass can be changed by connecting to the compass and sending the command '<Esc> B'. This halts the compass outputs and allows the following baud rates to be selected by typing a character from '0' to '6' [RD/5]:

- 0: 4800 baud rate
- 1: 9600 baud rate
- 2: 14400 baud rate
- 3: 19200 baud rate
- 4: 38400 baud rate
- 5: 57600 baud rate
- 6: 115200 baud rate

The highest baud rate that was available for the firmware version of the acquired compass was 38400. However this baud rate proved to be unstable as transmitted data could sometimes become unreadable. Thus the default 19200 baud rate was used for connection to the Arduino. The microcontroller located on the OS4000 compass saves what baud rate it has been configured and stores this information when the compass has been turned off. Thus baud rate reconfiguration does not have to occur each time the compass is powered on.

Figure 3.3 - OS4000 magnetic compass [RD/4]

3.4.2 Magnetic compass output sensor format

The OS4000 magnetic compass outputs its sensor data asynchronously via the compass's UART in serial ASCII format. The output sentence format has been configured to have the standard factory format output. This format is as follows:

Format**: $Chhh.hPpp.pRrr.rTtt.t*cc**

Example**: $C212.4P2.5R14.0T28.4*3A**

Where Chhh.h is the azimuth angle, Ppp.p is the pitch angle, Rrr.r is the roll angle, Ttt.t is the temperature and *cc is the checksum of the output sentence. This output sentence format will be read by the Arduino where the azimuth angle hhh.h will be extracted.

The rate at which this sentence is sent to the UART is controlled by the sentence output rate. This rate can range between -50 to +40 [RD/4]. The rate can be modified by sending the command <Esc> R to the compass's UART and inputting the required rate. The rate will be set to +40 which results in the compass output being sent 40 times per second.

3.4.3 Magnetic compass calibration

The OS4000 compass can be calibrated through hard iron and soft iron calibration. Hard iron calibration is a two tier process where the X, Y planes are calibrated followed by the Z plane calibration. X, Y plane calibration is executed with the command '<Esc> C'. When the compass is in this mode it needs to be rotated in the X, Y plane continually. Whilst it's being rotated the compass will output to the serial UART "XxYy" signifying that it is performing calibration. When the output changes to ".", the X, Y plane calibration is complete. The Z plane calibration is executed with the '<Esc> Z' command. When the compass is in the mode it needs to be tilted 90^{0} on its side. The compass will report similar information to the serial UART which was shown in the X, Y plane calibration except it will send "Zz" data. When the output changes to ".", the Z plane calibration is complete.

Soft iron calibration is performed in a similar fashion and can be invoked with the <Esc $> command. When the compass is in this calibration mode, it invites the user to align the compass to the cardinal points of a magnetic compass being North, South, East and West. At each alignment point the user must press the space bar to continue to the next point. Once the soft iron calibration is completed the compass will report the soft iron correction values to the UART. Both of these calibration routines will be executed once the compass is mounted in the quadrotor platform.

3.4.4 Magnetic compass Arduino functions

The following function has been implemented to read the compass heading on the Arduino:

- readCompass

The source file `arduserial.pde` contains the implementation of this function and can be seen in Appendix B. In addition, the source file `ardupass.pde` contains the Arduino code which passes serial commands from the Arduino to the OS4000 compass and vice versa. This enables a serial pass through to the compass so it can be reconfigured before test flights occur (refer to Appendix C).

3.5 Altitude sensor and battery voltage sensor

The altitude sensor which was chosen for the AHNS platform was the LV-MaxSonar-EZ0 sensor (refer to Figure 3.4). This sensor is based off ultrasonic technology as opposed to IR proximity sensor technology. IR based sensors were not chosen to be integrated on to the quadrotor platform due to the use of the Vicon system which also uses IR.

Figure 3.4 - LV-MaxSonar-EZ0 [RD/5]

3.5.1 Altitude sensor hardware interface

The LV-MaxSonar-EZ0 sensor provides altitude readings in 3 forms including: asynchronous serial, analogue to digital conversion (ADC) and pulse width modulation [RD/5]. The ultrasonic sensor connects to the Arduino via an ADC pin. The ultrasonic sensor itself outputs an analogue voltage with a scaling factor of (Vcc/512) per inch [RD/5]. Thus since the ultrasonic is powered by 5V the scaling factor is equal to 9.8mV per inch [RD/5]. The Arudino has 10 bit ADCs i.e. 1024 range such that an ADC read by the Arduino returns a number between 0 and 1023. Using these two pieces of information, an equation was developed to convert the ADC Arduino voltage reading (from 0 to 1023) to the altitude sensor reading (9.8mV per inch) in meters:

$$\text{Altitude (m)} = 0.0135 \times \text{adc}_{\text{read}} + 0.04155$$

3.5.2 Altitude sensor filtering

Since the altitude sensor reading is via an ADC, the reading will be very noisy. A moving average filter was developed to eliminate some of the noise. The moving average filter window is 5 points long and is filtered when the Arduino takes altitude readings.

3.5.3 Battery voltage sensor hardware interface

A voltage divider was implemented to convert the input battery voltage to a voltage in the range of 0-5V. The output of this voltage divider was connected to the ADC pin of the Arduino where the voltage could be read. An equation was developed to convert the ADC Arduino voltage reading to the battery voltage in volts:

$$\text{Battery Voltage (V)} = 0.0135 \times \text{adc}_{\text{read}} + 0.103465$$

3.5.4 Battery voltage filtering

The battery voltage reading needs to be filtered much like the altitude sensor due to its connection to an ADC. A moving average filter was also applied to the battery voltage reading to eliminate some of the noise. The moving average filter window was also 5 points long and is filtered when the Arduino takes battery voltage readings.

3.5.5 Altitude and battery voltage Arduino functions

The following functions have been implemented to read the altitude and voltages on the Arduino:

- readAltitude
- readVoltage

The source file `arduserial.pde` contains the implementation of these functions and can be seen in Appendix B.

3.6 Arduino

An Arduino board was required for sensor connection due to the limited number of UART ports available on the flight computer. The Arduino which was chosen to be put on the quadrotor platform was the Arduino Nano (refer to Figure 3.5). The Arduino Nano was selected due the mini-B USB connection located on its board. This allows easy access to the USB output on the Gumstix Pinto board which the Overo Fire connects to.

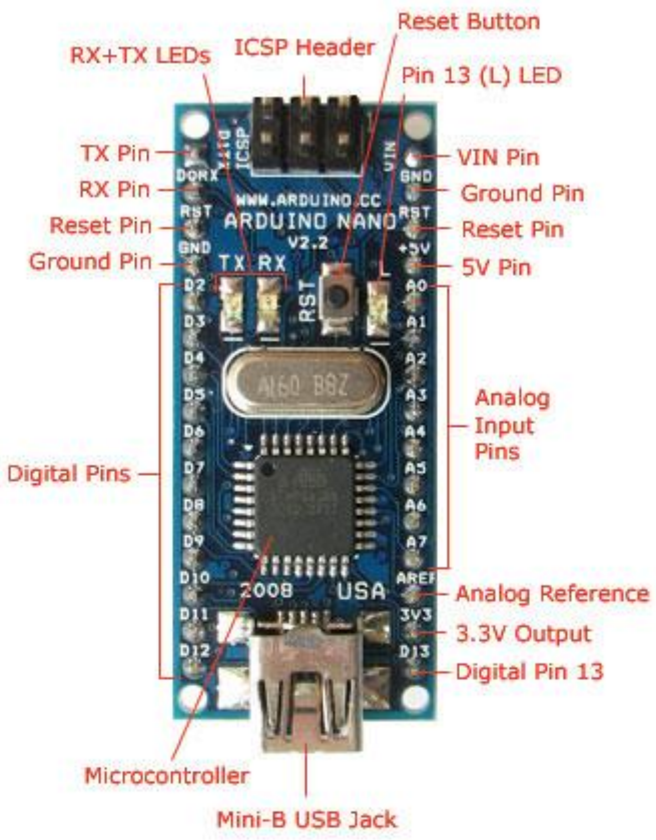


Figure 3.5 - Arduino Nano [RD/6]

3.6.1 Arduino sensors and hardware interface

The Arduino connects to the following sensors via the listed hardware interface: compass via UART, altitude sensor via ADC and the battery voltage via ADC. It also connects to the Overo Fire over UART. The Arduino Nano only contains one serial UART so the compass needs to be connected to 2 other digital pins to enable a software UART connection. The sensor data is transmitted from the Arudino Nano and sent to the Overo Fire which is running the flight computer. The Overo Fire connects to the Arduino Nano via the following UART configuration:

- Baud rate: 115200
- Data bits: 8 bits
- Start bits: 1 bit
- Stop bits: 1 bit
- Parity: No Parity

3.6.2 Arduino sensor data format

The Arduino transmits the sensor data asynchronously in the following format:

Format: **Chhh.h,Vvv.vvv,Aa.aaa**

Example: **C094.3,V11.53,A1.034**

Where Chhh.h is the compass heading, Vvv.vvv is the battery voltage and Aa.aaa is the altitude reading. The data is sent to the Overo Fire asynchronously to maximise the data transfer rate. The Arduino can also be reconfigured to send the data synchronously if desired.

3.6.3 Arduino functions

As was previously stated, the following functions have been implemented on the Arduino to allow for the connected sensors to be read:

- readCompass
- readAltitude
- readVoltage

The data from the Arduino is printed to the serial UART and sent to the Overo Fire via the following function:

- printOvero

These functions were implemented in the source file `arduserial.pde` and can be seen in Appendix B.

3.6.4 Arduino software library functions

The following functions have been implemented on the Overo Fire to connect to the Arduino and collect the data it sends:

- openArduSerial: Open the serial port between the Overo Fire and the Arduino.
- openArduSerialCan: Open the serial port between the Overo Fire and the Arduino so the Arduino can send data asynchronously (canonical mode).
- closeArduSerial: Close the serial port between the Overo Fire and the Arduino,
- getCompassHeading: Request the compass heading from the Arduino and receive the heading,

- getBatteryVoltage: Request the battery voltage level from the Arduino and receive the voltage.
- getAltitudeReading: Request the altitude reading from the Arduino and receive the altitude.
- getArduinoData. Request the compass heading, battery voltage level and altitude reading from the Arduino and receive all 3 values at once.
- getArduinoDataCan: Receive the asynchronously sent data (with the Overo Fire UART set to canonical mode) from the Arduino to the Overo Fire. The data sent in this way includes the compass heading, battery voltage level and the altitude reading.

The header file `arduserial.h` and the source file `arduserial.c` contain the implementation of these functions and can be found in Appendix D.

3.7 Flight computer processor

3.7.1 Flight computer hardware specifications

The flight computer is executed on the Gumsix Overo Fire (refer to Figure 3.6). The Overo Fire has the following specifications (refer to Table 3.3). The primary reason this hardware solution was chosen to run the flight computer was due its small size and weight compared with its processing power. The Overo Fire also contains a WiFi connection which was required to achieve HLO-6.

Table 3.3 - Gumstix Overo Fire specifications [RD/7]

Specification	Overo Fire
Processor	ARM Cortex-A8 OMAP3530
Clock speed	720 MHz
Memory	256MB RAM / 256MB Flash
Weight	5.6g
Size	17mm x 58mm x 4.2mm
Wireless Connectivity	• Bluetooth • Wi-Fi
Features	• I2C • PWM (6) • A/D(6) • UART • USB host

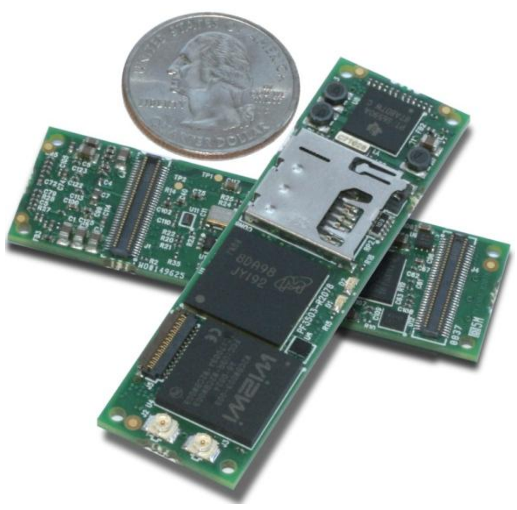


Figure 3.6 - Gumstix Overo Fire [RD/7]

3.7.2 Flight computer software architecture

The flight computer is structured using threads in the C programming language (through the `pthread` library). Each thread is responsible for managing a different process that needs to be executed concurrently. The threads implemented include:

- Arduino thread
- State Estimation thread
- Control thread
- MCU thread
- Downlink thread

All threads use mutex locks on the state variable that change such that update errors between the threads do not occur. All threads also have slight delays so that the threads can pass control back to the thread queue. The software architecture of the flight computer can be seen in Figure 3.7 below.

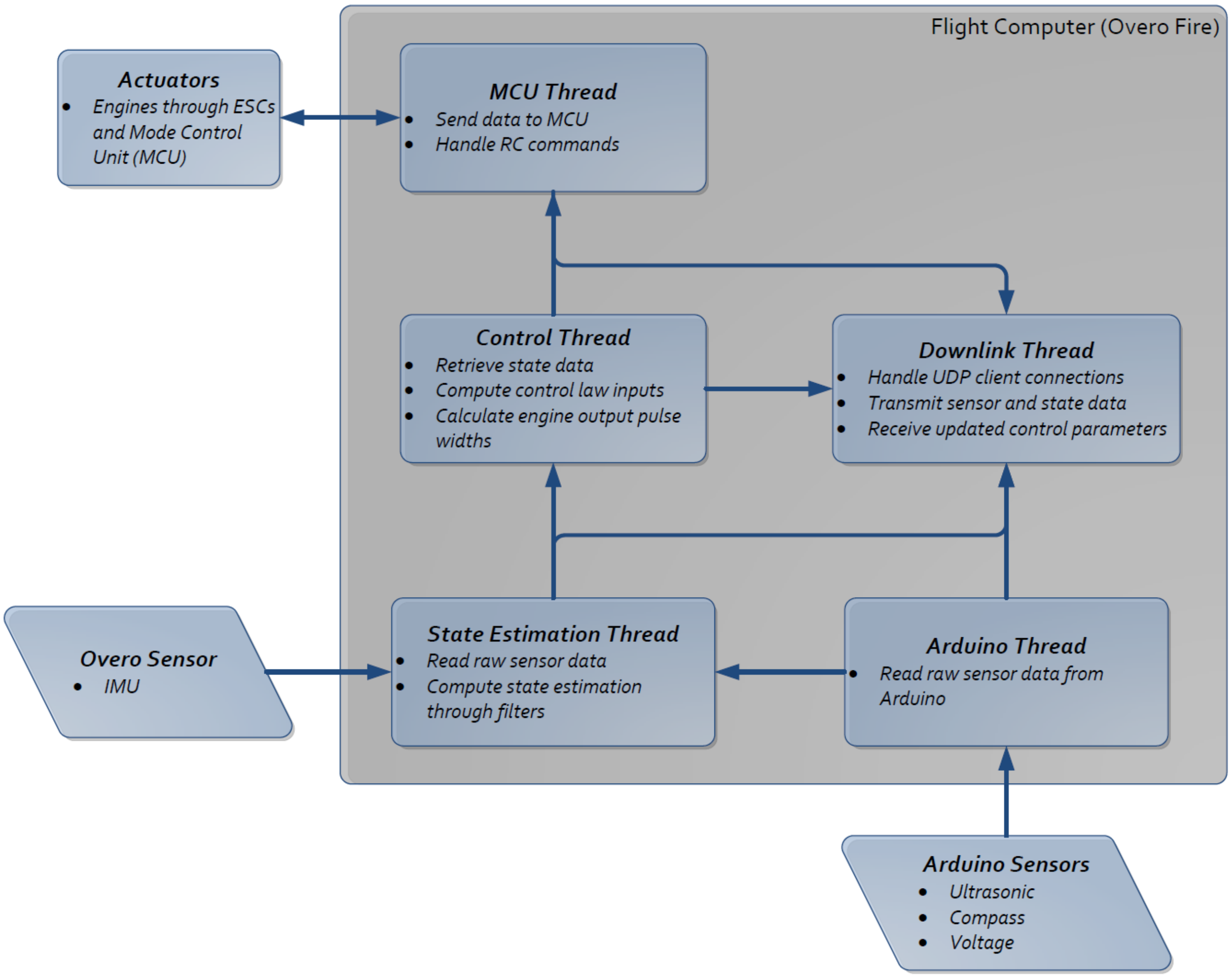
3.7.3 Flight computer functions

The following functions have been implemented for the flight computer to facilitate flight testing:

- updateCompassHeading: Updates the compass heading.
- updateArduinoData: Updates the Arduino data.
- updateIMUData: Updates the IMU data.
- sensorInit: Initialises the IMU and Arduino connections.
- mcuInit: Initialises the MCU connection.
- sendUDPData: Sends the UDP packet data to connected clients

- updateMCU: Updates the MCU
- updateControl: Updates the control loops

The header file `main.h` and the source file `main.c` contain the implementation of these functions and can be found in Appendix E.



Flight Computer (Overo Fire)
Actuators
Engines through ESCs and Mode Control Unit (MCU)
MCU Thread
Send data to MCU
Handle RC commands
Control Thread
Retrieve state data
Compute control law inputs
Calculate engine output pulse widths
Downlink Thread
Handle UDP client connections
Transmit sensor and state data
Receive updated control parameters
Overo Sensor
IMU
State Estimation Thread
Read raw sensor data
Compute state estimation through filters
Arduino Thread
Read raw sensor data from Arduino
Arduino Sensors
Ultrasonic
Compass
Voltage

Figure 3.7 - Flight Computer software architecture

3.7.4 Flight computer operating system configuration

The flight computer source files are built and executed on the Overo Fire which is running a Linux based operating system. This operating system is accessed by the Overo Fire through a SD card which is mounted on the Overo Fire board. The operating system which is used by the Overo Fire system needs to be configured and built under a system which contains the openembedded compiler environment. This cross compiler environment allows a Linux operating system to be built which will be compatible with the OMAP processor architecture on the Overo Fire board. Instructions on installing the openembedded compiler environment and how to build a Linux operating system console image can be found in [RD/8].

The following programs were chosen to be included on the flight computers operating system:

- Bash
- Bzip2
- Ckermit
- Devmem2
- Dhcp-client
- Dosfstools
- Fbgrab
- Fbset
- Fbset-modes
- Grep
- Gsl-dev
- I2c-tools
- Ksymoops
- Mkfs-jffs2
- Mtd-utils
- Nano
- Ntp
- Ntp update
- Openssh-misc
- Openssh-scp
- Openssh-ssh
- Omap3-writeprom
- Procps
- Socat
- Strace
- Subversion
- Sudo
- Syslog-ng

- Task-natve-sdk
- Task-proper-tools
- Vim

The recipe `julia-fc-image.bb` used to create this console image can be found in Appendix F. Furthermore one other program was also required to be downloaded and installed through the use of the 'opkg' program on the Overo Fire board:

- Build-essentials

This program allows the flight computer code to be built within the Overo Fire operating system instead of being cross compiled via the openembedded build environment.

3.7.5 Flight computer operating system installation

Once the operating system is built under the instructions found in [RD/8] it can then be placed on a SD card for execution. Instructions for formatting the SD card and placing the operating system onto it can be found in [RD/9].

3.7.6 UART1 pass-through

The Overo Fire is documented with having 4 serial UARTS which include:

1. UART0: spare serial connection
2. UART1: serial UART connected to the blue tooth module
3. UART2: serial UART connected to the console output of the Overo Fire
4. USB0: serial UART connected through the USB host controller on the Overo Fire

Thus the Overo Fire only has 2 serial connections that are freely available under standard operating system installation. The Overo Fire has to connect to 3 UART devices including:

1. IMU
2. Mode control unit (connected through UART0)
3. Arduino (connected through USB0)

This leaves the IMU without a serial connection. It was discovered that the pins connected to UART1 can be multiplexed to other GPIO pins on the Overo Fire board. This allows a pass through to be developed for UART1 if the blue tooth module is disabled. Multiplexing of the Overo Fire pins is handled by the u-boot program. This program is built automatically in the openembedded cross compile environment when a console image is created. Thus a patch for this program had to be created allowing the UART1 pins to be multiplexed to the GPIO pins. This patch was placed in the u-boot openembedded library files and was invoked during the cross compilation process of the console image. The patch `uart1.patch` that was used for this process can be seen in Appendix G.

Chapter 4 State Estimation

4.1 Introduction

Various parameters or states of the quadrotor platform need to be constantly measured for the platform to be controllable. This need is encapsulated in HLO-3 [RD/1] which specified that a state estimation method needed to be designed and implemented. The state estimation subsystem will be documented in the following steps:

- Identifying the states which need to be measured for the quadrotor platform
- Choosing the sensors that will measure the states
- Designing a state estimation methodology for each state (which will include low pass filtering and or Kalman filtering)
- Implementing the design which is suitable for the flight computer located on the Overo Fire or on the GCS.

4.2 System requirements

The following system requirements needed to be met to achieve HLO-3 (as specified in [RD/2]):

Table 4.1 - System requirements (relevant to state estimation) to met HLO-3

System Requirement	Definition
SR-B-04	The estimator shall provide Euler angle and rate estimation for the system at an average rate of 50 Hz.
SR-B-05	The estimator shall provide altitude estimation for the system at an average rate of 50 Hz.
SR-B-06	The estimator shall provide x and y estimation in an Earth fixed co-ordinate system at an average rate of 50 Hz.
SR-D-05	The airborne system shall receive and process measurement data from the state estimation and localisation sensors; supporting IMU, Camera, IR, Ultrasonic and Magnetic compass devices.

Hence the state estimation system needed to be designed to ensure that the stated system requirements have been met. This required a state estimation design for each individual state of the quadrotor platform. The design must also be implemented efficiently to guarantee that the estimated state will be calculated at an average rate of 50 Hz. Doing so will enable the quadrotor platform controller to issue high quality control commands.

4.3 Design and development

4.3.1 Quadrotor states

The following states of the quadrotor platform are required to be tracked to achieve autonomous capability (refer to Table 4.2). The table below also includes what senor was used to measure the platform state. The low level attitude state estimation was handled by the SensorDynamics 6 DOF IMU onboard the quadrotor platform. The X, Y, Z velocities and X, Y, Z displacements were calculated by the Vicon motion capture sensor. An alternative method of measuring the Z velocity and Z position was also provided by the altitude sensor. The estimation design for each state was considered and developed to create a high quality state estimation system.

Table 4.2 - Quadrotor Platform States and the Measurement Sensor

State	Sensor	State	Sensor
Roll rate $(\dot{\phi})$	X Rate gyro (IMU)	Z acceleration $(\ddot{z})$	Z Accelerometer (IMU)
Pitch rate $(\dot{\theta})$	Y Rate gyro (IMU)	X velocity $(\dot{x})$	Vicon
Yaw rate $(\dot{\psi})$	Z Rate gyro (IMU)	Y velocity $(\dot{y})$	Vicon
Roll (ϕ)	X Rate gyro and accelerometers (IMU)	Z velocity $(\dot{z})$	Vicon and Altitude sensor
Pitch (θ)	Y Rate gyro and accelerometers (IMU)	X displacement (x)	Vicon
Yaw (ψ)	Z Rate gyro (IMU) and compass	Y displacement (y)	Vicon
X acceleration $(\ddot{x})$	X Accelerometer (IMU)	Z displacement (z)	Vicon and Altitude sensor
Y acceleration $(\ddot{y})$	Y Accelerometer (IMU)		

4.3.2 Euler rate estimation $(\dot{\phi}, \dot{\theta}, \dot{\psi})$

The 3 Euler rates are measured by 3 gyroscopes located in the Sensor dynamics 6DOF IMU. These gyroscopes are aligned to the body axis of the quadrotor platform and hence can directly measure the required Euler body rates. The table below lists the gyroscopes characteristics as specified in the datasheet [RD/10]. As can be seen in Table 4.3, the gyroscopes are not perfect sensors and contain some noise error. This measurement error can be mitigated by applying an exponentially weighted moving average (EWMA) filter. This filter is described by the following equation:

$$\bar{x}_k = (1-\alpha)\bar{x}_{k-1} + \alpha x_k$$

Where $\bar{x}_k$ is the current mean value, $\bar{x}_{k-1}$ is the previous mean value, x_k is the current sensor reading and α is the filter constant or smoothing factor. The α parameter controls the degree of filtering or smoothing in the measurements and can range from 0 to 1. As α approaches 1 the current mean value $\bar{x}_k$ will rely more on the current sensor measurements. If α is equal to 1 then no filtering will take place. Conversely, as α approaches 0 the current mean value $\bar{x}_k$ will rely more on the previous sensor measurements and will filter out any large changes in the current mean. Hence the EWMA filter performs exactly like a discrete low pass filter and can be tuned by the smoothing factor α.

Table 4.3 - Gyroscope characteristics [RD/10]

Parameter	MR1	MR2	Unit	Condition
Measurement Range	± 100	± 300	$°/\mathrm{s}$	
Resolution	0.0039	0.0156	$(°/\mathrm{s})/\mathrm{bit}$	True 16 bit
Max RMS noise	0.2	0.3	$°/\mathrm{s}$	Bandwidth: MR1: 25Hz MR2: 75Hz
Max zero rate bias	± 0.5		$°/\mathrm{s}$	Zero setting at 25°C
Max temperature drift of zero rate bias	± 1.0		$°/\mathrm{s}$	Over full temp range
Max sensitivity error	± 2.0		%	Over full temp range
Max linearity error, versus best fit	± 0.2		%	Over full temp range

The EWMA filter can be shown to be equivalent to a discrete first order low pass filter. The Laplace transfer function of a first order low pass filter is equal to:

$$\frac{\bar{x}(s)}{x(s)} = \frac{1}{1+\tau s}$$

Where τ is equal to the time constant of the filter, $\bar{x}(s)$ is the filter signal and $x(s)$ is the current measurement. The above equation can be converted to the time domain using inverse Laplace transform properties:

$$x(t) = \tau \frac{d\bar{x}(t)}{dt} + \bar{x}(t)$$

The differential in the above equation can be discretised using the following approximation:

$$\frac{d\bar{x}(t)}{dt} \approx \frac{\bar{x}_k - \bar{x}_{k-1}}{T_s}$$

Where T_s is the time step between each measurement interval. Substituting this expression back into the time domain representation of the first order low pass filter yields:

$$x_k = \tau\left(\frac{\bar{x}_k - \bar{x}_{k-1}}{T_s}\right) + \bar{x}_k$$

Rearranging the above formula generates:

$$\bar{x}_k = \left(\frac{\tau}{\tau + T_s}\right)\bar{x}_{k-1} + \left(\frac{T_s}{\tau + T_s}\right)x_k$$

This equation can simplified by making a substitution for α:

$$\alpha = \left(\frac{T_s}{\tau + T_s}\right) \text{ and } (1-\alpha) = \left(\frac{\tau}{\tau + T_s}\right)$$

This creates the following formula when substituted into above formula:

$$\bar{x}_k = (1-\alpha)\bar{x}_{k-1} + \alpha x_k$$

Which is equivalent to the EWMA filter. Thus the EWMA filter performs a first order low pass filter on the gyroscope data and eliminates some of the noise components generated from temperature and platform vibration. 3 separate EWMA filters will need be to implemented for each gyroscope to give a good estimation for each Euler rate $(\dot{\phi}, \dot{\theta}, \dot{\psi})$. Corresponding α smoothing parameters will also need to be discovered as well during integration testing.

4.3.3 Euler angle estimation $(\boldsymbol{\phi}, \boldsymbol{\theta})$

The Euler angles ϕ and θ can be measured indirectly and directly from the IMU located on the platform. The 3 gyroscopes on the IMU measure the Euler rates $(\dot{\phi}, \dot{\theta}, \dot{\psi})$ which can be integrated at each time step to give an approximate or indirect measurement of the Euler angles (ϕ, θ, ψ). This approximation of the Euler angles will slowly drift over time due to temperature noise effects thus relying solely on gyroscopic data will produce poor quality estimates. The 3 accelerometers can also produce a coarse estimate of the Euler angles ϕ and θ through trigonometric calculations. Whilst this is a direct measurement of the Euler angles ϕ and θ it will also be of low quality due to the accelerometer noise.

Good quality estimates of the Euler angles ϕ and θ can be achieved by fusing these two streams of data together. These two streams of data are complementary in that:

- The gyroscopic Euler angle measurement will drift over time but the coarse estimate of the Euler angles ϕ and θ will not.
- The coarse estimate of the Euler angles ϕ and θ produced by the accelerometer will contain far more noise than is present with the gyroscopes Euler angle estimate.

Thus fusing the two low quality estimates will bound the gyroscopic drift and generate good quality estimates of the Euler angles ϕ and θ. The mechanism that performs this sensor fusion is the Kalman filter and can be implemented through a set of recursive mathematical equations. The theory of the Kalman filter will be presented first followed by the Kalman filter equations which will achieve Euler angle estimation of ϕ and θ.

4.3.3.1 Standard Kalman filter theory

A standard Kalman filter can be implemented via the following recursive equations. The equations are divided into two separate updates being the time update and the measurement update. The connection and control flow between these two updates can be seen in Figure 4.1. The equations required for the time update are:

$$\hat{x}_k^- = A_{k-1}\hat{x}_{k-1} + B_k u_k$$

$$P_k^- = A_{k-1}P_{k-1}A'_{k-1} + Q_{k-1}$$

Where $\hat{x}_k$ is the state estimate at time k, A_k is the state transition matrix, $\hat{x}_k^-$ is the one step ahead prediction of the state at time $k-1$, B_k is the control input matrix, u_k is the control vector, P_k^- is the state prediction covariance and Q_k is the process covariance matrix. The equations required for the measurement update are:

$$K_k = P_k^- H'_k (H_k P_k^- H'_k + R_k)^{-1}$$

$$P_k = (I - K_k H_k)P_k^-$$

$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - H_k\hat{x}_k^-)$$

Where K_k is the Kalman gain, H_k is the observation matrix, R_k is the measurement covariance matrix, P_k is the error covariance matrix and $\hat{x}_k$ is the Kalman filter estimate of the state. The Kalman filter needs to be initialised at $k = 0$ to begin the estimation process where values of $\hat{x}_0$ and P_0 need to be specified. P_0 is generally initialised to be a large number e.g.

$$P_0 = 1000I_{4\times 4}$$

The choice of $\hat{x}_0$ and P_0 are not critical to the operation of the Kalman filter since the filter will eventually become independent of these values given a large enough time scale. However the values of Q_{k-1} and R_k are extremely important to the operation of the Kalman filter and each matrix needs to be determined for each filter implementation.

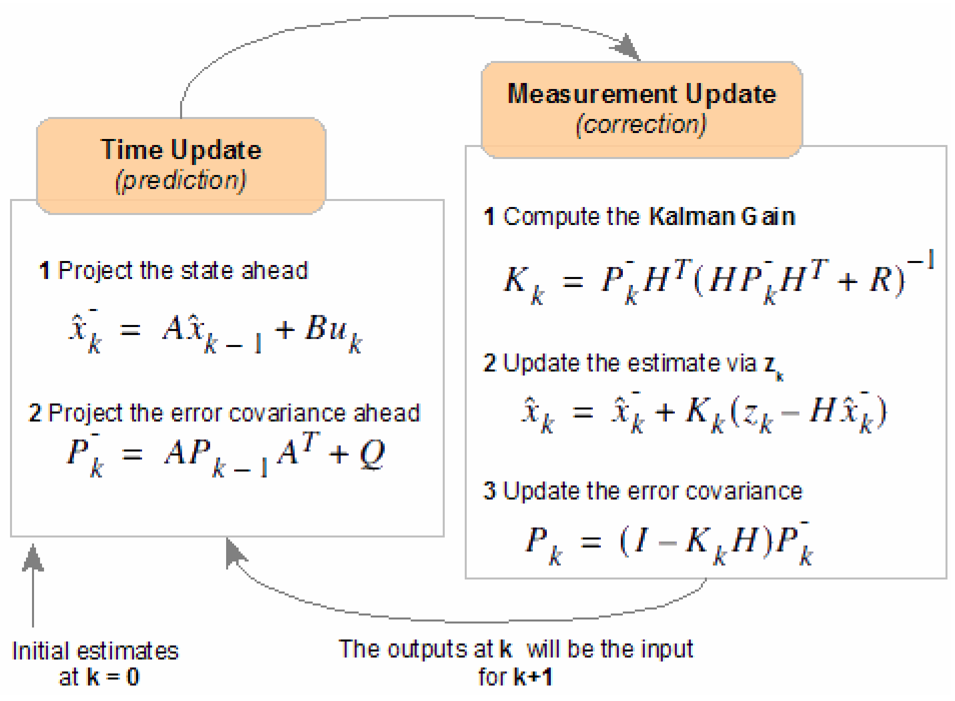

Time Update
(prediction)
1 Project the state ahead
$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_k$
2 Project the error covariance ahead
$P_k^- = AP_{k-1}A^T + Q$
Measurement Update
(correction)
1 Compute the Kalman Gain
$K_k = P_k^- H^T(HP_k^- H^T + R)^{-1}$
2 Update the estimate via z_k
$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-)$
3 Update the error covariance
$P_k = (I - K_kH)P_k^-$
Initial estimates at k = 0
The outputs at k will be the input for k+1

Figure 4.1 - Kalman Filter time and measurement updates

4.3.3.2 Kalman filter estimation model for ϕ and θ

The first step in Kalman filter design is to determine how to estimate the model of the system. The initial Kalman filter design in this section will involve a separate Kalman filter for each Euler angle being estimated. The main reason for this is speed of computation. Tthe more states a Kalman filter has the slower it will operate due to the matrix inversion required in the Kalman gain K_k equation. The estimation model for ϕ and θ are identical so only ϕ will be presented.

The following matrices and values need to be determined to develop the estimation model for ϕ: x_k, A_k, B_k, u_k, Q_k, H_k and R_k. The states x_k that will be tracked for the Kalman filter when estimating ϕ are:

$$x_k = \begin{bmatrix} \phi \\ \phi_{bias} \end{bmatrix}$$

Where ϕ is equal to the Euler angle being tracked and ϕ_{bias} is equal to the bias that has developed for the gyroscope measuring that Euler angle's rate (in this case it is ϕ). The state transition matrix A_k is equal to:

$$A_k = \begin{bmatrix} 1 & -dT \\ 0 & 1 \end{bmatrix}$$

Where dT is the time interval between each Kalman filter update step. The control input matrix B_k is equal to:

$$B_k = \begin{bmatrix} dT \\ 0 \end{bmatrix}$$

The control vector u_k is equal to the estimated Euler rate for the Euler angle being measured. In this case u_k will be equal to $\dot{\phi}$ which will be measured from the ϕ axis gyroscope after a EWMA filter has been applied. The process covariance matrix Q_k is equal to:

$$Q_k = \begin{bmatrix} \sigma_{\phi}^2 & 0 \\ 0 & \sigma_{\phi_{bias}}^2 \end{bmatrix}$$

Where σ_{ϕ} is the variance of the ϕ model estimate and $\sigma_{\phi_{bias}}$ is the variance of the ϕ_{bias} model estimate. The observation matrix H_k is equal to:

$$H_k = [1 \quad 0]$$

Finally the measurement covariance matrix R_k is equal to:

$$R_k = \sigma_{\phi_{measure}}^2$$

Where $\sigma_{\phi_{measure}}$ is the variance of the ϕ measurement.

4.3.3.3 Kalman filter time update for ϕ and θ

The time update step in the Kalman filter for ϕ and θ will be the same since the estimation models for ϕ and θ are identical. Thus the one step ahead prediction of the states $\hat{x}_k^-$ in the time update can be computed by substituting in the matrices and values defined in the estimation model:

$$\hat{x}_k^- = A_{k-1}\hat{x}_{k-1} + B_k u_k$$

$$\therefore \begin{bmatrix} \hat{\phi}_k^- \\ \hat{\phi}_{bias_k}^- \end{bmatrix} = \begin{bmatrix} 1 & -dT \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \hat{\phi}_{k-1} \\ \hat{\phi}_{bias_{k-1}} \end{bmatrix} + \begin{bmatrix} dT \\ 0 \end{bmatrix} \dot{\phi}$$

Thus the equations which define the one step ahead prediction of $\hat{\phi}_k^-$ and $\hat{\phi}_{bias_k}^-$ are equal to:

$$\hat{\phi}_k^- = \hat{\phi}_{k-1} - \hat{\phi}_{bias_{k-1}} dT + \dot{\phi} dT$$

$$\therefore \boldsymbol{\hat{\phi}_k^- = \hat{\phi}_{k-1} + (\dot{\phi} - \hat{\phi}_{bias_{k-1}}) dT}$$

$$\therefore \boldsymbol{\hat{\phi}_{bias_k}^- = \hat{\phi}_{bias_{k-1}}}$$

The time update is completed by computing state prediction covariance P_k^-. Substituting values of the estimation model into the P_k^- formula yields:

$$P_k^- = A_{k-1} P_{k-1} A'_{k-1} + Q_{k-1}$$

$$P_k^- = \begin{bmatrix} 1 & -dT \\ 0 & 1 \end{bmatrix} P_{k-1} \begin{bmatrix} 1 & 0 \\ -dT & 1 \end{bmatrix} + \begin{bmatrix} \sigma_\phi^2 & 0 \\ 0 & \sigma_{\phi_{bias}}^2 \end{bmatrix}$$

Let P_k be equal to:

$$P_k = \begin{bmatrix} P_{00} & P_{01} \\ P_{10} & P_{11} \end{bmatrix}$$

Substituting this value into the formula for P_k^- creates:

$$\begin{bmatrix} P_{00_k}^- & P_{01_k}^- \\ P_{10_k}^- & P_{11_k}^- \end{bmatrix} = \begin{bmatrix} 1 & -dT \\ 0 & 1 \end{bmatrix} \begin{bmatrix} P_{00_{k-1}} & P_{01_{k-1}} \\ P_{10_{k-1}} & P_{11_{k-1}} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -dT & 1 \end{bmatrix} + \begin{bmatrix} \sigma_\phi^2 & 0 \\ 0 & \sigma_{\phi_{bias}}^2 \end{bmatrix}$$

Therefore the equations which define the state prediction covariance P_k^- are equal to:

$$\therefore \boldsymbol{P_{00_k}^- = P_{00_{k-1}} - P_{10_{k-1}} dT - P_{01_{k-1}} dT + P_{11_{k-1}} dT^2 + \sigma_\phi^2}$$

$$\therefore \boldsymbol{P_{01_k}^- = P_{01_{k-1}} - P_{11_{k-1}} dT}$$

$$\therefore \boldsymbol{P_{10_k}^- = P_{10_{k-1}} - P_{11_{k-1}} dT}$$

$$\therefore \boldsymbol{P_{11_k}^- = P_{11_{k-1}} + \sigma_{\phi_{bias}}^2}$$

Note that with this implementation the term $P_{11_{k-1}} dT^2$ is generally ignored since dT^2 will be extremely small assuming small time step increments (which is true for +50Hz updates).

4.3.3.4 Kalman filter measurement update for $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$

The measurement update for ϕ and θ will be the same since the estimation model and time updates for ϕ and θ are identical. The first equation to be executed in the measurement update is to compute the Kalman gain K_k by the following equation:

$$K_k = P_k^- H_k' (H_k P_k^- H_k' + R_k)^{-1}$$

The inversion term is typically labelled S for innovation covariance. Substituting values for the estimation model into the formula for S creates:

$$S = H_k P_k^- H_k' + R_k$$

$$S = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} P_{00_k}^- & P_{01_k}^- \\ P_{10_k}^- & P_{11_k}^- \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \sigma_{\phi_{measure}}^2$$

$$S = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} P_{00_k}^- \\ P_{10_k}^- \end{bmatrix} + \sigma_{\phi_{measure}}^2$$

$$\therefore \boldsymbol{S = P_{00_k}^- + \sigma_{\phi_{measure}}^2}$$

Thus the Kalman gain K_k is equal to:

$$K_k = P_k^- H_k' (S)^{-1}$$

$$\begin{bmatrix} K_{00_k} \\ K_{10_k} \end{bmatrix} = \begin{bmatrix} P_{00_k}^- & P_{01_k}^- \\ P_{10_k}^- & P_{11_k}^- \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} (P_{00_k}^- + \sigma_{\phi_{measure}}^2)^{-1}$$

$$\begin{bmatrix} K_{00_k} \\ K_{10_k} \end{bmatrix} = \begin{bmatrix} P_{00_k}^- \\ P_{10_k}^- \end{bmatrix} (P_{00_k}^- + \sigma_{\phi_{measure}}^2)^{-1}$$

Therefore the equations which define the Kalman gain K_k are equal to:

$$K_{00_k} = P_{00_k}^- (P_{00_k}^- + \sigma_{\phi_{measure}}^2)^{-1}$$

$$\therefore \boldsymbol{K_{00_k} = \frac{P_{00_k}^-}{P_{00_k}^- + \sigma_{\phi_{measure}}^2}}$$

$$K_{10_k} = P_{10_k}^- (P_{00_k}^- + \sigma_{\phi_{measure}}^2)^{-1}$$

$$\therefore \boldsymbol{K_{10_k} = \frac{P_{10_k}^-}{P_{00_k}^- + \sigma_{\phi_{measure}}^2}}$$

The second measurement update equation is to modify the error covariance P_k described by:

$$P_k = (I - K_k H_k) P_k^-$$

Where I is the identity matrix with dimension equal to the number of states or $N = 2$. Substituting values for the estimation model into the above formula yields:

$$P_k = (\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} K_{00_k} \\ K_{10_k} \end{bmatrix} [1 \quad 0]) \begin{bmatrix} P_{00_k}^- & P_{01_k}^- \\ P_{10_k}^- & P_{11_k}^- \end{bmatrix}$$

$$\begin{bmatrix} P_{00_{k-1}} & P_{01_{k-1}} \\ P_{10_{k-1}} & P_{11_{k-1}} \end{bmatrix} = \begin{bmatrix} 1 - K_{00_k} & 0 \\ -K_{10_k} & 1 \end{bmatrix} \begin{bmatrix} P_{00_k}^- & P_{01_k}^- \\ P_{10_k}^- & P_{11_k}^- \end{bmatrix}$$

$$\begin{bmatrix} P_{00_{k-1}} & P_{01_{k-1}} \\ P_{10_{k-1}} & P_{11_{k-1}} \end{bmatrix} = \begin{bmatrix} P_{00_k}^- - P_{00_k}^- K_{00_k} & P_{01_k}^- - P_{01_k}^- K_{00_k} \\ P_{10_k}^- - P_{00_k}^- K_{10_k} & P_{11_k}^- - P_{01_k}^- K_{10_k} \end{bmatrix}$$

Therefore the equations which define the error covariance P_k are equal to:

$$\therefore \boldsymbol{P_{00_{k-1}} = P_{00_k}^- - P_{00_k}^- K_{00_k}}$$

$$\therefore \boldsymbol{P_{01_{k-1}} = P_{01_k}^- - P_{01_k}^- K_{00_k}}$$

$$\therefore \boldsymbol{P_{10_{k-1}} = P_{10_k}^- - P_{00_k}^- K_{10_k}}$$

$$\therefore \boldsymbol{P_{11_{k-1}} = P_{11_k}^- - P_{01_k}^- K_{10_k}}$$

The final equation which completes the measurement update and hence the Kalman filter process is the state estimate $\hat{x}_k$. The state estimate is described by the following formula:

$$\hat{x}_k = \hat{x}_k^- + K_k (y_k - H_k \hat{x}_k^-)$$

The innovation residual or error term in the above equation is denoted:

$$\tilde{y}_k = y_k - H_k \hat{x}_k^-$$

Where y_k is the coarse Euler angle measurement calculated from the accelerometers. The coarse measurement for ϕ is equal to:

$$\phi_{coarse} = \arctan\left(\frac{A_y}{\sqrt{A_x^2 + A_z^2}}\right)$$

The coarse measurement for θ is equal to:

$$\theta_{coarse} = \arctan\left(\frac{A_x}{\sqrt{A_y^2 + A_z^2}}\right)$$

Thus y_k can be represented as (demonstrating estimation of the Euler angle ϕ):

$$\tilde{y}_k = \begin{bmatrix} \phi_{coarse} \\ 0 \end{bmatrix}$$

Substituting the above expression and the estimation model values into the innovation residual equation $\tilde{y}_k$ yields:

$$\tilde{y}_k = \begin{bmatrix} \phi_{coarse} \\ 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \hat{\phi}_k^- \\ \hat{\phi}_{bias_k}^- \end{bmatrix}$$

$$\tilde{y}_k = \begin{bmatrix} \phi_{coarse} - \hat{\phi}_k^- \\ 0 \end{bmatrix}$$

Substituting this value and the estimation model values into the formula which describes the state estimate $\hat{x}_k$ creates:

$$\begin{bmatrix} \hat{\phi}_k \\ \hat{\phi}_{bias_k} \end{bmatrix} = \begin{bmatrix} \hat{\phi}_k^- \\ \hat{\phi}_{bias_k}^- \end{bmatrix} + \begin{bmatrix} K_{00_k} \\ K_{10_k} \end{bmatrix} \begin{bmatrix} \phi_{coarse} - \hat{\phi}_k^- \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} \hat{\phi}_k \\ \hat{\phi}_{bias_k} \end{bmatrix} = \begin{bmatrix} \hat{\phi}_k^- + K_{00_k}(\phi_{coarse} - \hat{\phi}_k^-) \\ \hat{\phi}_{bias_k}^- + K_{10_k}(\phi_{coarse} - \hat{\phi}_k^-) \end{bmatrix}$$

Thus the equations which describe the state estimate $\hat{x}_k$ are equal to:

$$\therefore \boldsymbol{\hat{\phi}_k = \hat{\phi}_k^- + K_{00_k}(\phi_{coarse} - \hat{\phi}_k^-)}$$

$$\therefore \boldsymbol{\hat{\phi}_{bias_k} = \hat{\phi}_{bias_k}^- + K_{10_k}(\phi_{coarse} - \hat{\phi}_k^-)}$$

The value for $\hat{\phi}_k$ is the Kalman filter estimate for the Euler angle ϕ at time interval k. Thus both the data streams of the gyroscope and the accelerometers have been combined to produce a high quality estimate of the Euler angle state. The code that implements the state estimates for ϕ and θ can be seen in Appendix H (includes a header file `kfb.h` and a source file `kfb.c`).

4.3.4 Euler angle estimation $(\boldsymbol{\psi})$

A similar design approach is used for estimating the Euler angle ψ as was designed for ϕ and θ. Two differences exist between the Kalman filter designs. The first is that the accelerometers cannot produce a coarse estimate for ψ that is usable. Instead, another sensor is needed to be employed that can directly measure the ψ Euler angle. This sensor was chosen to be a magnetic compass thus the coarse measurement for ψ in the Kalman filter measurement update is equal to:

$$\psi_{coarse} = \psi_{compass}$$

The measurement update of the Kalman filter is also different. A discontinuity exists for the Euler angle ψ in that it can routinely cross between $0°$ to $360°$ and vice versa. The Kalman filter measurement update needs to be aware of this behaviour otherwise it will perform full circular rotations when the discontinuity boundary has been crossed. This can be achieved by modifying the innovation residual or error term $\tilde{y}_k$ calculation. The following coding algorithm can be used to calculate the correct value for the error term if a boundary has been crossed by the quadrotor platform:

```
% Check if measurement is < 360 and estimate is > 0
If (($\psi_{compass}$ - $\hat{\psi}_k^-$) > 180)
    $\tilde{y}_k$ = $\psi_{compass}$ - (360 + $\hat{\psi}_k^-$);
    % Check if measurement is < 0 and estimate is > 360
    else If (($\psi_{compass}$ - $\hat{\psi}_k^-$) < -180)
        $\tilde{y}_k$ = $\psi_{compass}$ + (360 - $\hat{\psi}_k^-$);
        % Else normal measurement update
        else
        $\tilde{y}_k$ = $\psi_{compass}$ - $\hat{\psi}_k^-$;
      End
End
% Perform bound checking
$\hat{\psi}_k$ = mod($\hat{\psi}_k^-$ + ($\tilde{y}_k$*$K_{00_k}$),360);
$\hat{\psi}_{bias_k}$ = mod($\hat{\psi}_{bias_k}^-$ + ($\tilde{y}_k$*$K_{10_k}$),360);
```

The code that implements that state estimates for ψ can once again be seen in Appendix H (includes a header file `kfb.h` and a source file `kfb.c`). Note the time update for the Euler angle ψ is identical to that of ϕ and θ.

4.3.5 Euler angle estimation (Vicon alternative)

The Vicon system was used to verify the accuracy of the Euler angles obtained through the Kalman filtering process described in the previous sections. The Vicon system can provide the Euler angles of the quadrotor platform by querying the Vicon Tracker program. The function which is used to do this is `GetSegmentGlobalRotationEulerXYZ` as documented in [RD/11]. It is important to note that this function returns the Euler angles in the XYZ co-ordinate system hence the Euler angles are expressed in a world frame attached to the Vicon system. The attitude controller requires the Euler angles to be in a ZXY co-ordinate frame which is commonly used in aeronautical applications. Thus a rotation system needs to be constructed to transform the Euler angles from the XYZ co-ordinate system to the required ZXY co-ordinate frame.

The rotation system involves converting the XYZ co-ordinate system to an intermediate co-ordinate frame ZYX where the ZXY Euler angles can be calculated. The reason for this intermediate conversion to ZYX is that the Vicon system provides a rotation matrix which is equal to the ZYX co-ordinate rotation. This rotation can be built from the following 3 separate axis rotations:

1. Rotate about z by the yaw angle ψ
2. Rotate about y by the theta angle θ
3. Rotate about x by the phi angle ϕ

The overall rotation can be represented as:

$$R = R_1(\phi)R_2(\theta)R_3(\psi)$$

$$R = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\psi) & -\sin(\psi) \\ 0 & \sin(\psi) & \cos(\psi) \end{bmatrix}$$

$$\therefore \boldsymbol{R} = \begin{bmatrix} \boldsymbol{C(\phi)C(\theta)} & \boldsymbol{C(\phi)S(\theta)S(\psi) - S(\phi)C(\psi)} & \boldsymbol{C(\phi)S(\theta)C(\psi) + S(\phi)S(\psi)} \\ \boldsymbol{S(\phi)C(\theta)} & \boldsymbol{S(\phi)S(\theta)S(\psi) + C(\phi)C(\psi)} & \boldsymbol{S(\phi)S(\theta)C(\psi) - C(\phi)S(\psi)} \\ \boldsymbol{-S(\theta)} & \boldsymbol{C(\theta)S(\psi)} & \boldsymbol{C(\theta)C(\psi)} \end{bmatrix}$$

The function which returns this rotation is `GetSegmentGlobalRotationMatrix` as is documented in [RD/11]. From this resulting rotation matrix we can then use standard Direct Cosine Matrix (DCM) conversions to extract the Euler angles in the ZXY frame. These conversions are as follows:

$$\phi_{ZXY} = asin(R(3,2))$$

$$\theta_{ZXY} = -atan2(R(3,1), R(3,3))$$

$$\psi_{ZXY} = -atan2(R(1,2), R(2,2))$$

Where $R(i,j)$ is equal to the expression contained in the ith row and jth column in the derived rotation matrix R. These Euler angles from the Vicon system can then be used to compare the accuracy of the Euler angles obtained through Kalman filtering.

4.3.6 X,Y and Z accelerations $(\ddot{\boldsymbol{x}}, \ddot{\boldsymbol{y}}, \ddot{\boldsymbol{z}})$

The 3 accelerations in the x, y and z axes are measured by 3 accelerometers located in the Sensor dynamics 6DOF IMU. These accelerometers are aligned to the body axis of the quadrotor platform and hence can directly measure the required body accelerations. The table below lists the accelerometers characteristics as specified in the datasheet [RD/10]. As can be seen in Table 4.4, the accelerometers are not perfect sensors and contain some noise error. This measurement error can be mitigated by applying an exponentially weighted moving average (EWMA) filter which has been previously discussed.

Table 4.4 - Accelerometer characteristics [RD/10]

Parameter	MR1	MR2	Unit	Condition
Measurement Range	± 2	± 5	g	
Resolution	0.002031	0.004062	$(\mathrm{m/s^2})$ /bit	True 16 bit $1\mathrm{g} = 9.80665\ \mathrm{m/s^2}$
Max RMS noise	3	4	mg	Bandwidth: MR1: 40Hz MR2: 100Hz
Max zero g at RT	± 0.05		g	Zero setting at $25°\mathrm{C}$
Max temperature drift of zero g bias	± 0.06		g	Over full temp range
Max sensitivity error	± 2.5		$\%$	Over full temp range
Max linearity error, versus best fit	± 0.2		$\%$	Over full temp range

4.3.7 X,Y and Z position $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$

The Vicon sensor can directly provide the x, y and z displacement of the quadrotor platform. The function which is used to do this is `GetSegmentGlobalTranslation` as documented in [RD/11]. This function returns the difference between the centre of mass position of the platform to the origin of the Vicon system. The centre of mass position of the platform is specified by the user in the Vicon tracker program. The origin of the Vicon system is also specified by the user by performing the Vicon tracker calibration routine.

4.3.8 X,Y and Z velocity $(\dot{\boldsymbol{x}}, \dot{\boldsymbol{y}}, \dot{\boldsymbol{z}})$

The $\dot{x}, \dot{y}, \dot{z}$ velocities can be calculated from the Vicon sensor by taking the time derivative of the x, y, z position data. As an example, the calculation for the X velocity $\dot{x}$ will be equal to:

$$\dot{x}_k = \frac{x_k - x_{k-1}}{dT}$$

Where $\dot{x}_k$ is the estimate of the X velocity, x_k is the current measurement of the x displacement, x_{k-1} is the previous measurement of the x displacement and dT is the time interval between measurements. Additional filtering may be required for the X, Y and Z velocities depending on the quality of the time derivative calculation as mentioned in section 4.2. An EWMA filter will be implemented and tuned depending on the quality of the X,Y and Z velocity estimation data.

4.3.9 Z position and velocity $(z, \dot{z})$ (Ultrasonic alternative)

The z displacement of the quadrotor platform can also be measured from the ultrasonic sensor located on the platform. The ultrasonic sensor is strapped down to the quadrotor platform thus the z displacement will be a function of the ultrasonic measurement $z_{measure}$ and the Euler angles ϕ and θ. This can be described by the following equation:

$$z = \cos(\phi)\cos(\theta)z_{measure}$$

The ultrasonic measurement is obtained by ADC thus a filter will need to be applied to the measurements taken from the ultrasonic sensor. The filter employed will be an EWMA filter. The velocity can be calculated by taking the time derivative of the z position data from the ultrasonic sensor i.e.

$$\dot{z}_k = \frac{z_k - z_{k-1}}{dT}$$

Where $\dot{z}_k$ is the estimate of the X velocity, z_k is the current measurement of the z displacement, z_{k-1} is the previous measurement of the z displacement and dT is the time interval between measurements.

4.4 State estimation software library

The state estimation design as specified in this chapter utilises the following functions in the C programming language:

- attitudeFilterInitialiseB: Initialise the attitude Kalman filter for all axes.
- attitudeFilterB: Perform the Kalman filter routine for all axes.
- kFilterTimeUpdate: Execute the time update on the specified axes.
- kFilterMeasureUpdate: Execute the measurement update on the specified axes.
- kFilterMeasurePsiUpdate: Adjusted measurement update for ψ axis.
- coarsePitchAngle: Calculate the coarse pitch angle measurement from accelerometers.
- coarseRollAngle: Calculate the coarse roll angle measurement from accelerometers.

- accLPF: Low pass filter (EWMA) the accelerometer values.
- rateLPF: Low pass filter (EWMA) the rate values.
- compassLPF: Low pass filter (EWMA) the compass heading.
- altLPF: Low pass filter (EWMA) the ultrasonic readings.
- calibrateEulerAngles: Calibrate angular offset in Euler angles ϕ and θ by taking a 10 second average of the readings.
- printkFilterData: Print all Kalman filter data to a log file on the flight computer.

The header file `kfb.h` and the source file `kfc.c` contain the implementation of these functions and can be found in Appendix H.

Chapter 5 Flight Computer Testing

5.1 Acceptance tests

Testing the flight computer was important to achieve the system requirements as listed in section 3.2. Two acceptance tests were conducted by modifying certain inputs to the quadrotor platform and observing the outputs via the GCS. The sent outputs included sensor and state data, which were stored in numerous log files by the GCS for future MATLAB plotting and comparison. Meeting the system requirements for the flight computer required the following acceptance tests to be met:

Table 5.1 - Acceptance tests required to meet the flight computer system requirements

Acceptance Test	System Requirement	Description
AT-15	SR-D-05	Collect the following sensor data at an average rate of 50 Hz: • IMU • Arduino • Compass (connected via the Arduino) • Ultrasonic sensor (connected via the Arduino)
AT-16	SR-D-06	Collect the following sensor data at an average rate of 50 Hz: • Battery voltage sensor (connected via the Arduino) • Mode control unit (radio control status)

5.2 AT-15

Achieving this acceptance test involved the data collection from numerous devices including: IMU, Arduino, compass and ultrasonic sensor. The data collection and processing also had to be executed at an average rate of 50 Hz. The data collection for each device will be presented in the following sections either by command line outputs or logged data plotted in MATLAB.

5.2.1 IMU sensor data

5.2.1.1IMU reconfiguration

Before sensor data could be collected by the IMU, the IMU itself had to be reconfigured to allow:

- MR2 rate output
- MR2 accelerometer output

- 115200 baudrate

This was achieved by utilising the function `setIMUconfig` in the IMU software library (refer to Appendix A). The output of this was function as follows when executed on the flight computer Overo Fire board:

```
>> Success - Connected to IMU
>> Set CC0|OK!
>> Read C|C0
>> Set D7F|OK!
>> Read D|7F
>> Change baudrate 3 |OK!

>> IMU reconfigured
```

This reconfiguration process had to occur every time the Overo Fire was turned on.

5.2.1.2IMU 16 bit data collection and conversion

The following summarised gyroscopic rate and accelerometer data was gathered from the IMU after reconfiguration had occurred:

```
FFB9,FFB6,FFE4,FFDB,FFC1,0986,0088,0000,0000,2A
FFCF,FFC3,FFB8,FFD5,FFCB,096F,0089,0000,0000,0A
FF85,FFA7,FFE1,FFD4,FFCD,09AD,0089,0000,0000,1E
FFB2,FFB6,FFEE,FFD2,FFC7,09A6,0088,0000,0000,21
FF87,FFB6,000B,FFD9,FFC3,098C,0088,0000,0000,63
FFA9,FFB3,FFEE,FFD4,FFD6,097A,0089,0000,0000,1A
FFB5,FF90,FFCE,FFD6,FFCC,097C,0089,0000,0000,1B
FFD2,FFB5,FFD6,FFD9,FFD0,0992,0088,0000,0000,3B
FFAF,FFB3,FFE6,FFD4,FFCE,0961,0088,0000,0000,20
FF8B,FFA0,FFF1,FFDE,FFBE,097F,0088,0000,0000,0F
```

The conversion of this 16 bit data using the MR2 rate and accelerometer conversions led to the following output sentences:

Format:
| MR2 – Rate X | MR2 – Rate Y | MR2 – Rate Z | MR2 – Acc X | MR2 – Acc Y | MR2 – Acc Z |

```
-1.0920,-1.1388,-0.4212,-0.1462,-0.2518, 9.9032
-0.7488,-0.9360,-1.1076,-0.1706,-0.2112, 9.8097
-1.9032,-1.3728,-0.4680,-0.1747,-0.2031,10.0616
```

```
-1.2012,-1.1388,-0.2652,-0.1828,-0.2275,10.0331
-1.8720,-1.1388, 0.1716,-0.1544,-0.2437, 9.9275
-1.3416,-1.1856,-0.2652,-0.1747,-0.1665, 9.8544
-1.1544,-1.7316,-0.7644,-0.1665,-0.2072, 9.8625
-0.7020,-1.1544,-0.6396,-0.1544,-0.1909, 9.9519
-1.2480,-1.1856,-0.3900,-0.1747,-0.1990, 9.7529
-1.8096,-1.4820,-0.2184,-0.1340,-0.2640, 9.8747
```

This data was then plotted as can be seen in the following figures. These figures include outputs for MR2 Rate X (Figure 5.1), MR2 Rate Y (Figure 5.2), MR2 Rate Z (Figure 5.3), MR2 Acc X (Figure 5.4), MR2 Acc Y (Figure 5.5) and MR2 Acc Z (Figure 5.6). The outputs contain the effects of specific manoeuvres, which were applied to the quadrotor platform (i.e. rolling, pitching and yawing). The outputs gathered matched the manoeuvres that were applied at the time of data collection. These outputs were collected and processed in the state estimation thread at a mean update rate of 88.8 Hz (refer to Figure 5.7).

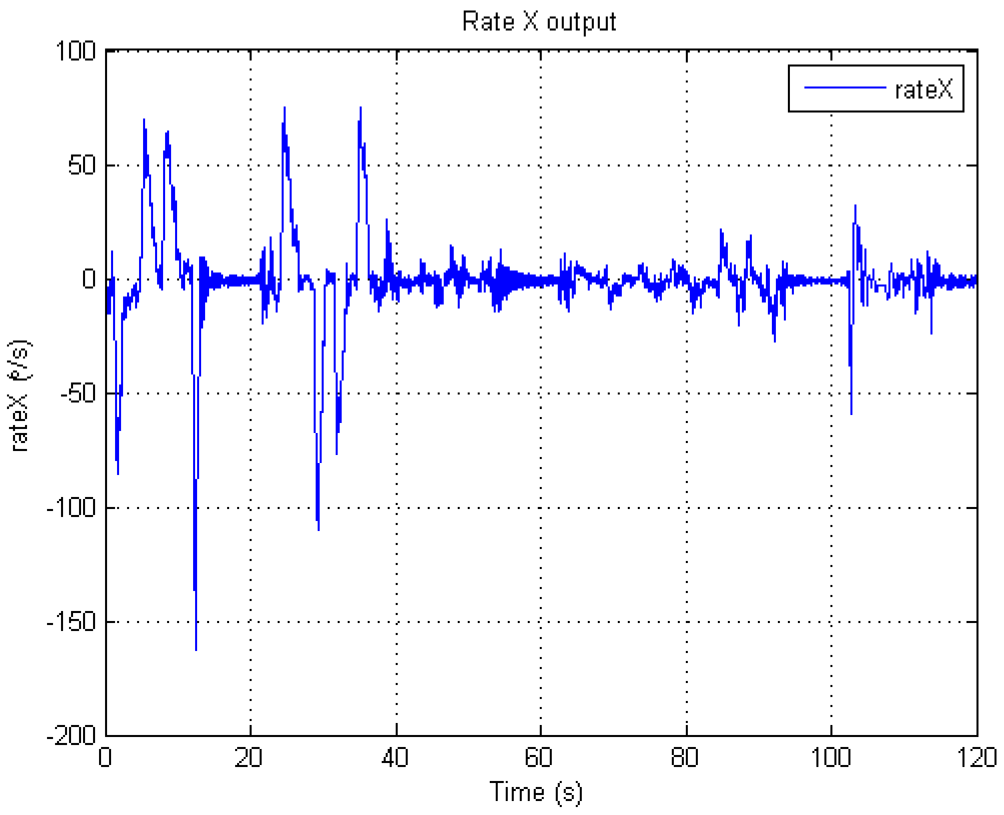


Figure 5.1 - MR2 Rate X sensor data (IMU)

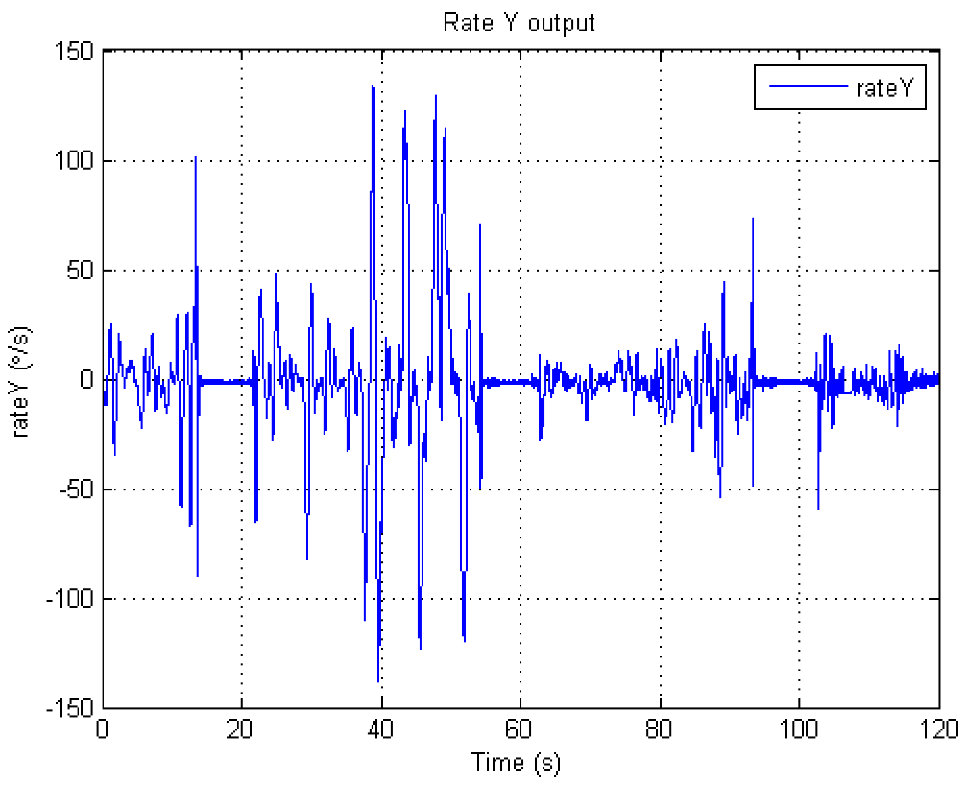


Figure 5.2 – MR2 Rate Y sensor data (IMU)

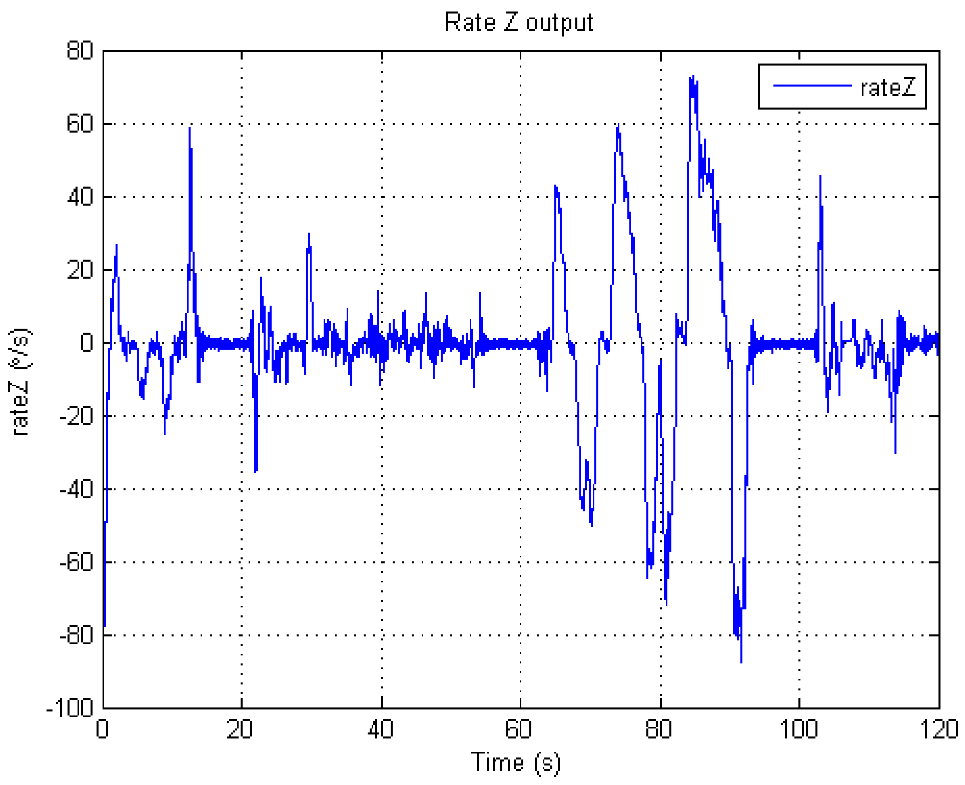


Figure 5.3 - MR2 Rate Z sensor data (IMU)

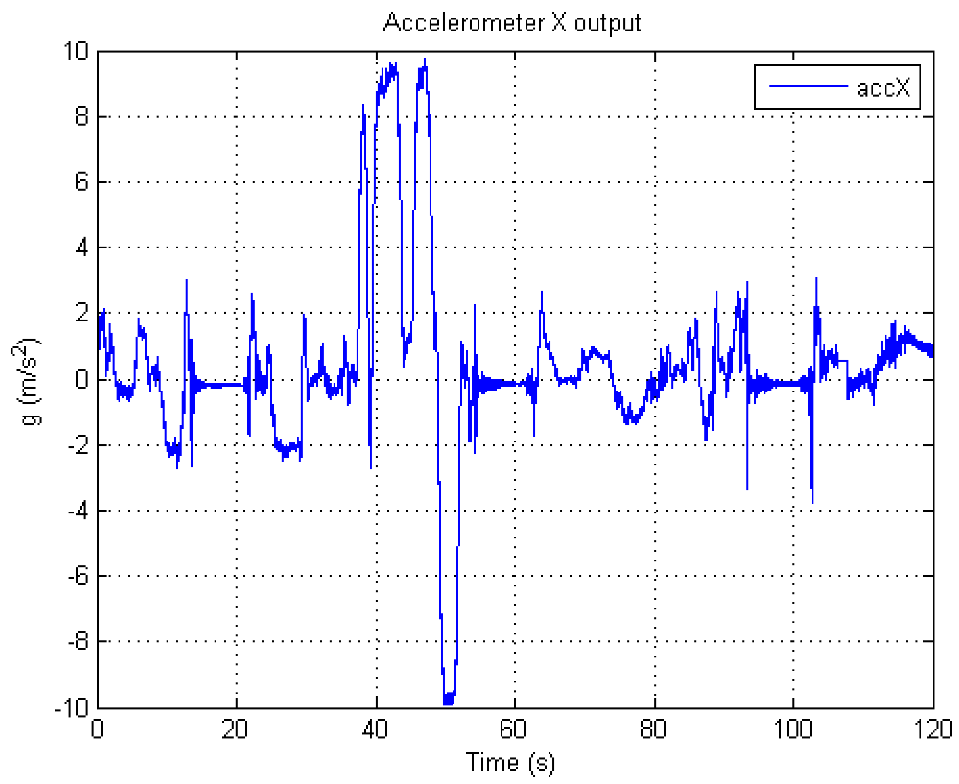


Figure 5.4 - MR2 Acc X sensor data (IMU)

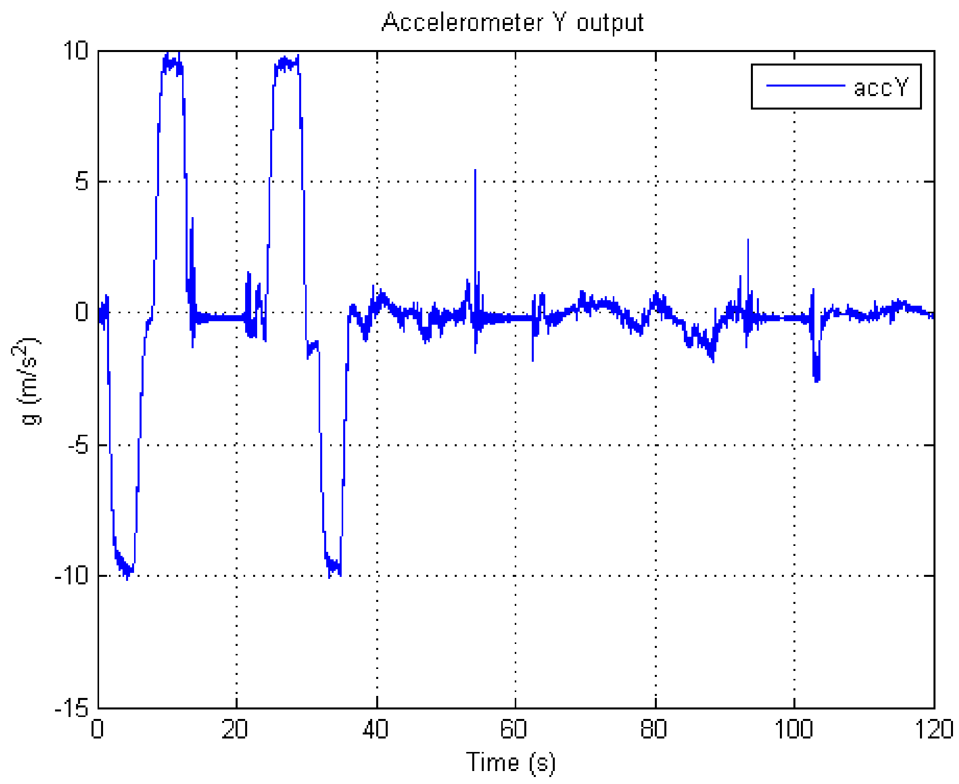


Figure 5.5 - MR2 Acc Y sensor data (IMU)

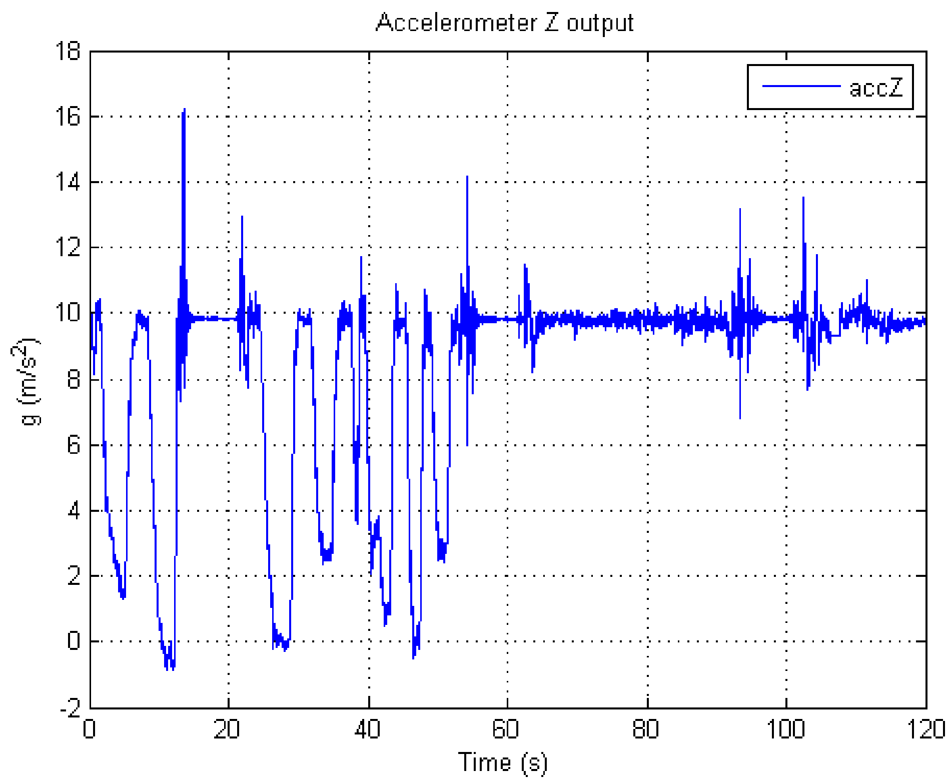


Figure 5.6 - MR2 Acc Z sensor data (IMU)

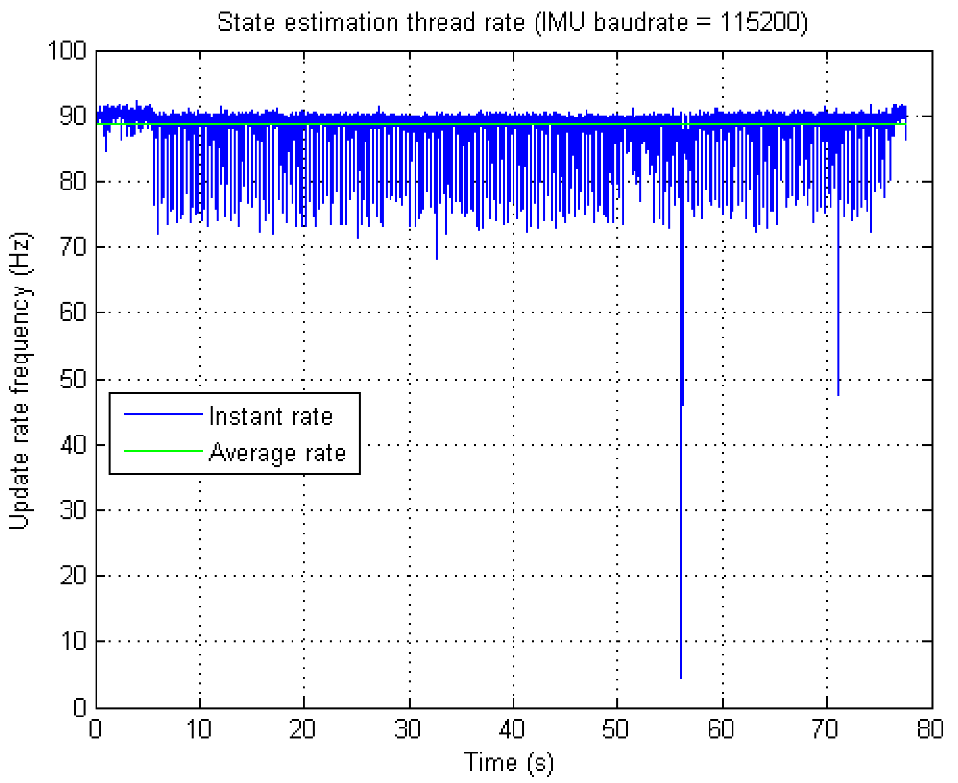


Figure 5.7 - State estimation thread update rate

5.3 Arduino sensors

5.3.1 Arduino sensor data

The Arduino has the following sentence sensor data output:

Format: **Chhh.h,Vvv.vvv,Aa.aaa**

The following summarised sensor output was obtained from the Arduino using the above format (from the flight computer command line output). This shows that the flight computer is successfully receiving the Arduino sensor data output:

```
C226.10,V14.085,A0.190
C226.20,V14.085,A0.312
C226.20,V14.085,A0.298
C226.20,V14.085,A0.312
C226.20,V14.085,A0.312
C226.20,V14.085,A0.312
C226.20,V14.085,A0.285
C226.20,V14.085,A0.312
C226.20,V14.085,A0.312
C226.20,V14.085,A0.298
C226.10,V14.085,A0.190
C226.20,V14.085,A0.312
C226.20,V14.085,A0.298
C226.20,V14.085,A0.312
C226.20,V14.085,A0.312
C226.20,V14.085,A0.312
C226.20,V14.085,A0.285
C226.20,V14.085,A0.312
C226.20,V14.085,A0.312
C226.20,V14.085,A0.298
```

5.3.2 Magnetic compass and ultrasonic sensor data

Satisfying AT-15 from the point of view of the Arduino involved the collection and processing of the magnetic compass and ultrasonic sensor at an average rate of 50Hz. The collected data from the magnetic compass and ultrasonic sensor (as seen by the command line output above) was plotted in MATALB (refer to Figure 5.8 and Figure 5.9). As can be seen in these figures, both sensors are reacting to changes to the quadrotor platform. The sensor data is collected and processed by the Arduino thread in the flight computer at a mean update rate of 91.1 Hz (refer to Figure 5.10).

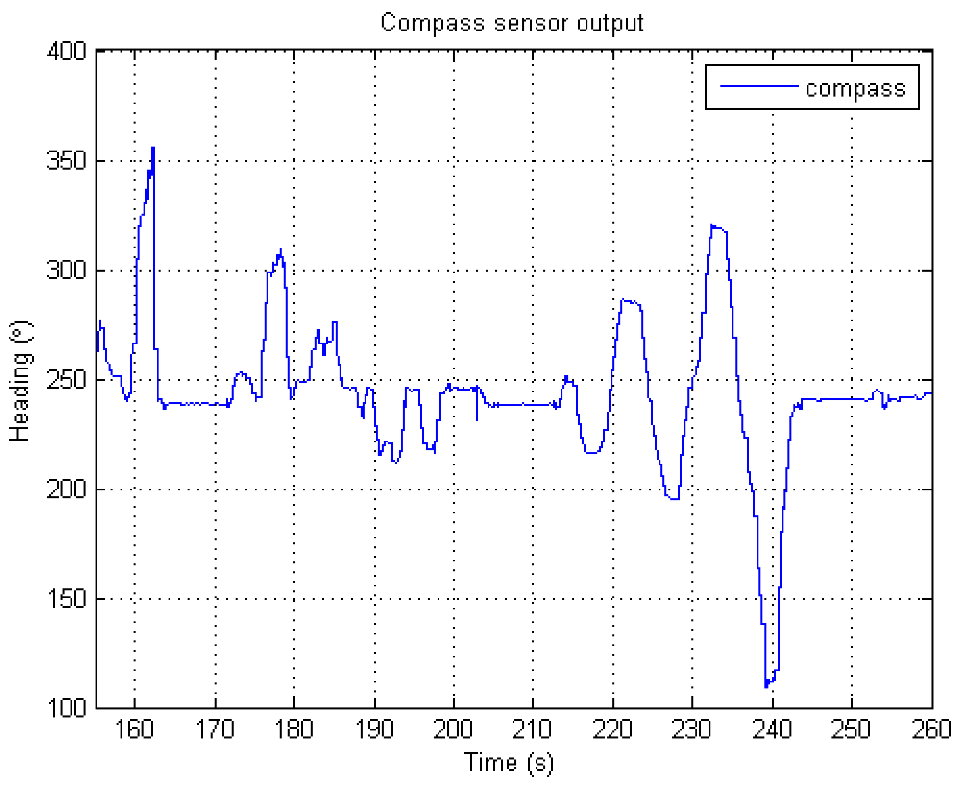


Figure 5.8 - Compass sensor output (from Arduino)

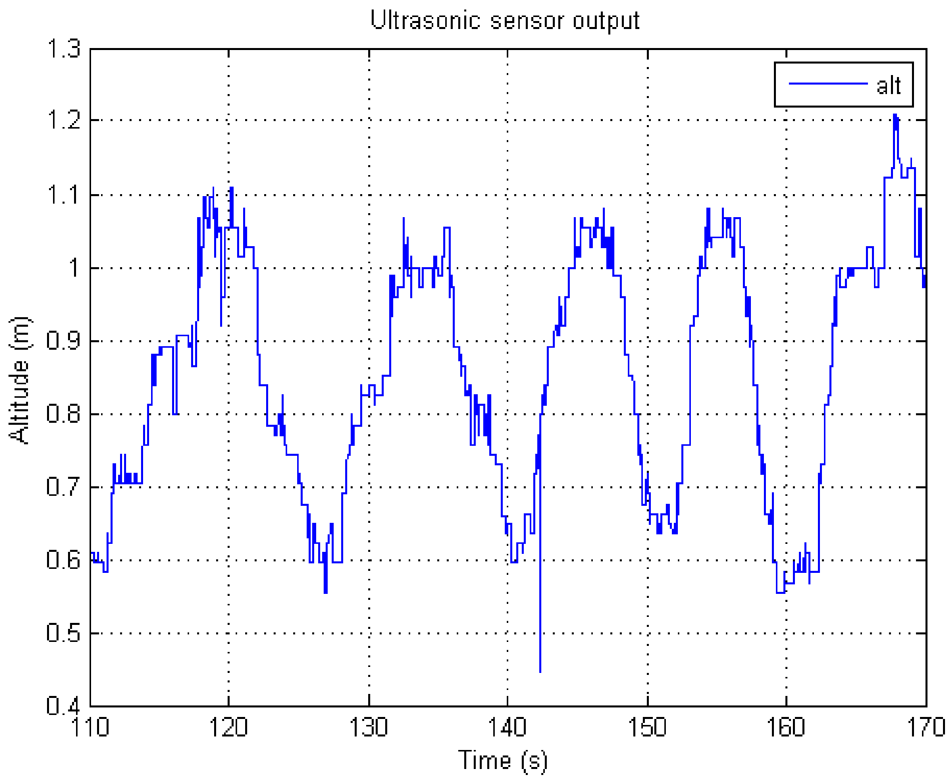


Figure 5.9 - Ultrasonic sensor output (from Arduino)

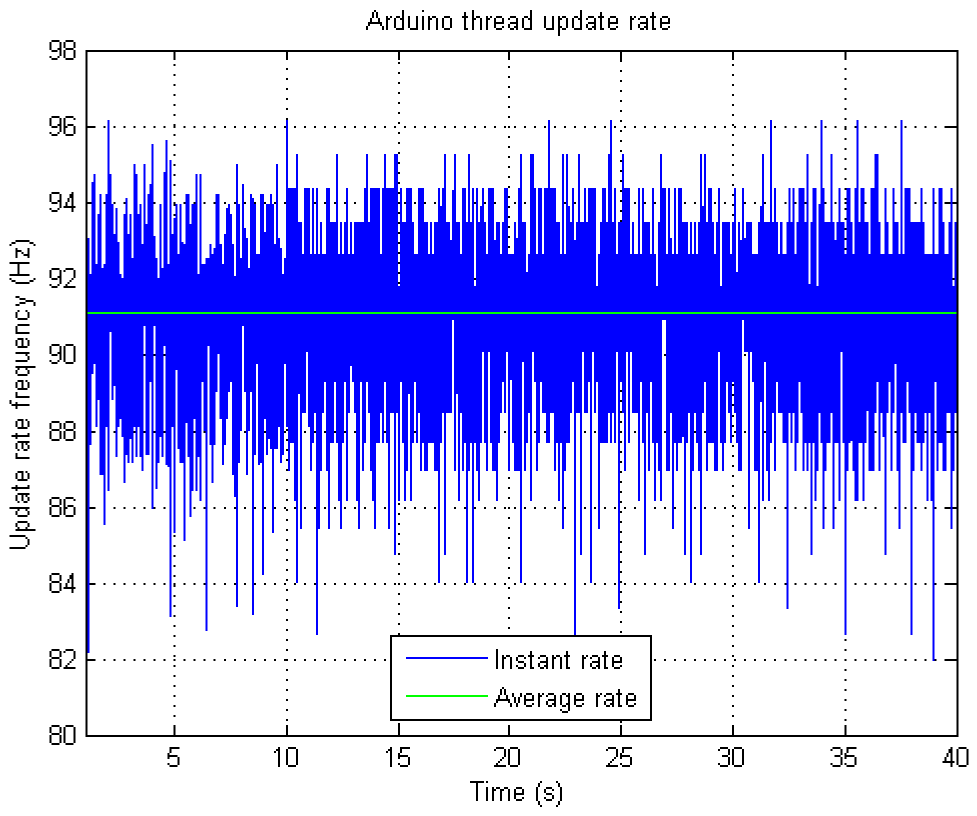


Figure 5.10 - Arduino thread update rate

5.4 AT-15 analysis

The following analysis can be made for AT-15:

- The IMU was reconfigured properly and the 16-bit data conversion is working as required. All rate and accelerometer data is being sent to the flight computer main process. The IMU data is being processed above the specified average rate of 50Hz since the state estimation thread is being updated at an average rate of 88.8Hz.
- The Arduino is collecting the magnetic compass and ultrasonic data and is transmitting the sensor data to the flight computer. This data is being collected and processed by the Arduino thread, which is being updated at an average rate of 91.1 Hz.
- The thread update plots (Figure 5.7 and Figure 5.10) show that on some time epochs the update rate of the threads slows below the average rate of 50Hz. This is due to the flight computer process being executed in an operating system environment. Over time, operating system overhead will begin to develop causing the flight computer threads to stall temporarily. Hence, the flight computer can never guarantee it will process data at required time intervals since it is running on a soft real time operating system.

Overall, all required sensor data for AT-15 was collected and processed. This included data from the IMU, magnetic compass and ultrasonic sensor. The processing of this data was handled by the state estimation thread and the Arduino thread which were updated at an average rate of 88.8 Hz and 91.1 Hz respectively. Whilst both thread update rates occasionally dropped below 50Hz, the average update rate is well above the average 50Hz update requirement. Thus AT-15 was successfully passed leading to the system requirement SR-D-05 being met.

5.5 AT-16

Successfully completing AT-16 involved the data collection from the battery voltage sensor and the radio control status from the mode control unit (MCU). The data collection for both the battery voltage sensor and the radio control status will be presented in the following sections by logged data plotted in MATALB.

5.5.1 Battery voltage sensor

The battery voltage sensor data was collected and processed by the flight computer in the Arduino thread (refer to Figure 5.11). As was previously shown in section 5.3, the Arduino is sending the flight computer magnetic compass data, battery voltage data and ultrasonic data. Thus the battery voltage is being sent by the Arduino and is being updated at an average rate of 91.1 Hz.

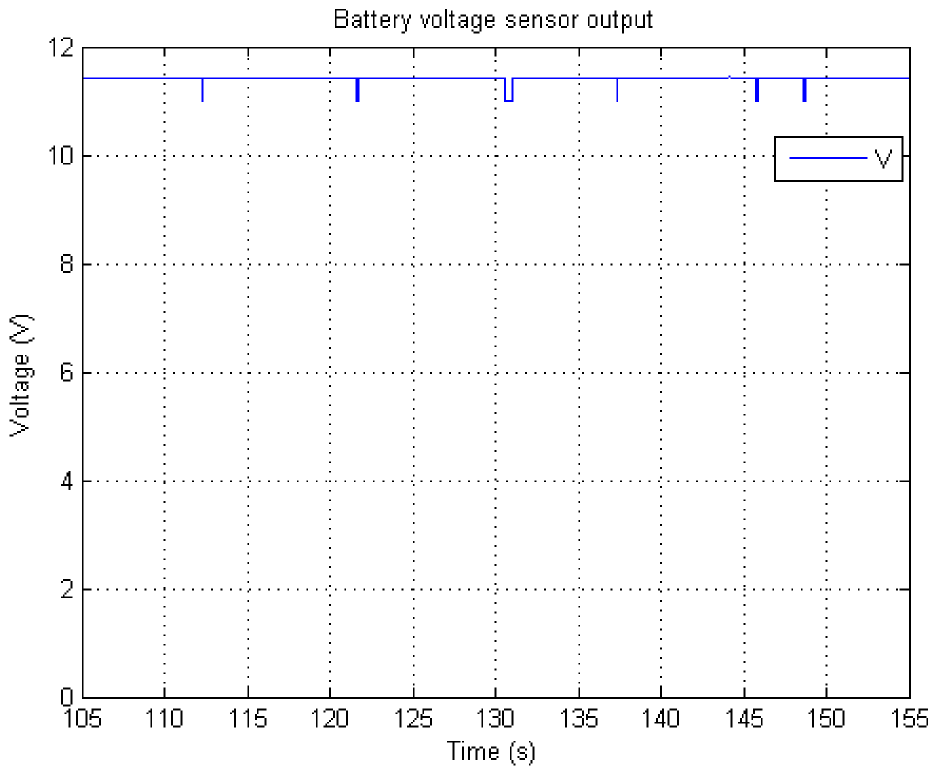


Figure 5.11 - Battery voltage sensor output (from Arduino)

5.5.2 MCU outputs

The radio control status was collected and processed by the flight computer in the MCU thread (refer to Figure 5.12). The MCU thread is being updated at an average rate of 223.4 Hz as can be seen in Figure 5.13.

5.6 AT-16 analysis

The following analysis can be made for AT-16:

- The Arduino is collecting the battery voltage data and is transmitting it to the flight computer. This data is being collected and processed by the Arduino thread, which is being updated at an average rate of 91.1 Hz.
- The MCU thread is collecting the radio control status data from the MCU. This data is being collected and updated at an average rate of 223.4 Hz.

All required sensor data for AT-16 was collected and processed. This included data from battery voltage sensor connected to the Arduino and the radio control status data from the MCU. The processing of this data was handled by the Arduino thread and the MCU thread which were updated at an average rate of 91.1 Hz and 223.4 Hz respectively. Hence, AT-16 was successfully passed leading to the system requirement SR-D-06 being met.

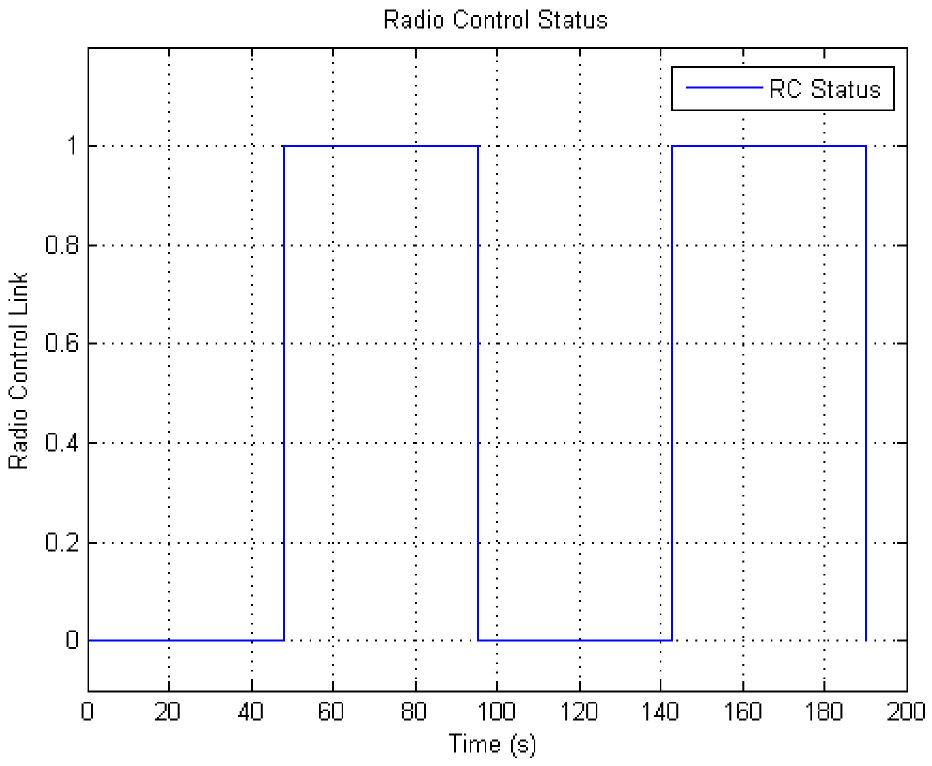


Figure 5.12 - Radio Control Status (MCU)

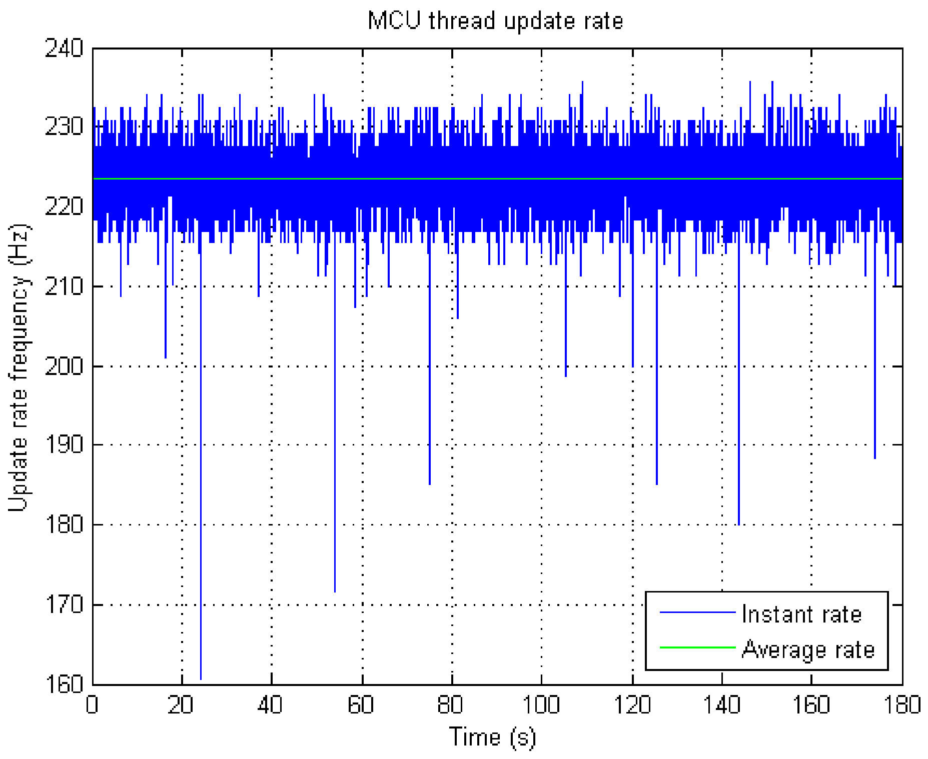


Figure 5.13 - MCU thread update rate

Chapter 6 State Estimation Testing

6.1 Acceptance tests

The state estimation design as specified in section 4 needed to be tested to achieve HLO-3 and certain system requirements. 3 acceptance tests needed to be completed to met the system requirements and can be seen in Table 6.1 below. The outcomes of these tests will be presented using the sensor and state data, which was collected by the GCS and stored in log files. These tests were conducted primarily at the ARCAA building where the Vicon motion tracking system is located.

Table 6.1 - Acceptance tests required to meet the state estimation system requirements

Acceptance Test	System Requirement	Description
AT-04	SR-B-04	Provide Euler angle and rate estimation at an average rate of 50Hz
AT-05	SR-B-05	Provide altitude estimation for the platform at an average rate of 50Hz
AT-06	SR-B-06	Provide X and Y position of the platform at an average rate of 50Hz

6.2 AT-04

Completing AT-04 required the Euler angle and Euler rates to be estimated at an average rate of 50Hz. The Euler rates are estimated from the outputs collected from the IMU gyroscopes. Testing of each Euler rate will be presented with the IMU output and the filter smoothing parameter α used. The Euler angles were calculated through the Kalman filtering design specified in section 4.3.3 and 4.3.4. The testing outcomes of each Euler angle will be presented including:

- The Kalman filter parameters used for each Euler angle (including filter smoothing parameter α used for the accelerometers).
- Stationary Euler angle estimate.
- Euler angle estimate when performing manoeuvres.
- Comparison between: Euler angle estimate, the Euler angle coarse measurement and the integration of the Euler rate.
- Verification of the Euler angle produced by the state estimator by comparing the estimate to the Euler angle measured by the Vicon system.

6.2.1 Euler rate $\dot{\phi}$

Filter smoothing parameter used: $\alpha = 0.9$. The Euler rate $\dot{\phi}$ can be seen in Figure 6.1 below.

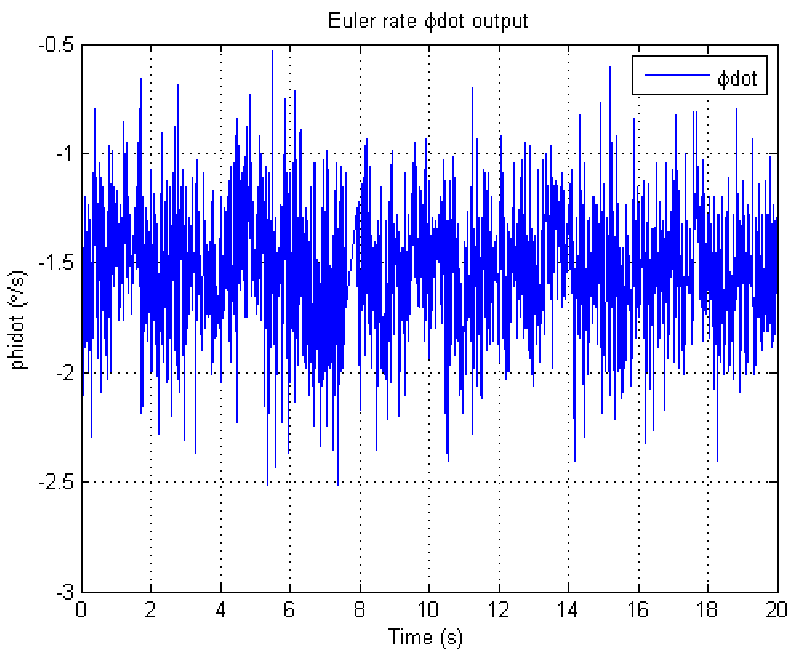


Figure 6.1 - Euler rate $\dot{\phi}$ output

6.2.2 Euler rate $\dot{\theta}$

Filter smoothing parameter used: $\alpha = 0.9$. The Euler rate $\dot{\theta}$ can be seen in Figure 6.2 below.

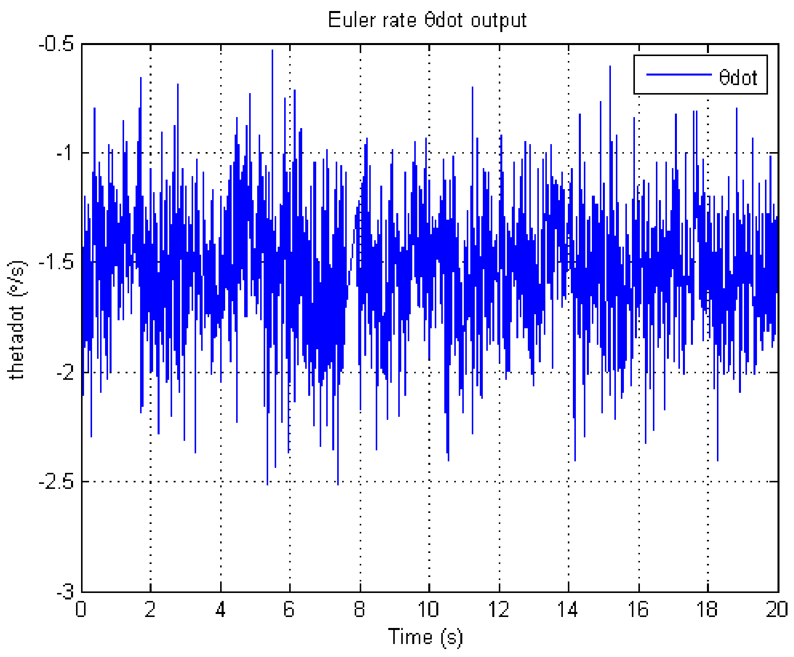


Figure 6.2 - Euler rate $\dot{\theta}$ output

6.2.3 Euler rate $\dot{\psi}$

Filter smoothing parameter used: $\alpha = 0.9$. The Euler rate $\dot{\psi}$ can be seen in Figure 6.3 below.

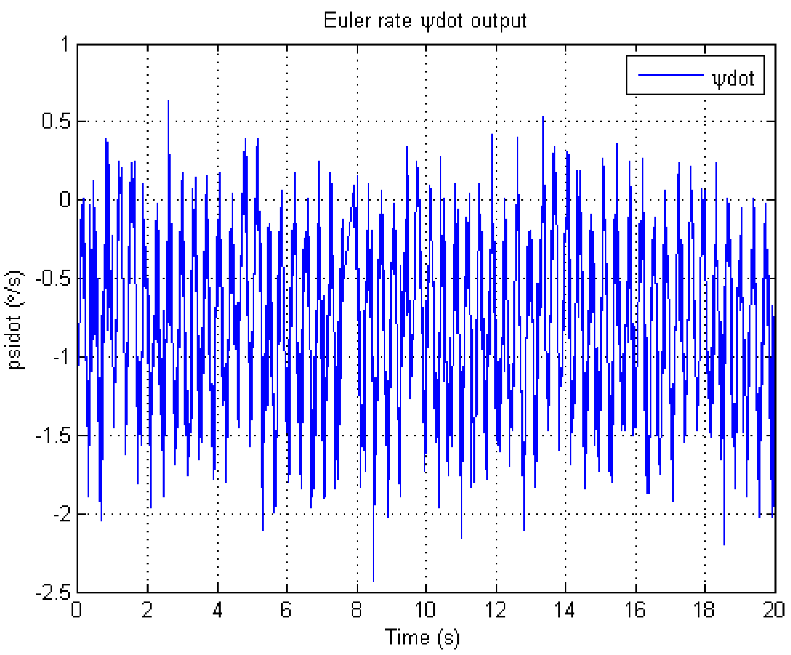


Figure 6.3 - Euler rate $\dot{\psi}$ output

6.2.4 Euler angle ϕ

Kalman filter parameters used for Euler angle ϕ included:

- Measurement covariance R: $\sigma^2_{\phi_{measure}} = 1.7°$
- Process covariance Q_k: $\sigma^2_{\phi} = 0.057296°$ and $\sigma^2_{\phi_{bias}} = 0.171887°/\mathrm{sec}$
- Initial error covariance: $P_0 = 1000 I_{4\times4}$
- Initial Kalman filter state estimate: $x_0 = \phi_{coarse}$ (initial coarse measurement from accelerometers)
- Filter smoothing parameter used: $\alpha = 0.1$

The value for $\sigma^2_{\phi_{measure}}$ was calculated by taking the average value of the ϕ_{coarse} measurement over a period of 60 seconds. The values for σ^2_{ϕ} and $\sigma^2_{\phi_{bias}}$ are extremely difficult to calculate experimentally. Thus standard values used by various quadrotor enthusiasts (e.g. Aeroquad and ArduCopter) were used for these parameters. The values used were $\sigma^2_{\phi} = 0.001\ \mathrm{rad} = 0.057296°$ and $\sigma^2_{\phi_{bias}} = 0.003\ \mathrm{rad/sec} = 0.171887°/\mathrm{sec}$. A filter smoothing parameter of $\alpha = 0.1$ provided a good balance between estimated Euler angle responsiveness and platform vibrations. Interesting plots for the Euler angle ϕ can be seen in Figure 6.4, Figure 6.5, Figure 6.6, Figure 6.7, Figure 6.8 and Figure 6.9.

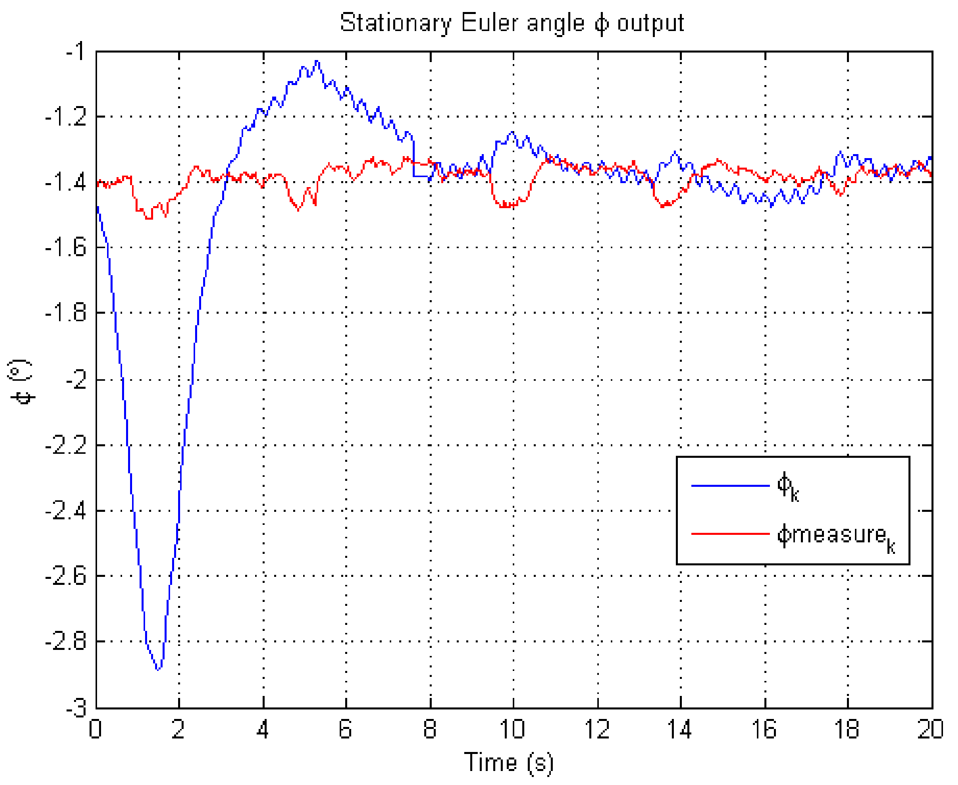


Figure 6.4 - Stationary Euler angle ϕ output

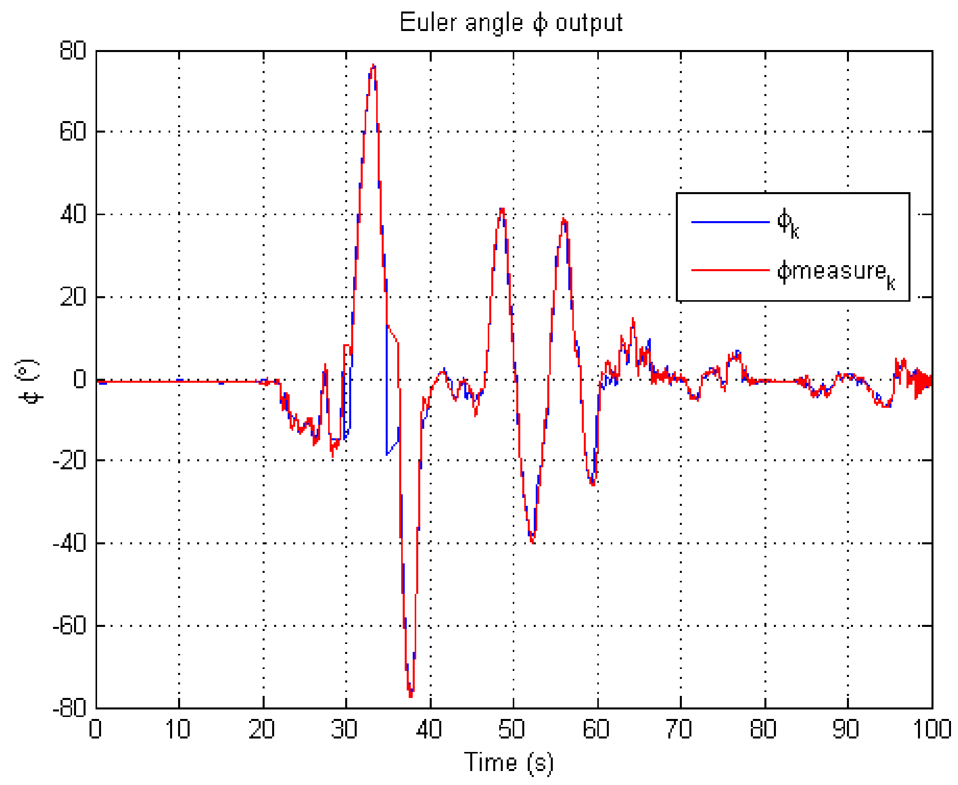


Figure 6.5 - Euler angle ϕ output

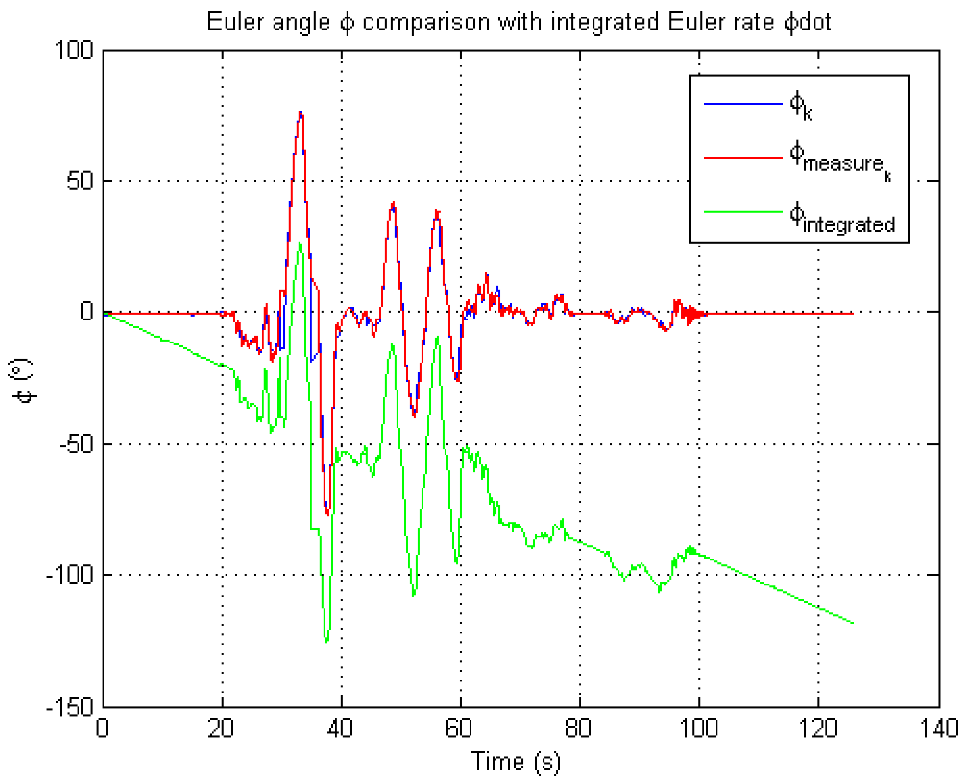


Figure 6.6 - Euler angle ϕ comparison with integrated Euler rate $\dot{\phi}$

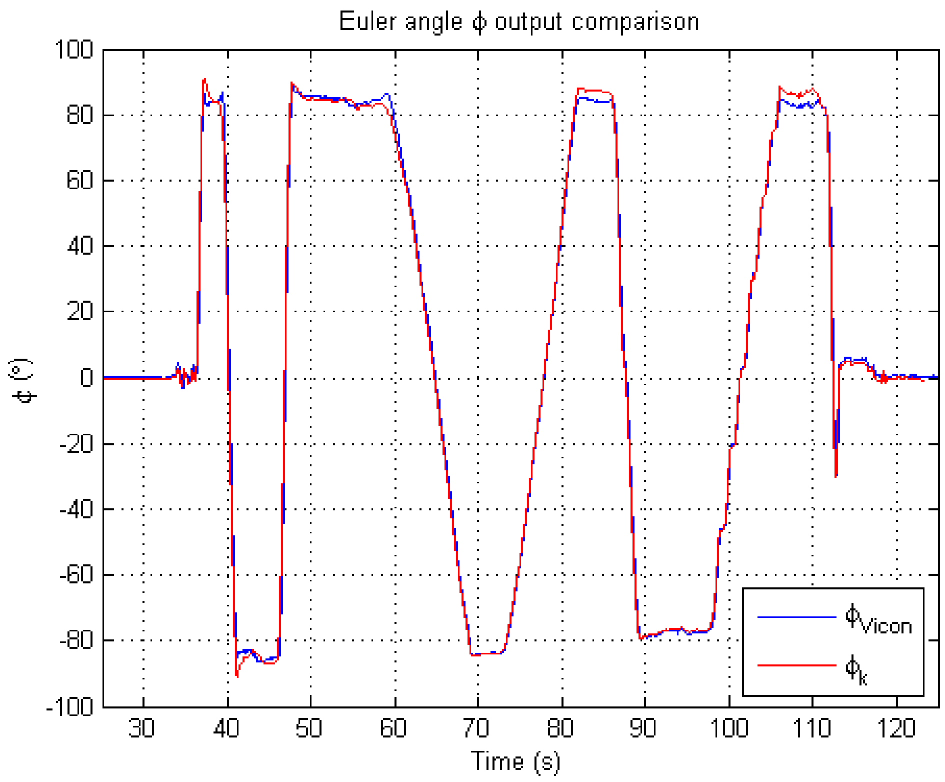


Figure 6.7 - Euler angle ϕ verification with Vicon

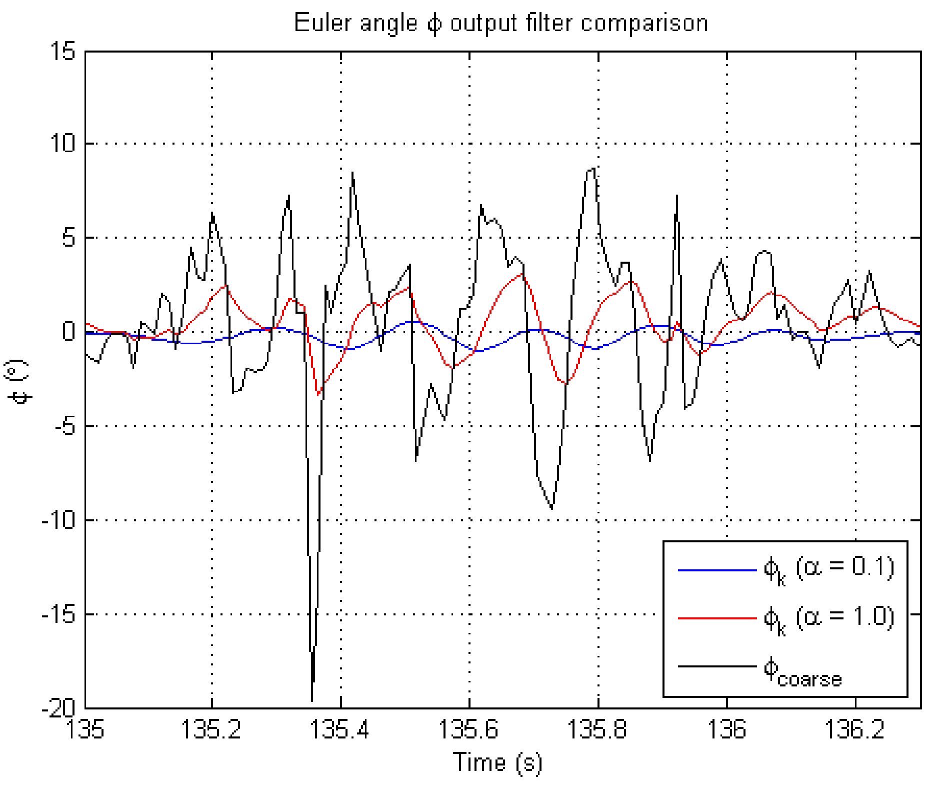


Figure 6.8 - Euler angle ϕ filter smoothing parameter comparison $\alpha = 0.1$

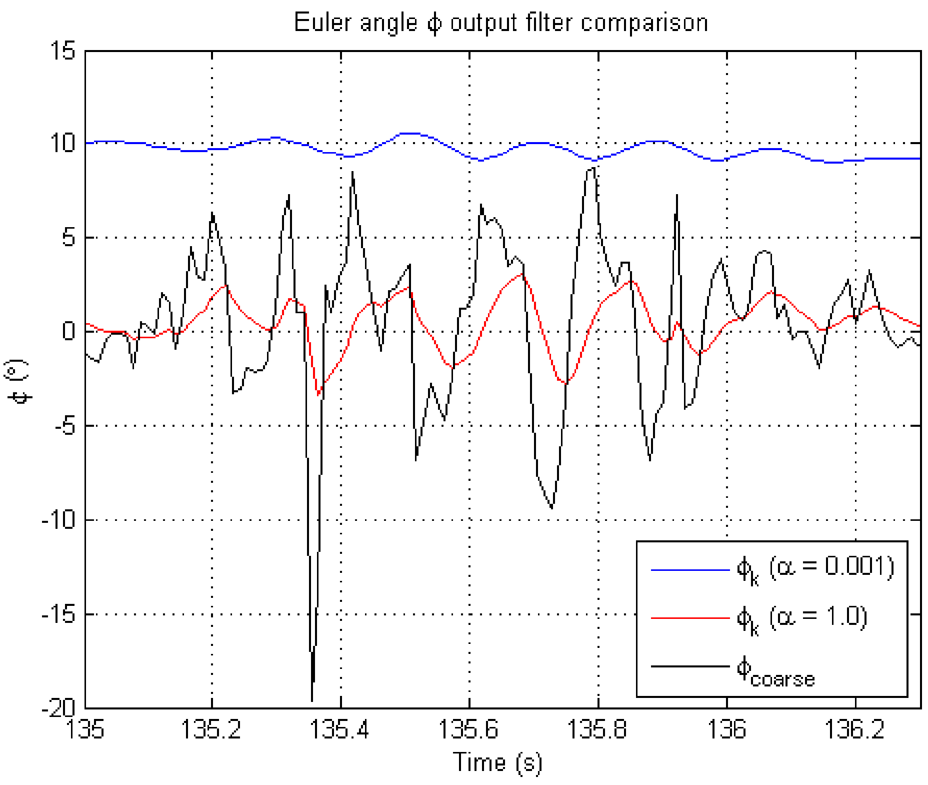


Figure 6.9 - Euler angle ϕ filter smoothing parameter comparison $\alpha = 0.001$

6.2.5 Euler angle θ

Kalman filter parameters used for Euler angle θ included:

- Measurement covariance R: $\sigma^2_{\theta_{measure}} = 1.8°$
- Process covariance Q_k: $\sigma^2_{\theta} = 0.057296\,°$ and $\sigma^2_{\theta_{bias}} = 0.171887°/\text{sec}$
- Initial error covariance: $P_0 = 1000 I_{4\times4}$
- Initial Kalman filter state estimate: $x_0 = \theta_{coarse}$ (initial coarse measurement from accelerometers)
- Filter smoothing parameter used: $\alpha = 0.1$

The value for $\sigma^2_{\theta_{measure}}$ was calculated by taking the average value of the θ_{coarse} measurement over a period of 60 seconds. The values for σ^2_{θ} and $\sigma^2_{\theta_{bias}}$ are extremely difficult to calculate experimentally. Thus standard values used by various quadrotor enthusiasts (e.g. Aeroquad and ArduCopter) were used for these parameters. The values used were $\sigma^2_{\theta} = 0.001\text{ rad} = 0.057296°$ and $\sigma^2_{\theta_{bias}} = 0.003\text{ rad/sec} = 0.171887°/\text{sec}$. A filter smoothing parameter of $\alpha = 0.1$ provided a good balance between Euler angle responsiveness and platform vibrations. Interesting plots for the Euler angle θ can be seen in Figure 6.10, Figure 6.11, Figure 6.12 and Figure 6.13.

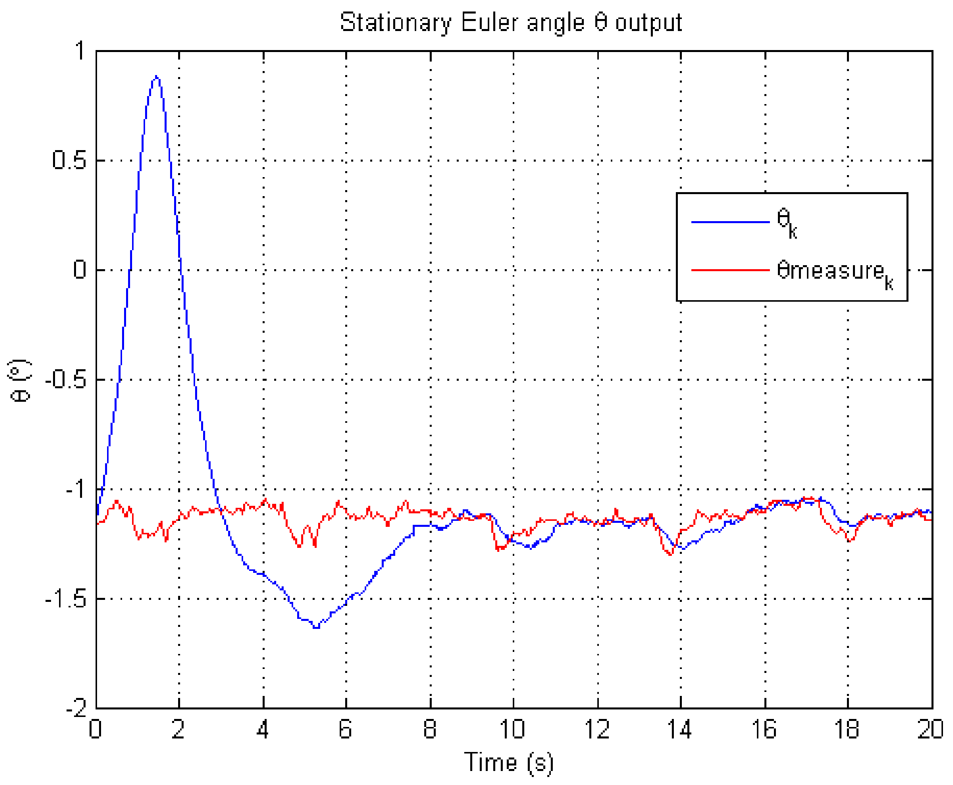


Figure 6.10 - Stationary Euler angle θ output

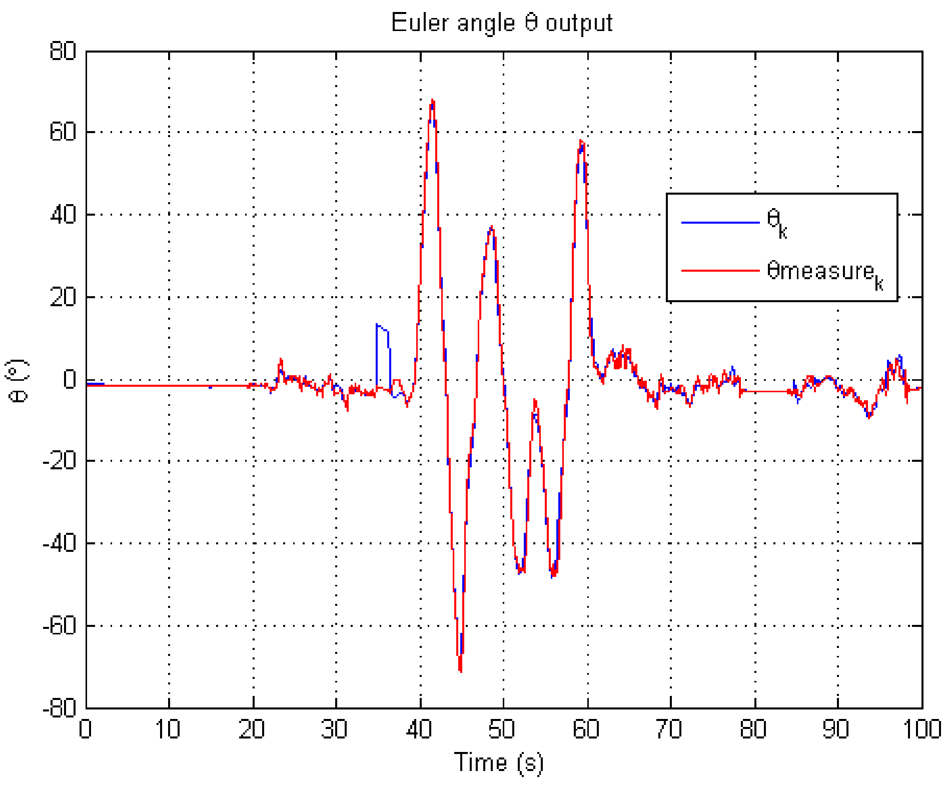


Figure 6.11 - Euler angle θ output

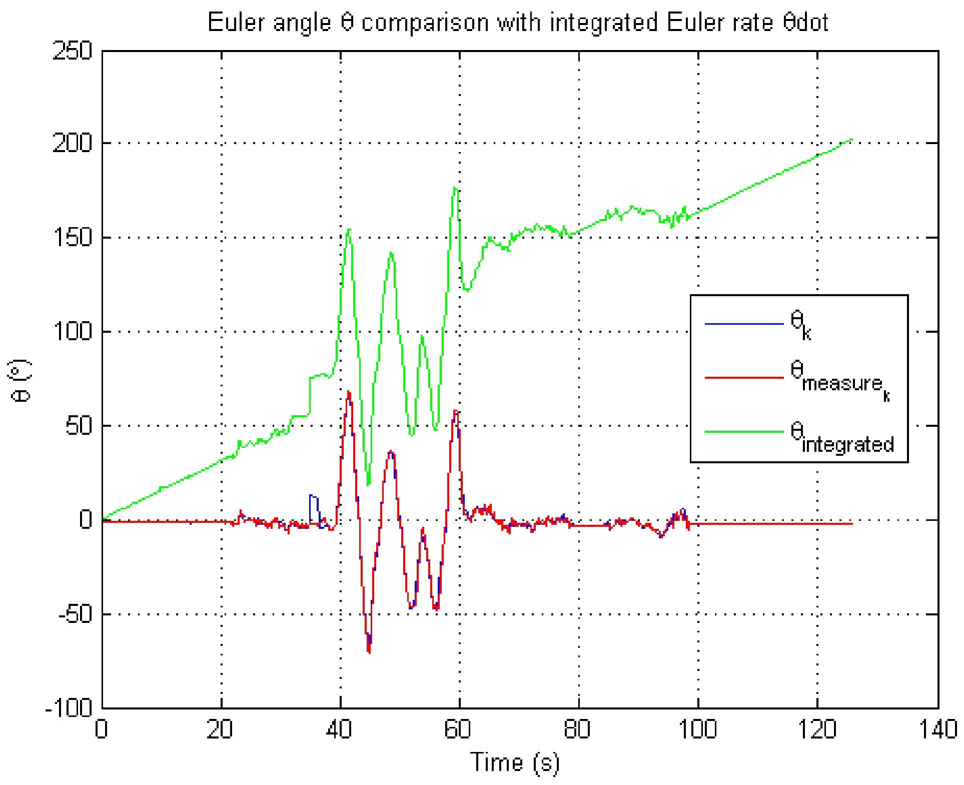


Figure 6.12 - Euler angle θ comparison with integrated Euler rate $\dot{\theta}$

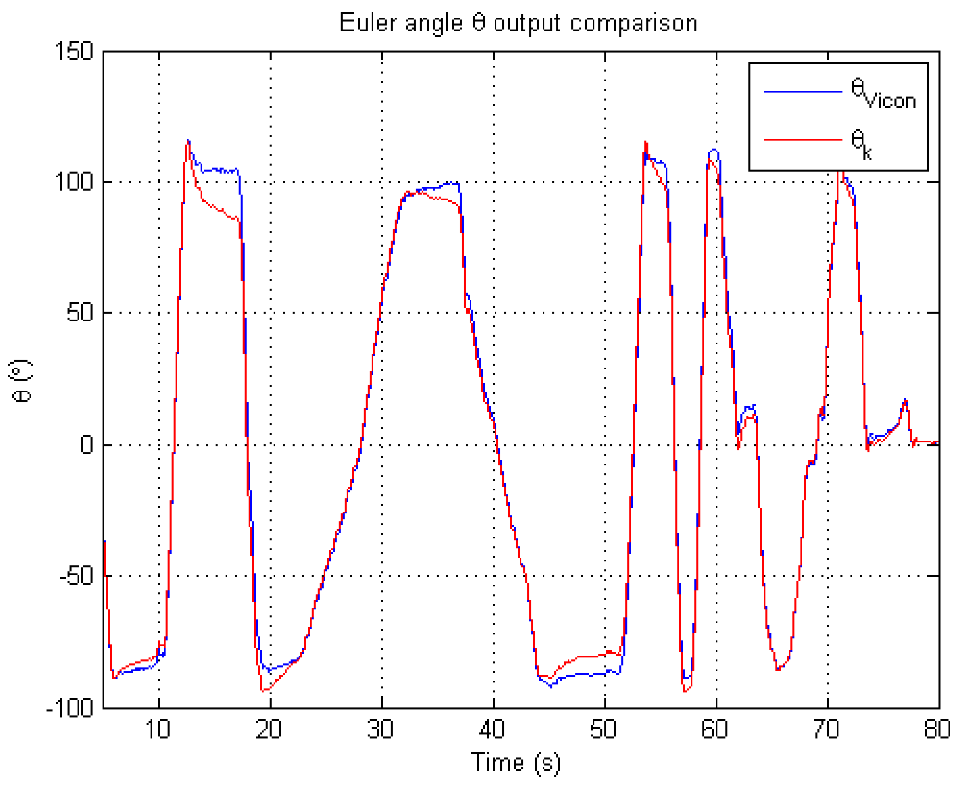


Figure 6.13 - Euler angle θ verification with Vicon

6.2.6 Euler angle ψ

Kalman filter parameters used for Euler angle ψ included:

- Measurement covariance R: $\sigma^2_{\psi_{measure}} = 5.4°$
- Process covariance Q_k: $\sigma^2_{\psi} = 0.057296$ ° and $\sigma^2_{\psi_{bias}} = 0.171887°/\mathrm{sec}$
- Initial error covariance: $P_0 = 1000 I_{4\times4}$
- Initial Kalman filter state estimate: $x_0 = \psi_{heading}$ (initial heading measurement from compass)
- Filter smoothing parameter used: $\alpha = 0.5$

The value for $\sigma^2_{\psi_{measure}}$ was calculated by taking the average value of the $\psi_{heading}$ measurement over a period of 60 seconds. The values for σ^2_{ψ} and $\sigma^2_{\psi_{bias}}$ are extremely difficult to calculate experimentally. Thus standard values used by various quadrotor enthusiasts (e.g. Aeroquad and ArduCopter) were used for these parameters. The values used were $\sigma^2_{\psi} = 0.001\ \mathrm{rad} = 0.057296°$ and $\sigma^2_{\psi_{bias}} = 0.003\ \mathrm{rad/sec} = 0.171887°/\mathrm{sec}$. A filter smoothing parameter of $\alpha = 0.5$ provided a good balance between Euler angle responsiveness and platform vibrations. Interesting plots for the Euler angle ψ can be seen in Figure 6.14, Figure 6.15, Figure 6.16, Figure 6.17 and Figure 6.18.

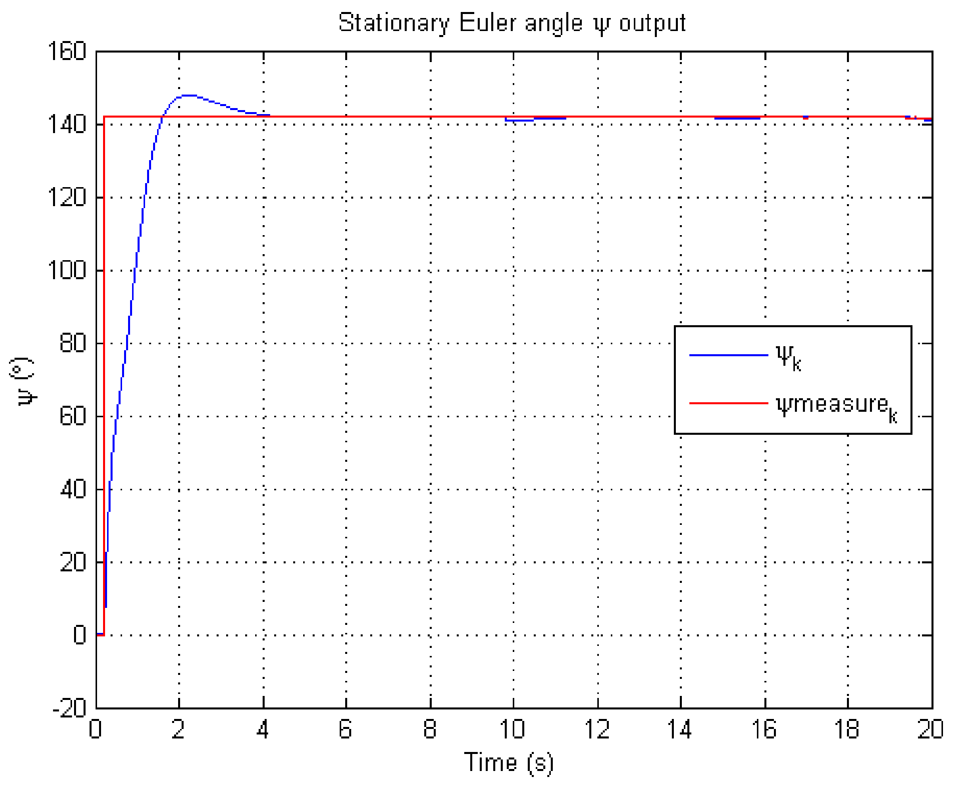


Figure 6.14 - Stationary Euler angle ψ output

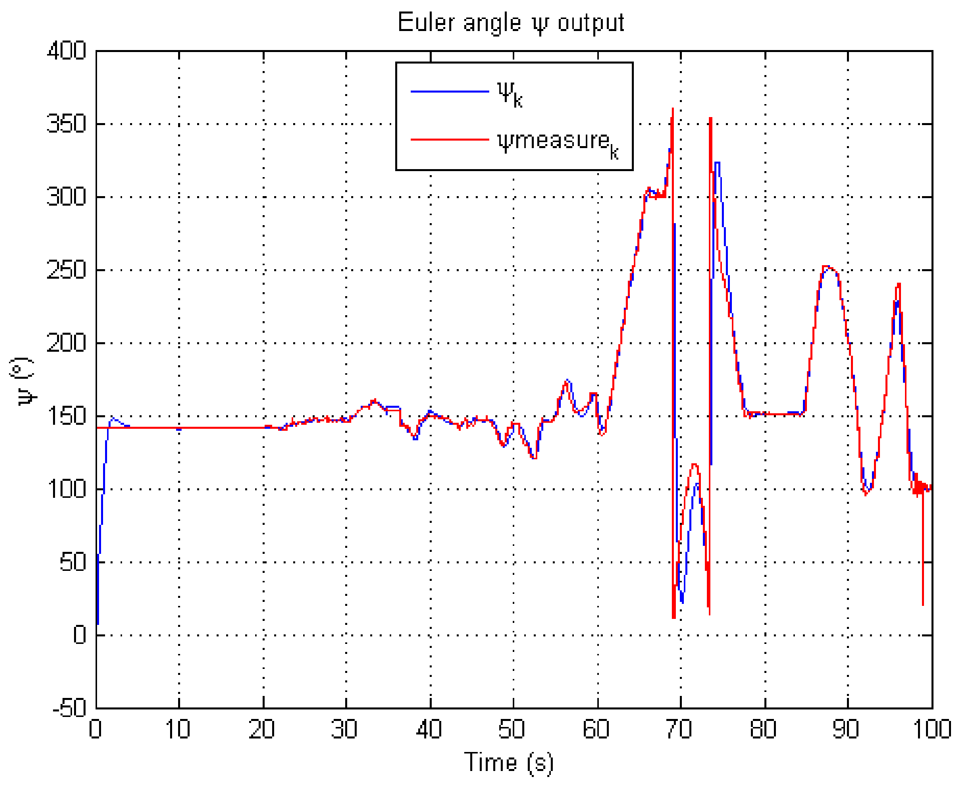


Figure 6.15 - Euler angle ψ output

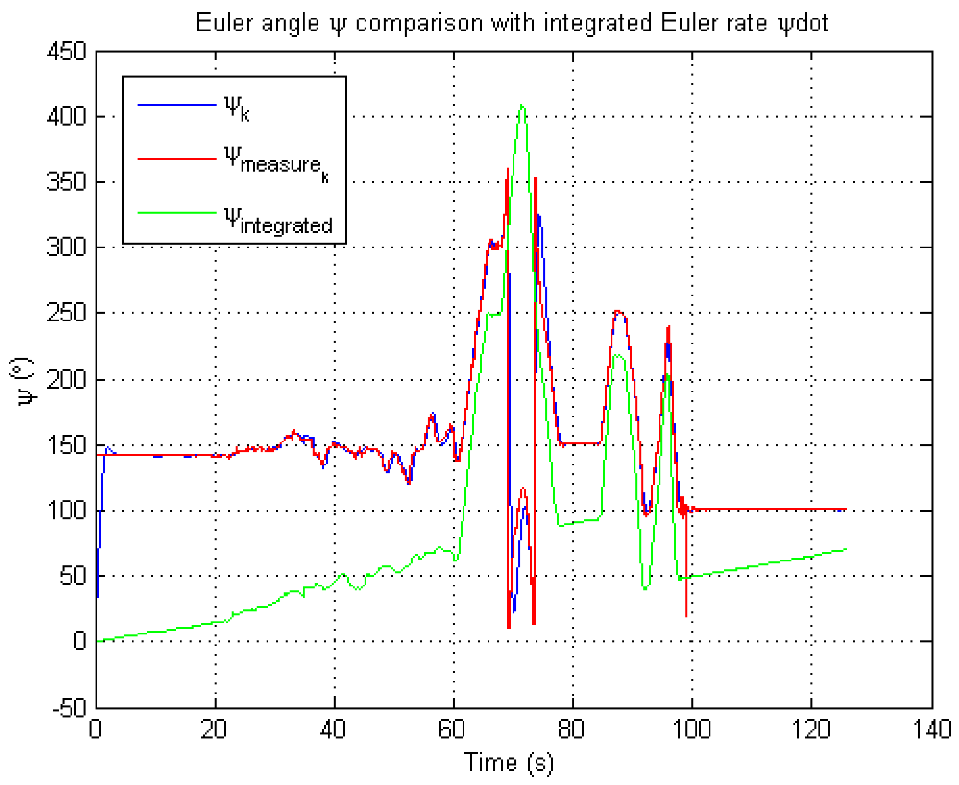


Figure 6.16 - Euler angle ψ comparison with integrated Euler rate $\dot{\psi}$

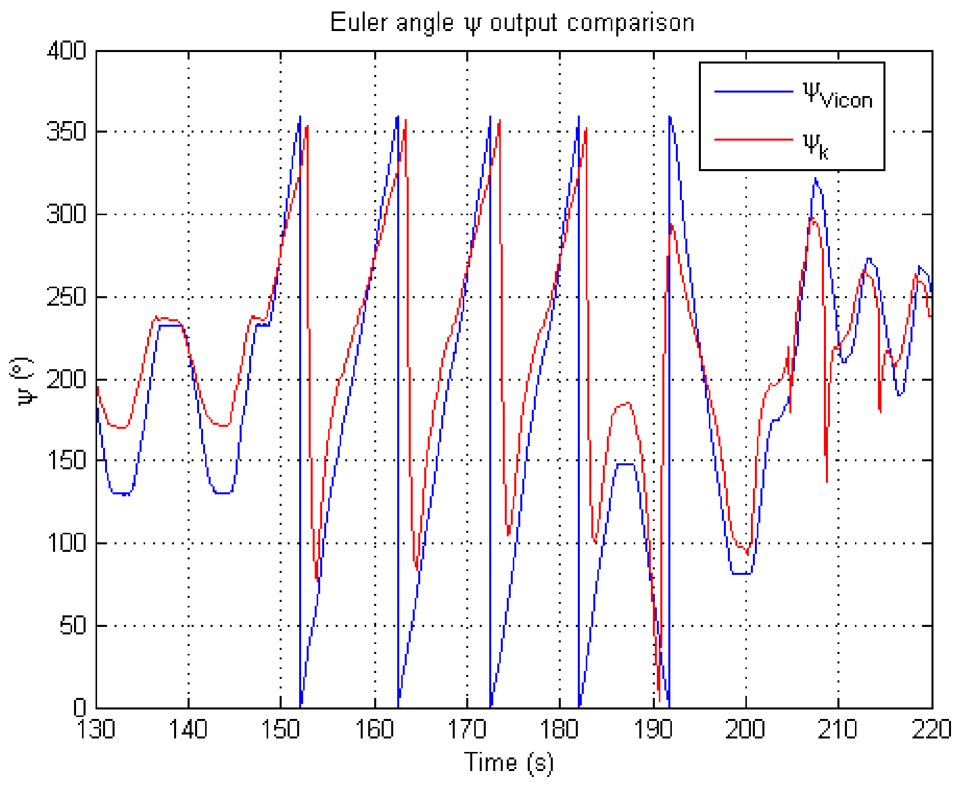


Figure 6.17 - Euler angle ψ output comparison (no discontinuity checking)

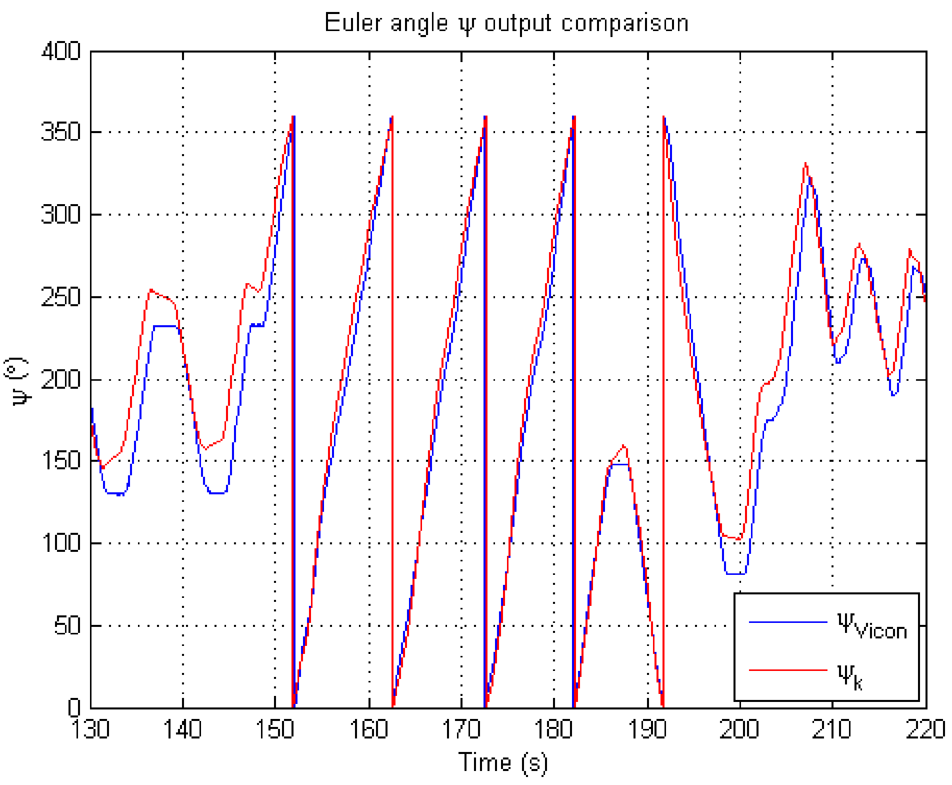


Figure 6.18 - Euler angle ψ output comparison (correct state estimation)

6.3 State estimation update rate

The state estimation process was handled by the state estimation thread. The average update rate of the state estimation thread was 88.8 Hz. The update rate of the state estimation thread can be seen in Figure 6.19.

6.4 AT-04 analysis

The analysis for acceptance test AT-04 will be presented for the Euler rates followed by each Euler angle.

6.4.1 Euler rates analysis

Figure 6.1, Figure 6.2 and Figure 6.3 showed that the Euler rates are being estimated by their individual state estimators. Very little filtering was required for these rates as the filtering smoothing parameters for each rate was only $\alpha = 0.9$. The state estimators for each of these rates are contained in the state estimation thread thus are being updated when the state estimation thread is updated. The average update rate of this thread is 88.8 Hz thus the Euler rates are being updated will above the required average rate of 50 Hz.

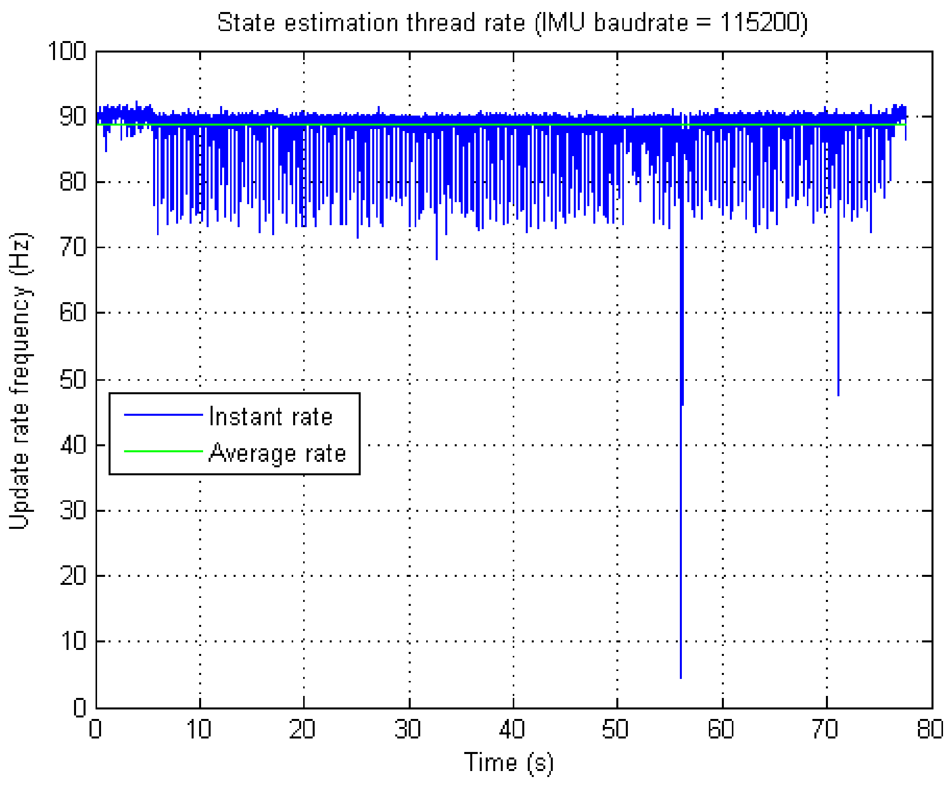


Figure 6.19 - State estimation thread rate

6.4.2 Euler angle ϕ analysis

The stationary Euler angle ϕ in Figure 6.4 showed that the Kalman filter estimate begins to track the Euler angle measurement closely after 8 seconds. Prior to that, the estimator had developed a large uncertainty in the reading thus it took a few seconds for the estimator and measurement to meet. After that both the estimate and the measurement for ϕ converged to a value of $\phi = -1.4°$. Thus the IMU had a mounting error on the platform in the ϕ axis of $-1.4°$ which was corrected by modifying the ϕ state estimator.

Figure 6.5 and Figure 6.6 showed that the Kalman filter has provided good estimates for the Euler angle ϕ. The gyroscopic bias that developed in the X rate gyro from the IMU is being eliminated. This can be seen in Figure 6.6 since the integrated Euler rate for ϕ drifts excessively as time progresses yet the state estimate for ϕ does not. The estimate also has eliminated the high frequency noise that exists in the coarse measurement for ϕ.

The Vicon system also has verified the accuracy of the state estimator for ϕ as seen in Figure 6.7. Only small differences exist between the state estimator and the Vicon system most notably as the platform approaches $\pm 90°$. However since the quadrotor platform will never reach this attitude in

ϕ, the state estimation for ϕ is producing high quality updates with the specified Kalman filter statistical parameters.

The filter smoothing parameter for the coarse measurement value for ϕ was also checked in Figure 6.8 and Figure 6.9. When a smoothing parameter of $\alpha = 0.1$ was used, the estimator produced the correct state estimate of near $0°$. However when the smoothing parameter is changed to $\alpha = 0.001$, the estimator failed to track the Euler angle ϕ. Hence a value of $\alpha = 0.1$ was permanently used to filter the coarse measurement value for ϕ. This value was generally used as the default value for all accelerometer filtering. This was because it provided a good balance between filtering high frequency noise (which resonated with the platform) and maintaining good coarse estimate responsiveness when the quadrotor platform manoeuvred in flight.

6.4.3 Euler angle θ analysis

A similar analysis for the Euler angle θ can be made as was conducted for ϕ. The stationary Euler angle θ in Figure 6.10 showed that the Kalman filter estimate for θ begins to track the coarse measurement after 10 seconds. After this time both the estimate and the measurement for θ converged to $\theta = -1.2°$. Thus the IMU also had a mounting error in the θ axis of $-1.2°$ which was corrected by modifying the θ state estimator.

Figure 6.11 and Figure 6.12 showed that the Kalman filter has provided good estimates for the Euler angle θ. Once again, the gyroscopic bias that developed in the Y rate gyro from the IMU is being eliminated. This can be seen in Figure 6.12 since the integrated Euler rate for θ drifts as time progresses yet the state estimate for θ does not. The estimate also has eliminated the high frequency noise that exists in the coarse measurement for θ.

The Vicon system also has verified the accuracy of the state estimator for θ as seen in Figure 6.13. Some differences exist between the state estimator and the Vicon system, most importantly as the platform approaches $\pm 90°$. However since the quadrotor platform will never reach this attitude in θ, the state estimation for θ is producing high quality updates with the specified Kalman filter statistical parameters.

6.4.4 Euler angle ψ analysis

Once again, a similar analysis for the Euler angle ψ can be made as was conducted for ϕ and θ. The stationary Euler angle ψ in Figure 6.14 showed that the Kalman filter estimate for ψ begins to track the Euler angle measurement from the compass after 4 seconds. Figure 6.15 and Figure 6.16 showed that the Kalman filter has provided good estimates for the Euler angle ψ. The gyroscopic bias that developed in the Z rate gyro is being eliminated again. This is seen in Figure 6.16 since the integrated Euler rate for ψ drifts as time progresses but the state estimate for ψ does not. The estimate also has eliminated the variations in the compass headings measurement since the estimate does not contain these variations.

The Vicon system has verified that the discontinuity check for Euler angle ψ has been successful. Figure 6.17 shows the output of the state estimator for ψ without the check between 0 ° and 360 °. As the compass measurement crossed the 0 ° to 360 ° boundary, the estimate for ψ rotated in the wrong direction. This resulted in no heading being predicted under an angle of 75 ° between the time periods of 150 to 180 seconds. However when the discontinuity check is being performed, the state estimate for ψ matches closely with that measured by the Vicon system (refer to Figure 6.18). Overall, the state estimator for ψ does a reasonable job in predicting the Euler angle ψ when compared with the Vicon system. Future tuning of the Kalman filter parameters and a different mounting location of the magnetic compass should result in more accurate state estimates for ψ.

6.4.5 AT-04 summary

All Euler angles have been successfully measured by the onboard attitude sensors. The state estimates for each axis have been verified by the Vicon system and are updating at an average rate of 88.8 Hz. Thus this acceptance test was successfully passed and the system requirement SR-B-04 was met.

6.5 AT-05

This acceptance test required the use of the Vicon system with the quadrotor platform. The Vicon tracker system supplies the altitude of the quadrotor platform in a Z translation from the origin. The origin of the Vicon system was set to be the middle of the ARCAA hanger where the flight tests were conducted. The Vicon Z translation output can be seen in Figure 6.20. The average update rate of the Vicon system for this test was 150.9 Hz and the update rate can be seen in Figure 6.21. The altitude estimation was also provided by the ultrasonic sensor as was described in section 4.3.9. The output of this ultrasonic data can be seen in Figure 6.22.

6.6 AT-05 analysis

The Z position of the quadrotor platform was successfully measured by both the Vicon tracker system and the altitude sensor. The Z position of the platform from the Vicon tracker is being updated at an average rate of 150.9 Hz. However the Z position of the platform from the ultrasonic sensor can only be updated at a maximum of 20Hz as is described in [RD/5]. Thus this acceptance test was successfully passed only once the Vicon system was available.

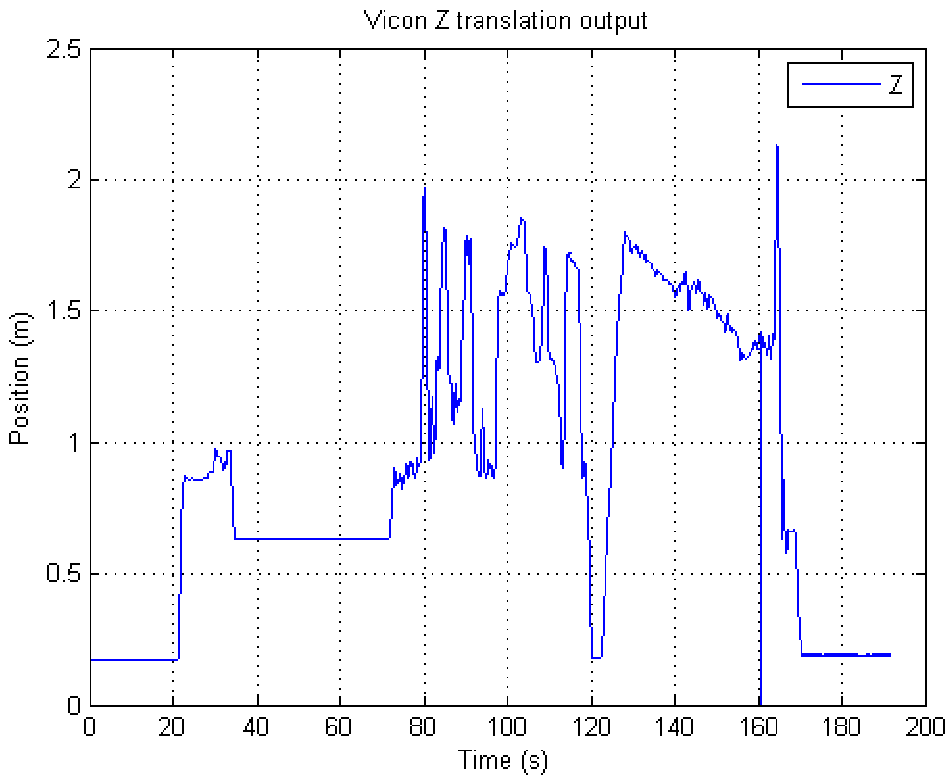


Figure 6.20 - Z position of the quadrotor platform (Vicon)

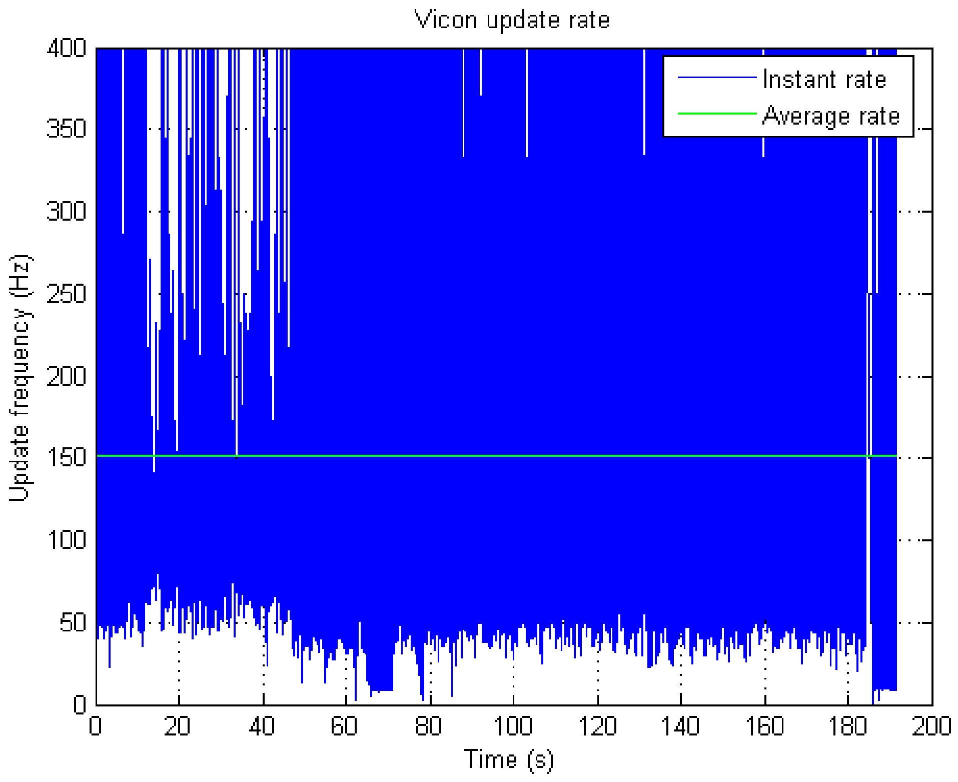


Figure 6.21 - Vicon update rate

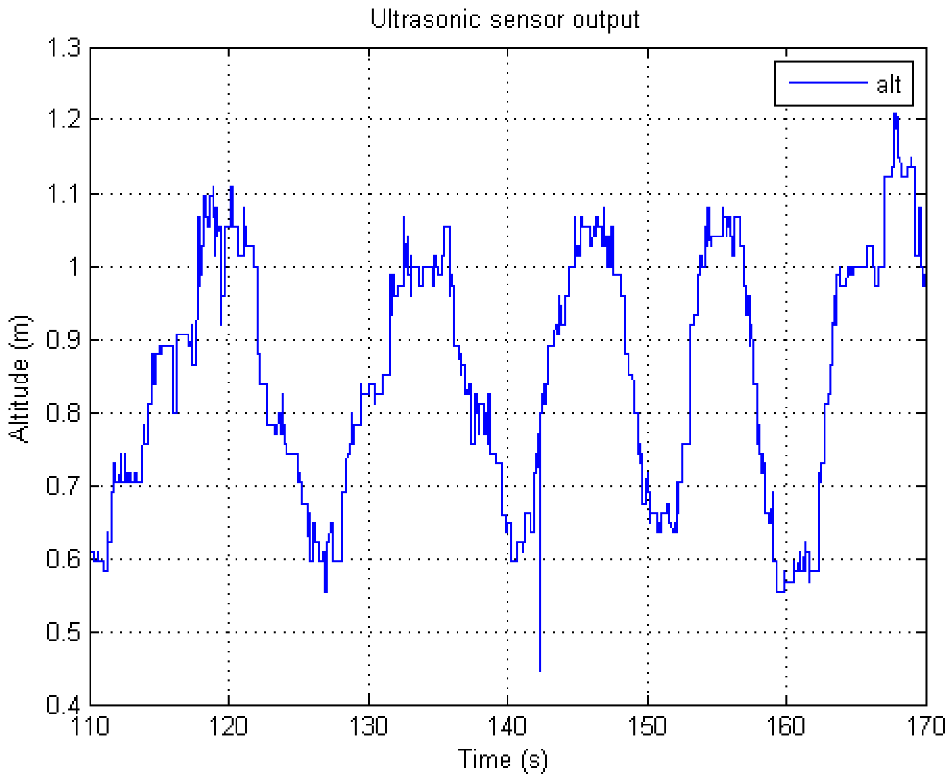


Figure 6.22 - Z position of the quadrotor platform (ultrasonic)

6.7 AT-06

Once again, this acceptance test required the use of the Vicon system with the quadrotor platform. The Vicon tracker program supplies the position of the quadrotor platform in X and Y translations from the origin. The origin of the Vicon system was set to be the middle of the ARCAA hanger where the flight tests were conducted. The Vicon X translation output can be seen in Figure 6.23 and the Vicon Y translation output can be seen in Figure 6.24. The average update rate of the Vicon system was 150.9 Hz. For completeness, the 3D trajectory of the quadrotor platform using X, Y and Z position information has been included in Figure 6.25.

6.8 AT-06 analysis

The X and Y position of the quadrotor platform was successfully received by the Vicon tracker system when the GCS was connected to it. These updates are being received at 150.9 Hz which is above the required average update rate of 50 Hz. Thus this acceptance test was successfully passed and system requirement SR-B-06 was met.

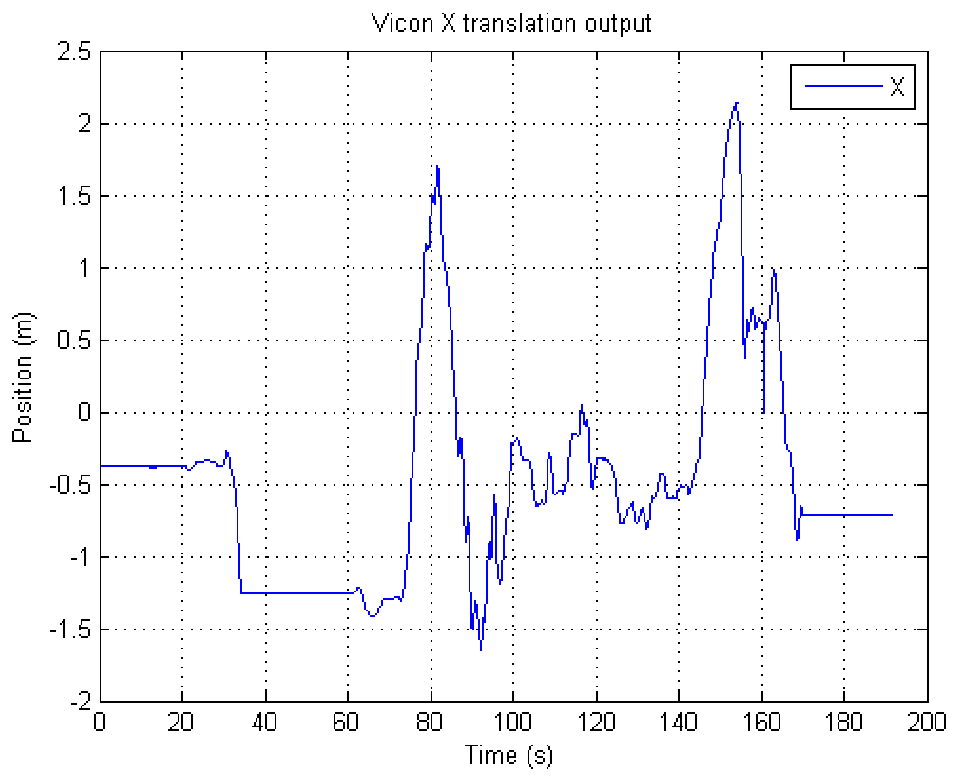


Figure 6.23 - X position of the quadrotor platform (Vicon)

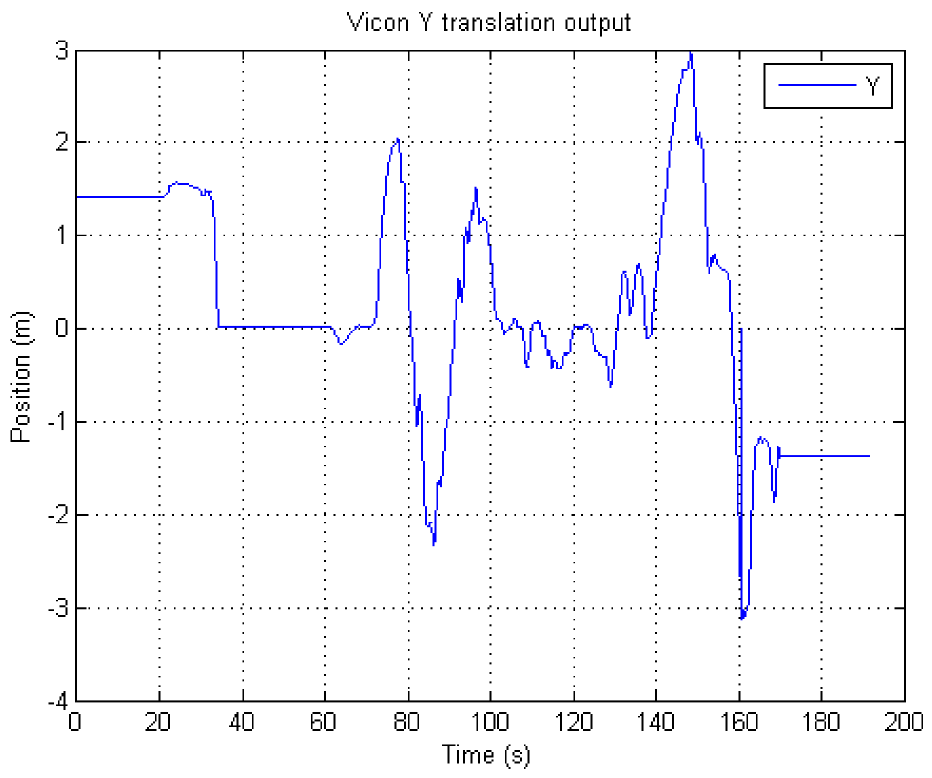


Figure 6.24 - Y position of the quadrotor platform (Vicon)

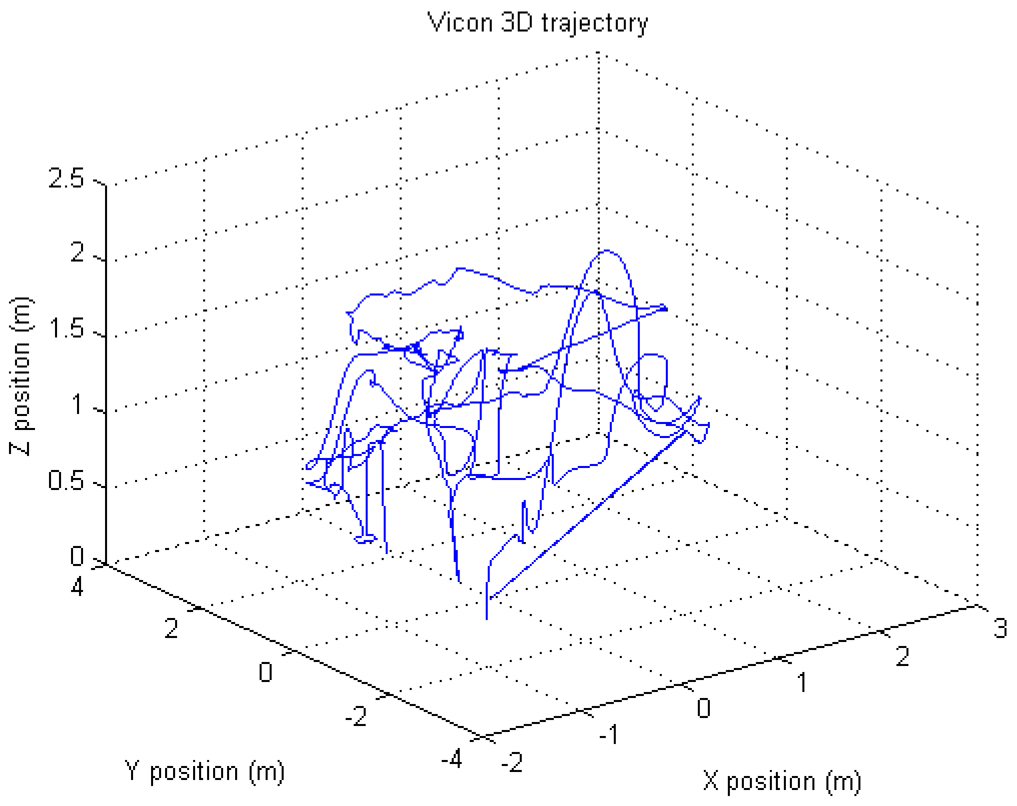


Figure 6.25 – 3D trajectory of the quadrotor platform (Vicon)uadrotor platform (Vicon)

Chapter 7 Conformance Matrix

7.1 Flight computer conformance

The conformance matrix for the flight computer can be seen below. As Table 7.1 shows all system requirements related to the flight computer subsystem were met and are listed as complete.

Table 7.1 - Flight computer conformance matrix

Requirement	Description	Status	Documentation
SR-D-05	Collect the following sensor data at an average rate of 50 Hz: • IMU • Arduino • Compass (connected via the Arduino) • Ultrasonic sensor (connected via the Arduino)	Complete	AHNS-2010-AP-TR-002
SR-D-06	Collect the following sensor data at an average rate of 50 Hz: • Battery voltage sensor (connected via the Arduino) • Mode control unit (radio control status)	Complete	AHNS-2010-AP-TR-002

7.2 State estimation conformance

The conformance matrix for state estimation can be seen below. All system requirements related to the state estimation subsystem were met and are listed as complete.

Table 7.2 - State estimation conformance matrix

Requirement	Description	Status	Documentation
SR-B-04	Provide Euler angle and rate estimation at an average rate of 50Hz	Complete	AHNS-2010-SE-TR-001
SR-B-05	Provide altitude estimation for the platform at an average rate of 50Hz	Complete	AHNS-2010-SE-TR-002
SR-B-06	Provide X and Y position of the platform at an average rate of 50Hz	Complete	AHNS-2010-SE-TR-002

Chapter 8 Conclusions

All system requirements for the flight computer subsystem and state estimation subsystem have been successfully completed. Thus a quadrotor platform that can accurately measure its attitude onboard and provide numerous state updates at an average rate of 50 Hz has been achieved. This allowed the quadrotor platform to be controlled with a respectable level of autonomy that future AHNS years can harness and improve upon. The remainder of this chapter will be dedicated to recommendations for future AHNS project revisions.

8.1 Recommendations

8.1.1 Application of systems engineering

Future AHNS members must be aware that a systems engineering methodology should apply to the project at all times. Since the AHNS project requires a large quantity of theoretical and technical research it becomes very tempting to ignore simple systems engineering principles. Future AHNS members are encouraged to review the HLOs and system requirements that were developed in the first weeks of the project as much as possible. If the direction or progress of the project changes then the customer of the project should be contacted and the HLOs and system requirements renegotiated. All prior systems engineering documentation should be revised if the HLOs or system requirements of the project change. Constant revision of system engineering documentation and communication with the customer are essential ingredients in delivering the required product to the customer.

8.1.2 Operating system overhead

Using the Overo Fire for the flight computer created considerable system update lag. This was because the Overo Fire runs a Linux based operating system and is thus utilising a soft real time system. The flight computer therefore could never guarantee that it would execute system updates within a specified time period. Precise system updates could be achieved by porting the flight computer to a dedicated processor. The dedicated processor should only be running the flight computer process, meaning that the entire system will be updated as fast as the dedicated processor allows.

Porting the flight computer code onto a non based operating system solution has ramifications for the communications subsystem. The communications protocol is built off a UDP client/server architecture, which can only be available when using an operating system. Thus the communications server located on the flight computer should stay on the Overo Fire but all other flight computer processes should be relocated to a dedicated processor. The Overo Fire will thus be transformed into a telemetry link so that clients can receive flight computer data. The connection between the Overo Fire and the dedicated processor should be implemented via the USB link on the Overo Fire pinto board. An added benefit of this new hardware architecture is that using a USB connection will remove the need for logic level shifting since the Overo Fire sends and receives serial data at 1.8V levels.

8.1.3 Accelerometer filtering

Accelerometer data sent by the IMU is completely raw ADC data transmitted via a serial UART. Under normal circumstances this accelerometer data is generally useful and can provide good coarse measurements for ϕ and θ. However during dynamic platform movement, vibrations began to develop which affected the accelerometer data readings. To reduce the effect of the vibrations, low pass filtering (or EWMA filtering) needed to be introduced. Care had to be taken when deciding on the level of low pass filtering by adjusting the smoothing parameter α. Too much smoothing caused the ϕ and θ estimates to become unresponsive even under static platform attitude manoeuvres. The use of low pass filters was not just limited to accelerometers, all sensors which used ADC data output had to be filtered using this method as well.

8.1.4 High level control using vision

Developing high level control using vision based sensors proved to be difficult to design. This was primarily because the quadrotor platform was built in house and no commercial off the shelf solutions were sought (as per the project specifications). Due to this and the strict project timeline, a stable quadrotor platform was only achieved towards the end of the project. The ability to design visual high level control (within the project timeline) would require a stable quadrotor platform to be available at the beginning of the project inception. Only then could visual based control be designed and implemented effectively leading to successful completion of HLO-2.

References

RD/1	AHNS-2010-SY-HL-001	AHNS, High Level Objectives of
RD/2	AHNS-2010-SY-SR-001	AHNS, System Requirements of
RD/3	Sensor Dynamics 6DOF IMU ACE board	Sensor Dynamics. 2009. Advanced Customer Evaluation Board. Available: http://ahns10.googlecode.com/svn-history/r551/docproject/unofficial_ahns/ACEBoard.pdf (accessed October 17 2010).
RD/4	Ocean Server OS4000	Ocean Server. 2008. Ocean Server Digital Compass Users Guide. Available: http://www.ocean-server.com/download/OS4000-Compass_s_Manual.pdf (accessed October 17 2010).
RD/5	LV-MaxSonar-EZ0	Max Botix Inc. 2010. LV-MaxSonar-EZ0 High Performance Sonar Range Finder. Available: http://www.maxbotix.com/uploads/LV-MaxSonar-EZ0-Datasheet.pdf (accessed October 17 2010).
RD/6	Arduino Nano	Ardunio. 2010. Arduino Nano. Available : http://arduino.cc/en/Main/ArduinoBoardNano (accessed October 17 2010).
RD/7	Overo Fire	Gumstix. 2010. Overo Fire COM. Available: http://www.gumstix.com/store/catalog/product_info.php?cPath=31&products_id=227 (accessed October 17 2010).
RD/8	Gumstix Overview	Gumstix. 2010. Setup and Programming of the Gumstix. Available: http://www.gumstix.net/Setup-and-Programming/view/Overo-Setup-and-Programming/Overview/111.html (accessed 17 2010).
RD/9	Gumstix file system	ETH PIXHAWK. Copy Linux filesystem on bootable SD card. Available: http://pixhawk.ethz.ch/wiki/tutorials/omap/copy_sd_card (accessed 17 2010).

RD/10	Sensor Dynamics 6DOF IMU	Sensor Dynamics. 2009. 6 DOF INERTIAL MEASUREMENT UNIT WITH CONTINUOUS SELF DIAGNOSIS. Available: http://www.sensordynamics.cc/images/content/file/product_linecards/1%205_6DoF_IMU_v1%207.pdf (accessed October 17 2010).
RD/11	Vicon DataStream SDK	Vicon. 2009. Vicon DataStream SDK 1.1.0 Developers Manual. Available: https://wiki.qut.edu.au/download/attachments/105028961/ Vicon+DataStream+SDK+Manual.pdf?version=1 (accessed October 17 2010).

List of Documentation

System Documentation

i.	AHNS-2010-SY-HL-001	AHNS 2010 High Level Objectives Document
ii.	AHNS-2010-SY-SR-001	AHNS 2010 System Requirements Document
iii.	AHNS-2010-SY-PM-001	AHNS 2010 Project Management Plan
iv.	AHNS-2010-SY-PM-002	AHNS 2010 Risk Management Plan

Subsystem Documentation

v.	AHNS-2010-AP-DD-003	Flight Computer Design Document
vi.	AHNS-2010-SE-DD-001	State Estimation Design Document
vii.	AHNS-2010-AP-TR-002	Flight Computer Test Report
viii.	AHNS-2010-SE-TR-001	Attitude State Estimation Test Report
ix.	AHNS-2010-SE-TR-002	Vicon State Estimation Test Report

Appendix A – IMU software library

Header file: `imuserial.h`

```
/**
 * @file   imuserial.h
 * @author Liam O'Sullivan
 *
 * $Author$
 * $Date$
 * $Rev$
 * $Id$
 *
 * Queensland University of Technology
 *
 * @section DESCRIPTION
 * IMU serial library header
 *
 */

#ifndef IMUSERIAL_H
#define IMUSERIAL_H

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>

// IMU baud rate (default)
#define IMU_BAUD_RATE_DEFAULT B38400
// IMU baud rate
#define IMU_BAUD_RATE B115200
// Delay between serial write and serial read (microseconds) (20000 =
38400)
#define IMU_DELAYRDWR 10000

int openIMUSerial(char* serialPort, int baudRate);
int closeIMUSerial();
int setIMUBaudRate(int baudRate);
int readCRBData(unsigned char reg);
int setCRBData(unsigned char reg, unsigned char value1, unsigned char
value2);
int getImuSensorData(double *rateXd, double *rateYd, double *rateZd, double
*accXd, double *accYd, double *accZd);
int startFreeRunningMode();
int stopFreeRunningMode();
int setIMUconfig();
#endif

// end imuserial.h
```

Source file: imuserial.c

```
/**
 * @file   imuserial.c
 * @author Liam O'Sullivan
 *
 * $Author$
 * $Date$
 * $Rev$
 * $Id$
 *
 * Queensland University of Technology
 *
 * @section DESCRIPTION
 * IMU serial library which implements:
 * - open and closing the serial connection with the IMU
 * - change the baudrate
 * - read or write CRB values
 * - request IMU data in one shot mode
 * - start or stop IMU data in free running mode
 *
 */

#include "imuserial.h"

// struct to save the port settings
struct termios currentSerialPort, previousSerialPort;
// specified serial port
int fd = 0;

// constants
// gyro scope offsets
const double mr1g = 0.0039;
const double mr2g = 0.0156;
// acceleration scope offsets
const double mr1a = 0.002031;
const double mr2a = 0.004062;
// gravity
const double g = 9.80665;
// temperature offset
const int tempOffset = 80;

/**
  * @brief Open the serial connection to the IMU
  */
int openIMUSerial(char* serialPort, int baudRate)
{

  // open the specified serial port
  fd = open(serialPort,O_RDWR | O_NOCTTY | O_NDELAY);
  // check if the serial port has been opened
  if (fd == -1)
  {
    // failed to open the serial port
    return 0;
  } else
    {
      //fcntl(fd, F_SETFL, 0);
    }
  // get serial port settings
  if(tcgetattr(fd, &previousSerialPort)==-1)
  {
    // failed to read the serial port settings
    closeIMUSerial();
    return 0;
  }
  // create a new memory structure for the port settings
  memset(&currentSerialPort, 0, sizeof(currentSerialPort));
  // set the baud rate, 8N1, enable the receiver, set local mode
  currentSerialPort.c_cflag = baudRate | CS8 | CLOCAL | CREAD;
  // ignore parity errors
  currentSerialPort.c_iflag = IGNPAR;
  currentSerialPort.c_oflag = 0;
  currentSerialPort.c_lflag = 0;
  // block until n bytes have been received
  currentSerialPort.c_cc[VMIN] = 0;
  currentSerialPort.c_cc[VTIME] = 0;
  // apply the settings to the port
  if((tcsetattr(fd, TCSANOW, &currentSerialPort))==-1)
  {
    // failed to apply port settings
    closeIMUSerial();
    return 0;
  }
  // successfully opened the port
  return 1;
}

/**
  * @brief Close the serial connection to the IMU
  */
int closeIMUSerial()
{
  close(fd);
  tcsetattr(fd, TCSANOW, &previousSerialPort);
  return 1;
}

/**
  * @brief Close the serial connection to the IMU
  */
int setBaudRate(int baudRate)
{
  // command to change baudrate
  unsigned char sCmd[6] = {'C','H','B','R', (char)baudRate,0x0D};
  if (baudRate > 3)
  {
    printf("Incorrect baudrate!\n");
    return 0;
  }
  // send the request to change the baud rate
  if (!write(fd,sCmd,6))
  {
    printf("Write failed\n");
    closeIMUSerial();
    return 0;
  }
  usleep(IMU_DELAYRDWR);
  // read the result
  unsigned char sResult[6];
  if(!read(fd,sResult,5))
  {
    printf("Read failed\n");
    closeIMUSerial();
    return 0;
  }
  // terminate with null character
  sResult[5] = 0x00;
  printf(">> Change baudrate %d |%s\n",baudRate,sResult);
  return 1;
}

/**
  * @brief Read the requested CRB register value
  */
int readCRBData(unsigned char reg)
{
  unsigned char sCmd[6] = {'R','C','R','B',reg,0x0D};
  // send the request to read the CRB reg value
  if (!write(fd,sCmd,6))
  {
    printf("Write failed\n");
    closeIMUSerial();
    return 0;
  }
  usleep(IMU_DELAYRDWR);
  // read the reg value returned
  unsigned char sResult[5];
  if(!read(fd,sResult,4))
  {
    printf("Read failed\n");
    closeIMUSerial();
    return 0;
  }
  // terminate with null character
  sResult[4] = 0x00;
  printf(">> Read %c|%s",reg,sResult);
  return 1;
}

/**
  * @brief Set the requested CRB register value
  */
int  setCRBData(unsigned  char  reg,  unsigned  char  value1,  unsigned  char
value2)
{
  unsigned char sCmd[8] = {'W','C','R','B',reg,value1,value2,0x0D};
  // send the request to read the CRB reg value
  if (!write(fd,sCmd,8))
  {
    printf("Write failed\n");
    closeIMUSerial();
    return 0;
  }
  usleep(IMU_DELAYRDWR);
  // read the reg value returned
  unsigned char sResult[6];
  if(!read(fd,sResult,5))
  {
    printf("Read failed\n");
    closeIMUSerial();
    return 0;
  }
  // terminate with null character
  sResult[5] = 0x00;
  printf(">> Set %c%c%c|%s",reg,value1,value2,sResult);
  return 1;
}

/**
  * @brief Request and receive IMU data (one shot mode)
  */
int getImuSensorData(double *rateXd, double *rateYd, double *rateZd, double
*accXd, double *accYd, double *accZd)
{
  unsigned char sCmd[7] = {'S','T','A','O','S','M',0x0D};

  if(!write(fd,sCmd,7))
  {
    printf("Write failed\n");
    closeIMUSerial();
    return 0;
  }

  unsigned char sResult[50];
  usleep(IMU_DELAYRDWR);
  if(!read(fd,sResult,49))
  {
    printf("Read failed\n");
    closeIMUSerial();
    return 0;
  }
  // calculate the CRC value
  unsigned char crcResult = 0;
  int i = 0;
  for(i=0; i<45;i++)
  {
    crcResult = crcResult + sResult[i];
  }
  crcResult = ~crcResult;

  short int rateX = 0;
  short int rateY = 0;
  short int rateZ = 0;
  short int accX = 0;
  short int accY = 0;
  short int accZ = 0;
  unsigned short int temp=0;
  unsigned char statusL=0;
  unsigned char statusH=0;
  unsigned char crc=0;

  //printf("crc given: %hx\n",crcResult);
  // terminate with null character
  sResult[50] = 0x00;
  //printf("%s",sResult);

  sscanf(sResult,   "%hx,%hx,%hx,%hx,%hx,%hx,%hx,%hhx,%hhx,%hhx",   &rateX,
&rateY, &rateZ, &accX, &accY, &accZ, &temp, &statusL, &statusH, &crc);
```

```
  // calculate the rates
  *rateXd = rateX*mr2g;
  *rateYd = rateY*mr2g;
  *rateZd = rateZ*mr2g;
  // calculate the acceleration
  *accXd = accX*mr2a;
  *accYd = accY*mr2a;
  *accZd = accZ*mr2a;
  // calculate the temperature
  temp = temp-tempOffset;

  return 1;
}

/**
  * @brief Start free running mode
  */
int startFreeRunningMode()
{
  unsigned char sCmd[7] = {'S','T','A','F','R','M',0x0D};

  if(!write(fd,sCmd,7))
  {
    printf("Write failed\n");
    closeIMUSerial();
    return 0;
  }
  return 1;
}

/**
  * @brief Stop free running mode
  */
int stopFreeRunningMode()
{
  unsigned char sCmd[7] = {'S','T','P','F','R','M',0x0D};

  if(!write(fd,sCmd,7))
  {
    printf("Write failed\n");
    closeIMUSerial();
    return 0;
  }
  usleep(IMU_DELAYRDWR);
  // read the result
  unsigned char sResult[6];
  if(!read(fd,sResult,5))
  {
    printf("Read failed\n");
    closeIMUSerial();
    return 0;
  }
  // terminate with null character
  sResult[6] = 0x00;
  printf("%s", sResult);
  return 1;
}

/**
```

```
  * @brief set IMU configuration for BAUD_RATE baudrate and the rate2/acc2
values
  */
int setIMUconfig()
{
  // set the rate 2 and acc2 values (C=C0,D=74)
  setCRBData('C', 'C', '0');
  usleep(IMU_DELAYRDWR);
  readCRBData('C');
  usleep(IMU_DELAYRDWR);
  setCRBData('D', '7', 'F');
  usleep(IMU_DELAYRDWR);
  readCRBData('D');
  usleep(IMU_DELAYRDWR);
  // set the baudrate to 115200 (results in a disconnect)
  // 0 = 19200, 1 = 38400, 2 = 57600, 3 = 115200
  setBaudRate(3);
  usleep(IMU_DELAYRDWR);
  printf(">> IMU reconfigured\n");
}

// end imuserial.c
```

Appendix B – Arduino sensor program

Source file: `arduserial.pde`

```
/**
 * @file   arduserial.pde
 * @author Liam O'Sullivan
 *
 * $Author$
 * $Date$
 * $Rev$
 * $Id$
 *
 * Queensland University of Technology
 *
 * @section DESCRIPTION
 * Arduino code to transmit to the overo the following:
 * 1. connect to compass and extract heading data
 * 2. connect to camera and gather image processing data
 * 3. connect to ultrasonic sensor and request height data
 * 4. perform adc read on battery voltage
 *
 */

#include "NewSoftSerial.h"

// compass transmit pin
#define COMPASS_TX 4
// compass receive pin
#define COMPASS_RX 5
// battery voltage adc pin
#define VOLTAGE_ADC 7
// altitude sensor adc pin
#define ALTITUDE_ADC 6
// compass baud rate
#define COMPASS_BAUDRATE 19200
// overo baud rate
#define OVERO_BAUDRATE 115200
// battery voltage read time
#define ADC_READ 5

// software serial compass object
NewSoftSerial compass(COMPASS_RX,COMPASS_TX);
// compass heading (0.0 - 359.9)
char compass_heading[10];
// compass buffer
char compass_buffer[10];
// compass index
int compass_index = 0;
// compss heading index
int compass_heading_index = 0;
// compass buffer index
int compass_buffer_index = 0;
// compass index difference
int compass_index_diff = 0;
int compass_received = 0;

// compass character
char compass_char = 0;
```

```
// overo command
char overo_com = 0;
// overo loop counter
int i=0;
// battery voltage reading
int adc_read = 0;
// battery voltage raw
double bat_voltage_raw = 0;
// transmitted battery voltage
double bat_voltage = 0;
// transmitted altitude reading
double altitude = 0;
// time counter adc
unsigned long current_time;
// previous time
unsigned long previous_time;
// battery voltage array
double bat_voltage_avg[5];
// battery voltage avg counter
int bat_voltage_avg_count = 0;
// altitude sensor array
double altitude_avg[5];
// altitude voltage
int altitude_avg_count = 0;
// moving average counter
double average = 0.0;

// arduserial functions
// read the serial compass data
int readCompass();
// read the battery voltage
int readVoltage();
// print data to the overo
int printOvero();

void setup()
{
  // setup delay time
  //delay(35000);
  Serial.begin(OVERO_BAUDRATE);
  pinMode(COMPASS_TX, OUTPUT);
  pinMode(COMPASS_RX, INPUT);
  // software serial (receive pin, transmit pin)
  compass.begin(COMPASS_BAUDRATE);
  // init the time
  previous_time = millis();
}


void loop()
{
  // check the time
  current_time = millis() - previous_time;
  readCompass();
  if (current_time > ADC_READ)
  {
      readVoltage();
      readAltitude();
      previous_time = millis();
  }
```

```
    if (compass_received == 1)
    {
      printOvero();
    }
  }

  // read serial data transmitted from the compass
  // Compass sentence: $C256.3P-0.7R-178.5T25.5*29
  int readCompass()
  {
    if (compass.available() > 0)
    {
      // read the compass character
      compass_char = compass.read();
      if (compass_char != '\n')
      {
        compass_buffer[compass_buffer_index++] = compass_char;
      }
      if (compass_char == '\n')
      {
        compass_heading_index = compass_buffer_index;
        for(i=0; i<compass_heading_index; i++)
        {
          compass_heading[i] = compass_buffer[i];
        }
        compass_buffer_index = 0;
        compass_received = 1;
      }
    }
  }

  // read the battery voltage via ADC
  int readVoltage()
  {
     adc_read = analogRead(VOLTAGE_ADC);
     bat_voltage_raw = (adc_read/double(1023))*5;
     bat_voltage = 2.7964*bat_voltage_raw + 0.103465;
     // store the voltage data in an array
     bat_voltage_avg[bat_voltage_avg_count] = bat_voltage;
     bat_voltage_avg_count++;
     if (bat_voltage_avg_count > 4)
     {
      bat_voltage_avg_count = 0;
     }
     average = 0;
     // compute the moving avg
     for(i=0; i<5; i++)
     {
       average = average + bat_voltage_avg[i];
     }
     // store the filter battery voltage
     bat_voltage = average/5.0;
  }

  // read the altitude sensor voltage via ADC
  int readAltitude()
  {
    adc_read = analogRead(ALTITUDE_ADC);
    altitude = 0.0135*adc_read + 0.04155;
    // store the voltage data in an array
    /*altitude_avg[altitude_avg_count] = altitude;
    altitude_avg_count++;
    if (altitude_avg_count > 4)
    {
     altitude_avg_count = 0;
    }
    average = 0;
    // compute the moving avg
    for(i=0; i<5; i++)
    {
      average = average + altitude_avg[i];
    }
    // store the filter battery voltage
    altitude = average/5.0;*/
  }

  // read serial data transmitted from the overo and reply with suitable
  response
  int printOvero()
  {
    // print the compass data
    Serial.print('C');
    // format the compass data
    // less than 10
    if (compass_heading[1] == '.')
    {
      Serial.print('0');
      Serial.print('0');
    }
    // less than 100
    if (compass_heading[2] == '.')
    {
      Serial.print('0');
    }
    for(i=0; i<compass_heading_index-1; i++)
    {
      Serial.print(compass_heading[i],BYTE);
    }
    Serial.print(",V");
    Serial.print(bat_voltage,3);
    Serial.print(",A");
    Serial.print(altitude,3);
    Serial.print(0x0D,BYTE);
    Serial.print(0x0A,BYTE);
  }

  // end arduserial.pde
```

Appendix C – Arduino serial pass-through

Source file: `ardupass.pde`

```
/**
 * @file   arduserial.pde
 * @author Liam O'Sullivan
 *
 * $Author$
 * $Date$
 * $Rev$
 * $Id$
 *
 * Queensland University of Technology
 *
 * @section DESCRIPTION
 * Arduino code implement a serial pass through to the compass
 */

#include "NewSoftSerial.h"

// compass transmit pin
#define COMPASS_TX 4
// compass receive pin
#define COMPASS_RX 5
// compass baudrate
#define COMPASS_BAUDRATE 19200
// overo baudrate
#define OVERO_BAUDRATE 115200

// sofrware serial compass object
NewSoftSerial compass(COMPASS_RX,COMPASS_TX);

void setup()
{
 Serial.begin(OVERO_BAUDRATE);
 pinMode(COMPASS_TX, OUTPUT);
 pinMode(COMPASS_RX, INPUT);
 // software serial (receive pin, transmit pin)
 compass.begin(COMPASS_BAUDRATE);
}

void loop()
{
 char c;
 // read the compass and print to overo
 if (compass.available() > 0)
 {
   c = compass.read();
   Serial.print(c);
 }
 // read from the overo and print to the compass
 if (Serial.available() > 0)
 {
   c = Serial.read();
   compass.print(c);
```

Appendix D – Arduino software library

Header file: `arduserial.h`

```
/**
 * @file   arduserial.h
 * @author Liam O'Sullivan
 *
 * $Author$
 * $Date$
 * $Rev$
 * $Id$
 *
 * Queensland University of Technology
 *
 * @section DESCRIPTION
 * Arduino serial library header
 *
 */

#ifndef ARDUSERIAL_H
#define ARDUSERIAL_H

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>

// Arduino baud rate
#define ARDU_BAUD_RATE B115200
// Delay between serial write and serial read (microseconds)
#define ARDU_DELAYRDWR 0

int openArduSerial(char* serialPort, int baudRate);
int openArduSerialCan(char* serialPort, int baudRate);
int closeArduSerial();
int getCompassHeading(double *compassHeading);
int getBatteryVoltage(double *batteryVoltage);
int getAltitudeReading(double *altitudeReading);
int  getArduinoData(double  *compassHeading,  double  *batteryVoltage,  double
*altitudeReading);
int   getArduinoDataCan(double   *compassHeading,   double   *batteryVoltage,
double *altitudeReading);

#endif

// end arduserial.h
```

Source file: `arduserial.c`

```c
/**
 * @file   arduserial.c
 * @author Liam O'Sullivan
 *
 * $Author$
 * $Date$
 * $Rev$
 * $Id$
 *
 * Queensland University of Technology
 *
 * @section DESCRIPTION
 * Arduino serial library which implements:
 * - open and closing the serial connection with the arduino
 * - getting the compass heading from the arduino
 * ... others
 *
 */

#include "arduserial.h"

// struct to save the port settings
struct termios currentSerialPort, previousSerialPort;
// specified serial port
int arduSerialfd = 0;

/**
  * @brief Open the serial connection to the IMU
  */
int openArduSerial(char* serialPort, int baudRate)
{
  // open the specified serial port
  arduSerialfd = open(serialPort,O_RDWR | O_NOCTTY | O_NDELAY);
  // check if the serial port has been opened
  if (arduSerialfd == -1)
  {
    // failed to open the serial port
    return 0;
  } else
    {
      //fcntl(arduSerialfd, F_SETFL, 0);
    }
  // get serial port settings
  if(tcgetattr(arduSerialfd, &previousSerialPort)==-1)
  {
    // failed to read the serial port settings
    closeArduSerial();
    return 0;
  }
  // create a new memory structure for the port settings
  memset(&currentSerialPort, 0, sizeof(currentSerialPort));
  // set the baud rate, 8N1, enable the receiver, set local mode
  currentSerialPort.c_cflag = baudRate | CS8 | CLOCAL | CREAD;
  // ignore parity errors
  currentSerialPort.c_iflag = IGNPAR;
  currentSerialPort.c_oflag = 0;
  currentSerialPort.c_lflag = 0;
  // no flow control
  currentSerialPort.c_cflag &= ~CRTSCTS;
  currentSerialPort.c_iflag  &=  ~(IXON  |  IXOFF  |  IXANY);  //  turn  off  s/w
flow ctrl

  currentSerialPort.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG); // make raw
  currentSerialPort.c_oflag &= ~OPOST; // make raw

  // block until n bytes have been received
  currentSerialPort.c_cc[VMIN] = 0;
  currentSerialPort.c_cc[VTIME] = 0;
  // flush the serial port
  tcflush(arduSerialfd, TCIFLUSH);
  // apply the settings to the port
  if((tcsetattr(arduSerialfd, TCSANOW, &currentSerialPort))==-1)
  {
    // failed to apply port settings
    closeArduSerial();
    return 0;
  }
  // successfully opened the port
  return 1;
}

/**
  * @brief Close the serial connection to the arduino
  */
int closeArduSerial()
{
  close(arduSerialfd);
  tcsetattr(arduSerialfd, TCSANOW, &previousSerialPort);
  return 1;
}

/**
  * @brief get the compass heading from the arduino
  */
int getCompassHeading(double *compassHeading)
{
  unsigned char sCmd[1] = {'C'};
  // send the command to the arduino
  if (!write(arduSerialfd,sCmd,1))
  {
    printf("Write failed\n");
    return 0;
  }
  usleep(ARDU_DELAYRDWR);
  // read the compass heading returned
  unsigned char sResult[6];
  if(!read(arduSerialfd,sResult,5))
  {
    printf("Read failed\n");
    return 0;
  }
  // terminate with null character
  sResult[5] = 0x00;
  // save to compass heading
  sscanf(sResult, "%lf", compassHeading);
  return 1;
}
```

```
/**
  * @brief get the battery voltage from the arduino
  */
int getBatteryVoltage(double *batteryVoltage)
{
  unsigned char sCmd[1] = {'V'};
  // send the command to the arduino
  if (!write(arduSerialfd,sCmd,1))
  {
    printf("Write failed\n");
    return 0;
  }
  usleep(ARDU_DELAYRDWR);
  // read the battery voltage
  unsigned char sResult[7];
  if(!read(arduSerialfd,sResult,6))
  {
    printf("Read failed\n");
    return 0;
  }
  // terminate with null character
  sResult[6] = 0x00;
  // save to battery voltage
  sscanf(sResult, "%lf", batteryVoltage);
  return 1;
}

/**
  * @brief get the altitude reading
  */
int getAltitudeReading(double *altitudeReading)
{
  unsigned char sCmd[1] = {'A'};
  // send the command to the arduino
  if (!write(arduSerialfd,sCmd,1))
  {
    printf("Write failed\n");
    return 0;
  }
  usleep(ARDU_DELAYRDWR);
  // read the altitude reading
  unsigned char sResult[6];
  if(!read(arduSerialfd,sResult,5))
  {
    printf("Read failed\n");
    return 0;
  }
  // terminate with null character
  sResult[5] = 0x00;
  // save to altitude reading
  sscanf(sResult, "%lf", altitudeReading);
  return 1;
}

/**
  * @brief get arduino data
  */
int  getArduinoData(double  *compassHeading,  double  *batteryVoltage,  double
*altitudeReading)
```

```
{
unsigned char sCmd[1] = {'O'};
  // send the command to the arduino
  if (!write(arduSerialfd,sCmd,1))
  {
    printf("Write failed\n");
    return 0;
  }
  usleep(ARDU_DELAYRDWR);
  // read the altitude reading
  unsigned char sResult[23];
  if(!read(arduSerialfd,sResult,22))
  {
    printf("Read failed\n");
    return 0;
  }
  // terminate with null character
  sResult[22] = 0x00;
  // save to altitude reading
  sscanf(sResult,   "C%lf,V%lf,A%lf\n",   compassHeading,   batteryVoltage,
altitudeReading);
  return 1;
}

int openArduSerialCan(char* serialPort, int baudRate)
{
  // open the specified serial port
  arduSerialfd = open(serialPort,O_RDWR | O_NOCTTY );
  // check if the serial port has been opened
  if (arduSerialfd == -1)
  {
    // failed to open the serial port
    return 0;
  } else
    {
      //fcntl(arduSerialfd, F_SETFL, 0);
    }
  // get serial port settings
  if(tcgetattr(arduSerialfd, &previousSerialPort)==-1)
  {
    // failed to read the serial port settings
    closeArduSerial();
    return 0;
  }
  // create a new memory structure for the port settings
  memset(&currentSerialPort, 0, sizeof(currentSerialPort));
  // set the baud rate, 8N1, enable the receiver, set local mode
  currentSerialPort.c_cflag = baudRate | CS8 | CLOCAL | CREAD;
  // ignore parity errors
  currentSerialPort.c_iflag = IGNPAR;
  currentSerialPort.c_oflag = 0;
  currentSerialPort.c_lflag = 0;

  // no flow control
  currentSerialPort.c_cflag &= ~CRTSCTS;
  currentSerialPort.c_iflag  &=  ~(IXON  |  IXOFF  |  IXANY);  //  turn  off  s/w
flow ctrl

  currentSerialPort.c_lflag &= ICANON;
  // block until n bytes have been received
  currentSerialPort.c_cc[VMIN] = 22;
  currentSerialPort.c_cc[VTIME] = 0;
  // flush the serial port
  tcflush(arduSerialfd, TCIFLUSH);
  // apply the settings to the port
  if((tcsetattr(arduSerialfd, TCSANOW, &currentSerialPort))==-1)
  {
    // failed to apply port settings
    closeArduSerial();
    return 0;
  }
  // succesfully opened the port
  return 1;
}

int  getArduinoDataCan(double  *compassHeading,  double  *batteryVoltage,
double *altitudeReading)
{
  unsigned char sResult[255];
  int res = 0;
  usleep(ARDU_DELAYRDWR);
  // read the arduino data
  res = read(arduSerialfd,sResult,255);
  //if(res < 25)
  //{
  //  printf("Read failure");
  //  return 0;
  //}
  // terminate with null character
  sResult[res] = 0x00;
  //printf("res:%d|",res);
  // save to altitude reading
  //int i = 0;
  sscanf(sResult,  "C%lf,V%lf,A%lf\r\n",  compassHeading,  batteryVoltage,
altitudeReading);
  return 1;
}
// end arduserial.c
```

Appendix E – Flight computer implementation

Header file: `main.h`

```
/**
 * @file   main.h
 * @author Liam O'Sullivan
 *
 * $Author$
 * $Date$
 * $Rev$
 * $Id$
 *
 * Queensland University of Technology
 *
 * @section DESCRIPTION
 * Flight computer header
 *
 */

#ifndef MAIN_H
#define MAIN_H

// server port
#define SERVER_PORT 2002
// server delay (milliseconds)
#define SERVER_DELAY 10

// arduino serial port
#define ARDU_SERIAL_PORT "/dev/ttyUSB0"
// compass read delay (milliseconds)
#define COMPASS_DELAY 0.1

// IMU serial port
#define IMU_SERIAL_PORT "/dev/ttyS1"
// IMU read delay(milliseconds)
#define IMU_DELAY 0.1

// MCU serial port
#define MCU_SERIAL_PORT "/dev/ttyS0"
// MCU delay (milliseconds)
#define MCU_UPDATE_DELAY 3
// MCU query delay (milliseconds)
#define MCU_QUERY_DELAY 250

// Control delay (milliseconds)
#define CONTROL_DELAY 2.5
#endif

// end main.h
```

Source file: `main.c`

```
/**
 * $Author: liamosullivan $
 * $Date: 2010-06-21 15:25:25 +1000 (Mon, 21 Jun 2010) $
 * $Rev: 193 $
 * $Id: main.c 193 2010-06-21 05:25:25Z liamosullivan $
 *
 * Queensland University of Technology
 *
 * @section DESCRIPTION
 * Flight computer main program
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include "main.h"
#include "server.h"
#include "state.h"
#include "imuserial.h"
#include "arduserial.h"
#include "kfb.h"
#include "control.h"
#include "mcuserial.h"
#include "MCUCommands.h"

// udp server
static Server server;
// state variable
state_t state;
// sensor data variable
sensor_data_t raw_IMU;
// compass heading
double compass_heading = 0.0;
// flight computer + engine state
fc_state_t fcState;
// autopilot state
ap_state_t apState;

pthread_mutex_t fcMut = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t apStateMut = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

// flight computer functions
int sensorInit();
int mcuInit();
// flight computer thread functions
void * sendUDPData(void *pointer);
void * updateIMUdata(void *pointer);
void * updateCompassHeading(void *pointer);
void * updateArduinoData(void *pointer);
void * updateMCU(void *pointer);
void * updateControl(void *pointer);

int main(int argc, char *argv[])
```

```
{
  // boolean variables to check serial connections
  int sensorReady = 0;
  int mcuReady = 0;
  // init the server
  server_init(&server, SERVER_PORT);
  // initialise the sensors
  sensorReady = sensorInit();
  // initialise the MCU
  mcuReady = mcuInit();
  // initialise the Kalman filter
  //attitudeFilterInitialise();
  // all systems operational - begin the flight computer
  if (sensorReady && mcuReady)
  {
    // create the flight computer threads
    pthread_t udpThread;
    pthread_t imuThread;
    pthread_t arduThread;
    pthread_t mcuThread;
    pthread_t controlThread;

    // create the server thread
    pthread_create(&udpThread, NULL, sendUDPData, (int*) 1);
    // create the imu thread
    pthread_create(&imuThread, NULL, updateIMUdata, (int*) 2);
    // create the arduino thread
    //pthread_create(&arduThread, NULL, updateCompassHeading, (int*) 3);i
    pthread_create(&arduThread, NULL, updateArduinoData, (int*) 3);
    //create the mcu thread
    pthread_create(&mcuThread, NULL, updateMCU, (int*) 4);
    //create the control thread
    pthread_create(&controlThread, NULL, updateControl, (int*) 5);

    // execute the udp server thread
    pthread_join(udpThread,NULL);
    // execute the state thread
    pthread_join(imuThread,NULL);
    // execute the compass thread
    pthread_join(arduThread,NULL);
    // execute the mcu thread
    pthread_join(mcuThread,NULL);
    // execute the control thread
    pthread_join(controlThread,NULL);

  }
  return 0;
}

// update the compass heading from the arduino
void * updateCompassHeading(void *pointer)
{
  double previous_compass_heading = 0.0;
  while(1)
  {
    // sleep the thread for updating the compass
    usleep(COMPASS_DELAY*1e3);
    // get the compass heading from the arduino
    pthread_mutex_lock(&mut);
    previous_compass_heading = compass_heading;
    if (!getCompassHeading(&compass_heading))
    {
      compass_heading = previous_compass_heading;
    }
    raw_IMU.psi = compass_heading;
    pthread_mutex_unlock(&mut);
  }
  return NULL;
}

// update the arduino data
void * updateArduinoData(void *pointer)
{
  double compass_heading_old = 0.0;
  double voltage_old = 0.0;
  double z_old = 0.0;
  double state_vz_previous = 0.0;
  // time stamp
  struct timeval timestamp;
  double startArduinoTime, endArduinoTime, diffArduinoTime;
  gettimeofday(&timestamp, NULL);
  startArduinoTime = timestamp.tv_sec+(timestamp.tv_usec/1000000.0);
  while(1)
  {
    // sleep the thread for updating the arduino
    usleep(COMPASS_DELAY*1e3);
    // get the data arduino
    pthread_mutex_lock(&mut);
    compass_heading_old = compass_heading;
    voltage_old = state.voltage;
    z_old = raw_IMU.z;
    if (!getArduinoDataCan(&compass_heading, &state.voltage, &raw_IMU.z))
    {
      compass_heading = compass_heading_old;
      state.voltage = voltage_old;
      raw_IMU.z = z_old;
    }
    gettimeofday(&timestamp, NULL);
    endArduinoTime = timestamp.tv_sec+(timestamp.tv_usec/1000000.0);
    diffArduinoTime = endArduinoTime - startArduinoTime;
    // calculate start time
    gettimeofday(&timestamp, NULL);
    startArduinoTime = timestamp.tv_sec+(timestamp.tv_usec/1000000.0);
    //printf(">> Ard: %f %f %f\n",compass_heading,state.voltage,raw_IMU.z);
    altLPF(&raw_IMU.z);
    // set the altitude to the filtered sensor date
    state.z = raw_IMU.z;
    // calculate altitude velocity
    state.vz = (state.z - state_vz_previous)/diffArduinoTime;
    // assign old altitude value
    state_vz_previous = state.z;
    // assign compass heading
    raw_IMU.psi = compass_heading;
    pthread_mutex_unlock(&mut);
  }
  return NULL;
}

// update IMU data
void * updateIMUdata(void *pointer)
```

```
{
  // time stamp
  struct timeval timestamp;
  // time between filter updates
  double startFilterTime, endFilterTime, diffFilterTime;
  // get IMU data to calculate initial state
  getImuSensorData(&raw_IMU.p,    &raw_IMU.q,    &raw_IMU.r,    &raw_IMU.ax,
&raw_IMU.ay, &raw_IMU.az);
  // initialise the Kalman filter
  attitudeFilterInitialiseB(&raw_IMU.ax, &raw_IMU.ay, &raw_IMU.az);
  // calculate filter start time
  gettimeofday(&timestamp, NULL);
  startFilterTime = timestamp.tv_sec+(timestamp.tv_usec/1000000.0);
  while(1)
  {
    // sleep the thread for updating the IMU
    usleep(IMU_DELAY*1e3);
    pthread_mutex_lock(&mut);
    // get the IMU sensor data
    getImuSensorData(&raw_IMU.p,   &raw_IMU.q,   &raw_IMU.r,   &raw_IMU.ax,
&raw_IMU.ay, &raw_IMU.az);
    // calculate the time elapsed since last update
    gettimeofday(&timestamp, NULL);
    endFilterTime = timestamp.tv_sec+(timestamp.tv_usec/1000000.0);
    diffFilterTime = endFilterTime - startFilterTime;
    // calculate filter start time
    gettimeofday(&timestamp, NULL);
    startFilterTime = timestamp.tv_sec+(timestamp.tv_usec/1000000.0);
    // perform the attitude filtering using the imu data
    //attitudeFilter(&raw_IMU.p,   &raw_IMU.q,   &raw_IMU.r,   &raw_IMU.ax,
&raw_IMU.ay,   &raw_IMU.az,   &state.p,   &state.q,   &state.r,   &state.phi,
&state.theta, &state.psi, compass_heading, diffFilterTime);
    attitudeFilterB(&raw_IMU.p,    &raw_IMU.q,    &raw_IMU.r,    &raw_IMU.ax,
&raw_IMU.ay,   &raw_IMU.az,   &state.p,   &state.q,   &state.r,   &state.phi,
&state.theta, &state.psi, &compass_heading, diffFilterTime);
    //printf(">> kf update : %f\n",1/diffFilterTime);
    state.trace = (1/diffFilterTime);

    pthread_mutex_unlock(&mut);
  }
  return NULL;
}

// initialise the sensors connected to the flight computer
// 1. IMU
// 2. Arduino (compass, camera, ultrasonic)
int sensorInit()
{
  int retSerial = 0;
  int retValue =0;
  // connect to the IMU
  retSerial = openIMUSerial(IMU_SERIAL_PORT, IMU_BAUD_RATE);
  if(retSerial)
  {
    retValue = 1;
    printf(">> IMU connected\n");
  } else
    {
      printf(">>   Error:   Cannot   connect   to   IMU   on   serial   port:
%s\n",IMU_SERIAL_PORT);
```

```
    }
  // connect to the arduino
  retSerial = openArduSerial(ARDU_SERIAL_PORT, ARDU_BAUD_RATE);
  if(retSerial)
  {
    retValue = 1;
    printf(">> Arduino connected\n");
  } else
    {
      printf(">> Error: Cannot connect to Arduino on serial port: %s\n",
ARDU_SERIAL_PORT);
    }

  return retValue;
}

// initialise connection to the MCU
int mcuInit()
{
  int retValue = 0;
  retValue = mcuOpenSerial(MCU_SERIAL_PORT,MCU_BAUD_RATE);
  if (retValue)
  {
    printf(">> MCU connected \n");
  } else
    {
      printf(">>   Error:   Cannot   connect   to   MCU   on   serial   port:
%s\n",MCU_SERIAL_PORT);
    }
  return retValue;
}

// send the UDP data packets to clients
void * sendUDPData(void *pointer)
{
  // udp packet type
  int udpType = 0;
  int init = 1;
  int count=0;
  struct timeval timestamp;
  double startServerTime = 0;
  double endServerTime = 0;
  double diffServerTime = 0;
  int dataLength;
  unsigned char buffer[512];
  // initialise startTime
  gettimeofday(&timestamp, NULL);
  startServerTime=timestamp.tv_sec+(timestamp.tv_usec/1000000.0);

  while(1)
  {
    int type=0;
    if(server_poll(&server,1))
    {
      udpType = server_get_packet(&server);
    }
    if(udpType < 0)
    {
      fprintf(stderr,"Read error from server");
      //return -1;
    }
    count++;
    usleep(SERVER_DELAY*1e3);
    //send data to the socket
    if(count%1 == 0)
    {
      init = 1;
    }
    if(init)
    {
      // calculate server delay time
      gettimeofday(&timestamp, NULL);
      endServerTime=timestamp.tv_sec+(timestamp.tv_usec/1000000.0);
      diffServerTime = endServerTime - startServerTime;
      // output the sent states
      //printf(">>       state       :       %f       %f       %f       %f       |
%f\n",raw_IMU.p,raw_IMU.q,raw_IMU.r,state.psi,(1/diffServerTime));
      // send the state packet
      server_send_packet(&server, HELI_STATE, &state, sizeof(state_t));
      // send the raw sensor data packet
      dataLength = PackSensorData(buffer, &raw_IMU);
      server_send_packet(&server, SENSOR_DATA, buffer, dataLength);
      // send the fc state packet
      pthread_mutex_lock(&fcMut);
      dataLength = PackFCState(buffer,&fcState);
      pthread_mutex_unlock(&fcMut);
      server_send_packet(&server, FC_STATE, buffer, dataLength);
      // send ap state packet
      pthread_mutex_lock(&apStateMut);
      dataLength = PackAPState(buffer,&apState);
      server_send_packet(&server, AUTOPILOT_STATE, buffer, dataLength);
      pthread_mutex_unlock(&apStateMut);
      init = 0;
      // reinitialise startTime
      gettimeofday(&timestamp, NULL);
      startServerTime=timestamp.tv_sec+(timestamp.tv_usec/1000000.0);
    }
  }
  return NULL;
}

void* updateMCU(void *pointer)
{
  static FILE *fidMCU = NULL;
  fidMCU = fopen("mcu.log","w");
  uint8_t mcuMode = 0;
  uint16_t readEngine[4];
  static int mcuDelayCount = 0;
  // time stamp
  struct timeval timestamp, timestamp1;
  double startMCUTime, endMCUTime, diffMCUTime;
  gettimeofday(&timestamp, NULL);
  startMCUTime = timestamp.tv_sec+(timestamp.tv_usec/1000000.0);

  while (1)
  {
    // Send Data
    pthread_mutex_lock(&apMutex);
    sendMCUCommands(&apMode, &apThrottle, &apRoll, &apPitch, &apYaw);
    pthread_mutex_unlock(&apMutex);
```

```
      // Get RC Data
      getMCUPeriodic(&mcuMode,readEngine);

      // RC Commands
      pthread_mutex_lock(&rcMutex);
      rcMode = mcuMode;
      rcThrottle = PWMToCounter(readEngine[0] - zeroThrottle);
      rcRoll = PWMToCounter(readEngine[1] - zeroRoll);
      rcPitch = PWMToCounter(readEngine[2] - zeroPitch);
      rcYaw = PWMToCounter(-readEngine[3] + zeroYaw);

      if (rcThrottle <= 0)
      {
        rollTrim = rcRoll;
        pitchTrim = rcPitch;
        yawTrim = rcYaw;
        //fprintf(stderr,">> RC Trims Set...");
      }
      pthread_mutex_unlock(&rcMutex);

      // Report to GCS
      pthread_mutex_lock(&fcMut);
      fcState.rclinkActive = mcuMode;
      fcState.commandedEngine1 = readEngine[0];
      fcState.commandedEngine2 = readEngine[1];
      fcState.commandedEngine3 = readEngine[2];
      fcState.commandedEngine4 = readEngine[3];
      pthread_mutex_unlock(&fcMut);

      // calculate the mcu thread update time
      gettimeofday(&timestamp1, NULL);
      endMCUTime = timestamp1.tv_sec+(timestamp1.tv_usec/1000000.0);
      diffMCUTime = endMCUTime - startMCUTime;
      startMCUTime = timestamp1.tv_sec+(timestamp1.tv_usec/1000000.0);
      fprintf(fidMCU,"%lf\n",1/diffMCUTime);

      usleep(MCU_UPDATE_DELAY*1e3);
    }

    return NULL;
}

void* updateControl(void *pointer)
{
  // initialise gains and parameters from files
  int i = 0, j = 0;
  FILE *gainsfd = NULL;
  double gains[6][3];
  FILE *parametersfd = NULL;
  double parameters[6][3];

  gainsfd = fopen("gains.ahnsgains","r");
  parametersfd = fopen("parameters.ahnsparameters","r");
  if (gainsfd && parametersfd)
  {
    for (i = 0; i < 6; ++i)
    {
      for (j = 0; j < 3; ++j)
      {
        fscanf(gainsfd, "%lf", &gains[i][j]);
        fscanf(parametersfd, "%lf", &parameters[i][j]);
      }
    }
    fclose(gainsfd);
    fclose(parametersfd);
  }
  else
  {
    fprintf(stderr,"void*   controlThread(void   *pointer)   ::   FILE   OPEN
FAILED");
    for (i = 0; i < 6; ++i)
    {
      for (j = 0; j < 3; ++j)
      {
        gains[i][j] = 0.0;
        parameters[i][j] = 0.0;
      }
    }
  }

  rollLoop.maximum = parameters[0][0];
  rollLoop.neutral = parameters[0][1];
  rollLoop.minimum = parameters[0][2];

  pitchLoop.maximum = parameters[1][0];
  pitchLoop.neutral = parameters[1][1];
  pitchLoop.minimum = parameters[1][2];

  yawLoop.maximum = parameters[2][0];
  yawLoop.neutral = parameters[2][1];
  yawLoop.minimum = parameters[2][2];

  xLoop.maximum = parameters[3][0];
  xLoop.neutral = parameters[3][1];
  xLoop.minimum = parameters[3][2];

  yLoop.maximum = parameters[4][0];
  yLoop.neutral = parameters[4][1];
  yLoop.minimum = parameters[4][2];

  zLoop.maximum = parameters[5][0];
  zLoop.neutral = parameters[5][1];
  zLoop.minimum = parameters[5][2];

  rollLoop.Kp = gains[0][0];
  rollLoop.Ki = gains[0][1];
  rollLoop.Kd = gains[0][2];

  pitchLoop.Kp = gains[1][0];
  pitchLoop.Ki = gains[1][1];
  pitchLoop.Kd = gains[1][2];

  yawLoop.Kp = gains[2][0];
  yawLoop.Ki = gains[2][1];
  yawLoop.Kd = gains[2][2];

  xLoop.Kp = gains[3][0];
  xLoop.Ki = gains[3][1];
  xLoop.Kd = gains[3][2];
```

```
yLoop.Kp = gains[4][0];
yLoop.Ki = gains[4][1];
yLoop.Kd = gains[4][2];

zLoop.Kp = gains[5][0];
zLoop.Ki = gains[5][1];
zLoop.Kd = gains[5][2];


// control thread states
double x = 0, y = 0, z = 0;
double vx = 0, vy = 0, vz = 0;
double phi = 0, theta = 0, psi = 0;
double p = 0, q = 0, r = 0;
// time stamp
struct timeval timestamp1;
// time between updates
double startControlTime, endControlTime, diffControlTime;
// calculate filter start time
FILE *fidc = fopen("control.log","w");
gettimeofday(&timestamp1, NULL);
startControlTime = timestamp1.tv_sec+(timestamp1.tv_usec/1000000.0);
while(1)
{
  //fprintf(stderr,"void* updateControl()");

  //Sort Vicon or State Data
  pthread_mutex_lock(&mut);
  pthread_mutex_lock(&viconMutex);

  x = viconState.x;
  vx = viconState.vx;

  y = viconState.y;
  vy = viconState.vy;
  if (zLoop.vicon)
  {
    z = viconState.z;
    vz = viconState.vz;
  }
  else
  {
    z = state.z;
    vz = state.vz;
  }

  if (rollLoop.vicon)
  {
    phi = viconState.phi;
    p = state.p;
  }
  else
  {
    phi = state.phi;
    p = state.p;
  }

  if (pitchLoop.vicon)
  {
```

```
  theta = viconState.theta;
  q = state.q;
}
else
{
  theta = state.theta;
  q = state.q;
}

if (yawLoop.vicon)
{
  psi = viconState.psi;
  r = state.r;
}
else
{
  psi = state.psi;
  r = state.r;
}
pthread_mutex_unlock(&viconMutex);
pthread_mutex_unlock(&mut);

// Update Guidance Loops
pthread_mutex_lock(&xLoopMutex);
pthread_mutex_lock(&yLoopMutex);
// Calculate Position Error in Vicon Frame
double xError_v = xLoop.reference - x;
double yError_v = yLoop.reference - y;
// Rotate Position Error Displacement Vector into Body Frame
double xError_b = xError_v * cos(psi) + yError_v * sin(psi);
double yError_b = xError_v * cos(psi) - yError_v * sin(psi);
// Rotate Velocity Vector into Body Frame
double vx_b = vx * cos(psi) + vy * sin(psi);
double vy_b = vy * cos(psi) - vy * sin(psi);
// Feed these errors to the PID Guidance Loops
// x Loop
updateGuidanceLoop(&xLoop,xError_b,x,vx_b);
if (xLoop.active)
{
  pitchLoop.reference = xLoop.output;
}
// y Loop
updateGuidanceLoop(&yLoop,yError_b,y,vy_b);
if (yLoop.active)
{
  rollLoop.reference = yLoop.output;
}
pthread_mutex_unlock(&yLoopMutex);
pthread_mutex_unlock(&xLoopMutex);

// Altitude Loop
pthread_mutex_lock(&zLoopMutex);
pthread_mutex_lock(&rcMutex);
// Assume the quad is horizontally level
double zError = zLoop.reference - z;

#define MAX_RATE zLoop.Ki
/*if (rcMode != MANUAL_DEBUG) // in ap mode and active
{
  if ((zError > 0) && (vz < MAX_RATE)) // needs to go up
```

```
      {
        zLoop.neutral += 1.0/60.0*0.001;
      }
      else if((zError < 0) && (vz > -MAX_RATE)) // needs to go down
      {
        zLoop.neutral -= 1.0/60.0*0.001;
      }
      printf("Accum = %lf",zLoop.neutral);
    }
    if (zLoop.neutral > 27)
    {
      zLoop.neutral = 27;
    }*/
    // add accumulator value to netural
    updateGuidanceLoop(&zLoop,zError,z,vz);
    pthread_mutex_unlock(&rcMutex);
    pthread_mutex_unlock(&zLoopMutex);

#ifdef _GYRO_
    // RC Gyro will do the mixing so roll, pitch and yaw loops will not be
active
    // Yaw Loop as Heading Hold
    pthread_mutex_lock(&yawLoopMutex);
    updateYawLoop(&yawLoop,phi,r);
    pthread_mutex_unlock(&yawLoopMutex);

    // Generate Commands for the MCU
    MutexLockAllLoops();
    pthread_mutex_lock(&apMutex);

    if (apMode == FAIL_SAFE)
    {
      apMode = FAIL_SAFE;
      //printf("CONTROL >> COMMANDED FAILSAFE");
    }
    else  if  (zLoop.active  &&  xLoop.active  &&  yLoop.active  &&
yawLoop.active)
    {
      apMode = RC_NONE;
    }
    else  if  (!zLoop.active  &&  xLoop.active  &&  yLoop.active  &&
yawLoop.active)
    {
      apMode = RC_THROTTLE;
    }
    else  if  (zLoop.active  &&  !yLoop.active  &&  xLoop.active  &&
yawLoop.active)
    {
      apMode = RC_ROLL;
    }
    else  if  (zLoop.active  &&  yLoop.active  &&  !xLoop.active  &&
yawLoop.active)
    {
      apMode = RC_PITCH;
    }
    else  if  (zLoop.active  &&  yLoop.active  &&  xLoop.active  &&
!yawLoop.active)
    {
      apMode = RC_YAW;
    }
    else   if   (!zLoop.active   &&   !yLoop.active   &&   xLoop.active   &&
yawLoop.active)
    {
      apMode = RC_THROTTLE_ROLL;
    }
    else   if   (!zLoop.active   &&   yLoop.active   &&   !xLoop.active   &&
yawLoop.active)
    {
      apMode = RC_THROTTLE_PITCH;
    }
    else   if   (!zLoop.active   &&   yLoop.active   &&   xLoop.active   &&
!yawLoop.active)
    {
      apMode = RC_THROTTLE_YAW;
    }
    else   if   (!zLoop.active   &&   !yLoop.active   &&   !xLoop.active   &&
yawLoop.active)
    {
      apMode = RC_THROTTLE_ROLL_PITCH;
    }
    else   if   (!zLoop.active   &&   !yLoop.active   &&   xLoop.active   &&
!yawLoop.active)
    {
      apMode = RC_THROTTLE_ROLL_YAW;
    }
    else   if   (!zLoop.active   &&   yLoop.active   &&   !xLoop.active   &&
!yawLoop.active)
    {
      apMode = RC_THROTTLE_PITCH_YAW;
    }
    else   if   (zLoop.active   &&   !yLoop.active   &&   !xLoop.active   &&
yawLoop.active)
    {
      apMode = RC_ROLL_PITCH;
    }
    else   if   (zLoop.active   &&   !yLoop.active   &&   xLoop.active   &&
!yawLoop.active)
    {
      apMode = RC_ROLL_YAW;
    }
    else   if   (zLoop.active   &&   yLoop.active   &&   !xLoop.active   &&
!yawLoop.active)
    {
      apMode = RC_PITCH_YAW;
    }
    else   if   (zLoop.active   &&   !yLoop.active   &&   !xLoop.active   &&
!yawLoop.active)
    {
      apMode = RC_ROLL_PITCH_YAW;
    }

    // Gyro will handle the inner loops
    apThrottle = zLoop.output;
    apRoll = yLoop.output;
    apPitch = xLoop.output;
    apYaw = yawLoop.output;

#else
    // Update Attitude Loops
    // No RC Gyro thus roll, pitch and yaw control loops in use
    pthread_mutex_lock(&rcMutex);

    double rcFactor = 1;
    if (rcMode == AUGMENTED) // use the gyro rates
    {
      // Roll Rate Control
      pthread_mutex_lock(&rollLoopMutex);
      rollLoop.reference = 2*8*rcFactor*rcRoll + zeroRoll;
      printf("%lf %d %lf",rollLoop.reference,rcRoll,zeroRoll);
      rollLoop.referenceDot = rollLoop.previousState; // Roll Loop D term
is now Kd*(pNew - pOld);
      // Scale Gyro Rates to 1000 to 2000 us based on +-300 deg/s IMU
      p = MapCommands(p*180.0/M_PI,300.0,-300.0,2000.0,1000.0);
      updateControlLoop(&rollLoop,p,p);
      // Scale Output to Timer Value
      rollLoop.output = PWMToCounter(rollLoop.output);
      pthread_mutex_unlock(&rollLoopMutex);

      // Pitch Rate Control
      pthread_mutex_lock(&pitchLoopMutex);
      pitchLoop.reference = -8*2*rcFactor*rcPitch + zeroPitch;
      printf("%lf %d %lf",pitchLoop.reference,rcPitch,zeroPitch);
      pitchLoop.referenceDot = pitchLoop.previousState; // Pitch Loop D
term is now Kd*(qNew - qOld);
      // Scale Gyro Rates to 1000 to 2000 us based on +-300 deg/s IMU
      q = MapCommands(q*180.0/M_PI,300.0,-300.0,2000.0,1000.0);
      updateControlLoop(&pitchLoop,q,q);
      // Scale Output to Timer Value
      pitchLoop.output = PWMToCounter(pitchLoop.output);
      pthread_mutex_unlock(&pitchLoopMutex);

    }
    else if (rcMode == AUTOPILOT) // use the filtered angles
    {
      pthread_mutex_lock(&rollLoopMutex);
      pthread_mutex_lock(&pitchLoopMutex);

      // TODO actually give these dedicated packets
      // at the moment these are coming from the angular loops
      levelRollLoop.Kp = rollLoop.maximum;
      levelRollLoop.Ki = rollLoop.minimum;
      levelPitchLoop.Kp = pitchLoop.maximum;
      levelPitchLoop.Ki = pitchLoop.minimum;

      // Level Control
      rollError = MapCommands(rcRoll,PWMToCounter(500),PWMToCounter(-
500),M_PI/4.0,-M_PI/4.0) - phi;
      pitchError = MapCommands(rcPitch,PWMToCounter(500),PWMToCounter(-
500),M_PI/4.0,-M_PI/4.0) + theta;
      levelAdjust[LEVEL_ROLL] = rollError*levelRollLoop.Kp;
      levelAdjust[LEVEL_PITCH] = pitchError*levelPitchLoop.Kp;
      // Check if the controller should try and return to hover
      if ((abs(rcRoll - rollTrim) > levelOff) || (abs(rcPitch - pitchTrim)
> levelOff))
      {
        levelRollLoop.integralError = 0.0;
        levelPitchLoop.integralError = 0.0;
      }
      else
      {
        levelRollLoop.integralError  +=  (rollError  *  diffControlTime)  *
levelRollLoop.Ki;
        if (abs(levelRollLoop.integralError) > levelLimit)
        {
          levelRollLoop.integralError = levelLimit;
        }

        levelPitchLoop.integralError  =  (pitchError  *  diffControlTime)  *
levelPitchLoop.Ki;
        if (abs(levelPitchLoop.integralError) > levelLimit)
        {
          levelPitchLoop.integralError = levelLimit;
        }
      }

      // Roll Control
      rollLoop.reference    =    2*8*rcFactor*rcRoll    +    zeroRoll    +
levelAdjust[LEVEL_ROLL];
      rollLoop.referenceDot = rollLoop.previousState; // Roll Loop D term
is now Kd*(pNew - pOld);
      // Scale Gyro Rates to 1000 to 2000 us based on +-300 deg/s IMU
      p = MapCommands(p*180.0/M_PI,300.0,-300.0,2000.0,1000.0);
      updateControlLoop(&rollLoop,p,p);
      // Scale Output to Timer Value
      rollLoop.output       =       PWMToCounter(rollLoop.output       +
levelRollLoop.integralError);

      // Pitch Control
      pitchLoop.reference   =   -2*8*rcFactor*rcPitch   +   zeroPitch   +
levelAdjust[LEVEL_PITCH];
      pitchLoop.referenceDot  =  pitchLoop.previousState;  //  Pitch  Loop  D
term is now Kd*(qNew - qOld);
      // Scale Gyro Rates to 1000 to 2000 us based on +-300 deg/s IMU
      q = MapCommands(q*180.0/M_PI,300.0,-300.0,2000.0,1000.0);
      updateControlLoop(&pitchLoop,q,q);
      // Scale Output to Timer Value
      pitchLoop.output       =       PWMToCounter(pitchLoop.output       +
levelPitchLoop.integralError);

      pthread_mutex_unlock(&pitchLoopMutex);
      pthread_mutex_unlock(&rollLoopMutex);
    }

    // Yaw Control
    pthread_mutex_lock(&yawLoopMutex);
    yawLoop.reference = 2*8*rcFactor*rcYaw + zeroYaw;
    yawLoop.referenceDot = yawLoop.previousState; // Yaw Loop D term is now
Kd*(rNew - rOld);
    // Scale Gyro Rates to 1000 to 2000 us based on +-300 deg/s IMU
    r = MapCommands(r*180.0/M_PI,300.0,-300.0,2000.0,1000.0);
    updateControlLoop(&yawLoop,r,r);
    // Scale Output to Timer Value
    yawLoop.output = PWMToCounter(yawLoop.output);
    pthread_mutex_unlock(&yawLoopMutex);

    pthread_mutex_unlock(&rcMutex);

    // Generate Commands for the MCU
    MutexLockAllLoops();
    pthread_mutex_lock(&apMutex);
    if (apMode == FAIL_SAFE)
    {
      apMode = FAIL_SAFE;
      //printf("CONTROL >> COMMANDED FAILSAFE");
    }
    else if (zLoop.active && xLoop.active && yLoop.active &&
yawLoop.active)
    {
      apMode = RC_NONE;
    }
    else if (!zLoop.active && rollLoop.active && pitchLoop.active &&
yawLoop.active)
    {
      apMode = RC_THROTTLE;
    }
    else if (zLoop.active && !rollLoop.active && pitchLoop.active &&
yawLoop.active)
    {
      apMode = RC_ROLL;
    }
    else if (zLoop.active && rollLoop.active && !pitchLoop.active &&
yawLoop.active)
    {
      apMode = RC_PITCH;
    }
    else if (zLoop.active && rollLoop.active && pitchLoop.active &&
!yawLoop.active)
    {
      apMode = RC_YAW;
    }
    else if (!zLoop.active && !rollLoop.active && pitchLoop.active &&
yawLoop.active)
    {
      apMode = RC_THROTTLE_ROLL;
    }
    else if (!zLoop.active && rollLoop.active && !pitchLoop.active &&
yawLoop.active)
    {
      apMode = RC_THROTTLE_PITCH;
    }
    else if (!zLoop.active && rollLoop.active && pitchLoop.active &&
!yawLoop.active)
    {
      apMode = RC_THROTTLE_YAW;
    }
    else if (!zLoop.active && !rollLoop.active && !pitchLoop.active &&
yawLoop.active)
    {
      apMode = RC_THROTTLE_ROLL_PITCH;
    }
    else if (!zLoop.active && !rollLoop.active && pitchLoop.active &&
!yawLoop.active)
    {
      apMode = RC_THROTTLE_ROLL_YAW;
    }
    else if (!zLoop.active && rollLoop.active && !pitchLoop.active &&
!yawLoop.active)
    {
      apMode = RC_THROTTLE_PITCH_YAW;
    }
    else  if  (zLoop.active  &&  !rollLoop.active  &&  !pitchLoop.active  &&
yawLoop.active)
    {
      apMode = RC_ROLL_PITCH;
    }
    else  if  (zLoop.active  &&  !rollLoop.active  &&  pitchLoop.active  &&
!yawLoop.active)
    {
      apMode = RC_ROLL_YAW;
    }
    else  if  (zLoop.active  &&  rollLoop.active  &&  !pitchLoop.active  &&
!yawLoop.active)
    {
      apMode = RC_PITCH_YAW;
    }
    else  if  (zLoop.active  &&  !rollLoop.active  &&  !pitchLoop.active  &&
!yawLoop.active)
    {
      apMode = RC_ROLL_PITCH_YAW;
    }

    // AP Handles the Inner Loop
    apThrottle = zLoop.output;
    apRoll = rollLoop.output;
    apPitch = pitchLoop.output;
    apYaw = yawLoop.output;
#endif
    pthread_mutex_unlock(&apMutex);

    // Update AP State
    pthread_mutex_lock(&apStateMut);
    apState.referencePhi = rollLoop.reference;
    apState.referenceTheta = pitchLoop.reference;
    apState.referencePsi = yawLoop.reference;
    apState.referenceX = xLoop.reference;
    apState.referenceY = yLoop.reference;
    apState.referenceZ = zLoop.reference;

    apState.phiActive = rollLoop.active;
    apState.thetaActive = pitchLoop.active;
    apState.psiActive = yawLoop.active;
    apState.xActive = xLoop.active;
    apState.yActive = yLoop.active;
    apState.zActive = zLoop.active;
    pthread_mutex_unlock(&apStateMut);

    MutexUnlockAllLoops();
    // calculate the control thread update time
    gettimeofday(&timestamp1, NULL);
    endControlTime = timestamp1.tv_sec+(timestamp1.tv_usec/1000000.0);
    diffControlTime = endControlTime - startControlTime;
    startControlTime = timestamp1.tv_sec+(timestamp1.tv_usec/1000000.0);
    fprintf(fidc,"%lf\n",1/diffControlTime);

    usleep(CONTROL_DELAY*1e3);
  }
  return NULL;
}
// end main.c
```

Appendix F – Openembedded flight computer receipe

Source file: `julia-fc-image.bb`

```
# console image for omap3

inherit image

DEPENDS = "task-base"

IMAGE_EXTRA_INSTALL ?= ""

BASE_INSTALL = " \
  task-base-extended \
 "

FIRMWARE_INSTALL = " \
#  linux-firmware \
  libertas-sd-firmware \
  rt73-firmware \
  zd1211-firmware \
 "

GLES_INSTALL = " \
#  libgles-omap3 \
 "

TOOLS_INSTALL = " \
  bash \
  bzip2 \
  ckermit \
  devmem2 \
  dhcp-client \
  dosfstools \
  fbgrab \
  fbset \
  fbset-modes \
  grep \
  gsl-dev \
  i2c-tools \
  ksymoops \
  mkfs-jffs2 \
  mtd-utils \
  nano \
  ntp ntpdate \
  openssh-misc \
  openssh-scp \
  openssh-ssh \
  omap3-writeprom \
  procps \
  socat \
  strace \
  subversion \
  sudo \
  syslog-ng \
  task-native-sdk \
  task-proper-tools \
  vim \
 "

IMAGE_INSTALL += " \
  ${BASE_INSTALL} \
  ${FIRMWARE_INSTALL} \
  ${GLES_INSTALL} \
  ${IMAGE_EXTRA_INSTALL} \
  ${TOOLS_INSTALL} \
 "

IMAGE_PREPROCESS_COMMAND = "create_etc_timestamp"

#ROOTFS_POSTPROCESS_COMMAND      +=      '${@base_conditional("DISTRO_TYPE",
"release", "zap_root_password; ", "",d)}'

# end julia-fc-image.bb
```

Appendix G – UART1 pass-through

Source file: `uart1.patch`

```
diff --git a/board/overo/overo.h b/board/overo/overo.h
index 576fc73..f47ba34 100644
--- a/board/overo/overo.h
+++ b/board/overo/overo.h
@@ -222,14 +222,14 @@ const omap3_sysinfo sysinfo = {
      MUX_VAL(CP(MMC2_DAT6),        (IEN     |    PTU    |   EN      |   M1))
/*MMC2_DIR_CMD*/\
      MUX_VAL(CP(MMC2_DAT7),        (IEN   | PTU | EN   | M4)) /*GPIO_139*/\
  /*Bluetooth*/\
-     MUX_VAL(CP(MCBSP3_DX),        (IEN   | PTD | DIS | M1)) /*UART2_CTS*/\
-     MUX_VAL(CP(MCBSP3_DR),        (IDIS | PTD | DIS | M1)) /*UART2_RTS*/\
-     MUX_VAL(CP(MCBSP3_CLKX),      (IDIS | PTD | DIS | M1)) /*UART2_TX*/\
-     MUX_VAL(CP(MCBSP3_FSX),       (IEN   | PTD | DIS | M1)) /*UART2_RX*/\
+     MUX_VAL(CP(MCBSP3_DX),        (IDIS | PTD | DIS | M0)) /*McBSP3_DX*/\
+     MUX_VAL(CP(MCBSP3_DR),        (IDIS | PTD | DIS | M0)) /*McBSP3_DR*/\
+     MUX_VAL(CP(MCBSP3_CLKX),      (IDIS    |    PTD    |    DIS    |    M0))
/*McBSP3_CLKX*/\
+     MUX_VAL(CP(MCBSP3_FSX),       (IDIS    |    PTD    |    DIS    |    M0))
/*McBSP3_FSX*/\
      MUX_VAL(CP(UART2_CTS),        (IEN   | PTD  | DIS  | M4))  /*GPIO_144 -
LCD_EN*/\
      MUX_VAL(CP(UART2_RTS),        (IEN   | PTD | DIS | M4)) /*GPIO_145*/\
-     MUX_VAL(CP(UART2_TX),         (IEN   | PTD | DIS | M4)) /*GPIO_146*/\
-     MUX_VAL(CP(UART2_RX),         (IEN   | PTD | DIS | M4)) /*GPIO_147*/\
+     MUX_VAL(CP(UART2_TX),         (IDIS | PTD | DIS | M0)) /*UART2_TX*/\
+     MUX_VAL(CP(UART2_RX),         (IEN   | PTD | DIS | M0)) /*UART2_RX*/\
      MUX_VAL(CP(UART1_TX),         (IDIS | PTD | DIS | M0)) /*UART1_TX*/\
      MUX_VAL(CP(UART1_RTS),        (IEN   | PTU | DIS | M4)) /*GPIO_149*/ \
      MUX_VAL(CP(UART1_CTS),        (IEN   | PTU  | DIS  | M4))  /*GPIO_150-
MMC3_WP*/\
```

Appendix H – State Estimation implementation

Header file: `kfb.h`

```
/**
 * @file   kfb.h
 * @author Liam O'Sullivan
 *
 * $Author: liamosullivan $
 * $Date: 2010-08-28 11:26:11 +1000 (Sat, 28 Aug 2010) $
 * $Rev: 324 $
 * $Id: kfc.h 324 2010-08-28 01:26:11Z liamosullivan $
 *
 * Queensland University of Technology
 *
 * @section DESCRIPTION
 * Kalman filter library (revision b)
 *
 */

#ifndef KFB_H
#define KFB_H

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>

// Kalman filter constants for phi (Q1=0.001,Q2=0.003)
#define PHI_ANGLE_Q 0.057296
#define PHI_GYRO_Q 0.171887
#define PHI_R 1.7
#define PHI_DIRECTION 1.0

// Kalman filter constants for theta
#define THETA_ANGLE_Q 0.057296
#define THETA_GYRO_Q 0.171887
#define THETA_R 1.7
#define THETA_DIRECTION -1.0

// Kalman filter constants for psi
#define PSI_ANGLE_Q 0.057296
#define PSI_GYRO_Q 0.171887
#define PSI_R 5.0
#define PSI_DIRECTION -1.0

// Euler angle calibration cycles
#define CYCLES 1000

// LPF alpha constants for accelerometers
#define ACCX_ALPHA 1.0
#define ACCY_ALPHA 1.0
#define ACCZ_ALPHA 1.0

// LPF alpha constants for gyro rates
#define RATEX_ALPHA 1.0
#define RATEY_ALPHA 1.0
#define RATEZ_ALPHA 1.0
```

```
// LPF alpha constant for compass reading
#define COMPASS_ALPHA 0.1

// LPF alpha constant for ultrasonic sensor
#define ALT_ALPHA 0.07

// Data logger flag
#define DATA_LOGGER 0

// struct definition for each axis
typedef struct _axis {
  // x state (tracking angle and gyro bias, 1x2)
  double X[2];
  // p covariance matrix (2x2)
  double P[2][2];
  // s term
  double S;
  // y measurement term
  double Y;
  // error term
  double err;
  // L gain matrix (1x2)
  double L[2];
  // Q process noise term (variance for angle and gyro bias, 1x2)
  double Q[2];
  // R measurement noise term
  double R;
  // direction of axis
  double direction;
  // angle offset
  double offset;
} axis;

// function definitions
int attitudeFilterInitialiseB(double *accXr, double *accYr, double *accZr);
int attitudeFilterB(double *rateXr, double *rateYr, double *rateZr, double
*accXr, double *accYr, double *accZr, double *rateXf, double *rateYf,
double *rateZf, double *phif, double *thetaf, double *psif, double
*compassZr, double dT);
int kFilterTimeUpdate(axis *axis_t, double *gyroRate, double dT);
int kFilterMeasureUpdate (axis *axis_t);
int kFilterMeasurePsiUpdate (axis *axis_t);
double coarsePitchAngle(double *accXr, double *accYr, double *accZr);
double coarseRollAngle(double *accXr, double *accYr, double *accZr);
int accLPF (double *accXr, double *accYr, double *accZr, double dT);
int rateLPF(double *rateXf, double *rateYf, double *rateZr);
int compassLPF(double *compass_heading);
int altLPF(double *altitude);
int calibrateEulerAngles(double *phif, double *thetaf, double *psif);
int printkFilterData(double *rateXr, double *rateYr, double *rateZr, double
*accXr, double *accYr, double *accZr, double dT);

#endif

// end kfb.h
```

Source file: kfb.c

```
/**
 * @file   kfb.c
 * @author Liam O'Sullivan
 *
 * $Author: liamosullivan $
 * $Date: 2010-08-28 11:26:11 +1000 (Sat, 28 Aug 2010) $
 * $Rev: 324 $
 * $Id: kfb.c 324 2010-08-28 01:26:11Z liamosullivan $
 *
 * Queensland University of Technology
 *
 * @section DESCRIPTION
 * Kalman filter library (revision b) which implements:
 * - Initialising the Kalman filter
 * - Performs the attitude filtering
 * - Low pass filtering for read IMU data
 */

#include "kfb.h"

// phi axis struct
axis phi_axis;
// theta axis struct
axis theta_axis;
// psi axis struct
axis psi_axis;
// accelerometer value storage (used for filtering)
double acc_previous[3];
// filtered accelerometer readings (from the raw sensor data)
double accXf = 0.0;
double accYf = 0.0;
double accZf = 0.0;
// gyro value storage (used for filtering)
double rate_previous[3];
double rate_current[3];
// compass heading value storage (used for filtering)
double compass_heading_previous = 0.0;
double compass_heading_current = 0.0;
// altitude value storage (used for filtering)
double altitude_previous = 0.0;
double altitude_current = 0.0;
// euler angle storage (calculation of filtered rate)
double angle_previous[3];
// calibration flag
int calib = 1;
// calibration cycle counter
int cyc_count = 0;
// phi and theta angle storage for calibration
double phi_sum = 0.0;
double theta_sum = 0.0;

double psi_angle = 0.0;

// function to initialise the values in the axis structures
int attitudeFilterInitialiseB(double *accXr, double *accYr, double *accZr)
{
  /* phi axis */
  // initialise phi angle to coarse roll angle
```

```
phi_axis.X[0] = coarseRollAngle(accXr,accYr,accZr);
// no initial bias
phi_axis.X[1] = 0.0;
// initialise covariance matrix to 1
phi_axis.P[0][0] = 1.0;
phi_axis.P[0][1] = 1.0;
phi_axis.P[1][0] = 1.0;
phi_axis.P[1][1] = 1.0;
// initialise S term
phi_axis.S = 0.0;
// initialise measurement term
phi_axis.Y = phi_axis.X[0];
// initialise correction term
phi_axis.err = 0.0;
// initialise the kalman gain
phi_axis.L[0] = 0.0;
phi_axis.L[1] = 0.0;
// initialise the Q terms
phi_axis.Q[0] = PHI_ANGLE_Q;
phi_axis.Q[1] = PHI_GYRO_Q;
// initialise the R term
phi_axis.R = PHI_R;
// initialise the direction term
phi_axis.direction = PHI_DIRECTION;
// initialise the offset term
phi_axis.offset = 0.0;


/* theta axis */
// initialise theta angle to coarse pitch angle
theta_axis.X[0] = coarsePitchAngle(accXr,accYr,accZr);
// no initial bias
theta_axis.X[1] = 0.0;
// initialise covariance matrix to 1
theta_axis.P[0][0] = 1.0;
theta_axis.P[0][1] = 1.0;
theta_axis.P[1][0] = 1.0;
theta_axis.P[1][1] = 1.0;
// initialise S term
theta_axis.S = 0.0;
// initialise measurement term
theta_axis.Y = theta_axis.X[0];
// initialise correction term
theta_axis.err = 0.0;
// initialise the kalman gain
theta_axis.L[0] = 0.0;
theta_axis.L[1] = 0.0;
// initialise the Q terms
theta_axis.Q[0] = THETA_ANGLE_Q;
theta_axis.Q[1] = THETA_GYRO_Q;
// initialise the R term
theta_axis.R = THETA_R;
// initialise the direction term
theta_axis.direction = THETA_DIRECTION;
// initialise the offset term
theta_axis.offset = 0.0;

/* psi axis */
// initialise psi angle to compass
psi_axis.X[0] = 0.0;
```

```
  // no initial bias
  psi_axis.X[1] = 0.0;
  // initialise covariance matrix to 1
  psi_axis.P[0][0] = 1.0;
  psi_axis.P[0][1] = 1.0;
  psi_axis.P[1][0] = 1.0;
  psi_axis.P[1][1] = 1.0;
  // initialise S term
  psi_axis.S = 0.0;
  // initialise measurement term
  psi_axis.Y = 0.0;
  // initialise correction term
  psi_axis.err = 0.0;
  // initialise the kalman gain
  psi_axis.L[0] = 0.0;
  psi_axis.L[1] = 0.0;
  // initialise the Q terms
  psi_axis.Q[0] = PSI_ANGLE_Q;
  psi_axis.Q[1] = PSI_GYRO_Q;
  // initialise the R term
  psi_axis.R = PSI_R;
  // initialise the direction term
  psi_axis.direction = PSI_DIRECTION;
  // initialise the offset term
  psi_axis.offset = 0.0;


  /* accelerometer previous values */
  acc_previous[0] = *accXr;
  acc_previous[1] = *accYr;
  acc_previous[2] = *accZr;

  /* gyro previous values */
  rate_previous[0] = 0.0;
  rate_previous[1] = 0.0;
  rate_previous[2] = 0.0;

  /* euler angle previous values */
  angle_previous[0] = phi_axis.X[0]*M_PI/180;
  angle_previous[1] = theta_axis.X[0]*M_PI/180;

  // initialise the text file for data logging
  if(!calib)
  {
    FILE *kfilterfd = fopen("kfilter.ahnskfilter","a");
    if(kfilterfd)
    {
      // print header information
      fprintf(kfilterfd,"### Kalman Filter data ###\n");
      fclose(kfilterfd);
    }
  }
  return 1;
}
int attitudeFilterB(double *rateXr, double *rateYr, double *rateZr, double
*accXr, double *accYr, double *accZr, double *rateXf, double *rateYf,
double *rateZf, double *phif, double *thetaf, double *psif, double
*compassZr, double dT)
{
  // LPF the accelerometer values
  accLPF(accXr,accYr,accZr,dT);
  //*rateZf = (*rateZr)*M_PI/180;
  //rateLPF(rateXr,rateYr,rateZr,rateXf,rateYf,rateZf);
  // Bound and LPF the compass value
  compassLPF(compassZr);
  // time update for phi axis
  kFilterTimeUpdate(&phi_axis,rateXr,dT);
  // time update for theta axis
  kFilterTimeUpdate(&theta_axis,rateYr,dT);
  // time update for psi axis
  kFilterTimeUpdate(&psi_axis,rateZr,dT);
  // calculate the coarse angle for phi from the sensors
  phi_axis.Y = coarseRollAngle(&accXf,&accYf,&accZf);
  // measurement update for phi axis
  kFilterMeasureUpdate(&phi_axis);
  // calculate the coarse angle for theta from the sensors
  theta_axis.Y = coarsePitchAngle(&accXf,&accYf,&accZf);
  // measurement update for phi axis
  kFilterMeasureUpdate(&theta_axis);
  // LPF the compass values
  compassLPF(compassZr);
  // allocate measurement angle for the psi axis
  psi_axis.Y = *compassZr;
  // measurement update for psi axis
  kFilterMeasurePsiUpdate(&psi_axis);
  // assign new phi angle (radians)
  *phif = (phi_axis.X[0]-phi_axis.offset)*M_PI/180;
  // assign new theta angle (radians)
  *thetaf = (theta_axis.X[0]-theta_axis.offset)*M_PI/180;
  // assign new psi angle (radians)
  *psif = fmod(((psi_axis.X[0]-psi_axis.offset)*M_PI/180),2*M_PI);
  // assign new phi rate
  //*rateXf = (*phif - angle_previous[0])/dT;
  // assign new theta rate
  //*rateYf = (*thetaf - angle_previous[1])/dT;
  // LPF the rate values (raw rate for the psi rate)
  //*rateZf = (*rateZr)*M_PI/180;
  // LPF the rate values (all raw)
  //*rateXf = (*rateXr)*M_PI/180;
  //*rateYf = (*rateYr)*M_PI/180;
  //*rateZf = (*rateZr)*M_PI/180;
  //rateLPF(rateXf,rateYf,rateZf);
  // update the previous angles
  angle_previous[0] = *phif;
  angle_previous[1] = *thetaf;
  // check if calibration is taking place
  if(!calib)
  {
    calibrateEulerAngles(&phi_axis.X[0], &theta_axis.X[0], psif);
  }
  // write all filtered data out to a file
  if(DATA_LOGGER)
  {
    printkFilterData(rateXr,rateYr,rateZr,accXr,accYr,accZr,dT);
  }
  return 1;
}

// Kalman filter time update for axis
int kFilterTimeUpdate(axis *axis_t, double *gyroRate, double dT)
```

```
{
  // compute the axis angle by unbiasing the gyro reading and integrating
  axis_t->X[0] += ((*gyroRate*axis_t->direction) - axis_t->X[1])*dT;
  // calculate the covariance matrix and integrate (discrete)
  axis_t->P[0][0] += (axis_t->Q[0] - axis_t->P[0][1] - axis_t->P[1][0])*dT;
  axis_t->P[0][0] += axis_t->Q[0] - (axis_t->P[0][1] + axis_t->P[1][0])*dT;
  axis_t->P[0][1] += -axis_t->P[1][1]*dT;
  axis_t->P[1][0] += -axis_t->P[1][1]*dT;
  axis_t->P[1][1] += axis_t->Q[1]*dT;
  axis_t->P[1][1] += axis_t->Q[1];
  // end time update
  return 1;
}

//  Kalman  filter  measurement  update  for  axis  (assume  coarse  angle  is
calculated)
int kFilterMeasureUpdate (axis *axis_t)
{
  // compute the error term
  axis_t->S = axis_t->P[0][0] + axis_t->R;
  // calculate the Kalman gain
  axis_t->L[0] = axis_t->P[0][0] / axis_t->S;
  axis_t->L[1] = axis_t->P[1][0] / axis_t->S;
  // update the covariance matrix (careful of computation order)
  axis_t->P[1][0] -= axis_t->L[1]*axis_t->P[0][0];
  axis_t->P[1][1] -= axis_t->L[1]*axis_t->P[0][1];
  axis_t->P[0][0] -= axis_t->L[0]*axis_t->P[0][0];
  axis_t->P[0][1] -= axis_t->L[0]*axis_t->P[0][1];
  // calculate the difference between prediction and measurement
  axis_t->err = axis_t->Y - axis_t->X[0];
  // update the x states (angle and gyro bias)
  axis_t->X[0]+= axis_t->err*axis_t->L[0];
  axis_t->X[1]+= axis_t->err*axis_t->L[1];
  // end measurement update
  return 1;
}

// Kalman filter measurement update for psi axis (boundary checking)
int kFilterMeasurePsiUpdate (axis *axis_t)
{
  // compute the error term
  axis_t->S = axis_t->P[0][0] + axis_t->R;
  // calculate the Kalman gain
  axis_t->L[0] = axis_t->P[0][0] / axis_t->S;
  axis_t->L[1] = axis_t->P[1][0] / axis_t->S;
  // update the covariance matrix (careful of computation order)
  axis_t->P[1][0] -= axis_t->L[1]*axis_t->P[0][0];
  axis_t->P[1][1] -= axis_t->L[1]*axis_t->P[0][1];
  axis_t->P[0][0] -= axis_t->L[0]*axis_t->P[0][0];
  axis_t->P[0][1] -= axis_t->L[0]*axis_t->P[0][1];
  //  calculate  the  difference  between  prediction  and  measurement  (perform
psi quadrant checking)
  if  ((axis_t->Y - axis_t->X[0])>180.0)  // ie the measurement is in < 360
and predict is above 0
  {
    axis_t->err = axis_t->Y - (360.0+axis_t->X[0]);
  } else if ((axis_t->Y - axis_t->X[0])<(-180.0)) // ie the measurement is
above 0 and predict is < 360
          {
            axis_t->err = axis_t->Y + (360.0-axis_t->X[0]);
```

```
        } else { // normal update
                 axis_t->err = axis_t->Y - axis_t->X[0];
               }
  // update the x states (angle and gyro bias)
  axis_t->X[0] = fmod((axis_t->X[0] + (axis_t->err*axis_t->L[0])),360.0);
  axis_t->X[1] += axis_t->err*axis_t->L[1];
  // handle negative sign
  if (axis_t->X[0] < 0.0)
  {
    axis_t->X[0] += 360.0;
  }
  // end measurement update
  return 1;
}

// function to return the coarse pitch angle based on the accelerometer
values (degrees)
double coarsePitchAngle(double *accXr, double *accYr, double *accZr)
{
  return atan2(*accXr,sqrt(*accYr * *accYr + *accZr * *accZr))*180/M_PI;
}

// function to return the coarse roll angle based on the accelerometer
values (degrees)
double coarseRollAngle(double *accXr, double *accYr, double *accZr)
{
  return atan2(*accYr,sqrt(*accXr * *accXr + *accZr * *accZr))*180/M_PI;
}

// function to low pass filter the accelerometer values
int accLPF (double *accXr, double *accYr, double *accZr, double dT)
{
  // LPF the X accelerometer value
  accXf = acc_previous[0]*(1-ACCX_ALPHA) + (*accXr * ACCX_ALPHA);
  // LPF the Y accelerometer value
  accYf = acc_previous[1]*(1-ACCY_ALPHA) + (*accYr * ACCY_ALPHA);
  // LPF the Z accelerometer value
  accZf = acc_previous[2]*(1-ACCZ_ALPHA) + (*accZr * ACCZ_ALPHA);
  // store the filtered values
  acc_previous[0] = accXf;
  acc_previous[1] = accYf;
  acc_previous[2] = accZf;
  return 1;
}

// function to low pass filter the rates values
int rateLPF(double *rateXr, double *rateYr, double *rateZr, double *rateXf,
double *rateYf, double *rateZf)
{
  rate_current[0] = *rateXr;
  rate_current[1] = *rateYr;
  rate_current[2] = *rateZr;
  // LPF the phi rate
  *rateXf   =   rate_previous[0]*(1-RATEX_ALPHA)   +   (rate_current[0]   *
RATEX_ALPHA);
  // LPF the theta rate
  *rateYf   =   rate_previous[1]*(1-RATEY_ALPHA)   +   (rate_current[1]   *
RATEY_ALPHA);
  // LPF the psi rate
  *rateZf   =   rate_previous[2]*(1-RATEZ_ALPHA)   +   (rate_current[2]   *
RATEZ_ALPHA);
  // save the new rates
  rate_previous[0] = *rateXf;
  rate_previous[1] = *rateYf;
  rate_previous[2] = *rateZf;
  return 1;
}

//  function  to  calibrate  the  euler  angles  obtained  from  the  IMU  by
computing an offset
int calibrateEulerAngles(double *phif, double *thetaf, double *psif)
{
  // calculate offset sum
  phi_sum += *phif;
  theta_sum += *thetaf;
  cyc_count++;
  // check if calibration has been completed
  if (cyc_count >= CYCLES)
  {
    // calibration complete, compute the mean for each axis
    phi_axis.offset = phi_sum/CYCLES;
    theta_axis.offset = theta_sum/CYCLES;
    // change calibration flag
    calib = 1;
    printf("Phi: %lf\n",phi_axis.offset);
    printf("Theta: %lf\n",theta_axis.offset);
  }
  return 1;
}

// function to print the kalman filter data
int printkFilterData(double *rateXr, double *rateYr, double *rateZr, double
*accXr, double *accYr, double *accZr, double dT)
{
  FILE *kfilterfd = fopen("kfilter.ahnskfilter","a");
  if(kfilterfd)
  {
    // print the time stamp
    fprintf(kfilterfd,"%lf,",dT);
    // print the raw gyroscope data
    fprintf(kfilterfd,"%lf,%lf,%lf,",*rateXr,*rateYr,*rateZr);
    // print the raw accelerometer data
    fprintf(kfilterfd,"%lf,%lf,%lf,",*accXr,*accYr,*accZr);
    // print phi axis data (angle,bias,measurement,offset)

fprintf(kfilterfd,"%lf,%lf,%lf,%lf,",phi_axis.X[0],phi_axis.X[1],phi_axis.Y
,phi_axis.offset);
    // print theta axis data (angle,bias,measurement,offset)

fprintf(kfilterfd,"%lf,%lf,%lf,%lf,",theta_axis.X[0],theta_axis.X[1],theta_
axis.Y,theta_axis.offset);
    // print psi axis data (angle,bias,measurement,offset)

fprintf(kfilterfd,"%lf,%lf,%lf,%lf\n",psi_axis.X[0],psi_axis.X[1],psi_axis.
Y,psi_axis.offset);
    // all data saved, close the file
    fclose(kfilterfd);
  }
  return 1;
}
```

```
int compassLPF(double *compass_heading)
{
  compass_heading_current = *compass_heading;
  if(*compass_heading > 360.0)
  {
    // bound the compass heading by applying previous reading
    *compass_heading = compass_heading_previous;
  } else
    {
      // reading ok, LPF the value
      *compass_heading   =   compass_heading_previous*(1-COMPASS_ALPHA)   +
(*compass_heading * COMPASS_ALPHA);
      compass_heading_previous = *compass_heading;
    }
  return 1;
}

int altLPF(double *altitude)
{
  altitude_current = *altitude;
  *altitude = altitude_previous*(1-ALT_ALPHA) + (*altitude * ALT_ALPHA);
  altitude_previous = *altitude;
  return 1;
}

// end kfb.c
```