Title: , Autonomous Helicopter Navigation System, Flight Computer Design Document

*“A Project”*

Prepared by Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

,

Checked by Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

,

Approved by Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

, Student Manager 2010

Authorised for use by Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Dr Luis Mejias, Project Coordinator

**QUT Avionics**

Queensland University of Technology

CRCSS-EESE, GPO Box 2434

Gardens Point Campus

Brisbane, Australia, 4001.

Telephone (+61 7) 3864 1772

Facsimile (+61 7) 3864 1517

e-mail luis.mejias@qut.edu.au

web <http://code.google.com/p/ahns10/>

This document is Copyright 2010 by the QUT. The content of this document, except that information which is in the public domain, is the proprietary property of the QUT and shall not be disclosed or reproduced in part or in whole other than for the purpose for which it has been prepared without the express permission of the QUT

**Revision Record**

|  |  |  |  |
| --- | --- | --- | --- |
| Document Issue/Revision Status | **Description of Change** | **Date** | **Approved** |
|  | Initial Issue |  |  |

**Distribution List**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Affiliation** | **Distribution Date** | **Approved** |
| Avionics Lab File Archive | QUT Avionics |  |  |

**Foreword**

The design of the flight computer is necessary to facilitate autonomous flight testing to occur. The flight computer will collect data from 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

The collection of this data involved numerous software libraries to be developed in C programming code. The functions of these libraries will be executed in the flight computer program which is organised through the use of threading. Threading is required for the flight computer since concurrent processes need to be run at the same time to achieve autonomous flight.

**Table of Contents**

Paragraph Page No.

[1 Introduction 8](#_Toc275460777)

[1.1 Scope 8](#_Toc275460778)

[1.2 Background 8](#_Toc275460779)

[2 Reference Documents 9](#_Toc275460780)

[2.1 QUT Avionics Documents 9](#_Toc275460781)

[2.2 Non-QUT Documents 9](#_Toc275460782)

[3 Hardware architecture 10](#_Toc275460783)

[4 IMU 11](#_Toc275460784)

[4.1 IMU hardware interface 11](#_Toc275460785)

[4.2 IMU architecture 11](#_Toc275460786)

[4.3 IMU CRB access 13](#_Toc275460787)

[4.4 IMU sensor data access 13](#_Toc275460788)

[4.5 IMU serial data output 14](#_Toc275460789)

[4.6 IMU serial sensor data conversion 15](#_Toc275460790)

[4.7 IMU library functions 15](#_Toc275460791)

[5 Magnetic compass 17](#_Toc275460792)

[5.1 Magnetic compass hardware interface 17](#_Toc275460793)

[5.2 Magnetic compass output sensor format 17](#_Toc275460794)

[5.3 Magnetic compass calibration 18](#_Toc275460795)

[5.4 Magnetic compass Arduino functions 18](#_Toc275460796)

[6 Altitude sensor and battery voltage sensor 19](#_Toc275460797)

[6.1 Altitude sensor hardware interface 19](#_Toc275460798)

[6.2 Altitude sensor filtering 19](#_Toc275460799)

[6.3 Battery voltage sensor hardware interface 19](#_Toc275460800)

[6.4 Battery voltage filtering 19](#_Toc275460801)

[6.5 Altitude and battery voltage library Arduino functions 19](#_Toc275460802)

[7 Arduino 21](#_Toc275460803)

[7.1 Arduino sensors and hardware interface 21](#_Toc275460804)

[7.2 Arduino sensor data format 21](#_Toc275460805)

[7.3 Arduino functions 21](#_Toc275460806)

[7.4 Arduino library functions 22](#_Toc275460807)

[8 Overo Fire 23](#_Toc275460808)

[8.1 Flight computer structure 23](#_Toc275460809)

[8.2 Flight computer functions 23](#_Toc275460810)

[8.3 Operating system configuration 24](#_Toc275460811)

[8.4 Operating system installation 25](#_Toc275460812)

[8.5 UART1 pass through 25](#_Toc275460813)

[9 Conclusions 27](#_Toc275460814)

[10 Recommendations 28](#_Toc275460815)

**List of Figures**

Figure Page No.

[Figure 3.1 - Hardware architecture (onboard) 10](#_Toc275458148)

[Figure 8.1 - Flight computer thread structure 23](#_Toc275458149)

**List of Tables**

Table Page No.

[Table 1 - MVRB contents [RD/4] 12](#_Toc275458150)

[Table 2 - Important CRB registers [RD/4] 13](#_Toc275458151)

**Definitions**

|  |  |
| --- | --- |
| AHNS | Autonomous Helicopter Navigation System |
| QUT | Queensland University of Technology |
| HLO | High Level Objective |
| SR | System Requirement |
| GCS | Ground Control Station |
| IMU | Inertial Measurement Unit |
| MCU | Mode Control Unit |
| MVRB | Measured Value Register Bank |
| CRB | Control Register Bank |
| ADC | Analog to Digital Conversion |
| UART | Universal Asynchronous Receiver Transmitter |
| SPI | Serial Peripheral Interface |
| OMAP | Open Multimedia Application Platform |
|  |  |

# Introduction

A flight computer implementation needs to be designed to deliver all airborne HLOs [RD/1] and their system requirements. This is seen in the following 2 requirements [RD/2]:

1. 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.
2. SR-D-06: The airborne system shall collect avionics system health monitoring information in the form of radio control link status, flight mode status and battery level.

Thus the implementation needs to be able to connect the Overo Fire (which is running the flight computer) to the IMU, camera, ultrasonic sensor, magnetic compass and mode control unit (MCU). The implementation also has to receive and process the data gathered from these sensors concurrently or when they are next available.

## Scope

The code design implementation considered in this document focuses solely on the Overo Fire and its connection to other devices and sensors. It also includes coding implementation for the Arduino but does not include any mode control unit (MCU) implementation details.

## Background

The flight computer will be implemented on the Overo Fire. The Overo Fire is a Gumstix based board that runs a TI OMAP 3530 ARM based processor. Thus all code that needs to be executed on this board will be required to be cross complied. This board also features a 720MHz clock, Bluetooth module and Wi-Fi connectivity. More information on the Overo Fire can be found in [RD/3].

# Reference Documents

## QUT Avionics Documents

|  |  |  |
| --- | --- | --- |
| RD/1 | AHNS-2010-SY-HL-001 | AHNS, High Level Objectives of |
| RD/2 | AHNS-2010-SY-SR-001 | AHNS, System Requirements of |

## Non-QUT Documents

|  |  |  |
| --- | --- | --- |
| RD/3 | 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/4 | 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/5 | 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/6 | 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/7 | 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). |

In the event of any conflict between this document and any RD referenced herein, such conflict shall be notified to Dr Luis Mejias.

In the following text, RD/x identifies referenced documents, where "x" denotes the actual document.

# Hardware architecture

The quadrotor platform has the following hardware architecture:



Figure .1 - Hardware architecture (onboard)

The flight computer process will be running on the Overo Fire board. To enable the flight computer to be run an operating system for the Overo Fire will need to be built and configured. Furthermore software libraries will need to be developed and designed for each sensor to collect their respective data inputs. The implementation of these libraries will require knowledge of how the sensors send their data and how their data can 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 is structured.

# IMU

## IMU hardware interface

The Sensordynamics 6DOF IMU is connected to the Overo Fire though UART. Originally it was designed for the IMU to be connected to the Overo Fire by SPI as is stated in the IMU datasheet. However under investigation it was found that the SPI is not yet implemented on the chip. 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 are:

* 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 has to be reconfigured to be using 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 will need to be implemented in the software library that enables the IMU ‘s baud rate to be reconfigured every time the Overo Fire is powered on.

## 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.

Table - MVRB contents [RD/4]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |  |
| 0x01 | MR1 – Rate Y | Angular rate along Y axis, measure range 1 | 16 bit / 2’s  complement |  |
| 0x02 | MR1 – Rate Z | Angular rate along Z axis, measure range 1 | 16 bit / 2’s  complement |  |
| 0x03 | MR1 – Accelerometer X | Acceleration along X axis, measure range 1 | 16 bit / 2’s  complement |  |
| 0x04 | MR1 – Accelerometer Y | Acceleration along Y axis, measure range 1 | 16 bit / 2’s  complement |  |
| 0x05 | MR1 – Accelerometer Z | Acceleration along Z axis, measure range 1 | 16 bit / 2’s  complement |  |
| 0x06 | MR2 – Rate X | Angular rate along X axis, measure range 2 | 16 bit / 2’s  complement |  |
| 0x07 | MR2 – Rate Y | Angular rate along Y axis, measure range 2 | 16 bit / 2’s  complement |  |
| 0x08 | MR2 – Rate Z | Angular rate along Z axis, measure range 2 | 16 bit / 2’s  complement |  |
| 0x09 | MR2 – Accelerometer X | Acceleration along X axis, measure range 2 | 16 bit / 2’s  complement |  |
| 0x0A | MR2 – Accelerometer Y | Acceleration along Y axis, measure range 2 | 16 bit / 2’s  complement |  |
| 0x0B | MR2 – Accelerometer Z | Acceleration along Z axis, measure range 2 | 16 bit / 2’s  complement |  |
| 0x0C | Temperature | Temperature | 16 bit, unsigned |  |
| 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. 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 - Important CRB registers [RD/4]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |

## 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 ***WCRBH1HH2***where ***H1*** is a 1 digit hex number representing the CRB register address and ***HH2*** 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 will be implemented in the software library that can enable the CRB registers of the IMU to be read or written.

## 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/4]:

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

## 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 will be equal to 0x7041 or in binary 0b0111000001000001. Hence when the 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 output format:

***| MR1 – Rate X | MR2 – Rate X | Temperature | Status\_L | Status\_H | Chk |***

The 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 should include the following MVRB registers:

* MR2 – Rate X
* MR2 – Rate Y
* MR2 – Rate Z
* MR2 – Accelerometer X
* MR2 – Accelerometer Y
* MR2 – Accelermatoer 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.

## 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 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 –
* MR2 Rates –
* MR1 Accelerations –
* MR2 Accelerations –

Continuing the previous example, once the rateX value is scaled (using the multiplier of 0.0156) 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.

## IMU 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.
* startFreeRunningMode: Starts the IMU in free running mode.
* stopFreeRunningMode: Stops the IMU from executing free running mode.

The header file imserual.h and the source file imuserial.c that implement these functions has been including in Appendix 1.

# Magnetic compass

## 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/5]:

* 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 for 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 which has available for the compass was 38400, thus is the baud rate which will be used when connecting to the Arduino. The microcontroller located on the OS4000 compass saves what baud rate it has been configured and saves this information when the compass has been turned off. Thus baud rate reconfiguration does not have to occur each time the compass is turned on.

## 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 from the compass 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/5]. 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.

## 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 90⁰ 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.

## 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 2. Also 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 3).

# Altitude sensor and battery voltage sensor

## Altitude sensor hardware interface

The LV-MaxSonar-EZ0 sensor provides altitude readings in 3 forms including: asynchronous serial, analog to digital conversion (ADC) and pulse width modulation. The ultrasonic sensor connects to the Arduino via an ADC pin. The ultrasonic sensor itself outputs an analog voltage with a scaling factor of (Vcc/512) per inch. Thus since the ultrasonic is powered by 5V the scaling factor is equal to 9.8mV per inch. 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:

## 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 directly after 5 altitude readings have been taken place on the Arduino.

## 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:

## Battery voltage filtering

The battery voltage reading needs to be filtered 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 directly after 5 battery voltage readings have been taken place on the Arduino.

## Altitude and battery voltage library Arduino functions

The following functions have been implemented to read the compass heading on the Arduino:

* readAltitude
* readVoltage

The source file arduserial.pde contains the implementation of these functions and can be seen in the Appendix 4.

# Arduino

## 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 Duemilanove 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 and sent to the Overo Fire which is running the flight computer. The Overo Fire connects to the Arduino via the following UART configuration:

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

## 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.

## Arduino functions

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

* readOvero

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

## Arduino 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 contains the implementation of these functions and can be found in the Appendix 5.

# Overo Fire

## Flight computer structure

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. 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 program and threading structure of the flight computer can be seen in the Figure below.

\

Figure .1 - Flight computer thread structure

## 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: Initalises 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 contains the implementation of these functions and can be found in the Appendix 6.

## 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/6].

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 used to create this console image can be found in the Appendix 7. 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.

## Operating system installation

Once the operating system is built under the instructions found in [RD/6] 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/7].

## 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 which 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 that was used for this process can be seen in the Appendix 8.

# Conclusions

This document presented the design of the flight computer and how it will be implemented via the use of threading. It included how to configure each sensor into a mode that is appropriate for the platform and how the Overo Fire will connect to it. It also showed the output format for each sensor that is connected to the Overo Fire board (or Arduino) and how the sensory information will be collected by the flight computer. Finally the configuration and installation of the Overo Fire operating system was described including what programs were used on the flight computer and how it was installed on the SD card.

# Recommendations

It is recommended that the flight computer design and code structure specified in this document be implemented on the Overo Fire. If any errors are found during implementation or testing, they should be mentioned and a new revision of this document generated. Particular care must be taken when implementing the threads on the flight computer such that deadlock or scheduling issues are avoided. The processes which these threads control (state estimation, sensor readings, UDP client) should be verified by checking that process inputs and outputs from the flight computer are correct.