

**La Trobe University**  
Bundoora, Victoria, Australia  
Faculty of Science, Technology & Engineering  
Department of Electronic Engineering



## **Autonomous Unmanned Aerial Vehicles**

### **ELE40ENP Project Thesis**

**Authors:**     **Robert Ross** (02578565)  
Bachelor Computer Science (Hons) /  
Bachelor Electronic Engineering

**Wade Tregaskis** (02557793)  
Bachelor Computer Science (Hons) /  
Bachelor Electronic Engineering

**Supervisors:**     Associate Prof. John Devlin  
Head of Electronic Engineering Department,  
La Trobe University

Mr. Paul Main  
Director, Systems Engineering Arts Pty Ltd

## **Abstract**

This thesis reports on the work done by Robert Ross & Wade Tregaskis for their fourth year engineering project, as part of their Computer Science (Honours) / Electronic Engineering double degrees. The project aimed to develop an autonomous UAV, using a model plane, microcontroller, and appropriate sensors.

The ATmega2561, an 8-bit RISC microcontroller from the Atmel AVR family [1], was chosen to power this system. A custom PCB was designed to incorporate the various sensors (GPS receiver, magnetometer, accelerometers, etc) and other systems (including a Bluetooth transceiver). Software for the system was written in C, compiled with ImageCraft C Compiler v7 for AVRs [2], and programmed into the AVR via Atmel's AVR Studio 4 using AVRISP hardware.

The completed system was mounted inside an "ElectraFun XP" model airplane, manufactured by J Perkins Distribution Ltd and available in Australia from Model Engines Australia Pty Ltd. Our system interfaced with the onboard servos and wireless control receiver, and was successfully flown numerous times. Autopilot was engaged during test flights and seen to be at least partly functional.

Flight data was recorded aboard the plane onto a MultiMedia Card (MMC) or Secure Digital Card (SDC), including the plane's actual flight path, sensor measurements, and other information.

## **Acknowledgements**

We would like to express our sincere thanks to Associate Professor John Devlin, our primary supervisor. From the outset John has enthusiastically supported, advised on and promoted our project both within the University and externally, resulting in appearances in The Age and The Herald Sun newspapers as well as on the TV show Totally Wild. John's technical expertise and engineering experience have immeasurably contributed to the success of the project.

We also acknowledge the wonderful assistance from Mr. Paul Main, our industry co-supervisor from Systems Engineering Arts Pty Ltd. Paul has provided numerous resources, particularly related to working with our embedded platform. We have greatly valued Paul's input as from the outset he has with a keen interest provided expert support and knowledge.

We extend our special thanks to Mr. Alvin Ng, General Manager of Dick Smith Electronics, who graciously provided a scholarship of \$200. Without this extended funding our project certainly would have had trouble getting off the ground.

We also extend our thanks to the Department of Electronic Engineering at La Trobe University for funding and supporting this research project. Within the department we would like to acknowledge the special assistance from Mr. Adam Console, Mr. Peter Stewart and Mr. Joe Hura.

Finally, special thanks go to our friends and families who have supported and helped us throughout this research project and more widely throughout our taxing double degrees.

# Contents

Abstract.....	i
Acknowledgements .....	ii
Contents .....	iii
List of Figures .....	v
Glossary .....	vi
1. Introduction .....	1
1.1 Project Objectives .....	1
1.2 Team member responsibilities.....	3
1.3 Organisation .....	3
2. Research .....	4
2.1 Flight Theory .....	4
2.1.1 Planes of motion .....	4
2.1.2 Aircraft Controls .....	5
2.1.3 Flight Manoeuvres .....	6
2.2 UAV Requirements.....	6
2.2.1 Navigation .....	7
2.2.2 Flight Control.....	7
2.2.3 Data Logging .....	9
2.2.4 Sensors.....	10
2.3 Commercial Applications.....	11
3. Design .....	12
3.1 Hardware .....	12
3.1.1 Navigation .....	12
3.1.2 Flight Control.....	13
3.1.3 Sensors.....	13
3.1.4 Data Logging and Communication .....	14
3.2 Embedded Software .....	14
3.2.1 Navigation .....	15
3.2.2 Flight Control.....	15
3.2.3 Sensors.....	17
3.2.4 MMC/SDC.....	17
3.3 PC based software.....	19
3.3.1 Flight Planning & Display .....	19
4. Implementation.....	21
4.1 Airframe .....	21
4.2 Hardware Implementation.....	21
4.2.1 Photo/Video Capability .....	22
4.2.2 Manual Switchover Circuitry .....	22
4.2.3 Sensors.....	23
4.2.4 Bluetooth Data Link .....	24
4.3 Embedded Software Architecture.....	25
4.3.1 Runloop .....	25
4.3.2 Interrupt-based I/O .....	27
4.3.3 Flight Control Algorithm.....	27
4.3.4 Servo Control.....	29
4.3.5 Data Logging .....	30
4.3.6 Magnetometer .....	37
4.4 Flight Planner .....	39
4.4.1 FlightPlanDocument .....	40



4.4.2 FlightPlan3DView .....	40
4.4.3 Google Maps WebView .....	41
4.4.4 GoogleMapCache.....	41
4.5 Budget .....	42
5. Results.....	43
5.1 Flight results .....	43
5.1.1 Bearing .....	43
5.1.2 Accelerometers .....	44
5.1.3 Elevation.....	45
5.1.4 Temperature & Pressure.....	46
5.1.5 Battery .....	48
5.2 Autopilot results .....	48
5.3 Logging results .....	49
6. Discussion .....	51
6.1 What went right .....	51
6.1.1 GPS .....	51
6.1.2 Manual override & emergency autopilot switch.....	51
6.1.3 Bluetooth .....	51
6.1.4 FAT support.....	52
6.2 What when wrong.....	52
6.2.1 Flash logging software over-designed and overly complex .....	52
6.2.2 Motor interference.....	53
6.2.3 Accelerometers .....	54
6.2.4 Pressure sensor's insufficient resolution.....	54
6.2.5 Manufacturing delays.....	54
6.2.6 ICC AVR7 bugs and limitations .....	54
6.2.7 Wireless transmitter .....	55
7. Conclusion .....	56
7.1 Future work .....	58
7.2 Parting Words.....	61
8. References.....	62
Appendix A: Schematics .....	64
Appendix A: Schematics .....	64
Appendix C: Flight log formatting.....	69
Appendix D: Example flight log .....	70
APPENDIX E: FAT File System Format - Summary .....	72
E.1 FAT Boot Sector .....	72
E.2 FAT Volume Layout .....	76
E.3 FAT.....	77
E.4 FAT Directories .....	77
Appendix F - Flight Planner Class Hierarchy.....	80
Appendix G - Google Maps WebView pre-defined HTML, Map.html.....	81

## List of Figures

Figure 1: Forces acting on the aircraft [7] .....	4
Figure 2: Planes of motion [7] .....	5
Figure 3: Plane Structure [7].....	5
Figure 4: Logging Stack Design .....	18
Figure 5: Fully Assembled Board .....	18
Figure 6: Photo taken from onboard UAV .....	22
Figure 7: Manual Override Circuitry.....	23
Figure 8: Main Runloop.....	26
Figure 9: PWM output scheme .....	29
Figure 10: Logging Stack .....	30
Figure 11: Magnetometer Calibration .....	37
Figure 12: Magnetometer uncalibrated chart.....	38
Figure 13: Magnetometer calibrated chart.....	38
Figure 14: Deviation of magnetometer from compass measurement.....	39
Figure 15: Flight Planner screenshot.....	39
Figure 16: Flight Planner relationship diagram .....	40
Figure 17: Matlab flight path plot .....	43
Figure 18: Flight #6, GPS vs Magnetometer bearing.....	44
Figure 19: Flight #5, Accelerometer vs Time.....	45
Figure 20: Flight #3, Elevation vs Time.....	46
Figure 21: Flight #3, Temperature & Pressure vs Time .....	46
Figure 22: Flight #3, Temperature & Pressure vs Altitude .....	47
Figure 23: Flight #6, Temperature & Pressure vs Time .....	47
Figure 24: Flight #3, Battery vs Time .....	48
Figure 25: Autopilot bearing tracking .....	49
Figure 26: DOSonCHIP Breakout board.....	53
Figure 27: MEMs Altimeter.....	60
Figure 28: Radio Modem Pair.....	61
Figure 29: FAT Bootsector .....	73
Figure 30: FAT Layout.....	76
Figure 31: FAT Directory Entry.....	78
Figure 32: Class Hierarchy .....	80

## Glossary

AMSL	Above Mean Sea Level, used to denote the reference point when talking about elevation or altitude, as the average sea level. What the average sea level is at any particular point is often contentious, but typically the GPS opinion is considered authoritative.
Bitbanging	A method of replacing dedicated communications hardware with a software implementation, typically involving manipulation of individual bits or pins.
Bytecode	A intermediate compiled form of software, which is interpreted or translated to machine code at runtime.
CCITT	International Telegraph and Telephone Consultative Committee (CCITT, from the French name "Comité consultatif international téléphonique et télégraphique". Renamed in 1992 to the ITU-T.
CRC	Cyclic Redundancy Check (also, Cyclic Redundancy Code). A means of computing a special code from data, such that it is very unlikely that a random number and combination of errors will modify the data in such a way that the code will not change. Used to reasonably assure correct data transfer.
GLU	OpenGL Utility Library, a collection of miscellaneous functionality for working with OpenGL.
Endianness	The ordering of bytes within a multi-byte structure. Little-endian systems place bytes in increasing order of significance as the address goes higher, while big-endian systems go the other way.
GPS	Global Positioning System.
ICC	ImageCraft C Compiler.
ITU	International Telecommunication Union, a standardisation body for international telecommunications protocols and methods.
ITU-T	ITU Telecommunication Standardization Sector, a branch of the ITU that deals specifically with telecommunication (as opposed to radio communication).
LiPO	Lithium Polymer, a common type of high-energy-density rechargeable battery.
MAC	Media Access Control, the layer of a communications stack which manages and possibly arbitrates access to the physical media.
MEMS	MicroElectroMechanical Systems. A somewhat ambiguous name describing electromechanical systems that are very small, down to nanometre scales.
MMC	MultiMedia Card.
NiMH	Nickel Metal Hydride, a common type of rechargeable battery.
NMEA	National Marine Electronics Association. Typically associated with the GPS, where they defined the most common receiver output format.
OpenGL	Open Graphics Library, the standard library and API for writing 3D software in the computer industry.
PHY	An abbreviation of "Physical Layer" in reference to communications systems, referring to the layer that manages the actual hardware, typically involving conversion of an arbitrary

	digital bitstream to and from the analogue communication method.
PID	Proportional, Integral, Differential. A type of control algorithm that utilises feedback to respond relative to the current error magnitude (proportional), rate of change (differential) and long-term average error (integral).
Refactoring	The process of reorganising (redesigning in place) a system without adding additional functionality, in order to improve one or more aspects of it (e.g. maintainability).
RS-232	A very commonly used low-speed interface for connecting two electronic devices together. Also known generically as “serial”.
SD	Secure Digital, a type of Flash memory card.
SDC	Secure Digital Card, a type of removable Flash memory.
SPI	Serial Peripheral Interface.
PWM	Pulse Width Modulation.
UART	Universal Asynchronous Receiver/Transmitter, a device for interfacing a communications method (e.g. RS-232) to a microcontroller or similar system.
UAV	Unmanned Aerial Vehicle.
USB	Universal Serial Bus, a high-speed serial bus interface designed for use by microcomputers, and more recently seeing use in smaller electronic systems. Typically a replacement for RS-232 and similar serial interfaces.
Xmodem	A file transfer protocol, very popular in the late 70’s and early 80’s, before being superseded by the Zmodem protocol, an evolutionary upgrade.

# 1. Introduction

UAVs (Unmanned Aerial Vehicles) are, by common definition, re-usable, powered, guided aircraft which do not carry people[3]. Traditionally these aircraft have been controlled in real time from a remote location via a radio link[3]; hence the synonymous use of the term "remote-controlled aircraft". Given this, one could argue they are still not really unmanned; the need for human piloting (albeit remotely) persists, negating their benefits in many scenarios.

In recent times there has been growing interest and research into autonomous UAVs, where control systems onboard the aircraft aid - or entirely subsume - the role of the human pilot[3, 4]. The intention in creating autonomous UAVs is to eventually obviate the need for direct human supervision. Onboard autonomous control systems provide increased responsiveness and can operate in dangerous environments, over larger distances, and without limits imposed by human limitations (such as fatigue)[4].

The purpose of this project was to create an inexpensive autonomous UAV using commercial off the shelf components: a hobbyist model plane, a microcontroller, a GPS receiver, a magnetometer, accelerometers and other sensors. It was posited that, using these primary sensors, the microcontroller onboard the aircraft should be able to pilot the aircraft; reducing human input to mission planning, takeoff and landing.

During operation, the UAV records data from various sensors, ranging from flight avionics to arbitrary payloads such as temperature sensors, cameras, etc. This data is stored onboard in non-volatile memory, for transfer to a PC once the plane lands.

UAVs have traditionally been used primarily and extensively by the military, and that use will only increasing in future, both in magnitude and scope. Additionally, as UAV technology becomes more widely available and less costly, commercial applications are increasingly viable. While the commercial market is small, it has fantastic growth potential. The world UAV market is widely predicted to exceed US\$5 billion annually within the next five years[4].

Although the fundamental concept behind this project - autonomous UAVs - has already been implemented by various manufacturers, to date these have been primarily high-end, military-orientated designs. In addition to exploring the low-end of the UAV design realm, this project required research in a range of fields including real-time control systems, fault tolerant design, navigational vectoring, miniaturisation and weight reduction.

## 1.1 Project Objectives

The direction and aims of the project took some time to settle upon. Initial suggestions included an airborne laser tag system, an automatic landing system, real-time visual object detection and ranging, and so forth. As these ideas were researched and evaluated, expectations were tempered somewhat, with some of the more advanced areas such as image recognition rejected on the basis of time and budgetary constraints.

Eventually, after much research and discussion with our supervisors, the formal objectives were specified, in three tiers. The *primary objectives* are the fundamental goals set to be achieved, which were expected to be completed to a satisfactory level. *Secondary objectives* were defined as extensions and enhancements on the primary objectives, which are desirable to explore but not critical to the success of the project. *Tertiary objectives* were defined to cover related project ideas that may have been researched as part of the project, but which were not planned for implementation due to time, budgetary and technical constraints. They were primarily indicators of where it was anticipated future work might be conducted.

### **Primary Objectives**

- Outfit a model aircraft with a navigation and control system consisting of a GPS receiver, a magnetometer, accelerometers and an AVR microcontroller.
- Write software for the microcontroller to sample each of the sensors and control the aircraft according to a pre-determined flight plan. (proviso: Aircraft will be launched and landed under manual control)
- Write computer software to develop a flight plan for the UAV to follow, with a communications link to facilitate transfer of the flight plan to the UAV.
- Support onboard recording of all flight data (e.g. GPS co-ordinates, magnetometer readings, etc) onto a MMC/SDC.
- Provide an in-flight wireless simplex communication channel to transmit data from the UAV back to a ground station (i.e. PC).
- Provide a control system to manually switch between autonomous control and human control.

### **Secondary Objectives**

- Addition of onboard sensors, with the sampled results stored on the onboard storage. Types of sensors to be investigated include battery monitors, temperature sensors and pressure sensors.
- Install an onboard wireless video camera to provide proof-of-concept video images of the visual data acquisition capabilities of the UAV.
- Implement Error Correcting Codes (ECC) for the wireless PC link transmission to ensure signal quality and attempt to recover corrupted data.
- Implement a cyclic redundancy check (CRC) for the wireless PC link to detect data corruption.
- Implement a failsafe autonomous control system, whereby if remote control is out of range the UAV will return to previous position where controller was in range, and progressively search for a signal.

### **Tertiary Objectives**

- Implement a visual landing system using a small camera and edge detection algorithms, to allow the aircraft to autonomously land on a well defined runway.
- Implement a collision detection system based on ultrasonic or laser measurements to detect the presence of upcoming objects and avoid them.

## **1.2 Team member responsibilities**

While the work of both team members overlapped in many cases, sections of the project were assigned to one or the other as their primary responsibility, as shown.

### **Robert Ross**

- Flight control & navigation software
- Sensors
- Piloting
- Hardware design & assembly
- Hardware Testing

### **Wade Tregaskis**

- Flight planning software
- Bluetooth
- Sensors
- MMC/SDC data logging
- Optimisation & testing

## **1.3 Organisation**

The organisation of this thesis as is follows:

Section 2 covers the research conducted at the start of the project. It begins with a brief lesson on basic flight theory, and then looks at the physical and functional requirements of the system. Lastly, a brief summary of the commercial interest and applications is included.

Section 3 presents the design of the key elements of the project, in particular the avionics hardware and the microcontroller software, as well as the flight planning software.

Section 4 explains the actual implementation, including problems encountered. It covers the three key areas of the previous section, as well as a discussion on the airframe and budgetary expenses.

Section 5 presents the results of our labour, covering the flight data recorded, autopilot test results and the data logging system (both Bluetooth and the onboard storage).

Section 6 discusses the results and the work of the project, including the perspectives and opinions of the authors. Notably, both technical achievements and shortcomings are presented, leading into the concluding remarks and future work found in Section 7.

Following the references in Section 8, several appendices are attached, providing hardware schematics, a listing of the attached CD contents, some sample flight log data and software implementation details.

## 2. Research

After early consultation with John Devlin the project objectives, as listed in Section 1.1, were settled upon. These objectives defined the requirements, and determined the areas that needed to be researched for the project. In order to meet some of the grander aims, such as autonomous flight, an understanding of fundamental aircraft motion and dynamics was necessary.

### 2.1 Flight Theory

During flight four forces (shown in Figure 1) are constantly acting upon the aircraft: lift, weight, thrust and drag[5]. Lift and weight are opposing forces, with lift being a force perpendicular to the flow of air over the aircraft wings (generated by differential pressures at different points on the wing), and weight being the downward force due to gravity[6, 7]. Thrust and drag are another pair of opposing forces, where thrust is a forward directed force created by the motion of the aircraft's propellers (or turbines/rockets) and drag is the aerodynamic force resisting the forward motion of the aircraft[6-8].

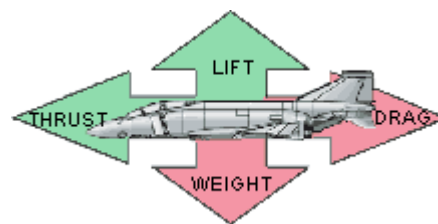


Figure 1: Forces acting on the aircraft [7]

#### 2.1.1 Planes of motion

The flight path of fixed wing aircraft is commonly resolved into three planes of motion; *yaw*, *pitch* and *roll*. Each of these planes of motion, as shown in Figure 2, operates along an orthogonal axis and is primarily controlled through a particular control surface[7, 8].

- *Yaw* describes the movement of the aircraft about the normal axis, which refers to the pivoting of the aircraft as modelled on a flat surface.
- *Pitch* refers to the movement of the aircraft around the lateral (or transverse) axis. The lateral axis is parallel to the wings, thus the pitch is used to describe the angle of the nose of the aircraft (angle of attack) with respect to centre of gravity.
- The *roll* (or bank) motion of the aircraft describes movement around the longitudinal axis (an axis parallel to the fuselage of the aircraft). An aircraft rolling can be characteristically observed with the wings of the aircraft alternately rising and dipping.



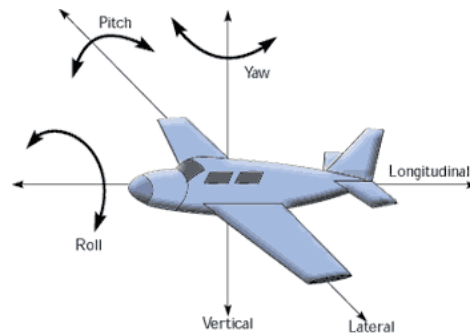


Figure 2: Planes of motion [7]

### 2.1.2 Aircraft Controls

Fixed wing aircraft are generally manoeuvred through use of three independent control surfaces. Each of these controls has an initial effect on one of the motions described previously, as well as a different secondary effect if the control surface settings are maintained[8]. In addition to these primary controls, several ancillary controls such as trims, flaps and throttle are also used. For unmanned aircraft the most commonly used of these ancillary controls is throttle.

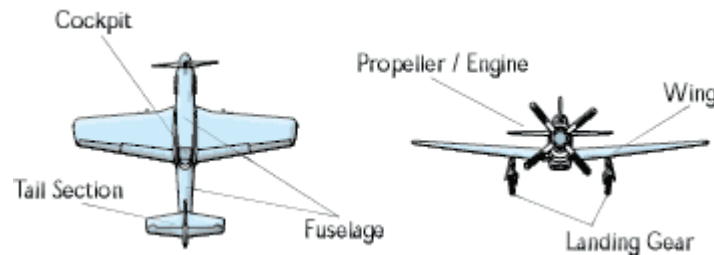


Figure 3: Plane Structure [7]

- The level of *throttle* (power to the engine) has both primary and secondary effects, akin to the main primary controls as discussed below. The primary effect of the throttle control is regulating the motor speed which relates to the thrust of the aircraft. The secondary effects are in pitch (increased throttle will cause the nose to pitch upward), a yawing affect (increased throttle will tend to make the plane yaw left), and a change in altitude (more throttle results in higher airspeed which increases lift).
- The *elevator* is a horizontal control surface positioned on the trailing edge of the horizontal stabiliser of the aircraft (which is normally found on the tail). The primary effect of the elevator is to change the pitch, where down elevator results in a decrease in pitch and up elevator results in an increase in pitch. The secondary effect of the elevator is to change the airspeed, which increases as the nose is pitched down and decreases as the nose is pitched up.
- *Ailerons* are horizontal control surfaces mounted on the wings of the aircraft that operate in an equal and opposite direction on each wing (one aileron will go up while the other goes down). The initial effect of the ailerons is to change the roll of the aircraft, with a secondary effect of causing the aircraft to yaw.
- The *rudder* is a vertical control surface normally found on the tail. The rudder has a primary effect of causing the aircraft to yaw, with a secondary effect of causing the aircraft to roll. The plane being used for this project only has throttle, rudder and elevator controls, so to cause the plane to roll; the secondary effect of the rudder is utilized.

### 2.1.3 Flight Manoeuvres

During basic flight training pilots are taught a series of manoeuvres which provide a basic skill set for flying powered aircraft[8]. Autonomous UAVs similarly need to understand and be able to perform these manoeuvres. Typically these manoeuvres include [8-10]:

- *Takeoff*: During takeoff, the aircraft needs to quickly gain momentum to ensure that sufficient lift is available to climb. Small UAVs are typically hand launched by giving them a gentle throw whilst applying full throttle. During this takeoff phase, the rudder/ailerons are used to ensure the plane doesn't bank overtly (as this reduces lift) and up elevator is gently applied to ease the aircraft into a gentle climb. Typically takeoff is performed into wind, to ensure maximum airflow over the wings – generating maximum lift, and increasing stability.
- *Landing*: UAVs are normally landed into wind, to provide maximum stability at low groundspeed. A typical landing begins by flying several around the landing area to gradually reduce altitude. The landing itself involves slowly decreasing altitude from a few meters above the ground, whilst throttling back power and stabilising the aircraft using the rudder.
- *Straight and Level Flight*: Straight and level flight describes flight when all the forces acting on the aircraft are balanced (thrust, drag, weight and lift). In this circumstance the aircraft will maintain a constant altitude, velocity and bearing. The main flight controls are used to maintain straight and level flight, and the throttle is adjusted to balance the thrust force with the drag force.
- *Turning*: Typically the ailerons are the primary control used to turn fixed wing aircraft, but in the case of small UAVs lacking ailerons, the secondary effect of the rudder (banking) is used. To turn the aircraft the rudder is used to bank the aircraft, which will cause the aircraft to turn. To complete the turn the rudder is moved in the opposite direction to return the plane to level.
- *Ascending/Descending*: The elevator is the key control involved in ascending and descending, but it doesn't operate in isolation. Specifically the throttle is important, as insufficient thrust when attempting ascent will result in a stall, and excessive thrust when descending may result in an undesirably fast rate of descent. Thus, typically when ascending the throttle would be increased, and the elevator would be gently pulled up. When descending the throttle may be lowered as the elevator is gently pulled down.
- *Stalling*: A stall is defined as the condition where the angle of attack (pitch of nose) exceeds the critical angle, resulting in a rapid reduction of lift. The critical angle of attack is the angle where the coefficient of lift is maximised, and any further increase in the angle of attack will result in a rapid reduction of lift. For light fixed wing aircraft, the critical angle of attack is typically at least 16 degrees. Several factors can influence the angle of attack required to stall, including weight (which acts in opposition to lift), airspeed (as higher airspeed increases lift) and the amount of bank (as overall lift is significantly reduced during banking).

## 2.2 UAV Requirements

While there are numerous subtle requirements for building an autonomous UAV, the most important ones are detailed in this subsection. Many of the requirements

evolved over the course of the project, due to new information and changing priorities; only the final requirements are presented here.

### **2.2.1 Navigation**

The UAV needs to know where it is, where it should be, and how to get there. Luckily this is, for the most part, very simple. The position of the UAV can be provided by a GPS receiver, waypoints can be specified in configuration files, and there are thankfully very few obstructions at typical operating elevations.

Initial research confirmed that modern GPS receivers are capable of achieving resolutions less than tens of metres, sufficient for our purposes, and with a typical update rate of 1Hz are sufficient (although not ideal) for navigation.

The UAV's flight path can be specified as a sequential list of co-ordinates in 3D space (latitude, longitude and elevation). There are only two navigational elements - bearing and altitude - to be computed and managed, which is easy to do, even on a minimal microcontroller.

Additional methods of navigating, such as visual recognition of landmarks, were investigated. The requirements for image recognition, even if only simple, are substantial - both in terms of necessary hardware as well as algorithmic complexity. Such systems were briefly researched, but not developed for the project.

### **2.2.2 Flight Control**

Flight control, as opposed to navigation, is the management of the control surfaces of the plane (and the motor) to perform flight manoeuvres (as covered in Section 2.1). It is concerned particularly with the stability of the plane - especially for straight and level flight - and responding to the needs of the navigation system.

On the ElectraFun XP the control surfaces are driven by electronic servos. Electronic servos consist of a DC motor mechanically coupled to a potentiometer to provide a feedback path. A pulse-width modulated (PWM) signal is used to drive the servo. The PWM signal has a period of 20ms and a 'high' pulse time of between 1.25ms and 1.75ms – which is used to control the position of the servo motor arm[11-14]. The servos used on the ElectraFun XP have a range of motion of approximately 180 degrees.

For the model airplane chosen, the motor speed controller utilises a servo-compatible interface, where longer pulses correspond to higher throttle.

As a sidenote, acrobatic flight - where the plane may perform a roll or loop - is far more difficult to manage than basic flight, and was never intended to be part of the project. Some allowances were made, however, for cases such as an accidental roll-over, although these were never fully implemented, nor tested.

### **Orientation**

In order to meet the control requirements, the plane needs several sensors. The GPS receiver is not useful for basic flight control, as it relates only to the actual position of the aircraft, not its orientation, pitch, immediate change in bearing, etc. For these, other sensors are required; an ideal sensor for this purpose is a gyroscope. As

summarised by the Wikipedia article for Gyroscopes, "A gyroscope is a device for measuring or maintaining orientation, based on the principle of conservation of angular momentum" [15].

Typically a two-axis gyroscope would be used, oriented to measure pitch (or attitude) and roll (or bank). A third axis (for yaw) is not strictly necessary, but could be useful if drift factors can be added as inputs to the control algorithms. Commercial aircraft avionics typically augment gyroscopes with accelerometers to create a complete Inertial Navigation System[16].

Unfortunately, research revealed many potential problems with use of a gyroscope. First and foremost, they are expensive - upwards of a hundred dollars for quality components. They are also fairly large, especially mechanical gyroscopes, and can have substantial power requirements (hundreds of mW). Given these problems, it was decided not to use a gyroscope, and instead attempt to suffice with only accelerometers. Accelerometers measure acceleration (including gravity), which is useful in many respects, but there are many sources of acceleration - gravity, change in speed and rotation. It was foreseen early on that decoupling all these from each other would be difficult, if possible at all. We optimistically hoped we could overcome the limitations in software.

## **Elevation**

While the GPS receiver can provide an approximate elevation once a second, the GPS is infamous for poor elevation accuracy, and John Devlin expressed strong reservations about its suitability. Since knowledge and control of elevation is rather critical in an aircraft, alternative means of measuring elevation were explored.

Commercial aircraft use a variety of systems for determining their elevation above ground level and mean sea level. Pressure is considered a fairly reliable determinant, particularly for relative elevation, although its resolution is very limited - typically it is used on scales of hundreds of metres, whereas for a small UAV a resolution of at least tens of metres is highly desirable.

A common method used around airfields is a ground based system of radio beacons, using triangulation to determine the plane's relative position in 3D space. Such systems can be extremely accurate within their field of operation, but are unsuitable for UAVs because the field of operation is potentially huge, and may include irregular terrain which is not suitable for such systems.

Another common method used is some form of radar or ultrasonic range-finder. Such systems typically provide accurate, position-independent determination of elevation above ground level. Their behaviour is, however, influenced by the reflecting surface below the aircraft, which is of some concern for a general use UAV. In particular, physical obstructions such as foliage may not be dense enough or have the ideal geometry for such systems, and may not appear, or appear and disappear sporadically. And beyond all this, there are mechanical issues with fitting any line of sight system such as these - these sensors must always be orientated vertically downwards, ideally across any possible range of rotation of the plane on any axis. This poses a significant design challenge. A compromise can be made, by which the system has a fixed orientation relative to the plane, and the software is aware of and accounts for the fact

that the plane will not always be correctly orientated to produce a useful reading. This represents a fairly significant amount of work, and it limits the freedom of the aircraft, placing further restrictions on the flight control.

It seemed unlikely that reliable determination of elevation could be achieved, beyond what is provided by the GPS. It was an easy decision to include a pressure sensor, given their availability and minimal cost, but it was never expected that this sensor would be useful for flight control.

## **Speed**

The groundspeed can be determined with reasonable accuracy using the GPS, provided it is above a certain threshold - approximately 16km/h, as a rule of thumb. It was not known what the typical cruise speed of the chosen aircraft was prior to building the system, so it was uncertain how useful the GPS would be for this purpose. Unfortunately, there are limited other means for accurately determining groundspeed. Typically some kind of stationary ground reference is required, whether by visual recognition, radio triangulation or radar. All these things, as noted previously, were not available to use.

This is not necessarily a problem, even if the GPS turns out to be insufficient, because it is the airspeed which is most important in terms of flight control. The airspeed can be reliably measured in a variety of ways, the most common of which is some kind of Pitot-static sensor - that is, a differential pressure sensor where one chamber is orientated in the direction of flight, and the other is sealed at a known pressure (e.g. 1 atmosphere). There are some practical issues with the use of Pitot-static sensors, in particular the fact that they have an alignment and so while using just one fixed parallel to the aircraft body is most often sufficient, it does not work reliably if there is significant cross-wind producing lateral movement of the aircraft, or in the case of a stall that results in significant downwards motion. Using multiple sensors aligned to different axes can resolve this, but becomes quite laborious, and expensive.

Ultimately it was felt that an onboard airspeed indicator was not strictly necessary, under the restriction that the plane be pre-dominantly operated at full throttle under autopilot, and that it not be operated during excessively windy conditions. The latter is a mechanical limitation of the plane itself, in any case.

### **2.2.3 Data Logging**

During the research phase of the project, and prior, several tests were conducted regarding equipping the plane with cameras, both still and video. These yielded promising results, proving at the least the suitability of model planes for such tasks. While ultimately it was not a primary aim of the project to include image acquisition, it was a consideration when it came to the data logging capabilities of the system. In particular, it encouraged the use of a high-capacity logging facility. Incidentally, testing the use of a wireless transmitter for the video camera revealed substantial problems with interference, which emphasised the importance of onboard data logging, given that a wireless link may not be reliable, or available at all (i.e. in true UAV operation over large distances).

The initial inclination was to use some kind of non-volatile memory. This could be fixed on the PCB (e.g. an EEPROM chip), an external device (e.g. connected via RS-

232, SPI or similar), or a removable device (e.g. a CompactFlash card). Additionally, hard drives were considered briefly. However, hard drives are relatively heavy, and quite expensive. They are also mechanical devices, with moving parts, which are not ideal for a potentially unstable platform such as a UAV. In particular, reliable onboard logging is most critical during a crash, which hard drives most definitely are not suited for. And lastly, hard drives require a significant amount of power, which directly impacts flight time.

A further possibility is a high-capacity SDRAM module, such as standard PC DDR/DDR2 memory. However, SDRAM is volatile (requires power to maintain its contents), which is clearly undesirable - in the event of a crash the internal systems may lose power.

Ultimately the decision was made to adopt some form of non-volatile memory. While it is entirely possible to acquire Flash memory or EEPROM chips to be included on the PCB itself, such an approach was considered unwieldy. The obvious approach, in contrast, was to use some form of removable memory card, such as CompactFlash or MMC/SDC. After researching the use of MMC/SDC in other projects and seeing the successes, the decision was made. MMC/SDC cards are small, widely available, and very cost effective in terms of capacity. CompactFlash cards, by comparison, are more expensive and less widely used - they are limited primarily to high-performance areas, which are far beyond our needs.

#### **2.2.4 Sensors**

While for this project the emphasis was on producing an autopilot for a UAV, the question of course is why? While our focus was on the system itself, not it's applications, we felt it important to provide some sort of proof of concept of at least one possible use, to demonstrate the relevance and significance of the project.

Most current UAV applications revolve around environmental measurement, surveying and image recording. UAVs have no capacity for carrying people by definition, and our focus is on small, light aircraft rather than larger cargo-capable ones, so transport of people and goods is not within our focus.

One non-military field where UAVs are increasingly applicable is meteorology[17]. With this in mind we researched what sensors could be added to provide basic meteorological measurements, with the proviso that they be reasonably inexpensive and simple to interface. The first sensor identified for possible inclusion was a pressure sensor. As noted, this was already considered as a potential means for determining elevation, but is also useful - even when not able to determine elevation - as a measure of relative pressure across varying latitude/longitude and time.

Another very simple sensor in this realm is for temperature. IC temperature sensors are cheap and very easy to use, and so one was included in the design. Apart from being useful as a meteorological instrument, temperature sensors are useful for monitoring devices which may generate excessive heat, notably the battery when heavily loaded and the motor during periods of high throttling.

Military use of UAVs is currently dominated by surveillance applications, featuring high-resolution image recording capabilities. While our budget did not allow the use

of similarly powerful equipment, we discovered we could acquire cheap video cameras and still cameras. These would demonstrate the capability of the UAV for this application; better equipment could be substituted in future as budgeting allows.

During the early stages of the project we conducted several test flights - under manual control - with alternately a wireless video camera or a digital still camera attached. This proved that we could provide video and still image capabilities. While in these tests there was no microcontroller or image processing electronics to interface with the cameras, it showed that the plane was able to fly with the equipment attached, and that this was an exciting application that could be pursued later.

## **2.3 Commercial Applications**

The future outlook for commercial UAVs is bright, with an estimated US\$5 Billion dollar global market for the year 2010[4]. Currently the market is segmented into two broad categories: military and commercial. The military market - which currently makes up around 95% of the global UAV market - has provided much of the funding and research into UAV technologies over the last few decades. UAVs are used by defence organisations for things like surveillance, scouting, and for reusable communications and radar networks. Currently the defence market is well served by established aerospace companies such as Lockheed Martin, Northrop Grumman and Boeing.

In contrast, the commercial UAV market is an area which has minimal development, but which shows great potential, particularly in areas such as agriculture, meteorology and asset monitoring services. Companies developing UAVs for the commercial market include Bell, Aerosonde and Yamaha. The use of UAVs in the commercial market is still in its infancy, with many research aircraft being created but far fewer aircraft ready for deployment, or able to meet the needs of potential customers. We suggest that, given appropriate resources, a mature form of our project could be aimed towards the commercial market as a versatile flight platform for autonomous unmanned flight.

### **3. Design**

A prototyping methodology was adopted for this project, given its research nature. This prototyping approach necessitated the creation of modular components (both hardware and software), which could be easily reused as the design changed. Each module was designed to be tested and optimised in isolation, prior to integration into the main system. The following sections cover the design of each module, drawing from the previous section which covered the foundational research that this project was based on.

Note that what is presented here is the design as it was foreseen. The actual implementations in some cases differ.

#### **3.1 Hardware**

The hardware design of this project was governed by three important principles; miniaturisation, reliability and robustness. These principles impacted on our component selection and guided important design choices such as the hardware manual switchover implementation.

The principle of miniaturisation was vitally important, as the weight budget for the whole electronic system had an upper limit in the order of 100 grams (empirically based on the flight performance of the aircraft whilst carrying different loads). To ensure a lightweight and sufficiently miniature design, first preference went to surface mount component packages.

As this system is a real-time vehicle control system, reliability is a vitally important consideration. Reliability was the key factor in the selection of a hardware implementation of the manual switchover circuitry, so that in the event of software malfunction the pilot would still be able to regain control of the aircraft. Increased reliability was also the principle behind ensuring sufficient decoupling was provided for each of the digital components by means of 100nF bypass capacitors. The choice of latch-up-resistant and diode-clamped CMOS logic devices from the 74HC series recognised the high reliability of such devices, making them a particularly good choice for critical path circuitry.

Finally, the requirement of robustness comes as a physical consequence, since the UAV experiences turbulence in the air and potentially significant forces – particularly on landing. Outsourcing board fabrication was one design decision based heavily on the perceived robustness and quality of the different boards – particularly when using plated through-hole constructions. Another design decision based on robustness was the choice of the GPS receiver, which uses a proper screw mounting (as opposed to the tape or clip-in competitors), providing good mechanical rigidity as well as ensuring a good electrical connection to the surface mount header. Given the major requirements for UAVs as outlined in Section 2.2, the following subsections seek to address the hardware elements of these design requirements.

##### **3.1.1 Navigation**

As noted in the research section, our primary and sole navigational aid is GPS. GPS receivers allow a user to augment their position by reading data from several of the 24



GPS satellites which orbit the earth [18]. The accuracy of their output is dependant on the number of simultaneous satellites a receiver can detect, which is in turn dependant on the quality of the GPS receiver. Given this, and whilst considering the key hardware design principles, we selected the lightweight and robust ET-202 from GlobalSat.

One problem with most commercial GPS receivers is the relatively slow update rate (1Hz). Concerned that such a slow update rate would detract significantly from the ability of the autopilot to control the aircraft, the decision was made to attempt to augment the GPS data with some faster updating sensors. To provide more frequent bearing updates a magnetometer was chosen. Magnetometers provide an indication of true north, which can be aligned to grid north based on the magnetic delineation factors for the specific areas of operation (an addition of 10 degrees is applicable for Victoria). Through use of twin-axis accelerometers and a pressure sensor it was hoped that altitude could be updated more frequently.

Each of the sensors used in this project interface directly with the microprocessor, an Atmel ATmega2561. The Atmel 8-bit AVR line, and the ATmega2561 in particular, was chosen because it provides all the I/O functionality required and Robert had previous experience with the AVR family. A hardware multiplexing system was chosen for switching between manual and autopilot control, primarily for increased reliability.

### **3.1.2 Flight Control**

The flight controls provided on the aircraft consist of two micro servos (for the rudder and elevator) and a motor-speed controller for regulating throttle. The signals driving these flight controls were to be hardware multiplexed between the AVR autopilot generated values and the RC receiver flight controls, as part of the hardware manual override. Further, the fact that the method of control is switched in hardware necessitates the need for inputs to the microprocessor indicating the current state of operation.

The algorithms used to drive the flight controls are covered further in the software design section, with the only hardware consideration being for efficient PWM generation requiring minimum CPU resources. The ATmega2561 provides several 16-bit counters with multiple triggered pins based on the contents of specific output compare registers – providing an efficient method of PWM generation.

### **3.1.3 Sensors**

For temperature sensing the LM61 IC sensor was chosen for its robustness and ease of use - it has linear, calibrated output which can be interfaced to the AVR using the built-in ADC. The chosen mounting point was inside the aircraft, on the PCB. When fitted inside the aircraft the sensor sits toward the centre of the fuselage, midway between the motor and the battery.

The final additional sensor measurement was that of battery voltage. Battery voltage is an important variable as it can be used as an indication of remaining flight time. For simplicity, a simple resistor voltage divider was chosen to interface the battery voltage measurement with the microcontroller, via the ADC.

### **3.1.4 Data Logging and Communication**

In the initial design an inexpensive 433MHz transmitter and receiver pair were chosen to provide communications functionality. During prototyping and testing, however, these modules were found to be severely lacking, both in range and reliability. In need of a replacement wireless system, the Spark Fun Electronics BlueSMiRF v1 [19] module was selected. It provides Class 1 Bluetooth v1.2 support, with far better reliability, higher data rate support and a quoted range up to 100m. It also provides a simple RS-232 interface, with hardware flow control, which allows it to be used almost transparently.

For the onboard storage, as discussed in the research section, MMC/SDC was chosen. MMC/SDCs support at least three physical interfaces - MMC 1-wire, MMC 4-wire, or SPI. The decision was made to use SPI because the ATmega2561 includes an SPI port to manage byte I/O. At 16MHz the AVR can transfer data via SPI at up to 1 MB/s, which seemed quite fast enough for our purposes. While the MMC 4-wire interface in theory is four times faster than SPI, it would require a software bit-banging implementation, and as a result it was concluded that this was both unnecessary and likely to offer no better performance than SPI.

Additionally, the SPI variant of the MMC/SDC protocol is slightly simpler, and has less stringent requirements (e.g. CRCs are optional), thus making the implementation simpler and faster. It has reduced capabilities, but still meets our requirements.

## **3.2 Embedded Software**

It was clear that we should write the software in C, at least where possible, given its size and complexity. Robert already had some experience with ImageCraft's C Compiler for AVR, and nominated it as the compiler and IDE to use. The alternatives included the free WinAVR, utilising gcc, and other commercial compilers such as CodeVisionAVR, Atmel's own AVR Studio 4, etc. Since John Devlin had a license for ICC6 for AVR, the cost was waived. Unfortunately we required the latest version, 7.08, for ATmega2561 support, but the free trial version proved sufficient.

The approach taken was to develop from the bottom-up, implementing each module with a simple, minimal API, and then integrating and utilising those as the main body of the program was later developed. Numerous modules were required, such as:

- GPS
- Logging
- Magnetometer
- PID
- Sensors (ADC)
- Servo control

Many of these could be developed independently - the Logging system, for example, was developed exclusively independently and under PC simulation for much of the project, well before the hardware was assembled.

### 3.2.1 Navigation

A basic requirement for an autonomous vehicle is instruction. In the case of an aerial vehicle, there are several approaches to providing instruction. One approach is to use a simple procedural language to issue commands to the aircraft [20], e.g. "attain altitude of X metres", "turn left 45 degrees", etc. While a novel and potentially powerful approach, it requires a complex set of tools to parse command lists into some form of bytecode, an onboard interpreter to execute that code, and potentially complex development of the actual command lists in order to perform even basic tasks. While the ability to make arbitrary decisions during flight is very powerful, it seems unnecessary for most uses of UAVs.

In particular, the typical autonomous UAV application is envisaged as having the aircraft fly a pre-defined route, performing particular tasks at various times and points along the way (e.g. taking photographs). For these applications, the aircraft need only a list of co-ordinates to fly to in sequence, with some optional metadata indicating payload operations to be performed at those co-ordinates.

Thus, the decision was made to use a simple co-ordinate list as the aircraft's flight instruction. These co-ordinates can be specified as latitude, longitude and elevation above mean sea level. This integrates well with the output of GPS receivers. It also provides a clear distinction between navigation and flight control, whereby navigation is controlled by the flight plan, while the actual flight control - including homing in on the current navigational waypoint - allows the critical control logic to be implemented in C, where it can be reused and heavily tested.

The navigation system is consequently quite simple. It is given a list of waypoints to travel to in sequence. Using the aircraft's position as determined by the GPS and other sensors, it can compute the desired bearing and altitude. It can then instruct the flight control system to manoeuvre to attain that bearing and altitude. A waypoint is considered to have been reached once the plane is within a set Cartesian distance from the point.

When the list of waypoints is exhausted, the navigation algorithm can return to the first waypoint, or perhaps it's first recorded position (presumably the airfield from which it was launched), or some other well-defined location, and enter into some form of holding manoeuvre - e.g. circle the return waypoint indefinitely, awaiting further instruction or manual control handover.

### 3.2.2 Flight Control

The flight control system, as discussed in Section 2.2.2, has a very important and quite difficult task to perform - it needs to satisfy the basic physical requirements of flight - appropriate banking angles, aircraft pitch, etc - while responding to navigational requirements.

It is somewhat intuitive to adopt a procedural approach to this - perhaps use a finite state machine to perform pre-defined manoeuvres in sequence, e.g. "bank 20 degrees left for 1 second per 30 degrees turn required", "pitch up 10 degrees, hold for 1 second, return to neutral pitch, hold for 1 second per metre of climb required", etc. This is similar to the procedural command approach mentioned previously, and is possibly quite useful for simple computer control of many vehicles - e.g. cars.

However, for an aircraft it is very poor. The aircraft is subject to all manner of outside forces, and its actions do not have well defined reactions. For example, the time taken to perform any task may be dependent on the current groundspeed, airspeed and wind. There is substantial freedom of movement in an aircraft, which makes simple procedural control difficult, if even possible at all, to implement successfully.

Nonetheless, at the other extreme trying to account for any possible contingency simultaneously is extremely difficult. An algorithm which has to allow for any starting orientation of the plane, any altitude and any airspeed will inevitably be far too complex, and likely not to work in practice. A compromise is necessary.

Assuming well controlled flight, the state of the plane at any particular time is likely to be predictable and steady - it is either flying on a given bearing (possibly changing altitude as it does so), or banking to a new bearing. If we also assume that the flight control algorithms do not, under reasonable circumstances, put the plane into any other state (e.g. flipped over), we can start to significantly reduce the complexity of the flight control. If these assumptions are met, the only additional assumption that needs to be made is that the plane is not operated in conditions that it cannot then handle (e.g. high winds).

These assumptions formed the basis of the flight control design. While they do limit the flexibility of the plane - it cannot perform acrobatic manoeuvres - they do massively simplify the system, and the resulting UAV is still applicable to most tasks.

Given this, the design focused on two basic states - adjusting altitude and adjusting bearing. Priority is given to one or the other at different times, e.g. the bearing is adjusted until it is correct within some tolerance, and then control focuses on correcting altitude as required.

Lastly, but critically, the actual driving of the plane's control surfaces needs an algorithm that can incorporate feedback on the actual performance of the plane, given how widely it can vary, and that can smoothly adjust the control surfaces to home in on a desired bearing or altitude, with close to critical damping, and without unnecessarily rapid changes that could provoke instability.

A common algorithm used in control systems, as suggested by John Devlin, is a PID (Proportional, Integral, Derivative) algorithm. This family of algorithms are used extensively in control systems that rely on feedback for evaluation and correction[21, 22]. They are relatively simple algorithms, easily implemented on a microcontroller. There is also a substantial body of reference material dealing with them, particularly how to tune them, which significantly favoured their use.

A PID algorithm controls one variable, so with two control surfaces (rudder and elevator) two independent PIDs are required. Additionally, for motor control a third PID might be used. However, the control surfaces on the plane, and the motor, must be operated synergistically to achieve desired outcomes. For example, the elevator needs to incline slightly while banking with the rudder, in order to maintain altitude. Consequently, the PIDs cannot necessarily be entirely independent, and must instead interact, or at least be controlled together.

### 3.2.3 Sensors

The GPS receiver has a RS-232 interface for bidirectional communication - GPS strings are read from the receiver in the ASCII, human-readable NMEA format, and configuration commands can be transmitted to it.

The easiest approach is to read the GPS data into a buffer, and process it line at a time. The human-readable format needs to be converted to machine types. The decision was made early on to avoid floating point types, given they do not have hardware support on the AVR and are thus quite costly. There also appeared to be compiler bugs pertaining to floating point use, which exhibited themselves severely in simulation. Instead, all GPS readings are converted to integers or longs, in a suitable format, i.e. instead of the usual latitude as degrees, e.g. 43.287362, the latitude is stored as ten thousands of a minute, e.g. 25972417.

The magnetometer uses SPI, conflicting with the MMC/SDC interface, given that the ATmega2561 has only a single SPI port. Through the use of individual chip selects both devices can share the single SPI port, although the magnetometer's SPI interface can only run at up to 1MHz (versus 25MHz for the MMC/SDC), requiring reconfiguration of the SPI interface when swapping between the two.

All other sensors - pressure, temperature, accelerometers and battery voltage - have analogue outputs, and so are fed into the AVR's ADC. All of these are straight-forward, single-channel interfaces, except the pressure sensor, which has a differential output. This complicates ADC configuration slightly, but not untowardly.

Since only one channel (whether referenced or differential) can be read at a time, the ADC must be operated in stages, reading each sensor one after the other. While it's certainly possible to read certain analogue sensors more often than others, all can be read quite quickly, so a simple design was chosen, whereby in each cycle each sensor is read in turn, and at the end all the new results are logged and made available to the other systems.

### 3.2.4 MMC/SDC

There was initially some uncertainty as to what level of support we should implement for MMC/SDC. The official file system of MMC/SDCs is FAT16 (although in the real world FAT32 is also used). Supporting FAT16/32 would allow us to copy data back and forth between the UAV and a PC with minimal fuss, as normal folders and files. However, a FAT driver is not a trivial thing to write - the FAT file system format is poorly documented, ambiguously implemented, and heavily abused by 3rd parties. In addition, when this decision was being made the intended microcontroller was the ATmega128, with half the resources of the ATmega2561 - 4K of SRAM and 128K of Flash program memory. Fitting a FAT driver into 4K of SRAM is easy enough on it's own, but doing so while leaving enough space for all the other systems seemed less trivial.

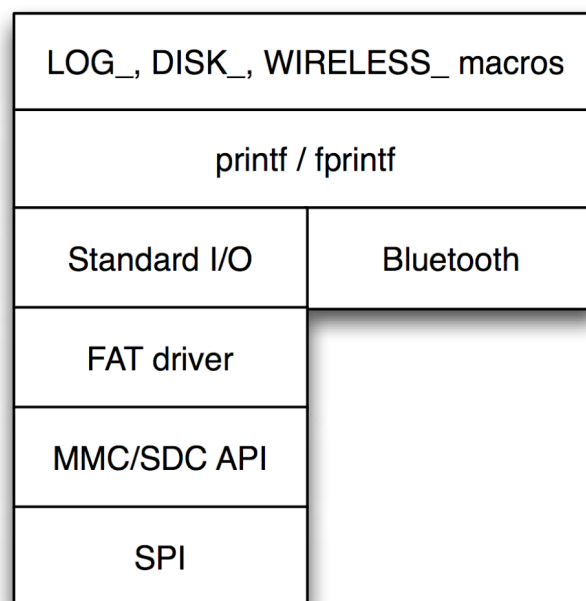
Other file systems, such as HFS or ext, were briefly considered, but are generally at least as complex as FAT, and are not supported under Windows.

One alternative considered was not to use a full file system, and either invent our own trivial one, or simply write data directly to the card as a single stream. This would

make the implementation on the AVR very simple. Unfortunately, while recorded data would be accessible on Linux or Mac OS X, it would not necessarily be easily manipulated. Furthermore, it would not be at all accessible under Windows without special software, whether written by us or acquired from a 3rd party. This was deemed unsatisfactory in any case, so the decision was ultimately made to support FAT16 (and, ideally, FAT32) on the AVR.

The implementation was presumed to take some time, and it was clearly not going to be wise to wait for the PCB to be designed and built before working on it. At the same time, it was a complex system that really needed testing throughout development - particularly given the aforementioned ambiguities in the FAT format, which required practical comparison with authoritative implementations (e.g. Windows & Mac OS X) to resolve. Thus, a somewhat creative and ultimately ingenious solution was found - write the FAT driver in a platform-agnostic manner, so that with the use of a minimal compatibility shim it could be compiled and tested on a PC (specifically, Mac OS X).

To facilitate this, the design was explicitly layered and modular. The hierarchy was envisioned as something like that shown in Figure 4, below.



**Figure 4: Logging Stack Design**

Under PC "simulation", as it was termed, the MMC/SDC layer (and below) would be replaced with an implementation that uses standard POSIX file APIs to interact with FAT16/32 disk images.

The fundamental unit of the logging system is the block; MMC/SDCs are divided at the hardware level into some number of blocks of a particular size - typically 512. On a properly formatted card, this block size matches the sector size used for the FAT file system. The whole I/O system is designed around blocks - it reads and writes a block at a time, caches zero or more blocks in SRAM at a time, operates on those blocks

one at a time, etc. At the higher levels an additional conceptual layer - Standard I/O - is added which is based on streams of arbitrary length, instead of blocks, as is traditional for userland file I/O and as is much more convenient for general use.

The highly layered approach also had the advantage that it allows additional file system formats to be supported, transparently to users of the system. While the intention was to support FAT16/32, there were concerns as to whether this would be completed in time. This design reserved the option of dropping FAT support and writing our own simple file system driver, with minimal changes to other code.

### **3.3 PC based software**

There was minimal software to be developed outside the onboard systems. The support for FAT16/32 MMC/SDCs would allow any suitable program to be used to read and write files used by the autopilot, such as the flight logs, flight plan and so forth. The PC software needed was for planning flights - creating waypoint lists - and if possible for plotting actual paths flown, as recorded in the flight logs.

#### **3.3.1 Flight Planning & Display**

As noted, the navigation system requires a list of waypoints to navigate between. While it's possible to manually construct such a list, it was decided early in the project to find or develop proper flight planning software. The ideal solution would be something like Google Earth - a 3D view of the Earth, including road maps, satellite imagery and ground elevation - with simple path creation capabilities. The ideal solution would in fact *be* Google Earth, were it not for the unfortunate limitation of Google Earth that you cannot modify the elevation of any particular point in a path - only the elevation of every point in the path simultaneously.

It seemed plausible a plug-in or hack could be developed for Google Earth to enable the desired functionality, but unfortunately the plug-in API is not officially documented or supported, and thus this approach is unpredictable.

Other existing packages offering some elements of the necessary functionality were already available, but none met all requirements, as simple as they were - to be inexpensive, to provide imagery of the Earth, and to provide the necessary path creation tools. Consequently, the decision was made to develop a custom solution.

While strictly speaking the custom solution need only display a 2D map of the Earth, it is obviously preferable to provide a 3D view of the flight path. Indeed, advanced features such as being able to fly through the flight plan were envisioned. The main priority, however, was in the essential flight planning functionality.

The Flight Planner application was developed in Cocoa on Mac OS X, given the relevant expertise of Wade Tregaskis, and the power of that development environment. It utilises industry-standard and portable OpenGL for 3D views.

Acquiring ground imagery was somewhat difficult. While it is possible to buy numerous massive databases of high resolution satellite imagery - and geographic information such as ground elevation, roads, borders, etc - it was seen as extremely

costly. There are also compatibility concerns between each of the many providers of such information; all tend to use their own proprietary formats.

At the same time, it is completely free and simple to access one particular database of imagery - Google's. In particular, Google Maps is an AJAX-based map viewer that operates in a standard web browser. More significantly, it has a public API and Google encourages its use by 3rd parties, including customisation. It has built-in capabilities for rendering paths, marking points of interest, and more.

Thus, it was trivial to include much of the necessary functionality within the application, utilising Google Maps operating in a WebKit view. WebKit is a framework built into Mac OS X that provides HTML rendering and JavaScript execution capabilities. It also integrates very well with Cocoa applications, allowing the application, written in Objective-C, to be intimately tied to the JavaScript within a WebView.

However, while it's easy to use Google Maps in this manner, to couple it with a 3D view of the Earth the map imagery itself needs to be extracted. This is not supported by Google, for obvious reasons; they don't want their imagery stolen. Much trial and error research went into discovering how to extract the map imagery from Google Maps. Several methods were found, documented and implemented.

The end design settled on a simple interface - four views, one of which would be a WebView containing Google Maps, the other three OpenGL views displaying the Earth and flight plans in 3D. All display the same data and reflect changes in each other. Multiple flight paths could be specified within each document, and manipulated in any of the views.

There were substantial difficulties implementing this, as discussed in Section 4.4.



## 4. Implementation

This section documents the implementation of the key modules which make up the project, and the problems that were encountered during implementation.

### 4.1 Airframe

The airframe used for the implementation of this project is a modified “ElectraFun XP” trainer aircraft, which typically retails in hobby stores for between \$125 and \$150. The ElectraFun XP measures 76cm in length and has a foam wing with a span of 104cm. The aircraft is powered by a Speed 360 electric motor driven by a Lithium Polymer battery (which resulted in a substantial improvement in weight and flight time over the standard NiMH battery). The original model was controlled by a 3 channel 27MHz receiver (for throttle, elevator and rudder), which was upgraded to a 5 channel Hitec Flash 5 radio system to provide additional channels for manual to autopilot switchover.

The total weight of the aircraft including the additional hardware is 513g, only 29g heavier than the stock aircraft, due mainly to the significant weight saving of 41g after upgrading to a LiPO battery. The aircraft is fitted out with micro-servos to control the rudder and elevator control surfaces, and a motor speed controller to control the throttle. The output for each of these control signals (throttle, elevator and rudder) is multiplexed between the receiver values (pilot control) and the autopilot values.

### 4.2 Hardware Implementation

Distinct from mechanical hardware (as discussed the previous section), this section discusses the electronic hardware implementation. The electronic hardware occupies a double-sided PCB measuring 10.2cm by 5cm (shown in Figure 5), a size governed predominately by the size of the fuselage. In addition, several components, notably the GPS, magnetometer and Bluetooth transmitter are mounted on parallel boards.

The board is directly interfaced with the 7.4V LiPO battery and uses two linear regulators to provide 5V and 3.3V supply voltages for the various components on the board. Unused space on both sides of the board is allocated as ground planes, which are extensively stitched together using vias. The centrepiece of the board is the ATmega2561, which we run at the maximum frequency of 16MHz using an external full-swing oscillator.

Several pushbuttons were also included on the board to provide useful functionality such as arming and I/O buffer flushing.



The following sections describe in detail the various hardware functions that have been provided, including the hardware manual switchover circuitry, the various sensors, and communications equipment.

#### **4.2.1 Photo/Video Capability**

To provide a brief proof-of-concept of the aerial surveillance capabilities of UAVs, the aircraft was fitted out with both a 2MP digital still camera and, at alternating times, a miniature wireless video camera. The video camera was coupled with a small 1.2GHz 200mW wireless transmitter which allowed aerial video to be received and recorded on the ground.



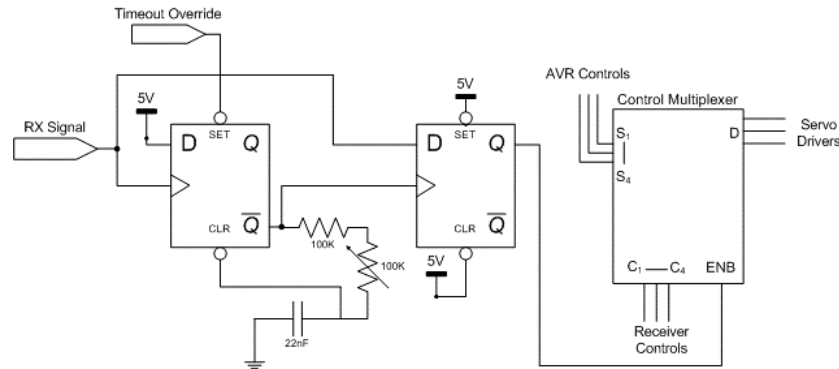
**Figure 6: Photo taken from onboard UAV**

The 2MP Dolphin digital camera was connected to a 555 Timer circuit and configured to take still photographs at 16 second intervals. A circuit similar to the manual switchover circuit could have been used to control the camera shutter, but the timing circuit was chosen for simplicity. The still photographs were stored in the camera's internal memory. A logical extension to this camera setup would be to use a camera module to interface with the microprocessor so that images could be stored on the MMC/SDC, and select images could possibly be transmitted over Bluetooth.

#### **4.2.2 Manual Switchover Circuitry**

As discussed in the design section, for safety reasons the switchover between autopilot to manual control was a function to be completed in hardware. Signals from the radio receiver and the microprocessor are multiplexed to the appropriate control surfaces. In the event of software failure, this should ensure that pilot on the ground will be able to switch over to manual control and pilot the aircraft – both quickly and safely. Control of the switch is assigned to channel 5 on the radio controller. All the channels on the radio controller produce a PWM signal with a period of 20ms and a pulse 'high' time between 1.25ms and 1.75ms. In hardware, the selection between manual and autopilot control is based on the width of the pulses.

To control the multiplexer switching input a fixed pulse of 1.5ms is locally generated onboard the aircraft, using a RC circuit. Using two D-Latches as shown in Figure 7 the pulse received from the radio receiver is compared to this artificially generated pulse. If the receiver pulse is wider than the RC generated pulse, the multiplexer is switched to manual pilot mode; otherwise it is set to autopilot mode.



**Figure 7: Manual Override Circuitry**

One further scenario needs to be considered: if the receiver loses the signal from the transmitter (e.g. transmitter dies or is out of range). Using the scheme listed above the future state of the aircraft would be the same as a current state (possibly in manual control when the pilot is unable to control the aircraft). To rectify this problem a 555 timer circuit was incorporated as a missing pulse detector. If no pulses are received for a period of 100ms (equivalent to missing 5 pulses), the circuit is forced into autopilot mode and a signal is sent to the microprocessor via an interrupt pin, warning that the manual control signal has been lost. In this condition the plane is programmed to enter a ‘mayday’ mode, where throttle is reduced and an emergency waypoint – somewhere safe and accessible - is set as the next course to fly towards.

### 4.2.3 Sensors

As discussed in the design section, the flight control system interfaces with a total of six different sensing devices using three different communications interfaces. The sensors can be divided into three categories: navigational sensors, meteorological sensors and general sensors.

#### Navigation Sensors

The sensors that could be described as navigation sensors include the GPS receiver, magnetometer and accelerometer. The GPS receiver is the primary navigation sensor and is used for most of the onboard flight control and data acquisition. The GPS is interfaced with UART0 on the ATmega2561, where the TX connection on the GPS is buffered and pulled up to meet the UART voltage ranges specified for the ATmega2561. An active external antenna is connected to the MMCX connector on the GPS receiver and is mounted forward of the wings at the top of the fuselage to maximise satellite reception. During the start-up routine, the GPS receiver is configured to transmit GPRMC and GPGGA ASCII strings at an update rate of 1Hz. These strings provide: UTC Time, Latitude, Longitude, Speed, Course, Date and Altitude. To make effective use of board space, the GPS receiver is mounted overlapping the SD card holder and the interface pins to the radio receiver.

The MicroMag2 magnetometer, which was included in an attempt to augment the GPS bearings (particularly at low ground speed), connects to the ATmega2561 using the SPI interface. Since the magnetometer is a 3.3V device, all signals from the AVR to the magnetometer (reset, chip select, SPI data out and SPI clock) are first passed through a voltage divider consisting of 1.8K and 3.3K resistors, to drop the 5V ‘high’ signal down to the order of 3.25V. The magnetometer has two axes (X and Y) which

are separately sampled (initiated using SPI commands) from which a bearing can be calculated in software. To save space the magnetometer is mounted at the front of the PCB (the furthest point from the electric motor) on a separate carrier board directly above the programming headers and accelerometer.

The accelerometer used in this project is the 1.5g MMA6260Q Freescale dual-axis accelerometer. Unfortunately the package for the accelerometer is a leadless QFN, which is difficult to hand solder, resulting in unreliable operation at the best of times (even after hot air soldering). Similar to the magnetometer, the accelerometer operates with a 3.3V supply and uses a voltage divider on the self-test control input. The accelerometer has two analogue outputs (X and Y), which are connected to ADC inputs PF4 and PF3 respectively.

### **Meteorological**

Meteorology refers to the scientific study of the atmosphere with a focus on weather processes, particularly for forecasting[17]. Meteorological sensors therefore make specific measurements of atmospheric conditions. For this project two simple meteorological sensors were interfaced; a pressure sensor and a temperature sensor.

Another Freescale device, the MPXM2202 pressure sensor was implemented as the first meteorological device. This sensor is piezoresistive based, providing a linear analogue output voltage proportional to the applied pressure. The MPXM2202 is powered from the 5V rail and is connected to the ADC inputs PF0 and PF1 in differential mode, with a lower voltage reference of 2.56V to improve measurement resolution. Unfortunately, this turned out to be insufficient; the differential signal needs to be pre-amplified before being sampled by the ADC.

The LM61 IC temperature sensor comes in a small SOT-23 package which makes good use of the limited board space. The LM61 is powered from the 5V supply rail and is flanked with a 100nF decoupling capacitor (as is each of the other digital components). The output from the LM61 is an analogue output, linearly proportional to the fuselage temperature, which is directly connected to the PF2 ADC input on the ATmega2561.

### **General Sensors**

From the final class of sensors, only a simple battery sensor was implemented. The battery sensor was configured using a simple matched voltage divider consisting of two 22K resistors to reduce the 7.4V LiPO battery voltage to within the 0 to 5V range. The comparatively low resistor values were used since the ADC input impedance is in the order of 10K, relatively close to the 11K calculated from the resistors of a Thevenin equivalent circuit.

#### **4.2.4 Bluetooth Data Link**

The BlueSMiRF Bluetooth module connects directly to UART1 on the ATmega2561. It provides a transparent bidirectional link between the AVR and whatever terminal device connects to the module via Bluetooth - typically a PC running terminal software. The current implementation is only used for logging from the UAV, and ignores incoming data.

There was significant difficulty in getting the BlueSMiRF module to work correctly. The first problem was that the so-called datasheet provides very little actual data about the module. For example, it shows on its schematics the CTS & RTS pins as active high, yet they are active low (it does say this, in some small print). It also has them back-the-front, which of course isn't much good. Once these problems were diagnosed and fixed, the implementation actually worked most of the time.

Only "most" of the time, because it was discovered that the Bluetooth module could easily be caused to hang by simply sending data to it too fast from the AVR. The only workaround found for this was to place the BlueSMiRF module into "fast" mode before sending too much data to it. In "fast" mode the hangs were far less frequent.

Another problem discovered was that if too much data is sent to the module while there is no Bluetooth connection - even if only slowly - it will fill up the module's internal buffer and the module, again, hangs. The only way to then reset the module is by power-cycling it, which in our design requires power-cycling the entire board. The datasheet does warn about this, but given there is no way to determine if there is a connection or not, it is difficult to avoid. Ultimately the introduction of the Arm button helped work around this problem, by ensuring we had time to connect to the Bluetooth module before outputting any data to it.

### **4.3 Embedded Software Architecture**

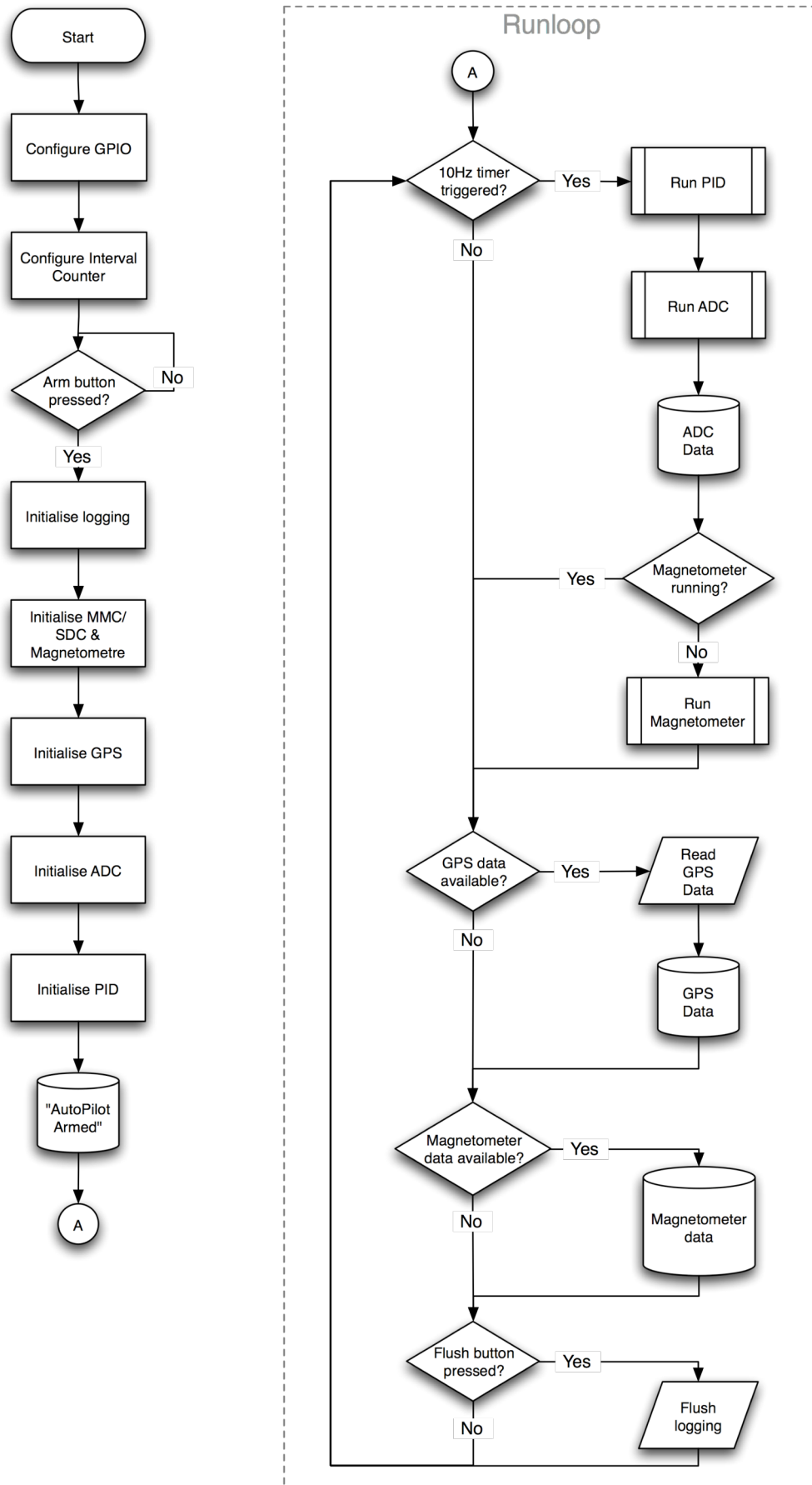
The software was implemented pretty much as designed, although many of the finer design details were left to trial and error to decide, and some changes were made as hardware problems were encountered, or performance issues discovered - e.g. CRCs for MMC/SDC data transfers.

As intended, ICC7 for AVR was used as the IDE and compiler for the project. Unfortunately, because the ATmega2561 was so new at the time we chose it, and represented a significant step up from previous AVRs (> 128K of program memory), compiler support was initially poor. Luckily ImageCraft released an updated version (7.08) just in time, which resolved the critical compiler bugs relating to the ATmega2561.

#### **4.3.1 Runloop**

The event-handling methodology chosen is that of the runloop - an infinite loop which essentially checks for inputs (e.g. new data from the GPS receiver) and then performs the appropriate action. Figure 8, next page, illustrates the main() function, which contains the initialisation process and runloop body.

Most of the runloop's inputs are interrupt-based, such as the arrival of new data from the GPS receiver, completion of an ADC conversion, the 10Hz PID timer, etc. The interrupts for those events set global flags, which are checked each time through the runloop. The runloop also polls for some events, such as indication of completion from the magnetometer.



**Figure 8: Main Runloop**

The advantage of this runloop architecture is that it provides a simple method for handling interrupt-driven events outside interrupt-time, and with priority. For example, the 10Hz timer has highest priority - it is checked first, and its actions executed first if necessary.

It is very important to be able to handle interrupt-based events outside interrupt time, for two main reasons:

- Firstly, many of the core systems cannot be used at interrupt-time. For example, the Bluetooth logging system can optionally use the interrupt-driven UART transmission system, which would not work properly if invoked from an existing interrupt.
- Secondly, many operations - such as logging - can take a substantial amount of time to perform. Critical systems, such as flight control, should not be blocked from execution for any significant period.

### **4.3.2 Interrupt-based I/O**

To ensure that time is not wasted polling various I/O devices, interrupts are used to handle the ADC conversions, UART data transfer and the output drive PWM signals for the servos. An interrupt-based 10Hz timer is also used to schedule period events, such as execution of the PID algorithms, initiation of ADC conversions, etc. Once the timer initiates these, through a series of global flags, the ADC “conversion complete” interrupts are used to step through each of the required conversions for a full conversions cycle. Likewise, the PID algorithms and magnetometer sampling are triggered by the timer.

Both of the UARTs are configured in a circular buffered arrangement, making use of the RX and DRE interrupts to facilitate efficient data transfer. UART0 is connected to the GPS receiver, and so will typically only receive approximately 140 characters (GPRMC and GPGGA strings) at one second intervals. With this irregularity in updates, interrupts proved to be a good choice for UART management. When the ‘CRLF’ end-of-line characters are detected a flag was set indicating that a complete GPS data string is now in the buffer and ready to be interpreted.

Conversely, UART1 connects to the Bluetooth module and is used solely for transmission. It is written to a line at a time, which is copied into the buffer, and then execution continues. The buffer is gradually emptied asynchronously as the UART transmits its contents byte by byte, using a “transmitter ready” interrupt.

### **4.3.3 Flight Control Algorithm**

As introduced in the design section, PID controllers are used to fly the aircraft in autopilot mode. PID controllers manipulate a control variable (in this case an aircraft control surface) by applying three mathematical functions to the current error deviance within the system (the difference between the desired value and the current value). These mathematical functions - covering the three elements of the PID; proportional, integral and derivative - are summed with their associated gains to provide an output, which is scaled and thresholded before driving the specific control

surfaces. As discussed in the previous section, a 10Hz timer is used to trigger the PID updates.

```
pValue = pid->pGain * error; //Compute proportional part

pid->iState += error;          //Sum error
if (pid->iState > pid->iMax){    //Threshold error to prevent
runaway
    pid->iState = pid->iMax;
}
if (pid->iState < pid->iMin){
    pid->iState = pid->iMin;
}
iValue = pid->iGain * pid->iState; //Compute integral part

dValue = pid->dGain * (currValue - pid->dState); //Compute Derivative
pid->dState = currValue;

return ((pid->decoupledGain * pid->ageGain * (pValue + iValue +
dValue)) >> 8);
```

The following parameters are used by the bearing PID:

```
pidBearing->pGain = 6;
pidBearing->iGain = 1;
pidBearing->iState = 0;
pidBearing->iMax = 10; //Threshold Value
pidBearing->iMin = -10; //Threshold Value
pidBearing->dGain = -8;
pidBearing->dState = 0;
```

These values may require further tuning to ensure consistent PID operation. Values have not yet been tuned for the altitude PID, but it is postulated that similar values would be used since the actual control output (i.e. a servo) is the same for the rudder and elevator.

The primary control surfaces (rudder and elevator) operate under decoupled control, where a separate PID is used to control each of the control surfaces. Using decoupled control both greatly simplifies the individual controller designs and mirrors some of the physical constraints (e.g. during a bank lift is reduced, which makes climbing more difficult). For simplicity the throttle, which is generally considered to be an ancillary control, is set to maximum for the normal autopilot case and minimum (no throttle) for the emergency mayday case (in order to slow down the aircraft and avoid injury caused by the propeller). Apart from the specific gains on each of the PID terms, an overall gain to each PID is assigned to assist with the decoupled control separation. An example of this would be the case where the elevator is the primary control surface: here the elevator gain is increased and the rudder gain is decreased.

Whenever the aircraft is switched into autopilot or begins to track a new waypoint, a 4 step algorithm is followed.

1. Firstly, an initialisation step ensures that all control surfaces are set as neutral to ensure smooth transition between manual and autopilot modes.



2. Through decoupled control the autopilot acquires the bearing for the next waypoint and steers the plane to track this bearing. To track the bearing a gain of 7 is assigned to the bearing PID and 2 is assigned to the altitude PID.
3. Once the bearing is being tracked (with error of less than 5 degrees for five consecutive cycles), focus is given to the altitude. For altitude tracking a gain of 8 is assigned to the altitude PID and 2 is assigned to the bearing PID.
4. After several consecutive cycles with low tracking error in the altitude (less than 8m, due to GPS constraints), the altitude and bearing PIDs are set to relatively low values – typically 3 to 4. They remain at these values to allow fine-tuning of the aircraft as it navigates towards the desired waypoint. When the aircraft reaches the waypoint, the process repeats from step 2.

Since the update rates from the GPS receiver are relatively slow (1Hz) an age gain has been implemented to reflect the increasing uncertainty of results between measurements. Each time GPS data is read, the ageGain variable is set to its maximum value – reflecting the accuracy and freshness of the current data. Each successive PID operation decrements the ageGain variable, so that each successive operation has a smaller effect – recognising that the data less accurately reflects the current state of the aircraft. After the separate gain terms are applied the outputs are assigned to the various output compare registers (OC3A, OC3B, OC3C) which in turn control the pulse widths for the various servos and motor speed control.

#### 4.3.4 Servo Control

As introduced in section 2.2.2, servos are PWM driven, with a period of 20ms and a pulse ‘high’ time of between 1.25ms and 1.75ms[12-14]. To generate these signals a 16-bit counter in the ATmega2561 is used in phase and frequency corrected PWM mode. In this mode the counter runs as dual slope counter, counting up to the TOP value (set to 2500) and then back down to zero. The three output compare registers (OC3A, OC3B and OC3C) each hold the current trigger threshold for the different PWM channels (throttle, rudder and elevator). Each trigger value is associated with a particular GPIO pin. When the counter is equal to the output compare value on an upward count the associated pin is set to ‘0’ (as shown in Figure 9). Conversely, pins are set to ‘1’ when their associated output compare register is equal to the counter on a downward count.

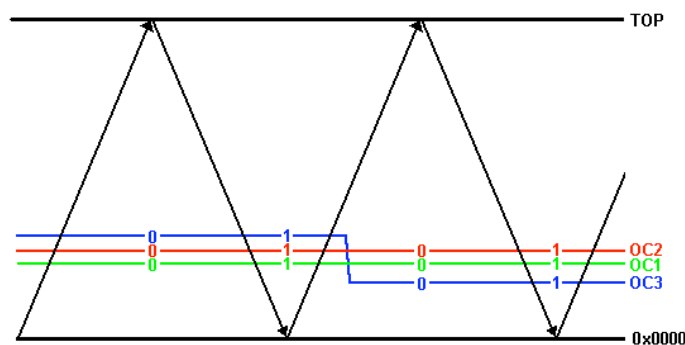
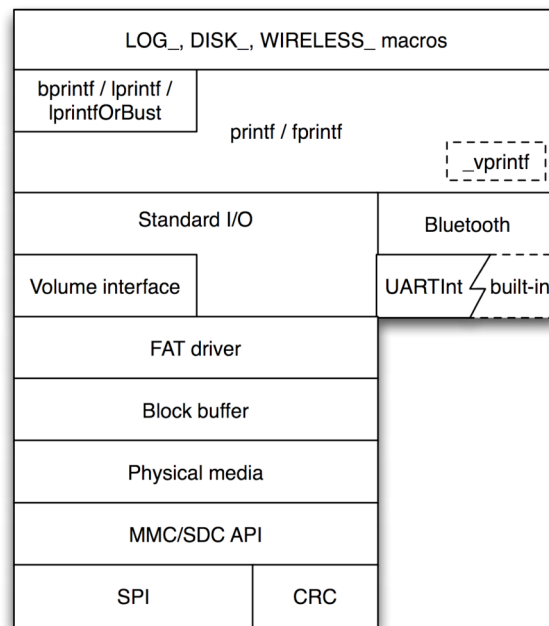


Figure 9: PWM output scheme

### 4.3.5 Data Logging

The data logging hierarchy is shown in full in Figure 10, next page.

There are two forms of data logging - logging to onboard storage, and logging via the Bluetooth wireless connection. These are exposed to the rest of the system as `fprintf` and `printf`, respectively. This simple, familiar interface makes logging trivial to use, and greatly aided debugging.



**Figure 10: Logging Stack**

### LOG macros

On top of `printf/fprintf` are a variety of macros which have greater semantic value. These are:

- `LOG_ERROR`: definite errors. These messages are typically logged first to Bluetooth, then to onboard storage.
- `LOG_WARNING`: possible errors, or irregular behaviour. These messages are typically logged first to Bluetooth, then to onboard storage.
- `LOG`: general operational messages. These messages are typically logged only to onboard storage.
- `LOG_DEBUG`: debug messages. These messages are typically logged only to onboard storage, and only in debug builds; in release builds they do nothing.

There are variants of each of these with a suffix of "2", which take one or more parameters after the main argument, the format string. This distinction is necessary due to the limited pre-processor used by ICC7 for AVR, which has no way to express a macro which takes zero or more variable arguments.

Additionally, there are two other families of logging macros, which use the prefixes "DISK\_" and "WIRELESS\_" instead of "LOG\_". These are for use by the onboard storage systems and the Bluetooth systems, respectively. They ensure that they do not perform any action which would result in recursion into the calling code - i.e. the DISK\_ family of logging macros will never log to onboard storage, as that could lead to an infinite loop. Additionally, the disk and wireless systems are not re-entrant.

The actual destination for messages logged to the onboard storage is set by calling the setLogFilePath function. If a destination is not set, messages that would go to onboard storage are sent to Bluetooth if possible, or failing that are lost. The main function sets the default log file, using this function, to "/Flight.log" after initialising the logging system.

### **printf/fprintf**

At the heart of the logging system are the printf and fprintf functions, which print to Bluetooth or to a specified file, respectively. Underneath, all logging functions ultimately call one or other of vprintf or vfprintf. These configure the output destination, whether Bluetooth or a file, and then call \_vprintf. \_vprintf performs the actual parsing of the format string and subsequent output. It calls the putchar function to output each character in turn. Depending on how \_vprintf is configured, putchar routes the character to the Bluetooth module or a file.

The choice to write our own version of printf, rather than use the built-in version provided with ICC7 for AVR, was somewhat arbitrary, somewhat accidental, and ultimately a good one. Initially the intention with logging was simply to output simple static strings to Bluetooth, primarily for debugging. It then became apparent that it would be very useful to be able to output program data (i.e. values of variables), in a human-readable format. So, basic formatter functionality was added. Initially the requirements were minimal, so there was no justification for using the rather large built-in printf. Over time, more capabilities were added. And once basic I/O for the onboard storage was functional, it was of course desirable to be able to fprintf to it.

Thus, the final implementation has ended up being very powerful and flexible – more-so than the built-in version, in fact. The built-in version is most likely more efficient, but we now rely on formatters and other features which it doesn't have, making it non-trivial to switch to.

### **Bluetooth**

putchar can output to Bluetooth in one of two different ways, as controlled by the USE\_HARDWARE\_FLOW\_CONTROL\_FOR\_BLUETOOTH define. If defined as true at compile time, the built-in Bluetooth interface is used. This uses the hardware flow control - CTS & RTS - provided by the BlueSMiRF module. It operates synchronously, blocking while the BlueSMiRF module indicates it is not ready, or while the previous character is still being transmitted by the UART.

The alternative mode uses the interrupt-based UARTInt module, which allows putchar to place each character into a buffer and return immediately. When the

UART is ready to transmit, it generates an interrupt which actually copies the next byte into the UART's buffer.

Both approaches were explored as it was felt that it might be more efficient and suitable to use the interrupt-based approach. The BlueSMiRF module did not seem to require hardware flow control; it typically always indicated it was ready to receive data. Thus, both approaches are functionally equivalent.

While the interrupt-based version requires a few dozen clock cycles for each interrupt (and thus each byte), in addition to the cost of placing the data into the buffer to start with, at maximum transmit rate to the BlueSMiRF module (250,000 baud) it takes 640 clock cycles on the AVR to transmit each byte. Thus, the blocking method may be wasting a significant number of cycles waiting, depending on how long it takes `_vprintf` to process each character. It is also better to take a little time at regular intervals, than block for a potentially long time doing everything all at once.

## **Standard I/O**

The Standard I/O API provides a fairly straight-forward file API. It provides a handful of functions for working with files - `openFile/closeFile`, `seekInFile`, `setFileLength` and `readFromFile/writeToFile`.

Files are identified by their path, using '/' to separate path components. The standard I/O API uses 32-bit types, to support files up to 4 GB in length (provided the underlying file system and storage medium supports such sizes).

Under the covers the Standard I/O module merely translates between the user and the underlying file system drivers. It performs generic parameter checking, converts the opaque types (e.g. `file_t`) to the underlying types (e.g. `struct File*`), and so forth. It determines which logical volume the file resides upon, and then invokes the appropriate driver for that volume to perform the actual file operation.

## **Volume interface**

Ideally it would be up to the "user" (code outside the logging system) to configure the logical volumes of the system. Since only one volume is currently needed, this is currently done automatically by Standard I/O. Nonetheless, the interface exists to do this explicitly and manually. The Volume interface module provides two functions for manipulating volumes - `createVolume` & `destroyVolume`.

`createVolume` takes as parameters the logical address and size of the volume, relative to the underlying storage medium, and attempts to find a driver that recognises the data within. It does this by iterating over all known drivers (as specified at compile-time in an internal data structure) and asking them one by one if they can read the volume. If a driver indicates it can, the appropriate internal structures are setup, the driver fully initialised for the new volume, and an opaque handle for the new volume returned.

`destroyVolume` simply attempts to remove all records of the given volume, provided there are no open files referencing it, etc. It is not currently used, but is provided for completeness.

## **FAT driver**

The bulk of the onboard storage system is contained within the FAT driver. It is a unified FAT driver, supporting FAT16 and FAT32. FAT12 is also provisioned for, although was not supported because there is no need presently.

In order to understand the driver, a basic understanding of the FAT file system format is necessary. Refer to appendix E for a basic summary. The step by step process by which the driver performs its work is too intricate to be detailed here; instead only the key attributes will be noted.

The FAT driver interacts with the actual storage via the block buffer, which divides the disk up into regularly sized blocks (typically 512 bytes). Its basic access pattern for retrieving data is to calculate the address of the required data, load the block which corresponds to that address, and then manipulate that block as necessary. Many structures within the FAT file system, primarily files themselves, may take up more than one block. In such cases, the FAT driver iteratively processes the blocks one at a time. For example, to write 1000 bytes to a typical FAT16 volume (with a 512-byte sector size) will require at least two blocks of the file to be written.

One self-imposed restriction of this implementation was that it should not use more than 1 block at a time, to allow it to operate within the absolute minimum amount of memory possible. This is not especially difficult to do, but is laborious and, if not balanced by other optimisations, does incur a significant performance penalty. In our case, use of the caching block buffer layer largely negates the performance penalty.

To augment the standard Volume and File structs, the FAT driver maintains its own list of FATVolume and FATFile structs, the pointer to which is stored in the driverData field of the Volume and File structs. FATVolume and FATFile also point to their parent Volume and File, respectively. This can be considered a poor-man's subclassing.

The FATVolume struct contains all the necessary information from the FAT boot sector - things like the sector size, start of FAT, start of data, sectors per cluster, etc. It is referenced frequently when performing operations on files, to retrieve key information.

The FATFile struct contains FAT-specific file information, and caches certain useful information, such as the cluster # containing the current file pointer, and the direct logical address of the file's directory entry (which is used frequently when writing to the file, to update the file's size as recorded in the directory entry).

The current implementation, while quite functional and thus far completely reliable, has significant room for improvement and optimisation. Section 6.2 details some of the perceived problems with the current implementation.

## **Block buffer**

The block buffer module's primary purposes is to manage blocks in memory - keeping track of the address of each block, whether it has been modified in memory, etc. Its secondary purpose is to cache blocks in memory and thus optimise MMC/SDC I/O. The module maintains its data internally using the Block struct, and provides to the

layers above an opaque reference to a block, which can be used to retrieve the block's data, mark the block as modified, free it, etc. The Block struct is defined as:

```
typedef struct {
    uint32_t logicalAddress;
    unsigned char referenceCount;
    unsigned char age;
    int isDirty:1;
    uint8_t data[BLOCK_SIZE];
} Block;
```

The logicalAddress and data fields are self-explanatory. The others are worth examining.

Blocks are reference counted, meaning each call to allocateBlock increments the reference count of that block, which must (at some time later) be decremented by a balancing call to freeBlock. The intention for this is to allow for recursive algorithms, and to allow flexibility with regards to the block size.

When a block is freed for the last time (i.e. its reference count decrements to zero) it becomes eligible to be reused for a different address. However, by default it remains in memory, so that if it is reused again it can be resurrected, and the data need not be read from the MMC/SDC again.

Blocks also have an age, which indicates how long since they were last used. Calls to allocateBlock, blockData, rereadBlock & markBlockAsModified set the age to 0. On every call to allocateBlock, every other block also has its age incremented. The age of the block is taken into account when determining which unused block to evict when it comes time to read in an uncached block; older blocks are evicted first. The replacement algorithm is thus an approximation of LRU (Least Recently Used).

Performance tests of the block cache show that even with only a small cache - e.g. 8 blocks - it reduces the number of MMC/SDC I/O operations by as much as two orders of magnitude.

The last field, isDirty, indicates whether or not the block's data has been modified in memory and not yet written back to the MMC/SDC. The initial implementation did not have this; whenever a block was modified in memory, it was immediately written back. This resulted in many more writes than strictly necessary, since the typical usage pattern for our purposes is to write one line at a time to storage, where each line is typically around 60-80 bytes; this resulted in seven or eight writes per block, instead of one. When it was realised that the logging performance was sub par, the behaviour was changed to delay writing back to the MMC/SDC for as long as possible. When the block is modified in memory, it is marked as dirty. The block is not actually written back to the MMC/SDC until it is replaced in the cache, or an explicit flush is triggered. For a sufficiently large cache (e.g. 8 blocks) this closely approximates optimal behaviour.

Additionally, the replacement algorithm favours clean blocks for eviction over dirty ones, on the assumption that blocks that are written to are likely to be written to again in the near future. For our usage pattern this is an accurate assumption.

A negative side-effect of this caching scheme, however, is that modifications are not necessarily written back to the card with any regularity, or in any bounded time. Thus, it is critically important to explicitly flush the block cache prior to removing the MMC/SDC. This was resolved using the "arm" button - if held down it will cause a full flush at the end of each iteration of the main runloop. Unfortunately, there is still a race condition in doing this, as the system does not stop using the card even when the button is pressed, so it is still possible the card may be removed midway through a write. Luckily, while this may cause the data being written to be lost, it is unlikely to corrupt any data already written and flushed to the card.

## **Physical media**

The Physical media layer is meant to abstract away the details of reading from and writing to the underlying medium - MMC/SDC in the current implementation. It defines the block size as used by the layers above, and can potentially modify it (depending on whether the MMC/SDC allows a different block size).

Since the only physical media supported are MMC/SDC, and they don't have any particularly onerous requirements, this layer is largely redundant.

## **MMC/SDC API**

This layer handles the actual MMC/SDC protocol. It implements the logic to send commands to the MMC/SDC, to interpret the response, and to perform data I/O. The MMC protocol over SPI uses a 6-byte command packet, which has a header and trailer, a 7-bit CRC of the whole packet, a 6-bit command code, and room for up to 32 bits of parameter data. Responses are one or more bytes; the first is typically a status byte indicating the general success or otherwise of the command, while other bytes may follow for particular commands (e.g. data I/O).

The SDPerformCommand function sends a command and returns the response. Sending a command is relatively straight-forward, but relatively lengthy. First, 8 SPI clocks must first be provided, to ensure the card has enough time to respond to its chip select being asserted. Then the 6-byte command packet is sent. Then up to 64 clock cycles may need to be given to the card before it returns the response byte. At the end of each command sequence (which, for data I/O, is more than just a simple command and response), the card must be disabled (by deselecting it's chip select) and an additional 8 SPI clocks provided to allow it time to finish up it's work.

Initialising the MMC/SDC is surprisingly complex and error-prone, and took a significant amount of trial and error to perfect. Initialisation is the only point at which MMCs and SDCs differ - the SDC format is a successor to MMC, and while they support the same basic protocol, they require a different initialisation command to be used, supposedly so that they are distinguishable from MMCs. Thus, the process requires that an SDC initialise be attempted first. If it fails, an MMC initialise is tried instead. So long as one or the other succeeds, the card is then ready for use.

In full, then, the initialisation process is:

1. Provide at least 128 SPI clocks and wait at least 1ms, to allow the card to boot up.
2. Issue CMD0 ("go idle") until the card responds that it is ready and idle. This should not take more than 100ms, so to handle the case where no card is inserted a timeout must be included.
3. Issue ACMD41 (CMD55 followed by CMD41 - "SD initialise") until the card responds that the command was successful and it is no longer idle. This should not take more than 500ms, so to handle the case where an SDC is not inserted, a timeout must be included.
4. If the previous command failed, issue CMD1 ("MMC initialise") until the card responds that the command was successful and it is no longer idle. This should not take longer than 500ms, so to handle the case where no valid card is inserted, a timeout must be included.
5. Provided either of steps 3 or 4 were successful, the card is now initialised and ready for use.

Once it is initialised, functionality which is conveniently wrapped up into the single, simple function `SDInitialise`, the MMC/SDC layer can be used. The most common functions are `SDReadBlock` and `SDWriteBlock`.

The MMC protocol uses a 7-bit CRC for command packets, as noted, and a variation of the CCITT 16-bit CRC (also known as the Xmodem CRC) for data transfers. The SPI version of the protocol does not require CRCs to be used for any command other than CMD0 (which has a fixed format and thus a well-defined CRC, which can be hardcoded for simple implementations). While our implementation does fully support the use of CRCs, their computation is extremely expensive on the AVR, relative to the cost of the actual I/O itself, and so they are disabled by default. They can be enabled by defining `USE_CRC_FOR_TRANSFERS` to be true at compile time.

## **SPI**

The SPI layer is very trivial - there is the `SPIInit` function to configure the SPI port, `SPIWrite` and `SPIRead` for the SPI I/O, and `setSPIDivider/getSPIDivider` for managing the speed of the SPI port (necessary because the magnetometer is limited to a 1MHz SPI clock, while the MMC/SDC can operate at up to 25MHz).

Nonetheless, there was a surprising amount of difficulty in getting SPI to work properly. It was found that the SPI CS needs to be asserted for the duration of an SPI read or write. If it changes during a read or write, a "write collision" error occurs, and the SPI port disables itself! This was a most perplexing intermittent problem for a long time, as we used GPIO pins for the magnetometer and MMC/SDC chip selects, so we were not explicitly configuring the SPI CS. Thus, it was an input by default, which meant it was floating. So, SPI operations would randomly fail. Once this problem was resolved, SPI worked fine.

## **CRC**

Two CRCs are required for our system at present - Xmodem CRC and CRC7, both for the MMC/SDC protocol. Wade had previously implemented a 16-bit CRC for CSE31NET, so the code from that was used as the basis for the AVR implementation.



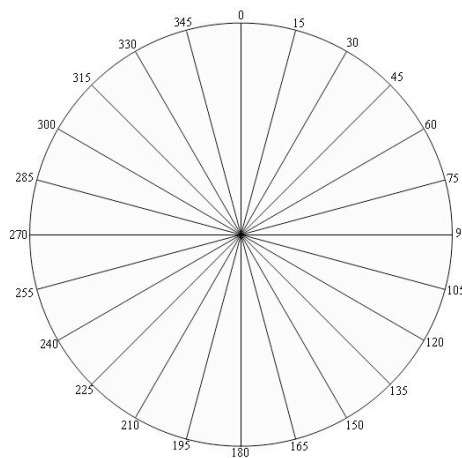
The CRC7 was then derived from the CRC16 implementation, which was a little tricky given the peculiar alignment (7 bit) of each "word".

Both implementations are correct and work fine on the AVR. However, they were found to be very slow. This was largely because ICC7 for AVR is very simple and produces very obese and suboptimal assembly code. It was also partly because the implementation was written to be simple and easy to read in C, without any optimisations. While some improvements were made, testing showed CRCs were simply taking too long to calculate - particularly the Xmodem CRC for the MMC/SDC data I/O, which was taking about 30ms (versus around 2.5ms for the actual data transfer itself). Thus, an alternative implementation was adopted, from AVR Libc, the gnu standard library for AVRs[23]. Their implementation turned out to be around 7x faster than ours. To use the AVR Libc implementation instead of ours, define `USE_AVRLIBC_CRC16` to be true at compile time.

In any case, it was eventually decided that CRCs were unnecessary, and so by default they are disabled. They are functional, however, and can be enabled for MMC/SDC use by defining `USE_CRC_FOR_TRANSFERS` to be true.

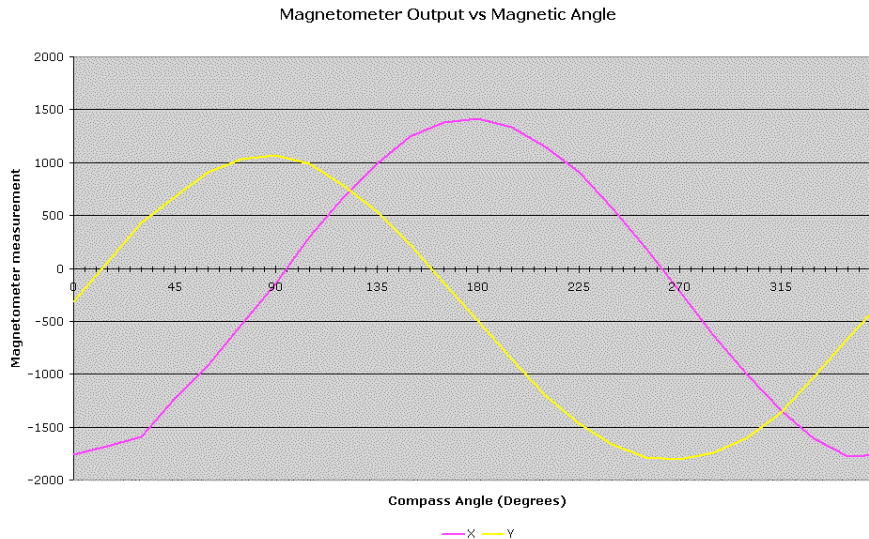
#### 4.3.6 Magnetometer

Before using the magnetometer's output, we wanted to ensure it was accurate. To do this, we printed out a circle divided into 24 wedges, such that each wedge is 15 degrees, as shown in Figure 11, below.



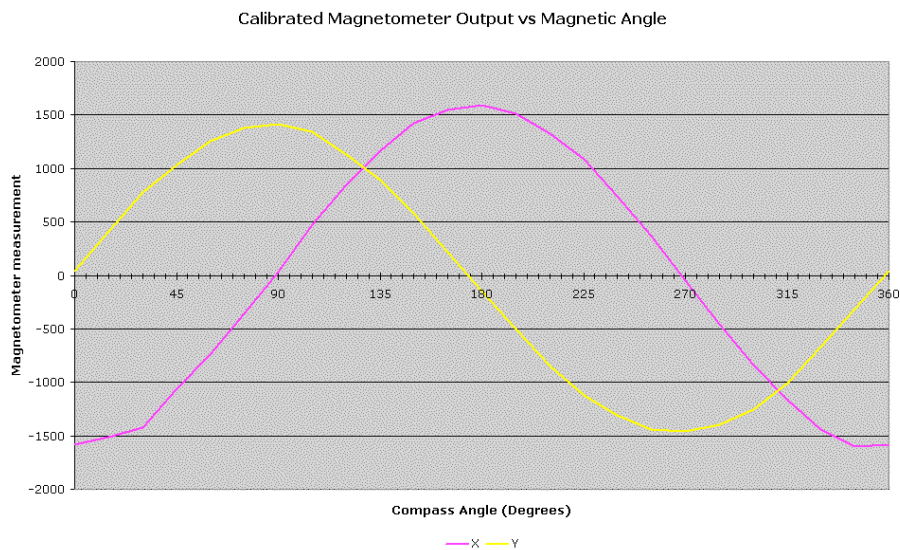
**Figure 11: Magnetometer Calibration**

Using a magnetic needle compass to determine magnetic north, we carefully aligned the printed sheet and secured it to the desk. We then aligned the PCB with the edge of each wedge in turn, noting the measurement from the magnetometer each time. These were recorded in a spreadsheet, and then charted, yielding the result shown in Figure 12, below.



**Figure 12: Magnetometer uncalibrated chart**

There is a clear DC offset for both X & Y readings, as well as a significant phase offset for the Y axis. The DC offset is easy to correct by adding a calibration constant to the measured value, which (for estimated calibration values) yields the results shown in Figure 13, on the following page.



**Figure 13: Magnetometer calibrated chart**

The deviation from "true" value (as determined by the compass) is shown in Figure 14. Clearly there is still some error due to the phase offset, but it is quite small, and certainly within the degree of error expected for the calibration process used. In the air, when the plane is not perfectly flat, there will of course be further deviation from "expected" values. In any case, the interference from the motor rendered the magnetometer unusable in practice, as noted previously.

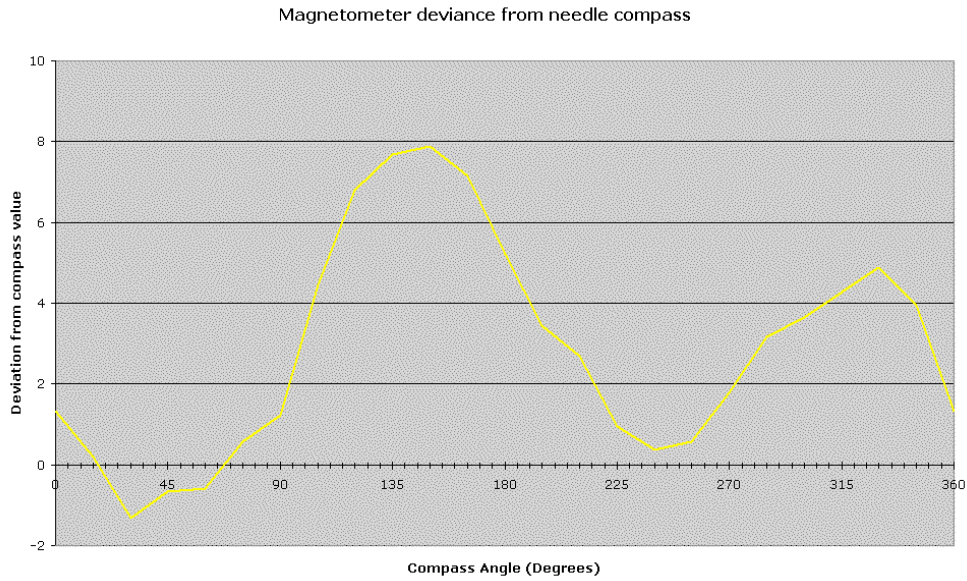


Figure 14: Deviation of magnetometer from compass measurement

#### 4.4 Flight Planner

The core functionality was implemented in the Flight Planner, although time did not allow for much more (in particular, texture mapping Google Maps imagery into the 3D views was not completed). The end result is shown in Figure 15 below.

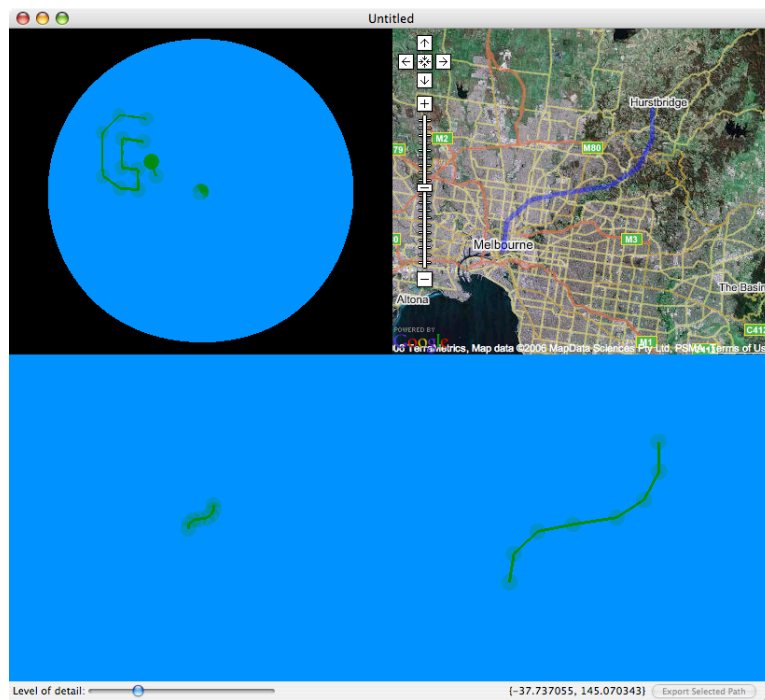


Figure 15: Flight Planner screenshot

The final implementation is relatively trivial, with the structure shown in Figure 16. Appendix F shows the full class hierarchy.

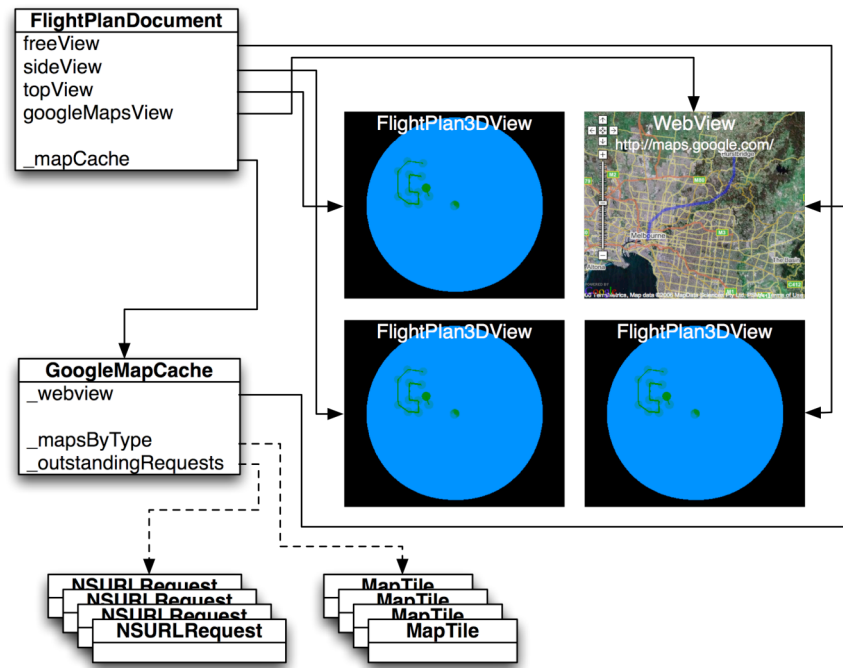


Figure 16: Flight Planner relationship diagram

#### 4.4.1 FlightPlanDocument

Most of the menial tasks like document management are handled automatically by Cocoa. The FlightPlanDocument class simply stores the current flight paths, providing an API for the views to modify these paths (e.g. in response to the user manipulating a point within the view). The FlightPlanDocument also handles saving and opening of documents, as well as the export of paths in UAV format.

The FlightPlanDocument needs to keep the four views synchronised, which includes keeping them centred on the same point, as well as reflecting changes to a path in all views. Whenever a shared property is changed (e.g. current position), the FlightPlan3DView or Google Maps WebView tells the FlightPlanDocument, which then informs the other views as necessary.

#### 4.4.2 FlightPlan3DView

The FlightPlan3DView is relatively straightforward. It needs to generate the geometry for the Earth and the paths. It then renders those. As noted, there was not sufficient time to implement texture mapping, so the Earth is rendered as a simple blue sphere. Paths are rendered as curved line segments following the great circle arc between their endpoints, and points are marked as spheres of a fixed size, relative to the viewer.

One particularly interesting detail is the generation of the Earth mesh. Originally the sphere-generating function of the GLU library was used, which while trivial to implement was not very flexible. And, most importantly, it generates the entire sphere. For most uses, the focus is on such a relatively tiny portion of the Earth's surface that it is essentially flat (requiring very few triangles to represent accurately),

and so small that to meet the detail requirements would require a mesh with many orders of magnitude more points than are practical.

So, a custom mesh generator was implemented, which generates only the portion of the sphere necessary to cover the current view, with a fixed number of points. Thus, the rendering load is constant, as the number of points is constant, and the smoothness of the Earth's surface increases as the user zooms in. This works very well in practice, although it can be computationally expensive if the view is changed rapidly, requiring many regenerations of the mesh. This could be optimised significantly in future; time did not allow for the current implementation.

#### **4.4.3 Google Maps WebView**

The Google Maps view serves two purposes - first, it was an easy first step to providing an interactive map prior to implementing the texture mapped 3D views. Second, it is the gateway to Google Maps through which imagery can be retrieved for external use.

The Google Map API is officially available and well documented [24], making it quite easy to use. It runs within any web browser that supports HTML, CSS and JavaScript. Mac OS X's WebKit framework meets these requirements, and can be embedded within a Cocoa application very easily. The architecture is thus that the actual view class in which Google Maps is rendered is a WebView, which loads a pre-defined local HTML file (Map.html, listed in appendix G) containing references and code for Google Maps. The Cocoa application, written in Objective-C, can interact with the WebView by executing JavaScript within it. It also exposes itself to the WebView using the Objective-C to JavaScript bridge, allowing JavaScript within the WebView to invoke methods on the exposed objects. This allows, for example, the FlightPlanDocument to be notified of changes in the Google Maps view position.

The Google Maps API also provides JavaScript classes and functions for creating vector paths, which are used to display the flight paths. At present it is not possible to manipulate these paths within the Google Map view itself; the Google Maps API does not provide such functionality.

#### **4.4.4 GoogleMapCache**

The GoogleMapCache's purpose is to satisfy requests for map imagery. Google Maps tiles its maps into squares 256 pixels on each side. Each image must be requested separately over HTTP, and as the desired level of detail increases, the number of map tiles to manage grows exponentially. The GoogleMapCache abstracts away all the management and downloading of map tiles. It provides a simple asynchronous interface - requests are issued for the tile or tiles which contain a given coordinate (or coordinate rectangle). The GoogleMapCache queues up requests to Google's web server for the necessary images, and as they download calls back to the original requestor with the data.

While the GoogleMapCache class has been implemented, it was not ultimately used, and has not been tested.

## **4.5 Budget**

### **Income:**

<b>Source</b>	<b>Amount</b>
La Trobe University Project Budget	\$200.00
La Trobe University Project Budget	\$200.00
Dick Smith Electronics Project Grant	\$200.00
<b>Total:</b>	<b>\$600.00</b>

### **Expenditure:**

<b>Item</b>	<b>Cost</b>
GPS	\$78.95
GPS antenna	\$22.95
Accelerometers	\$10.00
Magnetometer	\$49.95
Microprocessor (AVR)	\$12.00
Bluetooth RF Link	\$93.95
Sensors (temperature, pressure)	\$20.00
LiPO battery (x 2)	\$64.00
LiPO Charger	\$45.00
Servo connectors/Power connectors	\$20.00
Misc production/Mounting/small components costs	\$55.00
<b>Total Cost</b>	<b>\$471.80</b>

## 5. Results

While the project was not completed to the level we would have liked, what was done produced some fantastic results. The first stage of results was in getting the system operating onboard the aircraft during flight, albeit under manual control. This provided us with extensive sensor data and helped us discover problems (e.g. the motor's interference of the magnetometer) while the autopilot was still in early development. Changes could thus be made relatively cheaply, and tuning could be estimated prior to the first autopilot test.

We will now present our results, starting with the basic flight results, from the six test flights performed to date. There was no strict plan to the test flights; many were prompted by favourable weather, and the intention of most was simply to increase total flight time and the body of data collected.

### 5.1 Flight results

While one flight ended in disaster when the wing snapped (a rare but not unprecedented mechanical fault, a known problem with ElectraFun XP planes), most flights were entirely without such incidents and yielded promising results. An example flight log is shown in appendix D.

Using a basic parsing and graphing Matlab script, we are able to plot the path of the plane in 3D, as shown in Figure 17, below.

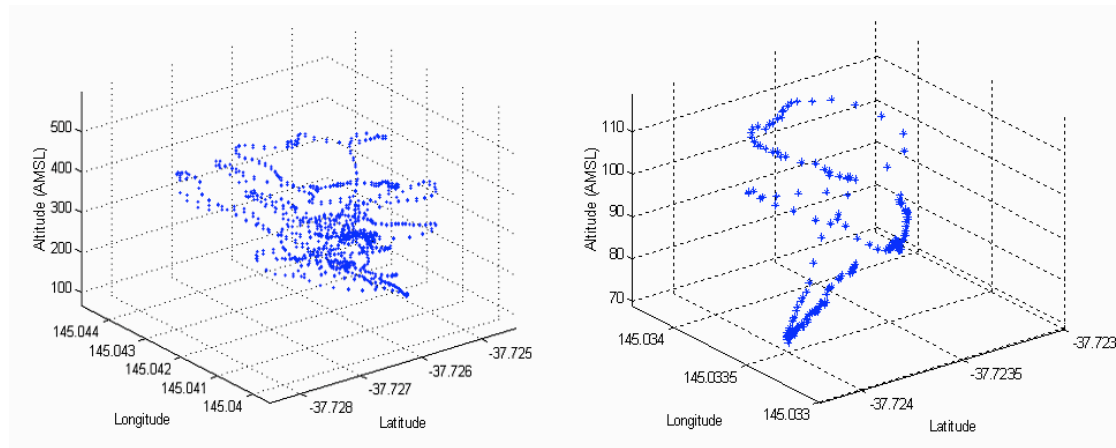


Figure 17: Matlab flight path plot

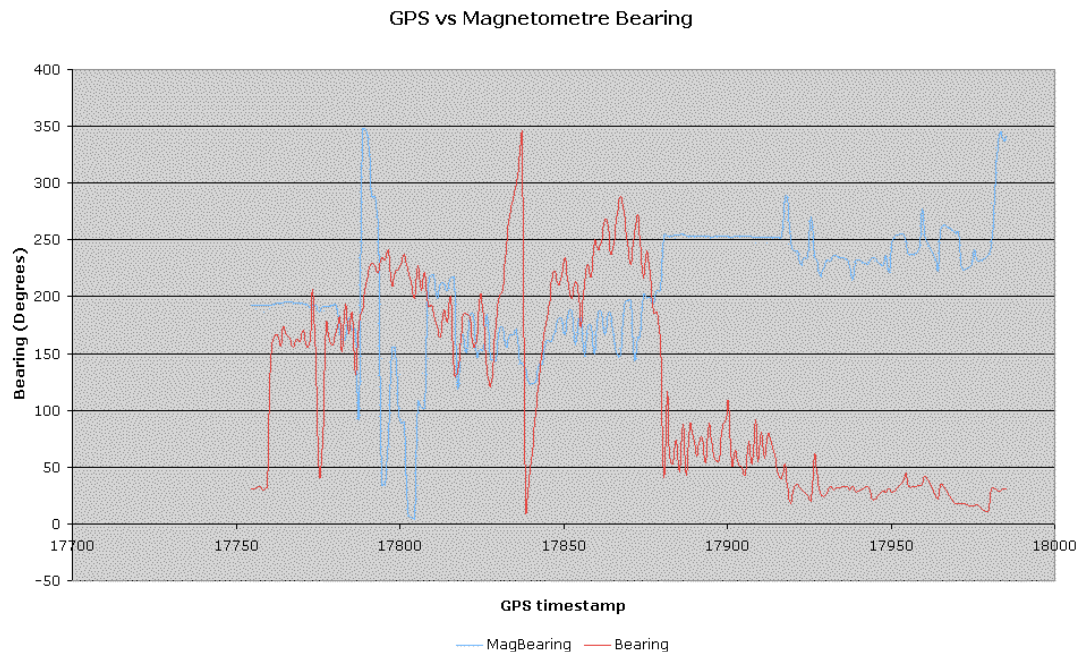
Using a small tool we wrote, `log2csv`, the log files for each flight were converted to a format Excel could import, and then charted. To start with, we'll look at the bearing measurements.

#### 5.1.1 Bearing

The bearing results are difficult to chart (in Excel at least), because of course they wrap around from 0 to 360 and vice versa. A typical result (taken from flight #6) is shown in Figure 18. Note the behaviour of the magnetometer, pinned to around 270 degrees for most of the flight. Prior to take off, and after landing, it appears to show a



correct value - indeed, given the plane is not moving at these times it the magnetometer bearing is likely far more accurate than the GPS bearing.



**Figure 18: Flight #6, GPS vs Magnetometer bearing**

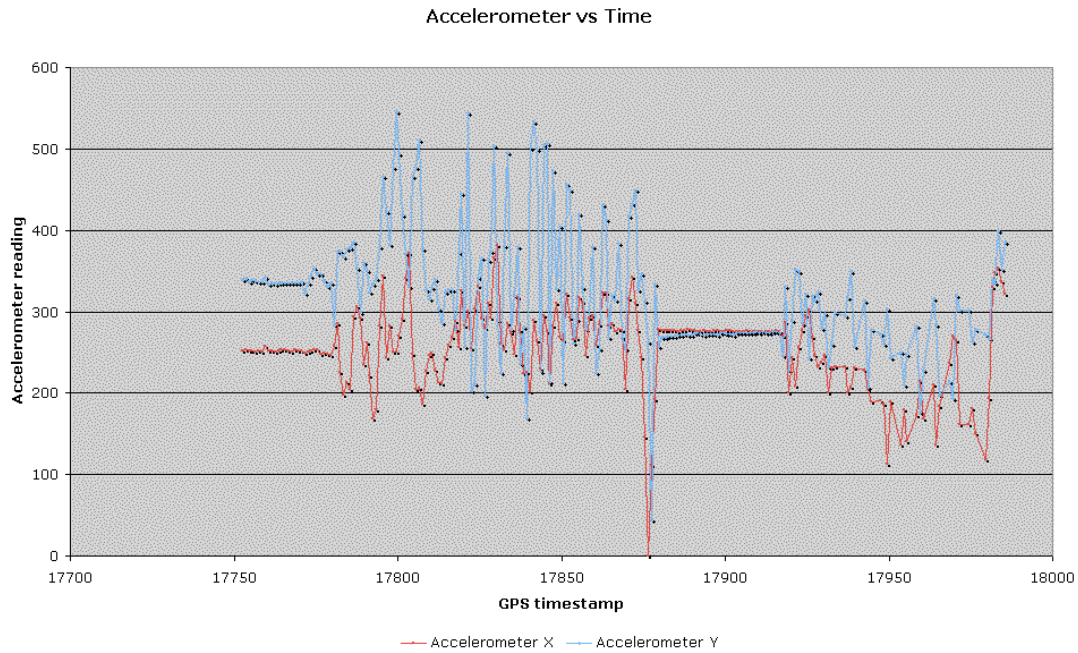
Nonetheless, the magnetometer was shown to be quite accurate during lab testing, as detailed in Section 4.3.6. Hopefully in future a solution can be found to prevent the interference seen above, so that the magnetometer can be used to it's full potential.

### 5.1.2 Accelerometers

The accelerometers were ultimately not used by the autopilot, but their readings were logged consistently nonetheless. Figure 19, next page, shows the recordings from flight #5, where the wing snapped. It is clear that there are several phases. At first the plane is held relatively still a little while, during preparation for takeoff. It is then launched, and the accelerometers reflect the movement of the plane as it rolls, pitches and yaws, as well as momentary bumps from strong gusts of wind. At only about 90 seconds into the flight, the wing snapped, sending the plane into a rapid downwards spiral into the ground, the impact of which is clearly visible at around the time 17875. There it sits for some time until it is retrieved, and the remaining measurements are of the period in which the plane was carried back to the start position.

Clearly the accelerometers work, although their fidelity and accuracy were not tested. The analysis work to find correlations between accelerometer readings and the plane's behaviour is yet to be performed.





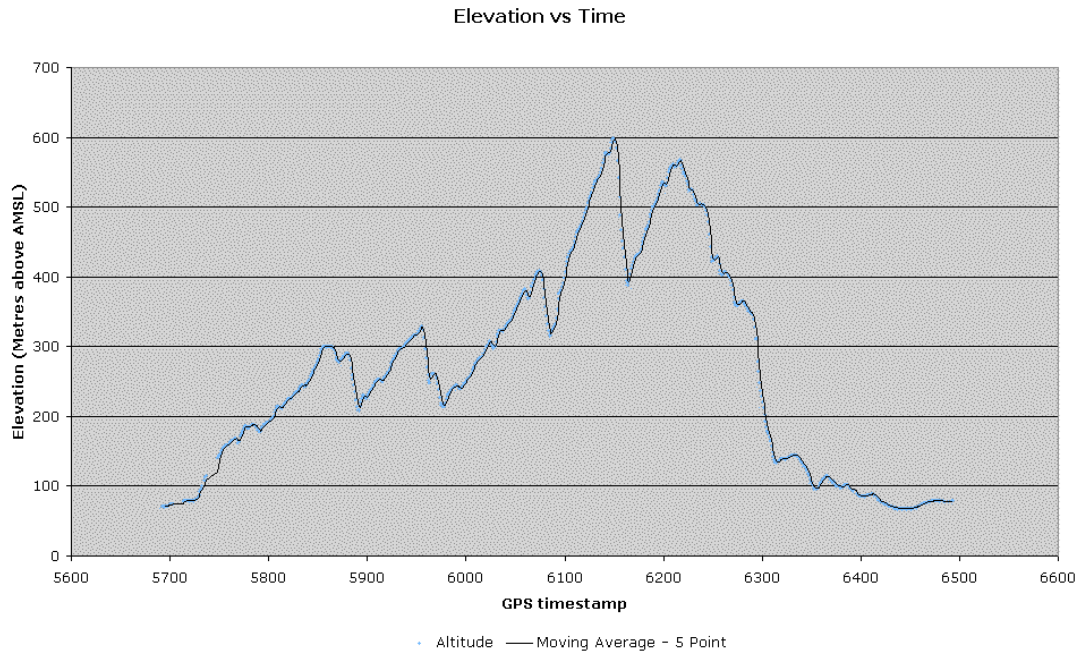
**Figure 19: Flight #5, Accelerometer vs Time**

### 5.1.3 Elevation

The GPS elevation measurement turned out to be surprisingly reliable and precise, although not particularly accurate. Values measured on the ground at the La Trobe sports fields are up to 40 metres off the actual value of 68 metres AMSL (taken by a formal survey conducted by the university). However, they are self-consistent within the time of a typical flight - when the plane returned to the launch point, it agreed with the values measured prior to takeoff of the same flight to within a few metres. This implies some kind of calibration needs to be applied at the start of each flight, to adjust for this fixed offset. However, the number of flights performed is not sufficient to be able to say, reliably, what the period of elevation fluctuations is or can be.

As seen in the chart (Figure 20, next page), in this particular flight we achieved an elevation AMSL of approximately 600 metres. Adjusting for the fixed offset noted previously, unfortunately, indicates we fell short of 600 metres by roughly 8 metres, although that's not to detract from the significance of the achievement - 600 metres is extremely high for a tiny model plane like ours. It is certainly not the actual limit of the plane, either, as the height was constrained by safety concerns; beyond a few hundred metres in elevation the plane becomes challenging to track, and obviously the result of a serious accident like a wing snapping is potentially very serious.

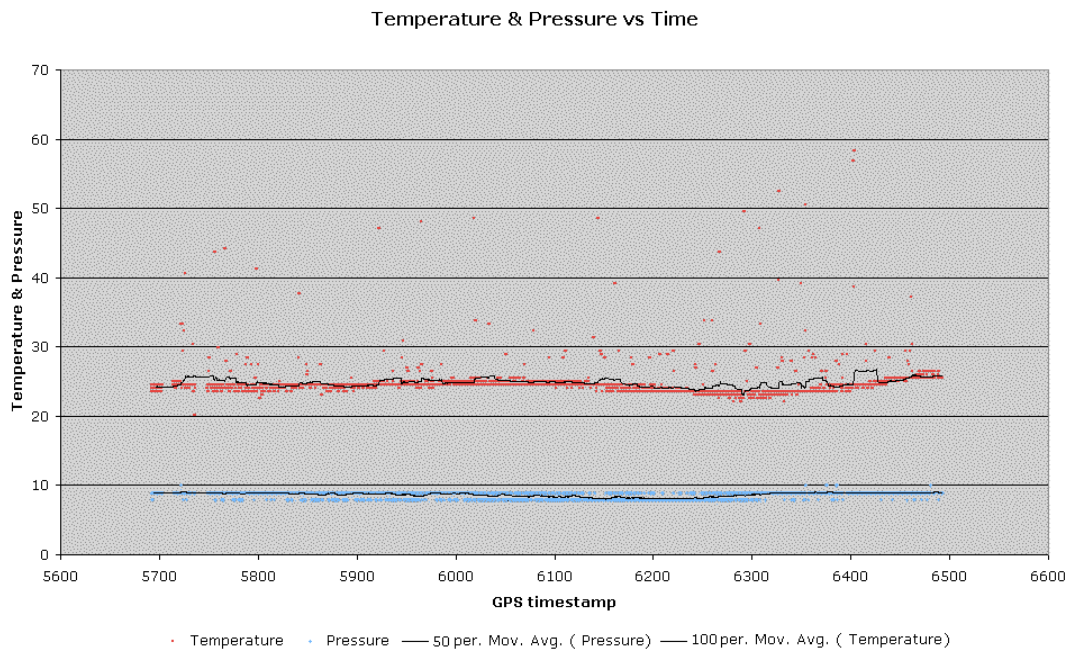
To emphasise the significance of this elevation achievement, consider that, in relative scale, 600 metres for a plane the size of ours is equivalent to 37,000 metres for a 747-400.



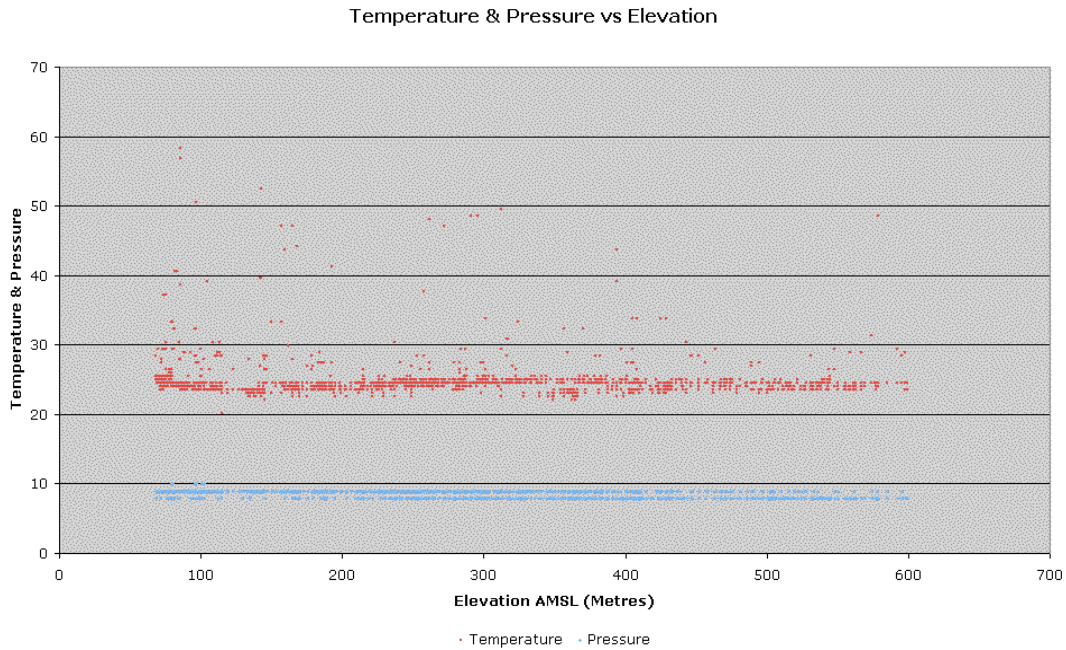
**Figure 20: Flight #3, Elevation vs Time**

#### 5.1.4 Temperature & Pressure

The temperature and pressure sensors worked well, although as noted they did not have any functional purpose. Figure 21, below, shows the temperature and pressure over time during flight #3, while Figure 22, next page, shows the sampled values plotted relative to elevation. Temperature values are in degrees, while pressure values are in unconverted ADC units (i.e. the value obtained direct from the ATmega2561's ADC).

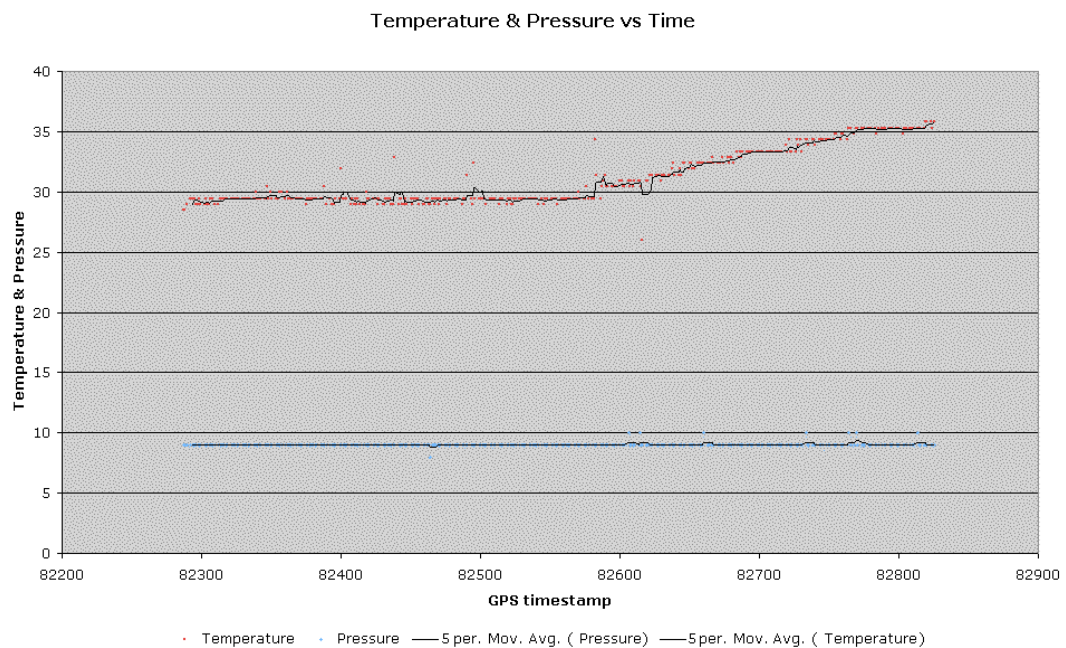


**Figure 21: Flight #3, Temperature & Pressure vs Time**



**Figure 22: Flight #3, Temperature & Pressure vs Altitude**

In the time domain it there is a clear fluctuation, albeit slow and slight, in the temperature. There are many factors involved in the temperature - including heat generated by the motor - so it is not clear what caused this fluctuation. Other flights have shown that in unusual circumstances (i.e. in flight #6 when the motor was caught in foliage, but still at full throttle) there is a significant rise in temperature, as shown in Figure 23 below.



**Figure 23: Flight #6, Temperature & Pressure vs Time**

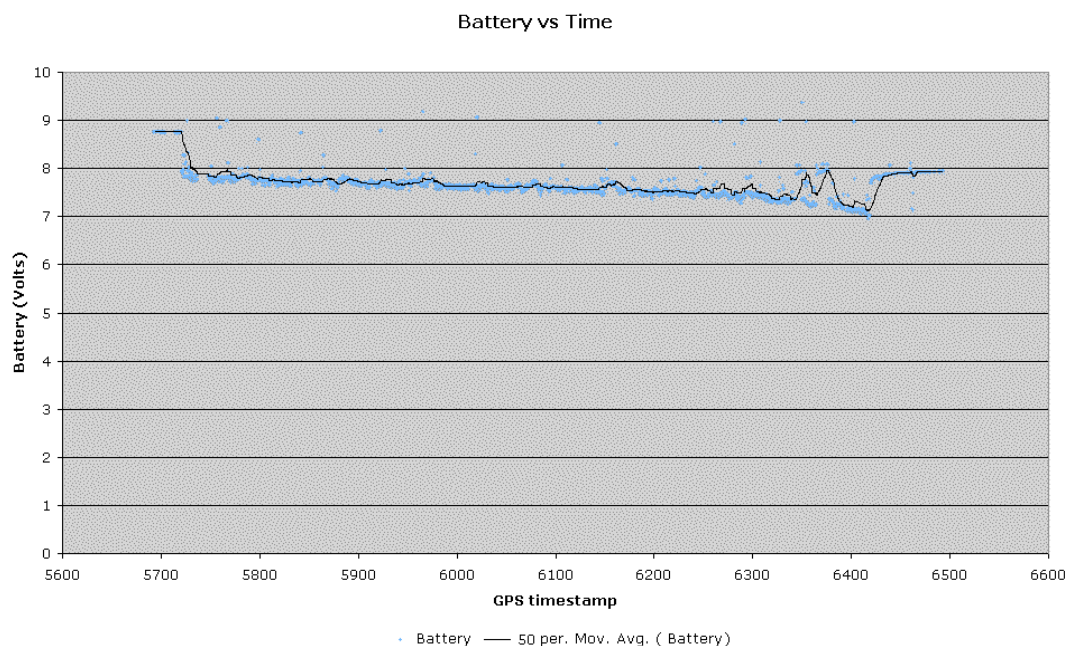
Pressure exhibits little discernable trend in the time domain, and is crippled by the lack of resolution. Nonetheless, the pressure over elevation as shown in Figure 22, above, shows a trend towards slightly lower pressure as elevation increases. An



aviation rule of thumb is that pressure drops by roughly 1% per 100 metres of increased elevation [20]. That rule of thumb has us expect a roughly 5% drop at peak elevation in flight #3. Unfortunately the resolution of the pressure measurement is too little to accurately verify, but it does seem to be of the right magnitude.

### 5.1.5 Battery

The battery monitor worked well, as demonstrated in Figure 24, below. As soon as the throttle ramps up, the battery voltage is dragged down by over 10% due to the high load. At around 6350 seconds, throttle is decreased in preparation for landing (which is typically done with reduced throttle or even un-powered). Once the plane hits the ground a built-in mechanical safety feature causes the motor to be disconnected, and the battery voltage jumps back up by 10% percent.



**Figure 24: Flight #3, Battery vs Time**

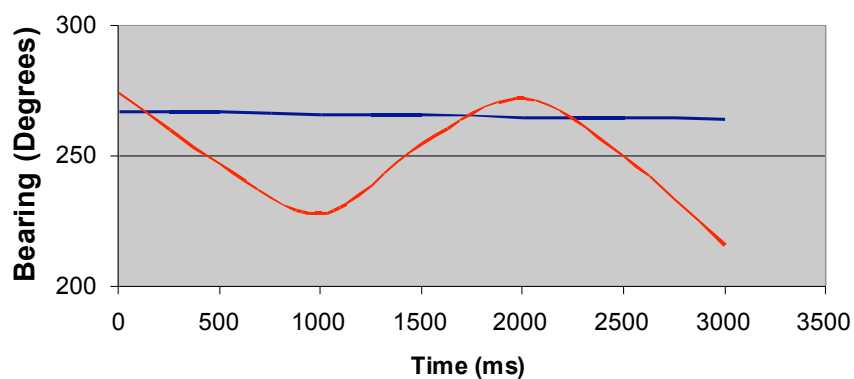
There is a clear decrease in voltage over the duration of the flight, which appears to be linear, indicating that this would indeed be a useful and meaningful way to monitor the battery status and remaining flight time. However, the LiPO batteries used do not have a very significant voltage drop (in contrast with NiMH batteries) until they are completely flat, at which point they rapidly drop by as much as 60%. This makes them very good overall, but does make their status slightly harder to determine.

## 5.2 Autopilot results

Contrasted against successful flight results presented in the previous section, the autopilot results are unfortunately not as stunning. The manual switching hardware was demonstrated to work reliably, both when the radio connection was lost and when the pilot manually switched over to automatic control. The disappointing results were in the performance of the PID algorithms, due largely to a lack of time in which to properly tune them. A significant disruption to our tuning flights was weather. On the first tuning flight, #5, the wing snapped through no fault of ours. Further, it was

also discovered that the control surfaces need to be reset to neutral positions each time autopilot is entered – otherwise erratic behaviour can ensue.

Despite the setbacks and lack of tuning, the bearing PID algorithm was shown to work on the ground using the magnetometer, and was observed to function at least partly in flight testing, albeit with overdamping as shown in Figure 25. Unfortunately with our flight tuning requiring fine flying weather, further tuning on the bearing PID was not possible, and no tuning was performed on the altitude PID. Another problem is that the using the GPS receiver, bearing and altitude updates are only available once per second, which tends to skew the PID controls and limit responsiveness. It was of course the purpose of the magnetometer to augment the GPS bearing information, giving faster bearing update information, but as examined in section 6.2.2 interference from the motor made the magnetometer all but useless at full throttle.



**Figure 25: Autopilot bearing tracking**

Despite the unfortunate lack of conclusive results, it seems that the decoupled control works well – with only one of the control surfaces moving significantly at a time, producing expected results, even without proper tuning. With faster update rates (preferably using gyros) and further tuning it is envisaged that the PID algorithms would behave far more robustly and reliably.

### **5.3 Logging results**

Both Bluetooth and onboard logging worked, on most occasions, successfully and well. Bluetooth logging turned out to be invaluable in the field, as once the system is assembled inside the plane and the wing attached, there is no direct access to it. Being able to observe the operating condition of the system at this point is of course critical to ensuring the system is ready for flight.

There were two Bluetooth transceivers used to connect to the BlueSMiRF module during the project - the built-in Bluetooth on a G4 Powerbook, and an unbranded USB Bluetooth dongle from eBay. Both worked well, although the range of the Powerbook's receiver was significantly less than that of the USB dongle, at approximately 40 metres as compared with 70 metres. The quoted range of the BlueSMiRF module is up to 100 metres, although this seems a little optimistic; perhaps with a high gain antenna on the receiver.

The onboard logging worked very well during testing and on most flights, particularly after the performance optimisations made during testing. Appendix D shows a small section of a flight log.

There were three flights where data from the onboard log was lost. In one - flight #6 it was a simple user error - the card was removed without holding down the "arm" button in order to flush the internal buffers, so some amount of recently written data was lost. The magnitude of the loss was surprising, but understandable in hindsight - the FAT table is accessed frequently, and so remains in cache virtually indefinitely. Thus, while the actual data was written to the card, the FAT table wasn't written back and so the log appeared far shorter than what was really written. Luckily, with the aid of file recovery tools we were able to recover the vast majority of the data.

On another occasion - flight #2 - logging stopped abruptly midway through the flight. The actual cause has not been determined, although it is theorised that a sudden jolt dislodged the card. The flight data up to that point supports this hypothesis - the plane was in a sharp and rapid dive just prior to the loss of logging. In future flights we used electrical tape to secure the card, just in case.

The third flight from which we lost logging was flight #4, where logging simply stops midway. There was more data written to the file on the card than the FAT indicated, which indicates in turn that the AVR simply stopped working at some point (as opposed to some kind of overriding corruption, of which there was no evidence). Even so, only a few dozen lines were lost from the log, indicating that logging did actually stop entirely shortly into the flight. Again, the cause is not known conclusively, but the current leading hypothesis is that the linear regulator supplying the AVR was briefly overloaded, causing the AVR to brown out and reset. The AVR would have then spent the rest of the flight sitting idle, waiting for the "arm" signal. This hypothesis arises because in this particular flight the servos were powered off the same regulator as the AVR, whereas in previous flights they were not. To address this for future flights, a separate regulator was added to supply the servos independently.

There was also the intention to add a special flag in EEPROM, that would be set at the start of the flight and manually cleared at the end, so that if the AVR were reset for any reason during the flight, it would see this flag is set and automatically arm itself, thus resuming operation. There was not enough time left this late in testing, however, to implement this.

But despite these setbacks - hopefully resolved now, as detailed - we did record excellent data, only a small portion of which has been presented here. The full log files and all charts for all flights are included on the accompanying CD.

## **6. Discussion**

In this section we will subjectively evaluate how the project went, and in particular highlight the areas where things went especially well or especially poorly. Many of these pitfalls in particular may be of assistance to future researchers, with a view to add to the pool of knowledge in the UAV arena.

### **6.1 *What went right***

There were a few aspects of the project which turned out to be either excellent design choices or just happy accidents that had positive consequences. They are discussed below, in no particular order.

#### **6.1.1 GPS**

The GPS receiver worked exceptionally well, providing reliable position data with very good precision (typically a variance of only a few metres). The interface, RS-232, was easily connected to one of the ATmega2561's built-in UARTs, and the software for reading from the GPS similarly trivial. Parsing the human-readable, ASCII GPS strings was more difficult, but the GPS receiver used also supports a binary format, which could be used in future to simplify processing.

When it became evident that the magnetometer was not usable, the GPS bearing was used instead. While the GPS bearing is not as responsive as the magnetometer (particularly at low ground-speed), it is immune to electromagnetic interference, and as such is perhaps more suited for measurements over long periods.

#### **6.1.2 Manual override & emergency autopilot switch**

Performing the manual override in hardware added significant complexity to the PCB, so it was somewhat contentious as to whether it was necessary or not; whether the switch could be controlled in software instead. As it turns out, there were several problems discovered during testing - including during flight tests - which resulted in complete software failure, conclusively demonstrating the importance of the hardware implementation. The implementation worked reliably throughout testing, and is an important safety assurance during flight tests.

#### **6.1.3 Bluetooth**

Given the failure of the initial wireless transmitter, it was uncertain whether that feature of radio communications would end up working. The assertion that the communications was an important feature, and the resulting retry with the BlueSMiRF module, turned out to be an excellent decision. The wireless functionality aided in debugging, particularly in the field where it was difficult - and at times entirely impossible - to access the PCB with a computer. One key proof of this is pre-flight preparation, when the PCB is mounted inside the plane, along with all the other components, and is not easily accessible. It was very handy to be able to connect wirelessly and watch the start-up sequence, as well as monitor the sensor outputs prior to and in the initial moments of flight.

Despite the limited range of Bluetooth making the system inaccessible over the ranges of normal flight, it was - even aside from the benefits already noted - an excellent

proof of concept. Future work could look at newer, more powerful Bluetooth modules, or alternative long-range equipment.

#### **6.1.4 FAT support**

Although we'll go on to criticise the actual FAT driver implementation in the next section, the decision to support FAT16/32 was an excellent one. Being able to take any MMC or SDC, insert it into the UAV, record data on it, and then take the card and read the data straight off it on any computer is a fantastic capability. It greatly aided debugging, where we could write a large amount of data to the card from the AVR, and then have that data up on a computer screen with just a few moments. Similarly, it is very handy to be able to retrieve the card after a flight, and simply copy off a single file containing the full flight log.

It should also be possible to read the MMC/SDC from a PDA and possibly even some modern mobile phones, allowing access in the field where a computer may not be available.

### **6.2 *What when wrong***

In unfortunate contrast to the successes noted thus far, there were many design choices that were in hindsight poor, or major problems which only became apparent during implementation and testing, or even at times after. We have been purposely highly critical of ourselves in this section, to emphasise and details these problems so that they won't be repeated in future.

#### **6.2.1 Flash logging software over-designed and overly complex**

In hindsight the software implementation of the onboard logging was overly complex, and over-designed. The Physical Media layer was ultimately redundant, and remains only as an artefact of an early design choice that didn't pan out as expected. It would be trivial to remove, but the emphasis - right up to the last moment - was on ensuring correct functionality of the system as a whole, not refactoring.

Similarly, although less importantly, the Volume interface module could have its essential functionality subsumed into the Standard I/O layer. The Standard I/O layer itself could be dropped, and the FAT driver used directly. This would remove the ability to support multiple file system formats simultaneously, which isn't at all important for our purposes, but also - and perhaps significantly - would require some of the Standard I/O functionality (such as parameter checking and its simplified API) to be merged into the FAT driver, enlarging the already substantial driver implementation. Again, all these are things that could be achieved relatively trivially, given the extra time we did not have.

The biggest source of unnecessary bloat is within the FAT driver itself. At over 3,000 lines of dense C, it is anything but succinct. This stems from the initial design being too overarching, and not being prepared to make certain assumptions - such as a fixed, 512-byte sector size, an assumption which in hindsight is obvious. Much of this stems from an initial inexperience with the FAT file system, and as such was probably inevitable. There was also an emphasis during its implementation to "optimise" it by performing most things inline, including very common operations like simply



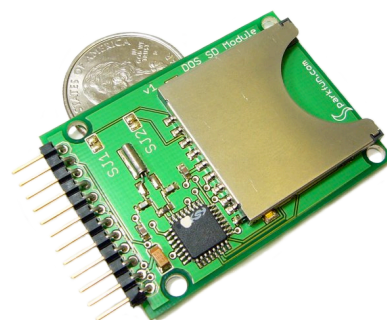
obtaining data from the physical media. This optimisation was most certainly premature, and there is little evidence to suggest it was successful. As a result, the code length was increased and its readability reduced. While it was refactored midway through development, which resolved some of the more glaring issues, there is still much work that could be done in this regard.

In particular, using SRAM buffers more liberally could save a lot of development time and effort, and most likely improve performance on the AVR. Again, they were discarded prematurely as part of the presumed optimisation. As it turns out - easily visible in hindsight - we use so little of the 8K SRAM; we could have used substantial buffers without detriment.

The official Microsoft specification for the FAT file system was not known to us until after the completion of the project. Not having an authoritative reference made development difficult, and subsequently raises real concerns about the accuracy of the driver. While comparing against the drivers in Windows XP and Mac OS X is a reasonable test, it is no substitute for following the full specification. Now that the full specification is available to us, after the fact, it is clear there are many issues with the current implementation. Most are minor, but nonetheless highlight the importance of obtaining such documentation early and understanding it well.

The final flaw we'll detail here was the failure to recognise that ICC7 for AVR is little-endian. While the AVR itself is an 8-bit microcontroller and thus has no intrinsic concept of endianness, for multibyte integers the compiler applies one. Since no endianness was assumed, data was read and ordered manually. This turns out to be extremely expensive using ICC7 for AVR, due to its very poor assembly generation, and bloated the compiled code size (and execution time) substantially. But yet again, it is relatively trivial to go through and utilise this assumption; it just wasn't realised soon enough to be fixed.

Perhaps to add insult to injury, just as the project was finishing up one of our preferred suppliers announced a new "DOSonCHIP" IC (Spark Fun SKU#: IC-DOSonCHIP) that connects directly to an MMC/SDC on one side, a UART or SPI port on the other, and performs all the FAT16/32 work automatically, providing a simple API to the microcontroller for manipulating files at a high level. If this little IC works as promised, it can replace our FAT driver and supporting infrastructure with perhaps only a few tens of lines of code. This was most distressing to us, but is of course inevitable in the ever-advancing, ever-miniaturising electronic world. And it certainly makes it easier for others to replicate our work in future.



### 6.2.2 Motor interference

The magnetometer was included even though there were concerns about interference due to the motor, which at full throttle draws approximately 6 amps. When the motor is fully powered, the magnetometer gives a fixed bearing, clearly demonstrating motor interference. However, even when the plane is at half throttle the magnetometer was shown to reliably measure the bearing. Testing prior to the PCB

design and fabrication could have revealed the problem, and the magnetometer could have been left off the PCB to save space, design time and power. It also would have allowed us to seek alternative means for bearing determination. While the ability to use the GPS bearing was a lucky save, it is not an ideal solution. Future work should investigate ways to integrate the magnetometer that avoid or at least reduce the interference, or a replacement. One possible solution would be to replace the brushed electric motor with a brushless motor, or a petrol motor, which should substantially reduce the level of interference.

Alternatively, the software could be adapted to only rely on the magnetometer when the throttle is below a certain threshold. During straight flight it may frequently be possible to lower the throttle, at least temporarily, in order to take measurements.

### **6.2.3 Accelerometers**

The Accelerometers were another disappointing sensor. When tested statically on the ground they act as tilt sensors. In motion however, the accelerometers are affected by the acceleration forces of the aircraft as well as the earth's gravitational field, rendering them far less useful as tilt sensors. A better choice would have been the inclusion of a gyroscope which would have provided stable tilt measurements during motion. Unfortunately by the time this problem was realised, neither the budget nor the time remaining allowed for the inclusion of a gyroscope.

### **6.2.4 Pressure sensor's insufficient resolution**

The pressure sensor chosen for the project was the Freescale MPXM2202 which is likely quite suitable for our purposes; indeed, the data sheet suggest it can be used in an altimeter. Unfortunately the output of the device didn't have enough amplification and so we were only able to reliably measure altitude changes in the order of thousands of meters. The inclusion of a pre-amplification stage, prior to inputting the signal to the ADC, would most likely resolve this issue.

Additionally, towards the end of the project, a pressure sensor (SCP1000-D01) was released by one of our suppliers with a purported precision of as fine as 9cm. Since the vertical error on the GPS due to vertical dilation is in the order of 10m, this device would hypothetically give us far more accurate altitude readings, and may have provided a suitable platform (when coupled with proximity sensors) for automatic landing systems.

### **6.2.5 Manufacturing delays**

Due to the complexity and intricacy of our design we decided to outsource the PCB fabrication rather than having the PCB's fabricated in the University Store. This decision lead to unexpected delays, which were the sum of both manufacturer delays and submission delays (not on our part). These associated delays resulted in a three week delay on top of the expected board delivery date and subsequently cut short our in-flight testing and algorithm optimisation time.

### **6.2.6 ICC AVR7 bugs and limitations**

The ATmega2561 was more or less brand new when we first acquired it, and represented a significant step up from previous ATmega AVR's, in that it includes 256 KB of Flash program memory. While it was opportune to use the ATmega2561,

given its extensive resources, it had the unfortunate down-side that many of the tools needed to be updated. In particular, ImageCraft added reliable support for the ATmega2561 only in version 7.08 of their compiler [25], which was released only on September 11th, 2006, when software development was already well underway. In hindsight, while things did work out fine in this regard in the end, it may have been wiser to use an older model ATmega, given we ultimately didn't use the majority of the 256 KB of program memory.

Additionally, the choice to use a demo version of a very expensive commercial compiler was unwise. Given that a commercial license was not a proviso of the project (although it could perhaps have been acquired nonetheless), this necessitated working around the limitations of the demo. Initially the only concern was the demo expiry - 45 days after installation. The intention was to reinstall the software after it expired, although as it turns out this requires reformatting the PC it is installed upon.

However, before the software expired another issue arose - that the demo is limited to 64 KB of program memory. Luckily, this limitation is implemented in the IDE itself, not the underlying compiler, and can be overridden by utilising the manual configuration section of the IDE's compiler settings. This took some time to work out, however.

Furthermore, it was then discovered that the compiler did not support any single section being larger than 64 KB. The solution to this was relatively trivial - dividing the MMC/SDC software off into its own section - but took a significant amount of time to find.

Luckily, the upside of these hacks was that when the trial period expired, and the compiler was supposed to revert to barely functional status, the aforementioned hacks overrode its limitations still, and so we were able to keep using it.

Nonetheless, the IDE and compiler themselves were very primitive and painful to use. The compiler supports only a limited part of the C standard, and has numerous bugs, which wasted significant amounts of time in devising workarounds and trying to rewrite perfectly valid C code to suite the compiler. The readability and consequently reliability of the code was reduced as a result of these workarounds.

Late in the project the possibility of switching to the free & open-source gcc for AVR (a.k.a. WinAVR) was considered, although by that time it was deemed too late to do so. Were the project to be repeated, or further work done on it, it would definitely be wise to at least try WinAVR instead.

### **6.2.7 Wireless transmitter**

In the original project design an inexpensive 433MHz wireless link was chosen from Laipac. Even after extensive trial and error prototyping, this module worked rather unreliably at a range of only a few metres at best. At John Devlin's recommendation we ceased development on these wireless modules and opted for Bluetooth, which as already discussed was an excellent choice.

## 7. Conclusion

This project was initiated with the ambitious goal of creating an autonomous UAV on a very limited budget and within a limited timeframe. To measure the success of the project, this section will review each of the objectives individually, and summarise with our overall opinion of the project.

### Primary Objectives:

- *Outfit a model aircraft with a navigation and control system consisting of a GPS receiver, a magnetometer, accelerometers and an AVR microcontroller.*

This was achieved and is verified by the data gathered from numerous test flights described in section 5.1.

- *Write software for the microcontroller to sample each of the sensors and control the aircraft according to a pre-determined flight plan. (proviso: Aircraft will be launched and landed under manual control)*

The microcontroller software samples and stores data from the various sensors mounted throughout the aircraft. The microcontroller, using decoupled PID controllers, is able to manipulate the control surfaces to follow a predetermined flight routes. The PID controls that run this need further tuning and would be greatly benefited from the use of an onboard gyroscope as mentioned in Section 7.1.

- *Write computer software to develop a flight plan for the UAV to follow, with a communications link to facilitate transfer of the flight plan to the UAV.*

Computer software was developed to allow flight plans to be created, showing planes overlaid on maps supplied by Google. The flight plan data is currently manually loaded into the aircraft on programming, but could be upgraded in future to allow wireless transfer of flight plans via the Bluetooth data link, or to be read from the MMC/SDC at initialisation time.

- *Support onboard recording of all flight data (e.g. GPS co-ordinates, magnetometer readings, etc) onto a MMC/SDC.*

Support for logging of flight data onto MMC/SDC works well, providing detailed logs which can be viewed on the supplementary CD.

- *Provide an in-flight wireless simplex communication channel to transmit data from the UAV back to a ground station (i.e. PC).*

The Bluetooth data-link is currently used as a simplex communications channel and provides data from the UAV to a ground based PC to a measured range of approximately 70m.

- *Provide a control system to manually switch between autonomous control and human control.*

A hardware based manual switchover system was implemented and shown to operate reliably.

Summary: Five primary objectives were met. The sixth, full autonomous operation, was in development but has not been proven successful yet.

## **Secondary Objectives**

- *Addition of onboard sensors, with the sampled results stored on the onboard storage. Types of sensors to be investigated include battery monitors, temperature sensors and pressure sensors.*

Pressure, temperature and battery voltage level sensors were employed onboard the UAV and provide data which is logged to the onboard SD card.

- *Install an onboard wireless video camera to provide proof-of-concept video images of the visual data acquisition capabilities of the UAV.*

A wireless video camera and a still digital camera were mounted onto the plane during the early design phases. The photographs and video acquired from these cameras can be viewed in the supplementary CD.

- *Implement Error Correcting Codes (ECC) for the wireless PC link transmission to ensure signal quality and attempt to recover corrupted data.*
- *Implement a cyclic redundancy check (CRC) for the wireless PC link to detect data corruption.*

Once switching over to the Bluetooth radio transmitter, the need for us to worry about this was removed; the Bluetooth module ensures reliable data transfer.

- *Implement a failsafe autonomous control system, whereby if remote control is out of range the UAV will return to previous position where controller was in range, and progressively search for a signal.*

The lost data signal detection was implemented in hardware and demonstrated to effectively trigger an emergency control state within the microcontroller. The PIDs which control autonomous flight need further tuning before the plane can be expected to navigate its way back toward the pilot.

Summary: Two secondary objectives have been met directly, two were met via a design change, and one is incomplete.

## **Tertiary Objectives**

- *Implement a visual landing system using a small camera and edge detection algorithms, to allow the aircraft to autonomously land on a*

*well defined runway.*

This objective was not attempted.

- *Implement a collision detection system based on ultrasonic or laser measurements to detect the presence of upcoming objects and avoid them.*

This object was not attempted.

Summary: None of the tertiary objectives were met (or attempted).

Final summary: Besides fulfilling the majority of the primary and secondary objectives, and thus establishing a reasonable measure a success, this project has been an exceptional learning experience, testing and developing our technical competence, teamwork and management skills. The autonomous UAV field is still very young, with many new developments and technologies that should be further investigated. The next section provides a brief outline on some of the logical extensions to our project, which we ourselves have dreamt up and consider interesting and important things to pursue.

## **7.1 Future work**

One aspect that was continually highlighted along the development of the project was the potential scope and application areas that beckon to be developed. Due to both budgetary and time restrictions we were not able to pursue many of these ideas, but have summarised them to provide direction for future autonomous UAV projects.

### **Complete Flight Planner**

The vast majority of the development time of the Flight Planner software was spent learning the ins and outs of Google Maps, with the intention of extracting imagery from Google Maps to be used in a custom 3D world view, as noted. While the final key step - applying this imagery to the 3D world - was not implemented due to time shortages, much of the necessary infrastructure is implemented, such that it is a logical follow-on for future work. The management of the imagery within OpenGL is not a trivial task, however, given it's huge volume in this application, so this is a task best left to someone with substantial OpenGL experience.

### **Differential GPS, Galileo, and more**

To improve the accuracy of the GPS, the signal can be calibrated in real time based on known points. This helps remove distortions from the ionosphere that typically reduce the accuracy of the system by an order of magnitude or more. Differential GPS, as this augmented system is known, can provide accuracy down to as little as a metre [26].

A rival system to GPS, named Galileo, is currently being implemented by the European Union. This modernised system is advertised as providing significantly higher accuracy - within a few metres, and even as little as a few tens of centimetres using the commercial service, or a differential service analogous to Differential GPS[4]. The Galileo system is expected to be operational from 2010 onwards, and

presents a very desirable future platform for a primary navigation sensor to be used in autonomous vehicles.

Furthermore, it is expected that many receivers will support both systems, to provide at the least greater reliability (since there are in sum more satellites in view at any given time). Additionally, there has been much excitement and preliminary work on combined GPS/Galileo receivers, that can use advanced algorithms and knowledge of the two systems to increase the accuracy by an additional order of magnitude over either of the systems separately - providing a resolution of tens of centimetres[27], even without using commercial services.

While the timeframe for all this is some way into the future, the increased accuracy invites new applications and capabilities that can be developed in anticipation of Galileo's deployment.

### **Use X-plane for modelling, flight simulation and flight planning**

Late in the project's development X-plane [28] was investigated in a general manner, looking at it's usefulness to the project. X-plane is a famously powerful and accurate flight simulator, which uses actual airflow modelling and advanced real-time physics to provide an impressively accurate simulation of reality. It is approved by the U.S. FAA for commercial pilot training, an impressive testimony to it's accuracy [29].

What is particularly attractive is that, because it performs actual airflow simulation, as opposed to most simulators which have simple, parameterised models, X-plane can simulate all manner of aircraft. It also includes an application, Plane-Maker, for constructing new aircraft. Thus, we could develop a simulation model of the UAV, down to the finest details.

Additionally, while X-plane doesn't appear to have a flight planning capability built-in, the author encourages 3rd party integration and extension of the software (which is, albeit, closed source) [30]. There is also a plugin API for writing C plugins [31]. Hopefully, these provide the necessary access to the program to implement flight planning.

X-plane runs on Windows, Mac OS X and Linux. It is commercial, but a trial version is available, and it is relatively inexpensive to purchase, at US\$69 (exact cost depends on chosen extras) [32].

### **Google Earth integration and possibly flight planning**

Google Earth was initially the prime candidate for our flight planning needs, given it provides a very powerful but simple interface, and runs on Mac OS X, Windows and Linux. It also has a vibrant community, generating additional data for the program, and providing related tools to extend it.

Unfortunately, as noted the small but critical failure of Google Earth is that it does not provide any way to adjust the elevation of a particular point within a path - only the elevation of the entire path. Thus, it doesn't really serve our purposes very well. Although the plug-in API is not officially supported, it may be possible nonetheless to develop a plug-in which can correct this glaring omission.

Additionally, it would be nice if GPS data from the plane could be plotted in Google Earth - possibly even in real time, using the wireless link. This may be achievable without plug-ins or 3rd party hacks, but will likely require some coercion of the plane's output into a format that Google Earth understands.

### **Improved altimeter**

While the pressure sensor had nothing like the necessary resolution, as noted, it did demonstrate changing pressure with elevation. As such, greater effort should be made in future to include high-resolution pressure sensors, as they clearly have the potential be useful flight control sensors.



### **Use a Gyroscope**

As mentioned in the discussion, the performance of our Autonomous UAV was severely restricted without the use of a gyroscope. For future projects at least a dual-axis gyroscope is recommended, as this would provide stable pitch and bank measurements, providing better flight control. SparkFun Electronics now provides suitable dual axis MEMs based gyroscopes on a breakout board for under \$70US. These are a relatively newly stocked line – at the outset of the project some price estimates we received for gyroscopes were in the order of \$150US - \$200US. The quality of newer, cheaper models needs to be carefully gauged.

### **Upgrade to more powerful micro**

While the ATmega2561 worked well, and reasonably met our current requirements, its limited performance was constraining in some places (e.g. CRC computation). It was difficult to meet the update rate requirements of the PID (10Hz), especially during debugging when the logging system was under heavy load and causing significant delays.

While the ATmega2561 will inevitably be supplanted with a more powerful successor sometime in the future, the 8-bit AVR line is ultimately not intended for the use to which we are trying to put it, and certainly will not be sufficient for future work. Atmel also have a family of 32-bit AVRs, which represent a relatively simple upgrade path from the current 8-bit family. The AT32AP7000 member of the 32-bit family maintains its microcontroller roots, with built-in SPI, UART, etc. It also provides built-in ethernet MACs and USB 2.0 PHYs. It runs at up to 133MHz, which should provide quite enough power for immediate future needs, and hardware DSP which could be extremely handy for image processing.

Alternatively, Atmel also provide a family of FPGA/AVR hybrids under the name FPSLIC, which combine an 8-bit AVR - much like we're already using - with an interconnected FPGA of up to 40K gates. This hybrid approach could be an efficient way to offload some of the work from the AVR, such as menial tasks like CRC computation and signal pre-conditioning.

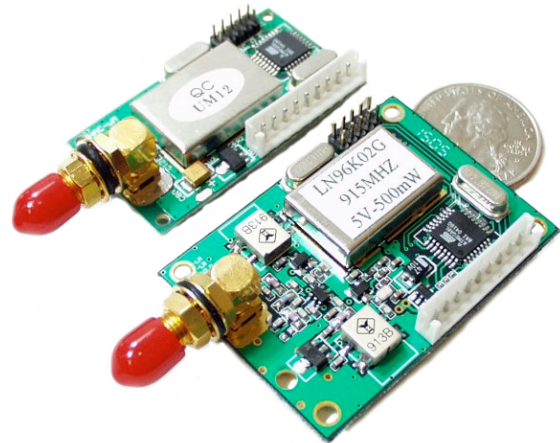
Alternatively, an entirely new processor may be used. At the start of the project the other primary contender was the gumstix line of general purpose Xscale boards. These provide a 32-bit Xscale processor (an ARM7 derivative) and run up to



400MHz, with built-in Bluetooth and other features. They represent a significant shift up in hardware abstraction, however; they run Linux and software would thus be written as kernel extensions and userland processes, as compared to the current model which is a monolithic, bare-to-the-hardware system. They do have a vibrant development community, and use standard headers for connecting expansion boards (which can be custom made).

### **Long range data link**

The Bluetooth data-link used in this project proved to be very reliable, but only over relatively short distances. For future projects of this type a high powered data link with a quoted range in the order of 1.5km should be investigated. These are available from SparkFun Electronics – our suppliers for the Bluetooth modules. They are more expensive – having a total link cost of three times that of the Bluetooth data-link, but could potentially provide constant communication with the UAV, rather than communication only during the takeoff stage.



### **Increase the size of the flight platform**

One of the major constraints within this project was weight. A future project using a larger plane – capable of carrying a larger payload - would open up the possibility for numerous enhancements. A larger aircraft could permanently carry a still camera or video camera, and would allow for the additional weight that the high power radio link would add.

## **7.2 Parting Words**

UAV technology is a rapidly growing area both in military research and civil applications. This project has laid the groundwork for future university UAV research and leaves numerous possibilities for optimization and improvement. As the technology continues to mature we can make one prediction – that more and more aircraft (and other vehicles in general) will have a microprocessor at the helm, rather than a pilot.

It is our hope that others can build upon this project, particularly future engineering students at La Trobe University. We look forward to seeing what the future holds for this burgeoning field.

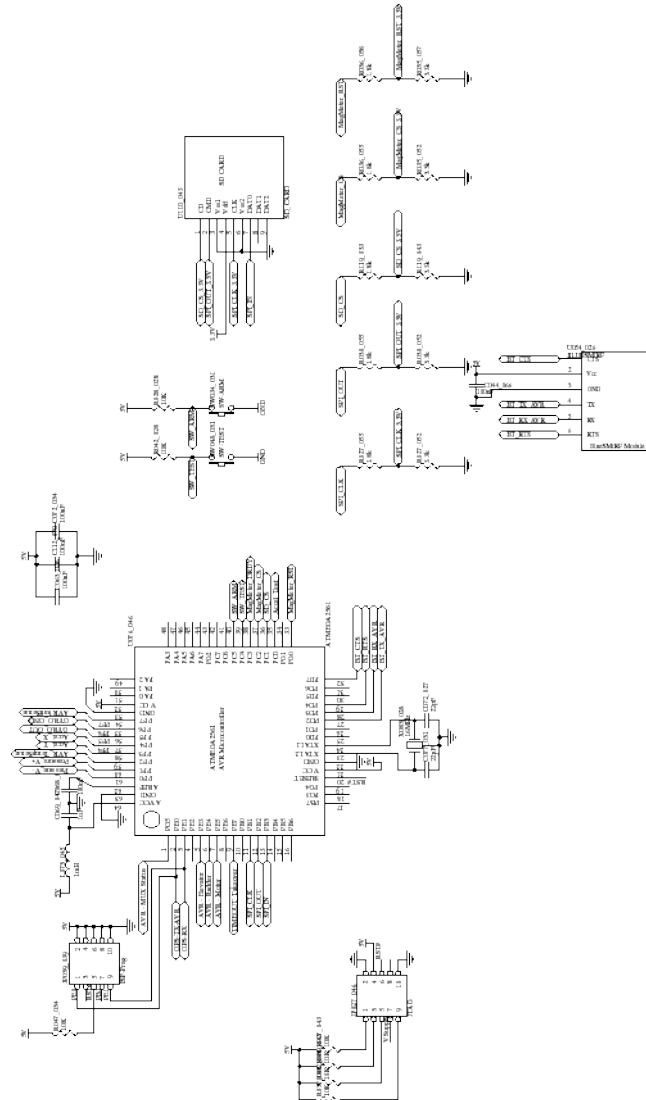
## 8. References

- [1] "ATmega2561", viewed 10/10/2006 [ONLINE]  
[http://www.atmel.com/dyn/products/product\\_card.asp?part\\_id=3631](http://www.atmel.com/dyn/products/product_card.asp?part_id=3631).
- [2] "Imagecraft embedded C Development Tools", viewed 10/06/2006 [ONLINE]  
<http://www.imagecraft.com/software/>.
- [3] "Unmanned Aerial Vehicle", Wikipedia, viewed 5/4/2006 [ONLINE]  
[http://en.wikipedia.org/wiki/Unmanned\\_aerial\\_vehicle](http://en.wikipedia.org/wiki/Unmanned_aerial_vehicle).
- [4] D. K. C. Wong, "Unmanned Aerial Vehicles (UAVs) - Are they ready this time? Are we?," Sydney Royal Aeronautical Society 25/05/2006 1997.
- [5] "Aviation", Wikipedia, viewed 10/11/2006 [ONLINE]  
<http://en.wikipedia.org/wiki/Aviation>.
- [6] "How Airplanes Work", viewed 10/11/2006 [ONLINE]  
<http://travel.howstuffworks.com/airplane1.htm>.
- [7] "NASA Aeronautics Tutorial", NASA, viewed 5/11/2006 [ONLINE]  
<http://virtualskies.arc.nasa.gov/aeronautics/tutorial/intro.html#Intro>.
- [8] "Effects and Operation of Controls," Pilot Training Notes: Civil Aviation Academy.
- [9] "Stall (Flight)", Wikipedia, viewed 10/11/2006 [ONLINE]  
[http://en.wikipedia.org/wiki/Stall\\_\(Flight\)](http://en.wikipedia.org/wiki/Stall_(Flight)).
- [10] "Landing", Wikipedia, viewed 10/11/2006 [ONLINE]  
<http://en.wikipedia.org/wiki/Landing>.
- [11] "Actuators - Servos", Society of Robots, viewed 23/5/2006 [ONLINE]  
[http://www.societyofrobots.com/actuators\\_servos.shtml](http://www.societyofrobots.com/actuators_servos.shtml).
- [12] "RC Servo", Wikipedia, viewed 6/4/2006 [ONLINE]  
[http://en.wikipedia.org/wiki/RC\\_Servo](http://en.wikipedia.org/wiki/RC_Servo).
- [13] "What is a servo?" Seattle Robotics Society, viewed 4/3/2006 [ONLINE]  
<http://www.seattlerobotics.org/guide/servos.html>.
- [14] G. Imahara, *Kickin' Bot - An illustrated guide to building combat robots*: Wiley, 2004.
- [15] "Gyroscope", Wikipedia, viewed 5/8/2006 [ONLINE]  
<http://en.wikipedia.org/wiki/Gyroscope>.
- [16] "Inertial guidance system", Wikipedia, viewed 20/10/2006 [ONLINE]  
[http://en.wikipedia.org/wiki/Inertial\\_navigation\\_system](http://en.wikipedia.org/wiki/Inertial_navigation_system).
- [17] "Meteorology", Wikipedia, viewed 9/10/2006 [ONLINE]  
<http://en.wikipedia.org/wiki/Meteorology>.
- [18] "GPS", Wikipedia, viewed 8/9/2006 [ONLINE]  
<http://en.wikipedia.org/wiki/GPS>.
- [19] "BlueSmiRF Module", SparkFun Electronics, viewed 7/8/2006 [ONLINE]  
[http://www.sparkfun.com/commerce/product\\_info.php?products\\_id=582](http://www.sparkfun.com/commerce/product_info.php?products_id=582).
- [20] J. Puscov, "Flight System Implmeentation in a UAV," in *Department of Physics* Stockholm, Sweden: KTH AlbaNova University Center, 2002.
- [21] "Understanding Servo Tune", National Instruments: NI Developer Zone, viewed 10/7/2006 [ONLINE] <http://zone.ni.com/devzone/cda/tut/p/id/2923>.
- [22] "PID Controller", Wikipedia, viewed 3/8/2006 [ONLINE]  
[http://en.wikipedia.org/wiki/PID\\_controller](http://en.wikipedia.org/wiki/PID_controller).
- [23] "CRC Computations", viewed 14/8/2006 [ONLINE]  
[http://www.nongnu.org/avr-libc/user-manual/group\\_util\\_crc.html](http://www.nongnu.org/avr-libc/user-manual/group_util_crc.html).
- [24] "Google Maps API Version 2 Documentation", Google Maps, viewed 28/6/2006 [ONLINE] <http://www.google.com/apis/maps/documentation/>.

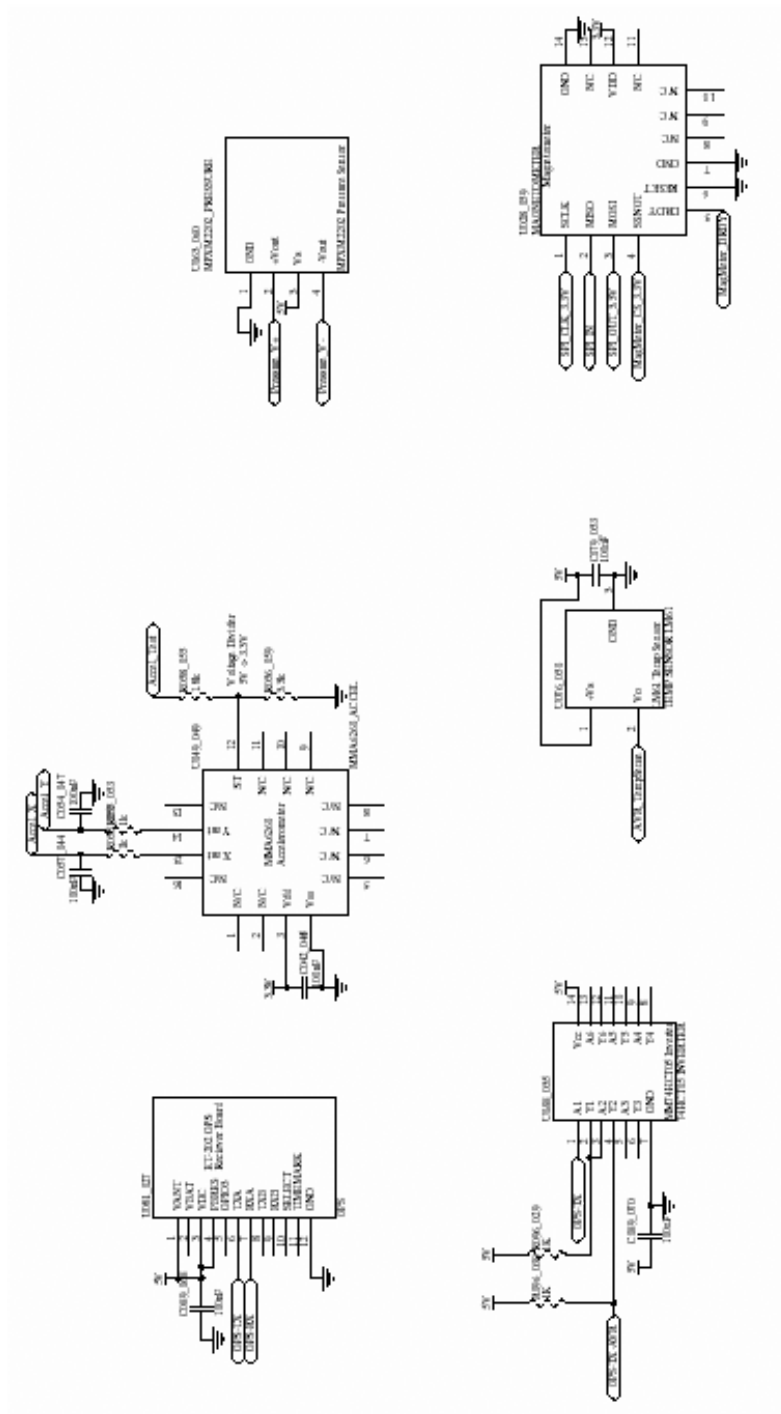
- [25] "ICCAVR Readme File", Imagecraft, viewed [ONLINE] <http://www.imagecraft.com/software/readmeAVR.txt>.
- [26] "Differential GPS", Wikipedia, viewed 8/9/2006 [ONLINE] [http://en.wikipedia.org/wiki/Differential\\_GPS](http://en.wikipedia.org/wiki/Differential_GPS).
- [27] K. d. Jong., ""Success Rates for Integrated GPS and Galileo Ambiguity Resolution", " in *Revista Brasileira de Cartografia*, 2002.
- [28] "X-Plane", viewed 7/9/2006 [ONLINE] <http://www.x-plane.com/>.
- [29] L. Kreider, "X-Plane Approval Notice," FAA, Ed., 2002.
- [30] "Hacking X-Plane", viewed 10/9/2006 [ONLINE] <http://www.x-plane.com/hacking.html>.
- [31] "The X-Plane Plugin SDK Home Page", viewed 12/10/2006 [ONLINE] <http://www.xsquawkbox.net/xpsdk/phpwiki/index.php>.
- [32] "X-Plane: Online Store", viewed 15/10/2006 [ONLINE] <http://store.x-plane.com/>.

# Appendix A: Schematics

## Microprocessor

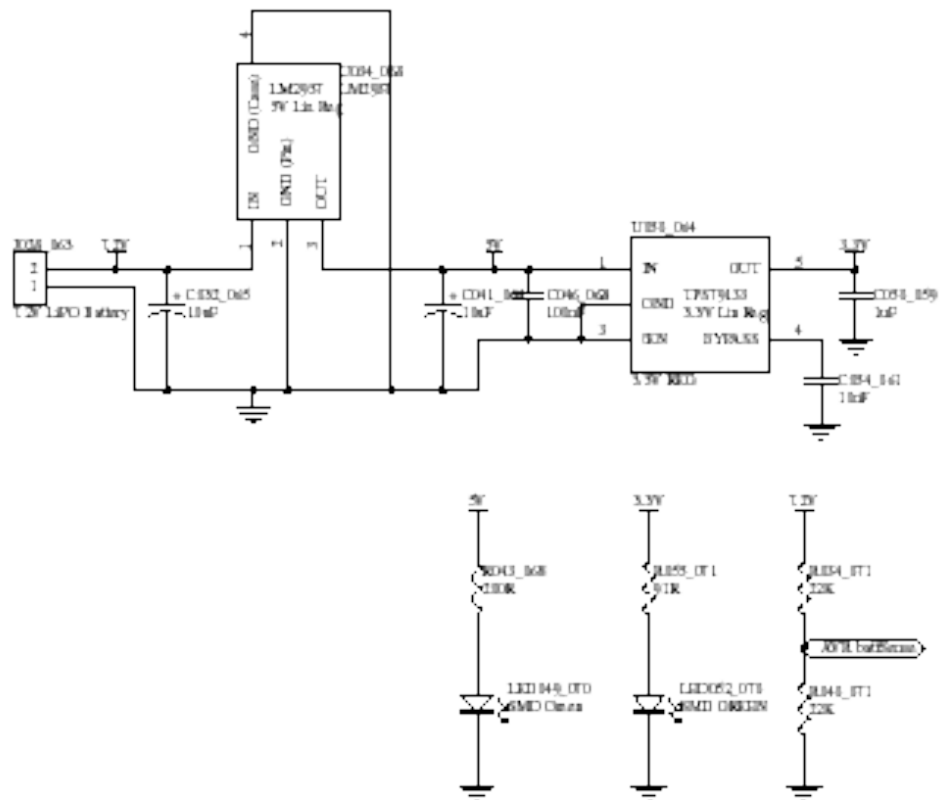


## Sensors

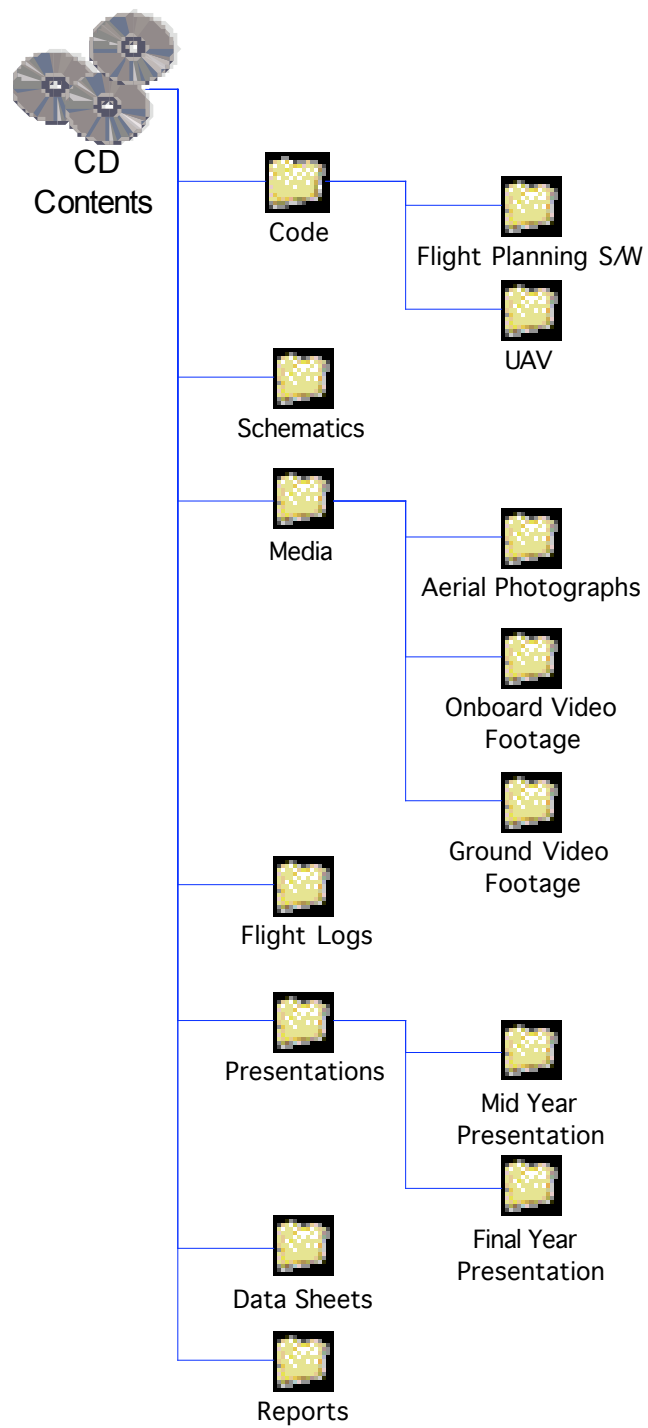


[illegible]

## Power Supply



## Appendix B: CD Contents





## Appendix C: Flight log formatting

Strings within the flight log are formatted as follows:

ADC Data:

"\$UAVADC,BA:720,TE:323,PR:9,AX:353,AY:453\*"

BA = Battery

TE = Temperature

PR = Pressure

AX = Accelerometer X

AY = Accelerometer Y

For GPS data:

"\$UAVGPS,TS:1794299271,LA:-22432475,LO:73183416,AL:93,BE:315,RG:1,GG:1\*"

TS = Time Stamp (multiplied by 1000)

LA = Latitude (multiplied by 60000)

LO = Longitude (multiplied by 60000)

AL = Altitude (MSL)

BE = Bearing (Degrees)

RG = RMCGood (1 = good, 0 = bad)

GG = GGAGood (1 = good, 0 = bad)

Magnetometer data:

"\$UAVMAG,BE:321,AX:-224,AY:716\*"

BE = Bearing (Degrees)

AX = Axis X value

AY = Axis Y value

## Appendix D: Example flight log

A small excerpt from Flight 3

Date 20-10-2006

Time 11:45am

La Trobe Sports Grounds

\$UAVGPS,TS:13515228,LA:-22635001,LO:87025964,AL:79,BE:288,RG:1,GG:1\*  
\$UAVADC,BA:896,TE:174,PR:9,AX:245,AY:310\*  
\$UAVADC,BA:896,TE:173,PR:9,AX:237,AY:316\*  
\$UAVGPS,TS:13516228,LA:-22635000,LO:87025956,AL:79,BE:278,RG:1,GG:1\*  
\$UAVMAG,BE:232,AX:-696,AY:175\*  
\$UAVADC,BA:897,TE:172,PR:9,AX:247,AY:328\*  
\$UAVADC,BA:897,TE:172,PR:9,AX:249,AY:348\*  
\$UAVGPS,TS:13517228,LA:-22635000,LO:87025949,AL:79,BE:284,RG:1,GG:1\*  
\$UAVADC,BA:896,TE:174,PR:9,AX:264,AY:287\*  
\$UAVMAG,BE:248,AX:-536,AY:319\*  
\$UAVADC,BA:897,TE:174,PR:9,AX:302,AY:342\*  
\$UAVGPS,TS:13518228,LA:-22634999,LO:87025942,AL:79,BE:288,RG:1,GG:1\*  
\$UAVADC,BA:896,TE:173,PR:9,AX:259,AY:331\*  
\$UAVADC,BA:896,TE:173,PR:9,AX:280,AY:318\*  
\$UAVGPS,TS:13519228,LA:-22634997,LO:87025936,AL:79,BE:296,RG:1,GG:1\*  
\$UAVMAG,BE:259,AX:-389,AY:317\*  
\$UAVADC,BA:897,TE:174,PR:9,AX:299,AY:329\*  
\$UAVADC,BA:811,TE:173,PR:9,AX:260,AY:282\*  
\$UAVGPS,TS:13520228,LA:-22634996,LO:87025931,AL:79,BE:305,RG:1,GG:1\*  
\$UAVADC,BA:812,TE:173,PR:9,AX:311,AY:331\*  
\$UAVMAG,BE:295,AX:174,AY:878\*  
\$UAVADC,BA:806,TE:174,PR:9,AX:327,AY:386\*  
\$UAVGPS,TS:13521228,LA:-22634996,LO:87025929,AL:79,BE:71,RG:1,GG:1\*  
\$UAVADC,BA:808,TE:173,PR:9,AX:294,AY:296\*  
\$UAVADC,BA:848,TE:191,PR:10,AX:321,AY:254\*  
\$UAVGPS,TS:13522228,LA:-22635001,LO:87025927,AL:80,BE:198,RG:1,GG:1\*  
\$UAVMAG,BE:319,AX:485,AY:474\*  
\$UAVADC,BA:816,TE:173,PR:9,AX:318,AY:239\*  
\$UAVADC,BA:800,TE:172,PR:9,AX:315,AY:320\*  
\$UAVGPS,TS:13523228,LA:-22635010,LO:87025913,AL:80,BE:248,RG:1,GG:1\*  
\$UAVADC,BA:848,TE:191,PR:9,AX:368,AY:440\*  
\$UAVMAG,BE:293,AX:253,AY:1474\*  
\$UAVADC,BA:828,TE:183,PR:8,AX:354,AY:301\*  
\$UAVGPS,TS:13524228,LA:-22635014,LO:87025886,AL:81,BE:267,RG:1,GG:1\*  
\$UAVADC,BA:802,TE:173,PR:9,AX:341,AY:348\*  
\$UAVADC,BA:832,TE:189,PR:9,AX:281,AY:209\*  
\$UAVGPS,TS:13525227,LA:-22635026,LO:87025867,AL:82,BE:196,RG:1,GG:1\*  
\$UAVMAG,BE:270,AX:-324,AY:687\*  
\$UAVADC,BA:802,TE:173,PR:9,AX:206,AY:162\*  
\$UAVADC,BA:920,TE:206,PR:9,AX:374,AY:533\*  
\$UAVGPS,TS:13526227,LA:-22635058,LO:87025877,AL:83,BE:150,RG:1,GG:1\*  
\$UAVADC,BA:815,TE:172,PR:9,AX:243,AY:240\*  
\$UAVMAG,BE:296,AX:82,AY:501\*

\$UAVADC,BA:851,TE:173,PR:8,AX:296,AY:292\*  
 \$UAVGPS,TS:13527227,LA:-22635099,LO:87025889,AL:85,BE:186,RG:1,GG:1\*  
 \$UAVADC,BA:821,TE:172,PR:9,AX:320,AY:402\*  
 \$UAVADC,BA:800,TE:173,PR:9,AX:311,AY:208\*  
 \$UAVGPS,TS:13528227,LA:-22635132,LO:87025877,AL:88,BE:210,RG:1,GG:1\*  
 \$UAVMAG,BE:298,AX:229,AY:830\*  
 \$UAVADC,BA:804,TE:173,PR:9,AX:310,AY:250\*  
 \$UAVADC,BA:800,TE:172,PR:9,AX:353,AY:396\*  
 \$UAVGPS,TS:13529227,LA:-22635158,LO:87025863,AL:92,BE:194,RG:1,GG:1\*  
 \$UAVADC,BA:811,TE:173,PR:9,AX:343,AY:385\*  
 \$UAVMAG,BE:303,AX:329,AY:831\*  
 \$UAVADC,BA:796,TE:172,PR:9,AX:346,AY:404\*  
 \$UAVGPS,TS:13530227,LA:-22635183,LO:87025862,AL:96,BE:172,RG:1,GG:1\*  
 \$UAVADC,BA:807,TE:172,PR:9,AX:339,AY:412\*  
 \$UAVADC,BA:810,TE:172,PR:9,AX:322,AY:427\*  
 \$UAVGPS,TS:13531227,LA:-22635204,LO:87025865,AL:100,BE:174,RG:1,GG:1\*  
 \$UAVMAG,BE:298,AX:127,AY:528\*  
 \$UAVADC,BA:801,TE:173,PR:9,AX:313,AY:310\*  
 \$UAVADC,BA:807,TE:173,PR:9,AX:288,AY:226\*  
 \$UAVGPS,TS:13532227,LA:-22635220,LO:87025860,AL:104,BE:217,RG:1,GG:1\*  
 \$UAVADC,BA:806,TE:173,PR:9,AX:299,AY:244\*  
 BAD GPS RMC String  
 \$UAVMAG,BE:296,AX:251,AY:1038\*  
 \$UAVADC,BA:797,TE:173,PR:9,AX:323,AY:284\*  
 \$UAVGPS,TS:13532227,LA:-22635220,LO:87025860,AL:109,BE:217,RG:0,GG:1\*  
 \$UAVADC,BA:793,TE:173,PR:9,AX:306,AY:179\*  
 \$UAVADC,BA:802,TE:185,PR:9,AX:352,AY:425\*  
 \$UAVGPS,TS:13534227,LA:-22635241,LO:87025846,AL:113,BE:158,RG:1,GG:1\*  
 \$UAVMAG,BE:292,AX:82,AY:769\*  
 \$UAVADC,BA:796,TE:171,PR:9,AX:348,AY:425\*  
 \$UAVADC,BA:804,TE:173,PR:9,AX:325,AY:433\*  
 \$UAVGPS,TS:13535227,LA:-22635245,LO:87025852,AL:115,BE:97,RG:1,GG:1\*  
 \$UAVADC,BA:804,TE:164,PR:9,AX:332,AY:432\*  
 \$UAVMAG,BE:284,AX:-70,AY:912\*  
 \$UAVADC,BA:805,TE:173,PR:9,AX:309,AY:433\*  
 \$UAVGPS,TS:13536227,LA:-22635245,LO:87025850,AL:118,BE:262,RG:1,GG:1\*  
 \$UAVADC,BA:796,TE:173,PR:9,AX:278,AY:400\*  
 \$UAVGPS,TS:13536227,LA:-22635245,LO:87025850,AL:141,BE:262,RG:1,GG:1\*  
 \$UAVADC,BA:804,TE:172,PR:9,AX:285,AY:210\*  
 \$UAVMAG,BE:305,AX:340,AY:733\*  
 \$UAVADC,BA:802,TE:172,PR:9,AX:325,AY:272\*  
 \$UAVGPS,TS:13548226,LA:-22635273,LO:87025512,AL:144,BE:164,RG:1,GG:1\*  
 \$UAVADC,BA:801,TE:172,PR:9,AX:309,AY:372\*  
 \$UAVADC,BA:804,TE:173,PR:9,AX:274,AY:204\*  
 \$UAVGPS,TS:13549226,LA:-22635301,LO:87025522,AL:145,BE:168,RG:1,GG:1\*  
 \$UAVMAG,BE:283,AX:-93,AY:700\*  
 \$UAVADC,BA:795,TE:171,PR:9,AX:292,AY:346\*  
 \$UAVADC,BA:795,TE:177,PR:9,AX:273,AY:199\*  
 \$UAVGPS,TS:13550226,LA:-22635327,LO:87025524,AL:147,BE:188,RG:1,GG:1\*

## APPENDIX E: FAT File System Format - Summary

This is a very brief overview of the FAT file system format. The authoritative reference on this topic is:

1) Microsoft FAT32 File System Specification [21], released as part of their EFI support initiative.

<http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx>

This document was not known of until after the implementation in this project. In lieu of the official references, the following were used:

1) "FAT16 Structure Information", by Jack Dobiash.

<http://home.teleport.com/~brainy/fat16.htm>

2) "FAT32 Structure Information", by Jack Dobiash.

<http://home.teleport.com/~brainy/fat32.htm>

3) "File Allocation Table" on Wikipedia.

<http://en.wikipedia.org/wiki/FAT16>

4) "FAT Boot Sector", by Alex Verstak.

<http://averstak.tripod.com/fatdox/bootsec.htm>

5) "File Allocation Table", by Thomas Kjoernes.

<http://home.no.net/tkos/info/fat.html>

6) "Detailed Explanation of FAT Boot Sector", Microsoft Knowledge Base article #140418.

<http://support.microsoft.com/kb/q140418/>

7) "How FAT Works", Microsoft TechNet.

<http://technet2.microsoft.com/WindowsServer/en/Library/50cd4ffc-1389-423d-9d02-1a898b2eb39f1033.mspx?mfr=true>

8) "Understanding FAT32 Filesystems", by Paul Stoffregen.

<http://www.pjrc.com/tech/8051/ide/fat32.html>

### ***E.1 FAT Boot Sector***

The first sector of every FAT volume is always the FAT boot sector. It contains vital information about the volume, such as its basic parameters (e.g. sector & cluster sizes) as well as identifying information (such as volume serial number and label).

The FAT boot sector differs slightly between FAT12/16 and FAT32. Both formats are shown in Figure N, below.

FAT12 & FAT16																	
Offset (Hex)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00	Jump Instruction			OEM Name									Bytes per sector		SpC	Reserved sector count	
10	#FATs	Max # of root dir entries		Total # of sectors Small		Media desc	Sectors per FAT		Sectors per track		Number of heads		Number of hidden sectors				
20	Total # of sectors Large				Phys drive#	Resv	Boot sig	Volume serial number				Volume label...					
30	...Volume label						FAT file system type										
....	Operating system boot code																
1F0															FAT boot sector marker		

FAT32																	
Offset (Hex)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00	Jump Instruction			OEM Name									Bytes per sector		SpC	Reserved sector count	
10	#FATs	Max # of root dir entries		Total # of sectors Small		Media desc	Sectors per FAT		Sectors per track		Number of heads		Number of hidden sectors				
20	Total # of sectors Large				Sectors per FAT				Flags		Vers Lo	Vers Hi	Root cluster number				
30	FSInfo sector number		Backup boot sector #		Reserved												
40	Phys drive#	Resv	Boot sig	Volume serial number				Volume label...									
50	...Volume label		FAT file system type														
....	Operating system boot code																
1F0															FAT boot sector marker		

Figure 29: FAT Bootsector

The meaning and usage of each field is:

### Intro (Yellow)

**Jump Instruction** - The FAT boot sector is technically executable; older systems wrote their boot loader code into the sector. Modern FAT boot sectors may still contain executable code, and are assumed as such, but must contain all the fixed fields as shown in Figure N. For backwards compatibility, the first three bytes of the sector are assumed to be a jump instruction which redirects execution to an appropriate location after the non-executable portion of the sector.

**OEM Name** - In theory, the name of the software which created the volume. In reality, this is typically always "MSWIN4.1".

### BIOS Parameter Block (BPB)

**Bytes per sector** - The number of bytes per sector. Typically 512.

SpC - Sectors per cluster. Typically 1 on small volumes such as MMC/SDCs.

Reserved sector count - The number of reserved sectors at the start of the volume, including the FAT boot sector. Must be at least 1, to cover the FAT boot sector. The FAT(s) begin immediately after the reserved sectors. Reserved sectors are typically used for backup FAT boot sectors, the FSInfo sector, or other 3rd party uses.

#FATs - The number of FATs (File Allocation Tables) on the volume. Typically 2.

Max # of root dir entries - The maximum number of root directory entries (i.e. the length of the root directory divided by 32, the size of each entry). Not used by FAT32, where it must be set to 0.

Total # of sectors Small - The total number of sectors on the volume (including reserved sectors), if that number is less than 65536 (i.e. if it fits) and the format is FAT12/16; for FAT32, this field is always unused regardless of the number of sectors, and must be set to 0. If set to 0 then the "Total # of sectors Large" contains the total number of sectors instead.

Media desc - A constant indicating the media type. Largely obsolete now; 0xF8 is typically used for permanent media, 0xF0 for removable media.

Sectors per FAT - The number of sectors in each FAT (i.e. the size of each FAT). The meaning is the same for FAT12/16 & FAT32, although the field is in different places and has a different size. FAT32 volumes must set the "old" field to 0.

Sectors per track - The number of sectors per track. Obsolete; a historical relic from an optimisation method that no longer applies.

Number of heads - The number of I/O heads available in the drive. Obsolete.

Number of hidden sectors - Ambiguously defined, only relevant on partitioned media, and in any case obsolete.

Total # of sectors Large - The total number of sectors on the volume, if that number is greater than 65535 or the volume is FAT32 formatted.

### **FAT32 Extensions to the BPB (Blue)**

Flags - Specifies certain attributes of the volume, and how it should be used, as explained below. All other bits are reserved and currently have no purpose.

Bits 0-3 - Index of the active FAT. Only meaningful if mirroring is disabled.

Bit 7 - If clear, FAT mirroring is enabled, otherwise it is disabled. FAT mirroring means all FATs should be updated in parallel; conversely, if it is disabled only the FAT specified by bits 0-3 should be modified.

Vers Lo - The FAT32 minor version. Typically 0.

Vers Hi - The FAT32 major version. Typically 0. The FAT32 version in theory allows for future extensions to FAT32, although in reality it has never been used and

is always 0.0. Drivers should refrain from using volumes that have a newer FAT version than the driver supports.

Root cluster number - The cluster number of the root directory.

FSInfo sector number - The sector number of the FSInfo sector, which must be within the reserved sector area.

Backup boot sector # - The sector containing a backup copy of the FAT boot sector, which must be within the reserved sector area.

### **Extended BIOS Parameter Block (EBPB) (Green)**

Phys drive# - The physical drive number. Obsolete.

Boot sig - FAT boot sector signature. Must be 0x29.

Volume serial number - A unique ID by which to identify the volume (in combination with the "Volume label"). Typically the current time is chosen as the value, when the volume is created.

Volume label - The label of the volume. There is also a special entry in the root directory which identifies the volume label; both should have the same value.

FAT file system type - A string naming the FAT file system in use, typically "FAT16", "FAT12" or "FAT32". In theory this should not be used to identify the format of the volume, although it is consistently set for volumes created by most tools, including Microsoft's.

### **Extras (Red)**

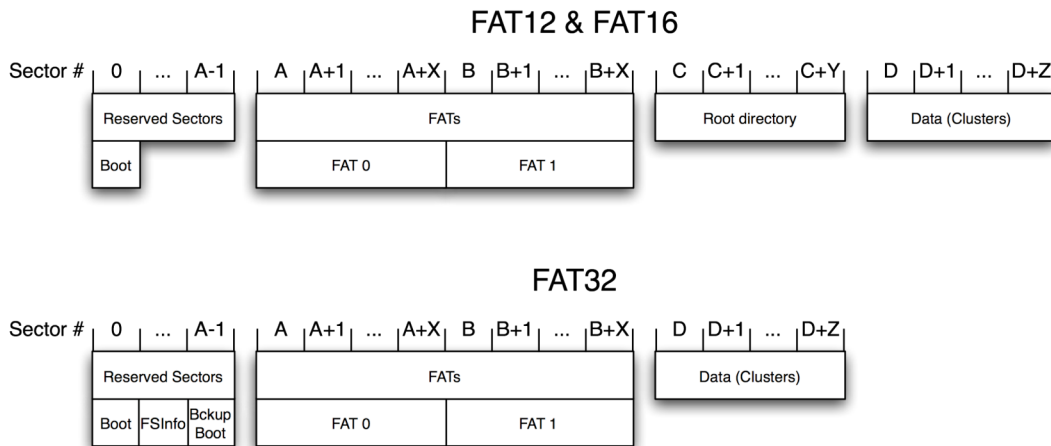
FAT boot sector marker - A marker identifying the sector as a FAT boot sector. Has the value 0x55 0xAA. Commonly assumed to be the last two bytes in the sector, but this is incorrect; it is always at 0x1FE, while the sector size is not necessarily 512 bytes.

As noted, a lot of this is cruft, obsolete and useless. The important fields are the ones that tell us what the size of everything is: bytes per sector, sectors per cluster, sectors per FAT, total number of sectors, reserved sector count and number of FATs. For FAT12/16 the maximum number of root directory entries is also important, while for FAT32 the root cluster number is similarly essential.

Determining exactly which FAT format is used is tricky. There are complex rules defined by Microsoft for distinguishing between the formats, which we won't bother detailing here. As noted, it's usually fine to just read the "FAT file system type" field and compare that against known values (e.g. "FAT32"). For this project that worked correctly with all volumes tested, which were formatted under Windows XP and Mac OS X 10.4.

## E.2 FAT Volume Layout

FAT volumes are divided into either four (FAT12/16) or three (FAT32) main sections, as shown in Figure N, below. In this section we list all values in terms of sectors. To convert to a byte offset, multiply by the value in the "Bytes per sector" field.



**Figure 30: FAT Layout**

The formula for working out these offsets and sizes is as follows:

A = "Reserved sector count"

X = "Sectors per FAT"

B = "Reserved sector count" + "Sectors per FAT"

[More generally, the offset of FAT number N is "Reserved sector count" + (N x "Sectors per FAT")]

FAT12/16:

C = "Reserved sector count" + ("#FATs" x "Sectors per FAT")

Y = ("Max # of root dir entries" x 32) / "Bytes per sector"

[Note: Y will not always be a whole number. It should be rounded up; any extra space in the last block of the root directory cannot be used.]

D = "Reserved sector count" + ("#FATs" x "Sectors per FAT") + ("Max # of root dir entries" x 32) / "Bytes per sector"

FAT32:

D = "Reserved sector count" + ("#FATs" x "Sectors per FAT")



## **E.3 FAT**

The actual FAT (File Allocation Table) is where the file system does all its real work. The FAT is simply an array of numbers of a fixed size - 12 bits for FAT12, 16 bits for FAT16, 32 bits for FAT32. The index of the number is the associated cluster number. Its value can indicate many things, such as the next cluster in a chain, that the cluster is unused, that the cluster contains a bad sector or sectors, etc.

As mentioned, the FAT deals in terms of clusters, not sectors. A cluster may be equivalent to a sector, if "Sectors per cluster" is 1, but this is typically not the case. Clusters are indexed starting from the beginning of the data section, although the first cluster in the data section is cluster #2, not #0; 0 and 1 are special cluster numbers, which don't actually have a physical representation on disk. The first two entries in the FAT of course correspond to these two special clusters, and are consequently not actually used; their values are typically used to represent other information about the disk, although that information is not important to FAT drivers generally.

The types of values each entry in the FAT can have are:

	FAT12	FAT16	FAT32
Available	000	0000	00000000
Reserved	001	0001	00000001
User Data	002 - FF6	0002 - FFF6	00000002 - 0FFFFFFF6
Bad Cluster	FF7	FFF7	0FFFFFFF7
End Marker	FF8 - FFF	FFF8 - FFFF	0FFFFFFF8 - 0FFFFFFF

The "User Data" type is the particularly interesting one; values in this range indicate the next cluster in a chain. The "chain" is the basic building block of files; each file has a cluster number that refers to the first cluster in a chain; additional clusters are added to the chain as necessary to extend the size of the file.

So, for example, the root directory (on a FAT32 volume) typically starts with the first real cluster, cluster #2. The root directory typically has few entries, so it may take up only one cluster; thus the value of entry #2 in the FAT will be "End Marker".

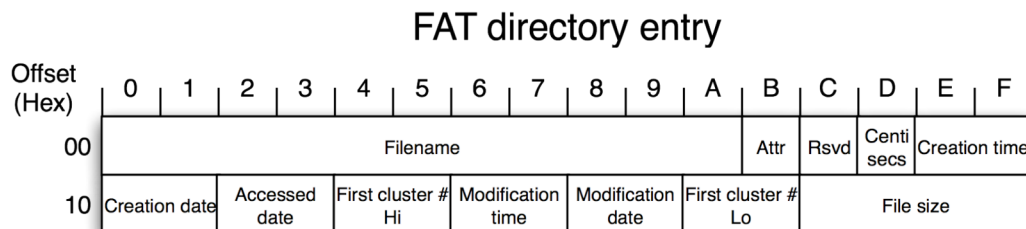
The root directory may contain a file, "Flight.log", whose directory entry indicates cluster #7 as the file's first cluster. Since this file is very large, it spans many clusters. If the next cluster in the chain is #8, then the value of FAT entry #7 will be 8. If the third cluster in the file is #13, the value of FAT entry #8 will be 13. And so forth.

All cluster chains must be terminated with an "End Marker". No valid cluster chain may contain "Available", "Reserved" or "Bad Cluster" clusters.

## **E.4 FAT Directories**

A FAT directory is essentially just like a file as far as the basic driver is concerned - it is just a chain of clusters. In the case of a directory the contents of the "file" are of a

particular format, and can point to other "files" (whether actual files or other directories). A directory is only identified as such by being either the root directory (as specified in the FAT boot sector) or by being linked to from another directory, with the "Directory" attribute set. We'll now look at the structure of each directory entry, to help explain.



**Figure 31: FAT Directory Entry**

Each directory entry is 32 bytes long, and has the structure shown in Figure N, above. Luckily, this structure is more or less the same between all FAT formats. The only difference between FAT32 and FAT12/FAT16 is that the "First cluster # Hi" is of course meaningless under FAT12/FAT16. Luckily, it is essentially unused in most FAT12/FAT16 implementations; OS/2 and early versions of Windows NT used it to refer to an Extended Attributes cluster, but that usage is understandably rare today.

The name field is more than just the human-readable name; the first byte of it indicates the status of the directory entry. If this first byte is a period (0xE5) then the directory entry is currently unused. If it is zero, then not only is the entry unused but it is the last entry in the directory. All directories (except the root directory on FAT12/FAT16 volumes) must be terminated with such a directory entry. A value of 0x05 indicates the first byte really is a period (0xE5), and that the entry is thus used.

Any other values indicate the directory entry is in use, and are the first byte of the filename. Having said this, however, valid filenames may not contain:

- Any values less than 0x20 (excepting the special case of 0x05, as detailed above)
- Any of the values: 0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, or 0x7C.

Filenames should contain uppercase letters only, and any unused portions should be filled with spaces (0x20). The filename is divided into two implicit parts - the first 8 bytes are the pure name (e.g. "FLIGHT "), while the last 3 are the file extension (e.g. "LOG"). The file extension may be empty (all spaces), but the pure name may not (i.e. the filename ".LOG" is invalid and cannot be used).

The remaining fields of the directory entry are:

Attr - The attributes of the entry, which determines both the entry type and additional information, as follows:

Bit 0 - Entry is read-only if this bit is set.

Bit 1 - Entry is a system file if this bit is set (should be protected from the user; maybe not shown at all).

Bit 2 - Entry is hidden if this bit is set (should not be displayed to the user).  
Bit 3 - Entry is a volume label (should only occur once, in the root directory, and should match the volume label as set in the FAT boot sector).  
Bit 4 - Entry is a directory.  
Bit 5 - Entry has been modified since last archive.  
Bits 6 & 7 - Reserved.

Centi secs - Additional resolution for the creation time, in centiseconds (hundreds of a second). Valid values are in the range 0 to 199 inclusive.

Creation time - The time of day at which the file was created, in the bizarre time format:

Bits 0 to 4 - Double-second count, from 0 to 29 inclusive.

Bits 5 to 10 - Minutes, from 0 to 59 inclusive.

Bits 11 to 15 - Hours, from 0 to 23 inclusive.

Creation date - The date on which the file was created, in the format:

Bits 0 to 4 - Day of month, from 1 to 31 inclusive.

Bits 5 to 8 - Month of year, from 1 to 12 inclusive.

Bits 11 to 15 - Years from 1980, from 0 to 127 inclusive (1980 - 2107, respectively).

Accessed date - The date on which the file was last accessed, in the same format as the creation date.

First cluster # Hi - The high 16-bits of the entry's cluster number. Only valid for FAT32; for FAT12/16 should be ignored.

Modification time - The time at which the file was most recently modified (written or created), in the same format as the creation time.

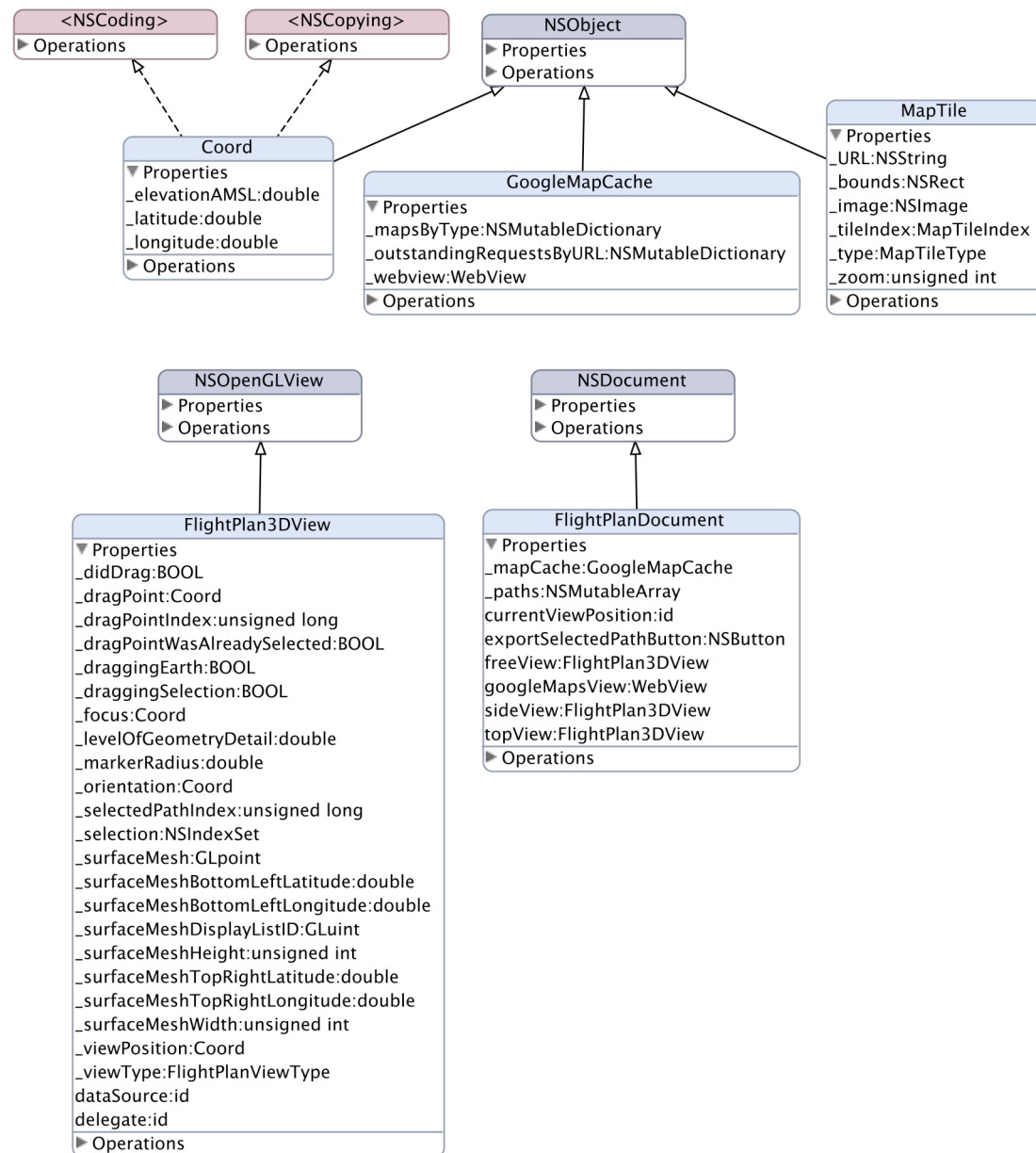
Modification date - The date on which the file was most recently modified (written or created), in the same format as the creation date.

First cluster # Lo - The low 16-bits of the entry's cluster number.

File size - The length of the file in bytes. This is always 0 for directories.

In addition to user-created directory entries, every directory should contain two special directories, for files name "." and "..". These should point to the start of the current directory and the start of the parent directory, respectively. The later may point to 0, in the case of the root directory which has no parent directory.

## Appendix F - Flight Planner Class Hierarchy



**Figure 32: Class Hierarchy**

## Appendix G - Google Maps WebView pre-defined HTML, Map.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
  <title>Google Maps</title>
  <script
src="http://maps.google.com/maps?file=api&v=2&key=ABQIAAAA8b1O
q4C6AoRMOT0VAs83ehSeVCIRwDGHUSJPDdRTzrUmoDeFLBRe9yOLAEVwkP
9WAlS2oUGQ6EX4cA" type="text/javascript"></script>
  <script type="text/javascript">

//<![CDATA[

function resizeMap() {
  mapDiv.style.width = document.style.width;
  mapDiv.style.height = document.style.height;
  googleMap.checkResize();
}

function load() {
  if (GBrowserIsCompatible()) {
    mapDiv = document.getElementById("map");
    googleMap = new GMap2(mapDiv);
    googleMap.setCenter(new GLatLng(37.4419, -122.1419), 13);
    googleMap.setMapType(G_HYBRID_MAP);
    googleMap.addControl(new GLargeMapControl());
    GEvent.addListener(googleMap, "move", function() {
      currentPosition = googleMap.getCenter();
      window.FlightPlanDocument.mapPositionChangedTo(currentPosition.lat(),
currentPosition.lng());
    });
    GEvent.addListener(googleMap, "load", function() {
      window.FlightPlanDocument.googleMapViewLoaded();
    });
  }
}

//]]>
</script>
</head>
<body onload="load()" onresize="resizeMap()" onunload="GUnload()">
  <div id="map" style="margin-left: 0px; margin-right: 0px; margin-top: 0px;
margin-bottom: 0px; padding-left: 0px; padding-right: 0px; padding-top: 0px;
```

```
padding-bottom: 0px; position: absolute; top: 0px; left: 0px; width: 100%; height:
100%;"></div>
</body>
</html>
```