Autonomous Fixed Wing Unmanned Aerial Vehicle

By

Bhathiya Fernando

Thesis

Submitted by Bhathiya Fernando

in partial fulfilments of the Requirements for the Degree of

Bachelor of Digital Systems with Honours (1200)

Supervisor: Dr. Gordon Lowe
Associate Supervisor: Assoc. Prof. Bijan Shirinzadeh

Clayton School of Information Technology

Monash University

November 2006

Contents

List of Figures

List of Tables

Autonomous Fixed-Wing Unmanned Aerial Vehicle

Bhathiya Fernando
Bhathiya.Fernando@csse.monash.edu.au
Monash University, 2006

Supervisor: Dr. Gordon Lowe
glowe@infotech.monash.edu.au
Associate Supervisor: Assoc. Prof. Bijan Shirinzadeh
Bijan.Shirinzadeh@eng.monash.edu.au

Abstract

In recent times autonomous unmanned aerial vehicles, UAV, have been increasingly deployed in military, civilian and academic applications. In order to fly autonomously these vehicles consist of many integrated software and hardware subsystems. The following thesis describes the design and development of many of the subsystems which when integrated and deployed in a small fixed wing aircraft will enable it to fly autonomously.

A software architecture was designed which separates the various subsystems and facilitates their interaction via a specified interface. This architecture was designed with the intention of minimizing computational resources consumed. The architecture is also adaptable to different flight platforms without modification of subsystems unspecific to the flight platform's dynamics. It also allows for the introduction of new modules and or sub-systems without affecting a given configuration of a system, for example the introduction of a new control algorithm or a new sensor module.

Sensory capabilities adequate to facilitate control of a fixed wing aircraft were explored. Software modules for the operation of a Digital Compass, Inertial Measurement Unit and Global Positioning System were created including a viable sensory solution integrating GPS and Digital Compass. Furthermore a framework for a more robust sensor fusion is made available for future work.

A navigation system that provides position and attitude information within a threshold tolerable for the operation of a fixed wing aircraft was implemented based on the sensory data available.

Software to receive operational data from the craft on the ground was created along with a prototype communications data-link design. This software plots the position and attitude of the craft in a graphical user interface. This is done using data transmitted by the craft via the data link in real-time. Functionality to start and terminate the operation of the craft over the data-link is also implemented.

Autonomous Fixed-Wing Unmanned Aerial Vehicle

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Bhathiya Fernando
November 7, 2006.

Acknowledgements

I would like to thank the following people:

- My family for their support.
- The Honours students for providing comic relief throughout this year
- Gary Evans for all his help with hardware related issues
- Tim Ferguson for his time and help in everything Linux
- My supervisors Assoc. Prof. Bijan Shirinzadeh and Dr. Gordon Lowe for their support of this project

Bhathiya Fernando

Monash University
November 2006

1. Introduction

The development of a fixed-wing unmanned aerial vehicle (UAV) is a joint project between Monash University's Faculty of Information Technology and Faculty of Mechanical Engineering's Robotic and Mechatronic Research Laboratory. This is an ongoing project aimed at developing a low-cost fixed wing UAV that can fly autonomously and be applied to real world tasks.

Autonomous UAV are aerial vehicles which operate without being manually controlled by a human operator. Initially development of UAV technology was the domain of military organisations; however in recent times, fuelled by the ever increasing computational power and falling cost of embedded computing and sensor hardware, UAV are being increasingly used for academic and commercial purposes.

The following thesis involves the design and development of a system which when deployed in a small fixed wing aircraft will enable it to fly autonomously. This requires the development of several integrated hardware and software subsystems. The development of such a system in its entirety requires and draws upon expertise from various fields of study. These may include but are not restricted to computer science, electrical and aerospace engineering. As the author's background is in computer science, his contribution and consequently the work described in this thesis will primarily address areas of the project relating to this field of study.

Although this is an ongoing project, the work carried out in the past has been purely investigative in nature and little implementation work has been carried out. The focus of this project is on the design and development of software architecture, ground station software, and the sensor, navigation and data link subsystems, to provide a foundation for future work.

1.1 Thesis Overview

In chapter 1 the direction and motivations for this research are described. Chapter 2 discusses the developed software architecture used to organize and facilitate the interaction of the various sub-systems. Chapter 3 describes the design and development of a prototype communications data link. Chapter 4 presents the development and integration of sensors for the purposes of Navigation. In particular it discusses sensor fusion and inertial navigation. Chapter 5 describes the methods and subsystems created for operation of the UAV control surfaces. Chapter 6 describes a ground control program. The thesis in concluded in Chapter 7.

As the various chapters deal with significantly different material, each chapter will introduce its material, describe the implementation and discuss results and future work.

2. Software Architecture

This chapter describes a modular architecture for the software components carrying out the various tasks needed for UAV operation. This architecture provides for future expansion and modification, inter module communication, and flight platform independence.

2.1 Background

The software components operating on the UAV on-board computer are required to interact with each other, as well as hardware. This often involves integrating sub-systems designed to work independently, for example sensors from different vendors (Horowitz et al, 2002). This operation and interaction must be carried out in a highly coordinated and time constrained manner (Hong et al, 2005). In fact these operations have hard real-time constraints (Dollery, 2001). Due to the high level of interaction between the various components, replacing sub-systems or incorporating new subsystems often results in system failure and requires a complete design overhaul (Horowitz et al, 2002).

Due to the problems posed by developing such systems, entire programming languages have been developed with the aim of overcoming the difficulties involved. These tools depend on other, sometimes proprietary, software packages being available and that the user is competent in using these tools.

Giotto is one example that has been used for UAV applications. It requires the actual "functional" code to read from sensors and control algorithms to be written in another programming language such as C or Oberon whilst the programming of the real-time system is in Giotto. In addition most projects completed with Giotto have been carried out by interfacing Giotto to Simulink. Kirsch et al successfully converted an existing rotary-wing flight controller to Giotto. Figure 1 demonstrates the structure of a Giotto based system. Kirsch et al implemented their system on a port of Giotto for HelyOS, a proprietary real time operating system, (Kirsch et al, 2002).

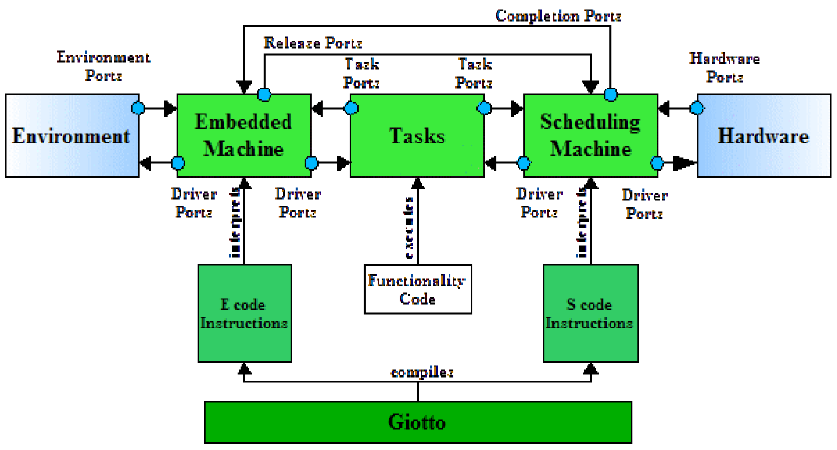


Figure 1: The Giotto System (Giotto, 2006)

Systems such as the one described, serve as a layer of abstraction that hides the unnecessary details of the underlying implementation. These systems are known as a platform (Sangiovanni-Vincentelli et al, 2001).

This section describes the development of a programming library written in C which is used to implement an architecture that provides the functionality to overcome the difficulties discussed. In particular this library provides a programming API that simplifies the creation of separate subsystems and their methods of communication. The architecture designed and implemented facilitates the introduction of new sub-systems and the substitution of existing sub-systems. This programming library and architecture will be used as a platform for developing the UAV.

2.2 Design Considerations

2.2.1 Adaptability and Flight Platform Independence

There are currently three different aircraft in the mechanical engineering laboratory. The developed system should be easily adaptable to any of these or other flight platforms that may be purchased at a future point. That is, the greater part of the system should be non-specific to a particular flight platform. This is a consideration not taken into account in many UAV projects, which are for the most part developed to suit one particular flight platform. Furthermore addition of new sensory capabilities, introduction of new control algorithms should be done with minimal re-engineering of other non-related software modules.

Consider a solution that is developed for a particular fixed-wing aircraft that successfully enables it to fly autonomously. If the same sensor hardware is then transferred to another similar craft, the greater part of the existing software solution should be applicable to the new flight platform. In fact the only software module specific to the new flight platform is the control algorithm. Ideally the old control algorithm should be easily "plugged-out" and a new control algorithm "plugged-in". Now consider another scenario whereby a new sensor is added to an existing system. New software modules should be easily inserted into the system to read from and process the new sensor information and run perfectly alongside the existing modules.

In order to achieve the above-mentioned objectives, the various tasks carried out to operate the craft should be divided into separate modules. These modules should easily interact with each other with no dependence on one another's implementation specifics. A suitable software architecture must be designed and implemented that is conducive to these goals. This architecture should be designed in such a fashion that the system is easily be adapted, changed, and evolved. Entirely new modules, for example to read and process from new sensor hardware, should be easily added with out adversely affecting the system. The necessary methods for expansion and module creation must be made available. This will ensure it remains as a viable platform for future development.

2.2.2 Communications between Modules and Data Integrity

Modules do not need to communicate directly. However there must be some mechanism that allows for the flow and sharing of data. Each module may require some input. This input may be produced by another module, and in turn, this module may produce output that is the input for other modules. There are two scenarios by which this data sharing occurs. In one scenario a module produces many outputs of the same kind for example readings from a sensor, which are processed by another module, one at a time. In the second scenario, two or more modules may operate on the same data structure. These two scenarios are referred to as the consumer/producer problem and mutual exclusion respectively.

Also consider the situation where two different modules are operating on the same data. If one module is half-way between performing some operation on a data structure and at this point another module then begins writing to this data structure, it is clear that the data will then be corrupted. It is clear that there must be a mechanism in place to ensure that this scenario is not allowed to occur. Access to a data structure must be exclusive to a module until it has completed a particular operation on that data structure. The mechanism to allow this functionality is called a semaphore. Two semaphores are required for the consumer producer problem, a semaphore to indicate how many data units are available for processing and another semaphore to indicate the available spots to store new data units. A single semaphore is needed to address the mutual exclusion problem. – either allowing or disallowing access to the data structure in question. For more information on these topics please refer to (Silberschatz et al, 2001)**.**

2.2.3 Efficiency

Another aspect of the software design that must be enforced is the efficient utilization of computational resources. If a process isn't making progress because some resource is unavailable it should then not consume CPU resources until the required resource is available to it. Also, if a process has completed its task for some given time period it should not consume any CPU resources till there is useful work for it to do. This mechanism is referred to as *blocking* and/or *sleeping* (Silberschazt et al, 2001).

2.3 Implementation

The designed software architecture divides all operations to be carried out into "plug-ins". Each plug-in is loaded dynamically at run-time and executed in a separate process. An execution control process is responsible for starting and terminating the execution of all other processes.

Plug-ins create shared-memory locations to store output data. Plug-ins access previously created shared memory locations to access data created by other plug-ins. All shared memory locations and semaphores can be accessed via an integer *key* provided the key is known.

The execution control process creates a hash table, which is then used to keep track of all shared memory locations. This hash table is also stored in shared memory. The hash table is easily accessed by all plug-ins as the key for its shared-memory location is a pre-defined constant. For more information on hash tables and other data structures mentioned in this chapter refer to (Weiss, 1997)**.**

A given shared-memory location is identified by a string. Each location created has a unique string identifier. When a shared memory location is created, a string identifier is provided along with the size in bytes. The hash value of this string identifier is used to calculate an integer. This integer is the position index in the hash table for that shared memory location's entry. This index is also the integer key used to access the shared memory location and its semaphores. In this way unique keys for semaphores and shared memory are generated from a string identifier.

A similar process can be used to retrieve an entry for a shared memory location given its string identifier. This information can then be used to access that location.

Semaphores are used to control access to a given shared memory location. If a plug-in is waiting on a semaphore to allow access to a shared memory location it will be put to *sleep* until the resource is available. This ensures that CPU resources are used efficiently. Furthermore sensor data is only periodically available. Plug-ins reading from sensors are made to *sleep* until sensor data is available by the operating system. In turn plug-ins waiting to process sensor data will *sleep* until the corresponding semaphore indicates there is sensor data to process. In this way CPU utilization is kept to a minimum.

New plug-in modules can be created to interface with existing plug-ins given the string identifiers of their output locations and the data they contain. A programming interface was created to automate the creation of shared memory locations and their associated semaphores.

The following sections describe the above mentioned ideas in detail as well as the overall operation of the implemented software architecture.

2.3.1 Plug-ins

To enable autonomous flight, there are several tasks that need to be carried out in-flight. These include but are not limited to: gathering data from various sensors, processing sensor data, performing various algorithms such as sensor fusion, control algorithms, communications. All these tasks have to be carried out concurrently. There are two possible ways to achieve this concurrency. By separating each task into a separate process or running them within one process as separate threads. To save the overhead of running a thread scheduler and a process scheduler and to keep all operations as separate as possible it was decided that these operations be run as separate processes.

In fact all operations were divided into separate "plug-ins". Plug-ins are loaded dynamically at run-time, given a list of plug-ins to be loaded. No knowledge of a plug-in's operation is required for loading and execution. A plug-in is implemented as a shared library which can be loaded at run time. Each shared library must provide the following four functions:

```
void    __attribute__ ((constructor))    plugin_load      (void);
void    __attribute__ ((constructor))    plugin_unload    (void);
void                                     run_plugin       (void);
void                                     init_plugin      (void);
```

In Linux and QNX the following standard functions are used to open, close and export functions from shared libraries.

```
#include <dlfcn.h>

void    *dlopen     (const char *filename, int flag);
char    *dlerror    (void);
void    *dlsym      (void *handle, const char *symbol);
int     dlclose     (void *handle);
```

Under the current operating system platforms – Linux, QNX – the constructor and destructor are called automatically when a shared library is opened and closed respectively.

The *Execution-Control Process* creates a new process each time a plug-in is loaded. The new process then loads a given plug-in library. The constructor is called automatically upon loading the library. The new process then initializes the plug-in by calling init_plugin (). The plugin is then executed by repeatedly calling run_plugin (). Upon termination, the destructor function is called automatically.

2.3.2 Shared Memory and Semaphores

In both Linux and QNX, each process is given its own virtual address space in which to store and manipulate variables etc. A shared memory location is a memory location that can be mapped into the virtual address space of one or more processes. Data in a shared memory location is available to any process that has mapped that shared memory into its address space. A shared memory segment may have a different memory address in different processes.

Semaphores are operating systems mechanisms used to protect the integrity of shared data and are also used to synchronize the operation of several processes. The two most common operations on a semaphore are *post* and *wait.* A *post* on a semaphore increments its value and a *wait* on a semaphore decrements its value. The value of a semaphore can never be less than zero. A process attempting to perform a *wait* on a semaphore that is zero will block from execution until the function can be performed (that is a *post* is performed on the same semaphore by another process).

The following section outlines how data is shared and protected between different plug-ins. As already stated the data flow and sharing scenarios fall under either consumer/producer or mutual exclusion.

Consumer Producer

In this scenario one plug-in creates a shared memory location to store many units of data of the same kind. In this example we shall assume this data is a reading from a sensor. These sensor readings are gathered at regular time periods and stored in shared memory. However we do not want to worry ourselves with wondering if the last sensor reading has been processed yet (ideally it should have been, however it is best not to make this assumption). For this reason it was decided that the data be stored in a circular queue. The entire shared memory location is a circular queue data structure as depicted in figure 2.

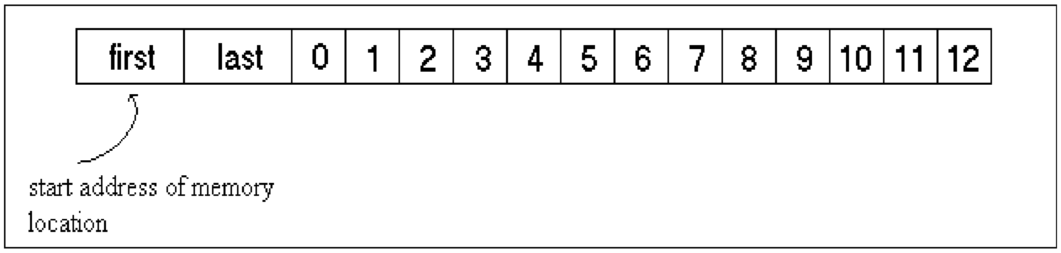


Figure 2: Circular Queue Shared Memory Location

First and last are integer variables which indicate which of the elements currently in the queue was placed in first and which was placed in last. These are effectively indexes into the queue, 0,1,2 ... The positions 0..12 (actual queue size is much larger) are positions in the queue in which data can be placed.

There are two semaphores associated with a shared memory location of this type, which we shall call SEM_0 and SEM_1. SEM_0 indicates the number of data items in the queue. Conversely SEM_1 indicates the number of free positions in the queue. SEM_0 is initialized to zero and SEM_1 is initialized to the number of positions in the queue. The following pseudo code describes how a producer module would insert a data item into the circular queue:

```
wait ( SEM_1 )
insert item  into queue
post ( SEM_0 )
```

The following pseudo code demonstrates how a consumer module would remove and item from the queue.

```
wait ( SEM_0 )
insert item  into queue
post ( SEM_1 )
```

Mutual Exclusion

Mutual exclusion is the situation in which some plug-in stores a data structure in a shared memory location. For example the navigation system might store a data structure containing position and velocity data which will be needed by other modules from time to time.

This scenario requires only one semaphore, SEM_0. This semaphore is initialized to 1. The following pseudo code describes how a plug-in gets exclusive access to this type of shared memory.

```
wait (SEM_0)
perform operations on data structure
post (SEM_0)
```

2.3.3 The Hash Table

All shared memory locations are associated with a unique string, for example "IMUSensorData". When a shared memory location is to be created, the plug-in creating the shared memory must first register this string identifier in the hash table.

A hashing function is used to compute an integer or "hash value" from the string identifier. This integer is the position index in the hash table for that shared memory entry. This index is also the integer key used to access the shared memory location and its semaphores. In this way unique keys for semaphores and shared memory are generated from a string identifier. If a shared memory location already exists in the index calculated from a string identifier of a new location, this shared memory is allocated the next empty position in the hash table and the key produced is not the same as the hash value of the string identifier. The corresponding entry contains the string identifier, the key (as described above the may not be the same as the hash value of the string identifier) and the size in bytes of the shared memory. The plug-in creating the shared memory uses the key produced by this process to create shared memory and semaphores.

A plug-in needing to access existing shared memory need only know its string identifier. This can be used to retrieve the corresponding entry from the hash table. This entry will contain the key used to attach to the share memory and to access the associated semaphores.

2.3.4 Programming Interface

Both Linux and QNX provide the following standard functions to operate on shared memory locations:

```
#include  <sys/ipc.h>
#include  <sys/shm.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>
#include  <sys/types.h>
#include  <sys/shm.h>

        ….

int     shmget      (key_t key, size_t size, int shmflg);
void    *shmat      (int shmid, const void *shmaddr, int shmflg);
int     shmdt       (const void *shmaddr);
int     shmctl      (int shmid, int cmd, struct shmid_ds *buf);
```

The following are standard Linux functions for creating and performing operations on semaphores:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
            ….
int     semctl      (int semid, int semnum, int cmd, ...);
int     semget      (key_t key, int nsems, int semflg);
int     semop       (int semid, struct sembuf *sops, unsigned nsops);
```

As the operation of these functions is tedious a programming interface was created that would create the shared memory locations and semaphores, and perform the necessary operations easily. These functions are displayed in the code listing below:

```
#include "shmem.h"
…….
#define SHM_CIRC_QUEUE 0
#define SHM_SINGLE_DS  1

void shm_create_location (char *identifier
                         , int ds_size
                         , int type, struct shm_access_t *acc);

void shm_open_location (char *identifier
                       , int type
                       , struct shm_access_t *acc);
```

The above functions can significantly reduce the effort of creating and setting up shared memory locations and their semaphores. The *identifier* argument is the unique string identifier of the shared memory location to be created or opened. The *type* argument is one of two pre-defined constants, SHM_CIRC_QUEUE or SHM_SINGLE_DS. Theses indicate weather a circular queue is to be created or a single data structure respectively. If a circular queue is created, *size* indicates the size of each element in the queue; otherwise it indicates the size of the shared memory location to be created. This parameter is not need to open a shared memory location, as it was stored in the hash table when the shared memory was created. The last argument to both these functions is a pointer to a structure. This structure is defined as follows.

```
#include "shmem.h"
....
struct shm_access_t {
int *first;
int *last;
void *circ_queue;
void *addr;
int semid;
};
```

After calling the functions to create or open shared memory, the information to access that shared memory is stored in this structure. If a circular queue was requested then *first* is a pointer to the index of the first (next) element in the queue, *last* is a pointer to the index of the last element in the queue, *circ_queue* is a pointer to the circular queue, *semid* is the semaphore identifier for the pair of semaphores to access the queue and *addr* is ignored. If the shared memory created or opened contains a single data structure, then *addr* is a pointer to the data structure and *semid* is the semaphore identifier for the semaphore to access this data structure and other members of the structure are ignored.

The other part of the programming interface created performs the required operations on semaphores.

```
#include "semaphore.h"
....
int     sem_create          ( int key, int semnum );
int     sem_open            ( int key, int semnum );
void    sem_setval          (int semid, int semnum, int value);
int     sem_getval          (int semid, int semnum);
void    sem_wait            (int semid, int semnum);
void    sem_post            (int semid, int semnum);
void    sem_waitzero        ( int semid, int semnum);
void    sem_remove          ( int semid );
```

2.4 Examples of Expansion Modification

The first example demonstrates how the implemented architecture can facilitate expansion of a developed system. The second example demonstrates the flight platform independence provided by the architecture.

2.4.1 Adding a New Sensor

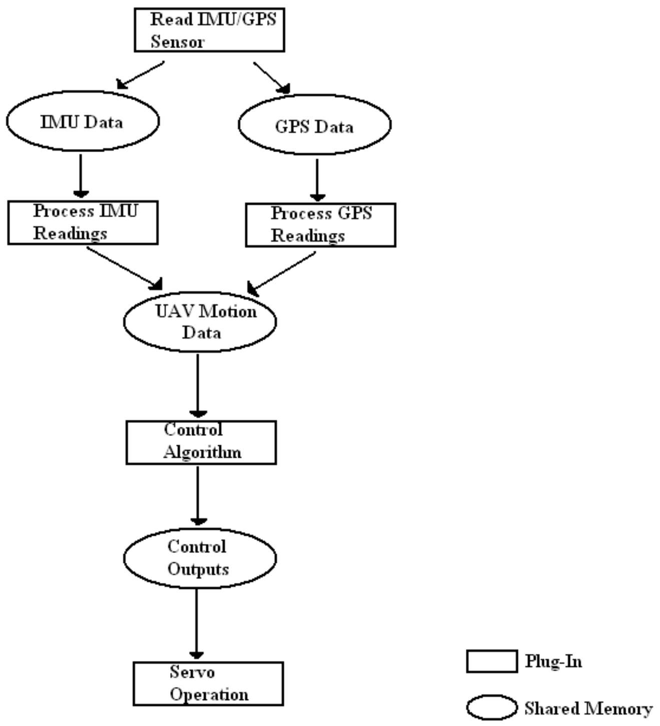


Figure 3: Example arrangement of plug-ins

Figure 3 depicts an arrangement of interacting plug-ins. The rectangles are plug-ins and the ellipses are shared memory. Now consider inserting a module to read from a new sensor. This process is inserted into the system as shown in figure 4 below. This example demonstrates how a new subsystem can be inserted without affecting the operation of existing components.

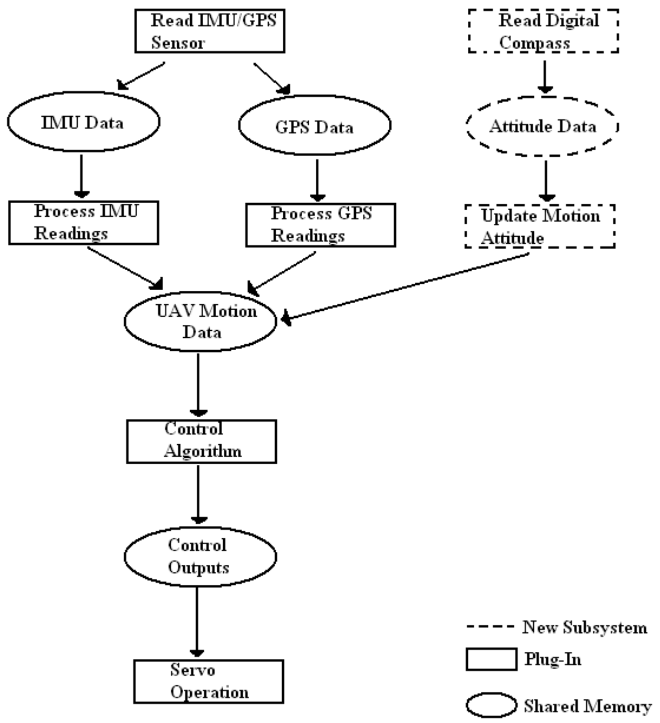


Figure 4: Insertion of new sensor

2.4.2 Installing Into a New Flight Platform

If the UAV system depicted in figure 4, including sensor hardware, was transferred to a different flight platform it would require a new control algorithm. This is because any control algorithm must take into account the particular dynamic flight behavior specific to the flight platform. In this example, for simplicity we will assume that the new flight platform has a similar control surface configuration therefore the old servo operation plug-in does not require substitution. The old control algorithm can simple be replace with a new on. In this way any subsystem can be substituted without affecting the behavior of other subsystems.

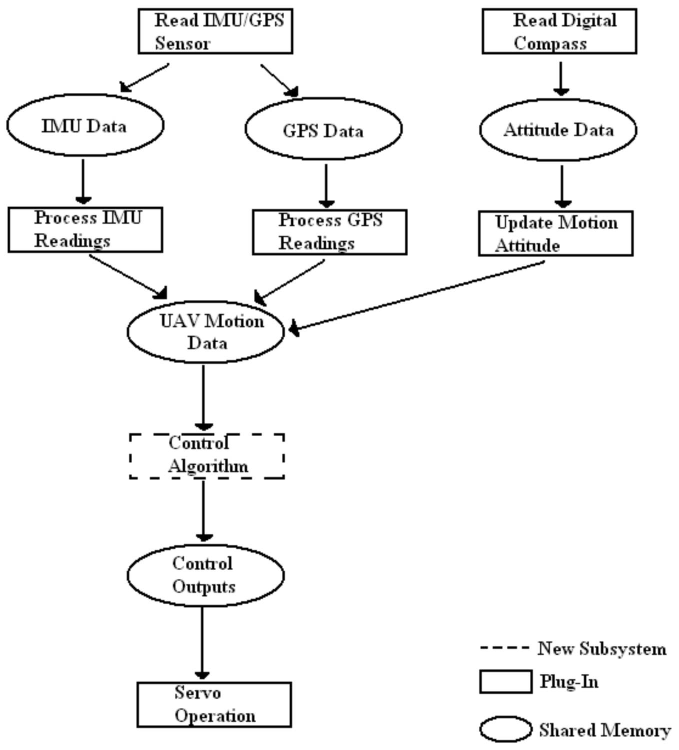


Figure 5: Substituting a new control Algorithm

2.5 Results

All software components for the developed for the UAV were designed using the programming interface and architecture described in this chapter. The advantages it was created to provide were experienced throughout the project.

For example a plug-ins to read from sensors in real-time as well as from saved data in a file were created. These were very easily substituted by simply instructing the executing control process which plug-in it was to load by editing a configuration file, and was very helpful in the testing and development. A new sensor, a digital compass, was also purchased towards the end of the project. This sensor was easily integrated as described in figure 4.

Figure 6 shows the current configuration for the UAV. It depicts the various plug-ins that have been implemented. It also depicts the inter-action between software on the UAV and software on the ground via a data link. The details of the Data-Link and Base station software will be discussed in chapters 3 and 6 respectively.

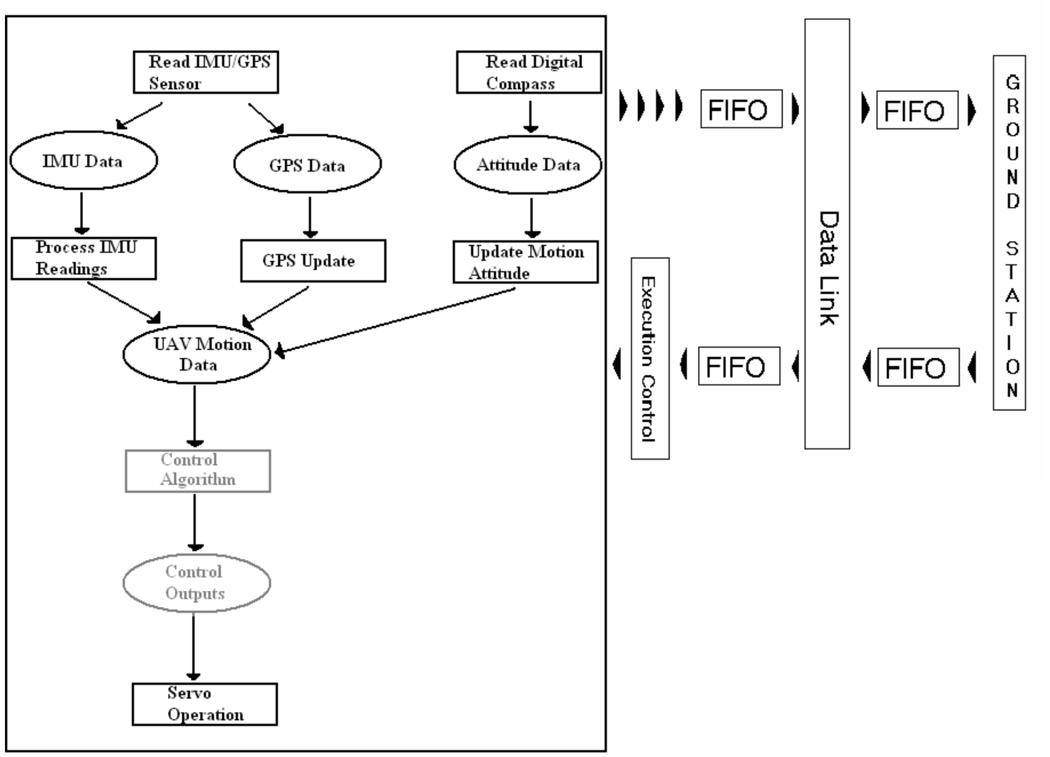


Figure 6: Current UAV architecture and configuration

Also note that the current control algorithm was only created to demonstrate functionality and provide mock control outputs so the servo operation plug-in could be tested.. The development of a control algorithm requires knowledge of aerospace and the development of a practical control algorithm was not a focus of this project.

The GPS update depicted is one of two plug-ins, a Kalman filter or a simple GPS update module. Towards the end of the project a configuration using only GPS and digital compass was decided upon and the inertial measurement unit was ignored (refer to chapter 4).

3. The Data-Link

The ability to communicate with a ground or base station is a necessary requirement for any UAV. This functionality is provided by the data-link which consists of communication hardware and software. This chapter describes a prototype data-link which has been designed.

3.1 Background

There is a variety of possible solutions which may be used to provide communications functionality. Operating range, data rate, reliability and cost are strongly interacting factors in data link design. Furthermore any data link should provide for a) an up-link that allows the ground station to exert some degree of control over the UAV b) a downlink that provides UAV status and sensor data to the ground station (Saeedipour, Azlin, Sathyanarayana, 2005).

Various technologies have been successfully used to implement the data-link however radio frequency (RF) transceivers and receivers combined with modems appear to be the most common solution for low cost UAV. There are also many low-cost commercial off the shelf components available in this configuration; the most popular of which is wireless LAN or WiFi, also known as IEEE 802.11b, and wireless RS-232 (Brown et al, 2004). In the 2006 AUSVI competition, a UAV competition for students from various universities, every team used a data-link of this type (AUVSI, 2006).

Examples of more advanced and consequently more expensive technologies used to provide the data-link are Ultra Wide Band technology used more commonly for ground penetrating radar (Fontana et al), and laser communications (Ortiz et al, 1999). Despite the advantages they offer, due to the expensive cost of these more advanced technologies they are not viable solutions for this project. Radio-modem solutions remain the most ubiquitous for academic and even commercial applications.

As this project aims to develop a low-cost UAV, viable options for communication are Wifi or wireless RS-232. The prototype data-link designed is targeted at wireless RS-232 via a radio modem. These devices provide communications distances in the range of 20km and mechanisms for reliability such as packet acknowledgement and frequency hopping (Downey et al, 2006). Kits including a pair of radio modems can be purchased for under $499 US (Maxstream Products, 2006).

3.2 Implementation

Unfortunately the hardware required for communication, a radio modem, was not available during the course of this project. However a foundation for future work was established. A data-link framework was designed as well as a simple protocol for sending and receiving messages. This simple data-link and protocol was implemented and its performance indicated that it would be a viable solution.

The data link software designed was built around a RS-232 serial connection. This system was tested via a serial cable connecting two computers, one acting as the UAV and the other the ground station.

Although this data-link only serves to validate the proposed concept, it is more than simply a proof of concept exercise. If a wireless RS-232 radio modem were integrated into the system, it would communicate with the existing system via the same means as the serial cable in this simple data link. Specifically, under both QNX and Linux it would be accessed, opened, read and written to, as a file descriptor in the same manner as the serial communications port used for the test data-link.

3.2.1 Structure

The structure of the data-link implemented is shown in Figure 7. It can be seen that the means for sending and receiving data on both ends of the data-link is available. Messages are received by reading from the receive buffer. Writing to the send buffer transmits messages. Further information on the FIFO buffers is given in the next section.

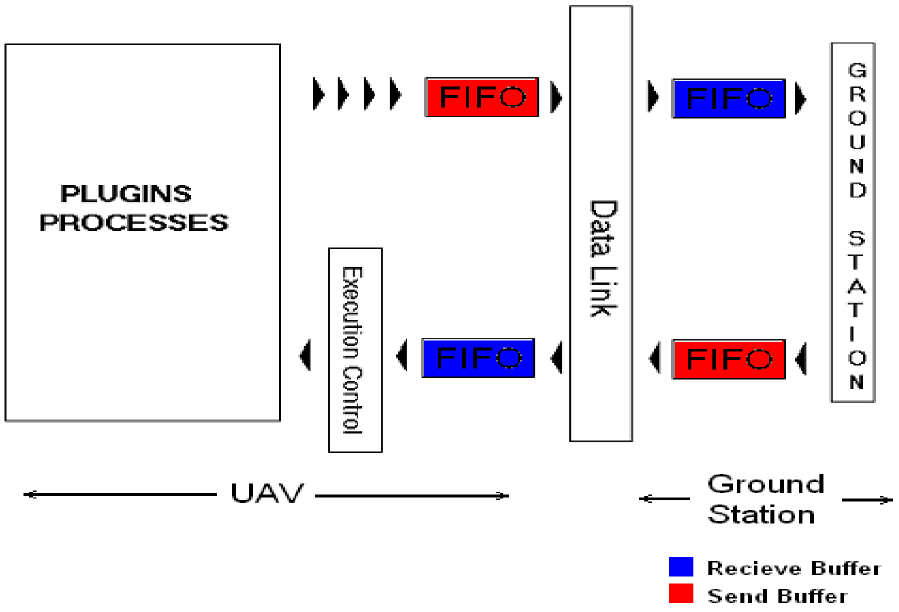


Figure 7: Structure of Data-Link

All messages sent by the Ground Station are commands. These commands are received and carried out by the Execution control process. At the moment these commands are limited to starting and shutting down the UAV plug-ins.

All messages sent by the UAV are from the various active plug-ins. These messages carry information about the status of the vehicle. Typical examples of such messages are: position and attitude data from the navigation system plug-in, fuel levels and temperatures. At the current stage of development theses messages are limited to position and attitude data. As various plug-ins may attempt to write to the data-link at a time, a semaphore is used to control access to the data-link. This access control situation is one of mutual exclusion (refer to chapter 2).

The amount of data that can be transmitted in a time period is a factor that must be taken into account during the use of the data-link. The data link must not be overwhelmed. This limitation must be considered when deciding how often and how much data is transmitted.

3.2.2 First In First Out (FIFO) Buffers

A First In First Out file is a special file available on Unix-like operating systems. This file is viewed as having two ends - one end for writing information to and the other end for reading information from. A program opening a FIFO file for writing will block until the FIFO is also opened by another program for reading and vice-versa. A FIFO is created using the following command:

mkfifo <*filename*>

It may seem that using a FIFO between the data-link and the process communicating may be an unnecessary addition. However, this is not the case.

First and fore most, performing any communication function should not hinder a process' progress. Blocking from execution until transmission or reception of a message is complete, is undesirable. A process should be able to send any data as required and complete any work it is required to do, without the speed of the communications link being a bottleneck. Writing to the serial port directly is bound to hamper the timely execution of the various plug-ins as the operating system will block them from execution until the message has been transmitted. Writing to a file on the hard-disk is considerably faster than writing to the serial port directly.

In this implementation, plug-ins access the FIFO via a semaphore (obtained via a pre-defined key). They write any messages to be transmitted to the FIFO (much faster than writing to the serial port directly). A background process is constantly running "between" the FIFO and the serial-port (this process is viewed as part of the data link and is not depicted in figure 7 for simplicity). This process continuously reads bytes from the FIFO and writes them to the serial port. It is this process that blocks from execution to allow for the slow speed of the serial port. A similar background process reads bytes from the serial port and writes them to the receive FIFO.

Also there may be short periods of heavy traffic over the communications link. Without the use of a FIFO, this would lead to a disastrous situation whereby processes queue up to access the serial port directly, each blocking until their message has been transmitted. As some processes inevitably depend on the progress of other processes for their own operation, this situation can potentially degrade the timely operation of the entire system. The FIFO buffer allows data accumulated during periods of heavy traffic to be dispensed at a steady pace, allowing for the link to "catch-up" as the traffic load is alleviated. However the capability of the data-link must be considered when implementing plug-ins, in order to ensure that the link is not overwhelmed.

3.2.3 Protocol

In order for the receiving party to recognize messages, some convention for the format of messages must be established. In this implementation a simple message format has been created. All messages are 25 bytes in length. These are all null terminated strings. If the actual string is shorter than 25 bytes then the message must be padded to make up the expected length.

As stated in the previous sections messages from the UAV come from the various active plug-ins running. These messages all begin with a three letter identifier. Once this identifier has been recognized the ground station software will know how the information in the remainder of the message is formatted. The remainder of the message is the data being communicated and is dealt with in a pre-determined way.

Messages from the ground station are commands. These must belong to a set of commands recognized by the execution control process. These commands once recognized are then dealt with in a pre-defined way. Table 1 provides examples of messages from both the UAV and the ground station.

Message Transmitted	Type	Information Conveyed
"ATT 82.5 33.6 18.9"	**UAV to Ground: Attitude Update**	**Roll=82.5 Pitch=33.6 Yaw=18.9**
"POS 145.8292 -37.1583"	**UAV to Ground: Position Update**	**Longitude = 145.8292 Latitude = -37.1583**
"START"	**Ground to UAV: Start Plug-ins**	**-----------------------------**
"STOP"	**Ground to UAV: Stop Plug-ins**	**-----------------------------**

Table 1: Message format and meaning

3.3 Results

A prototype data link was designed and implemented. The data link provided the expected functionality when used for communications between the UAV software system and the ground station software. However, it is not as yet mature enough to be a final solution.

3.4 Future Work

A fully operational UAV will require a functional data-link. A prototype for this sub-system has been created however further development is required.

A radio modem must be purchased and integrated into the architecture presented. This is expected to be a trivial task as it will require only minor modifications to the source code already created. Also mechanisms will have to be developed to enforce reliability in the link. This will most likely include some packet acknowledgement system if it is not already integrated into the hardware.

The current protocol is not adequate. It was developed as a demonstrative exercise and is not suitable for the transmission of large amounts of data. This is primarily due to the fact that messages are of fixed length resulting in unnecessary utilization of the data-link when padding short messages with extra bytes.

Also compression and decompression of data before transmission and after reception should be implemented as a mechanism to reduce the amount of data-transfer via the communications link. A mature data link will also require that some form of encryption be carried out on transmitted data to provide for secure communications. This compression/decompression and encryption/decryption will have to be carried out by the background processes running between the FIFO’s and the radio modem.

4. Navigation

The navigation subsystem provides information about the path, trajectory and motion of the vehicle, particularly position, velocity and attitude of the UAV. This data will be used by higher-level subsystems such as control and mission planning as a basis for decision-making. This chapter describes the development of the Navigation subsystem. This includes the integration of sensor hardware and the different configurations that have been experimented with.

4.1 Background

Any operational task will require the UAV to travel to various destinations. In order for the UAV to function autonomously, it is a fundamental requirement that it should know its position relative to some reference point (be it destination or initial position). This functionality is provided by the navigation sub-system. More specifically, navigation is concerned with determining the position of the UAV relative to some desired position, in order to facilitate the higher level guidance and control systems. Guidance is concerned with getting to the destination and control with staying on track (Grewal, Weill, 2001).

The UAV keeps track of information pertaining to its motion by processing data from various sensors. Information about position, velocity and attitude can be obtained via sensors internal to the craft or receiving data relative to local or global beacons (Dollery, 2001).

The use of sensors to measure internal parameters in order to calculate the trajectory of the UAV is known as dead reckoning (Mckerrow, 1991). Dead reckoning as a method of measuring position and velocity tends to result in errors which accumulate over time and require updates from other sensors for correction. Dead reckoning for navigation in aerospace and marine applications is traditionally achieved through the use of accelerometers and gyroscopes mounted within the craft (Grewal et al). These devices are used to measure the accelerations and angular rates of the craft along all three axes and are usually incorporated into an Inertial Measurement Unit. This is referred to as inertial navigation (Fernandex, Macomber, 1962).

GPS or Global Positioning Systems make use of 24 satellites as global beacons to deduce the position of an entity. This technology and its more accurate counterpart, Differential GPS, were designed to facilitate the navigation of civilian, commercial and military entities around the globe. This makes it highly applicable as a solution to provide the navigational functionality of the UAV (Grewal et al).

Data from various sensors inevitably contain some degree of noise. This noise is most often a related to the sensor technology itself. Sensor aiding or sensor fusion refers to combining data from various sensors and is an effective way to increase the reliability of sensor data (McKerrow, 1991). The standard method of sensor fusion for avionic navigation applications is via Kalman Filtering (Grewal et al).

Much work has been undertaken into implementing low-cost navigations systems for UAV applications. The fusion of GPS and Inertial Measurement Units to implement GPS aided Inertial Navigation Systems is well established and is the most common method used in UAV projects of this nature (AUVSI., 2006).

4.2 Inertial Navigation System

An inertial navigations system, INS, relies on data from an Inertial Measurement Unit to measure acceleration and angular rates in the body frame (Kumar, 2004). A set of Navigation Equations is then used to calculate the body frame velocities and attitude. This information is then used to determine the UAV's position and velocity in the navigation frame.

Due to noise in IMU measurements an INS is usually aided by GPS readings via a Kalman Filter. Table 2, describes the kind of errors occurring in IMU readings. As IMU readings are integrated to calculate position, velocity and attitude, these errors grow very quickly and without bound if not compensated for.

Alignment errors	roll, pitch and heading errors
Accelerometer bias or offset	a constant offset in the accelerometer output that changes randomly after each turn-on.
Accelerometer scale factor error	results in an acceleration error proportional to sensed acceleration.
Nonorthogonality of gyros and accelerometers	the axes of accelerometer and gyro uncertainty and misalignment.
Gyro drift or bias (due to temperature changes)	a constant gyro output without angular rate presence.
Gyro scale factor error	results in an angular rate error proportional to the sensed angular rate
Random noise	random noise in measurement

Table 2: Sensor generated errors in the INS, (Kumar, 2004)

4.2.1 Reference Frames

There are many possible implementations of an Inertial Navigation System. The Inertial Navigation System used for this project is described in (Kumar, 2004). There are two coordinate frames used in this INS; the body and navigation frame. The body frame refers to a coordinate frame centered in the body of the craft with axes oriented as depicted in figure 8. As the UAV moves the body frame will undergo some rotation and translation relative to the navigation frame.

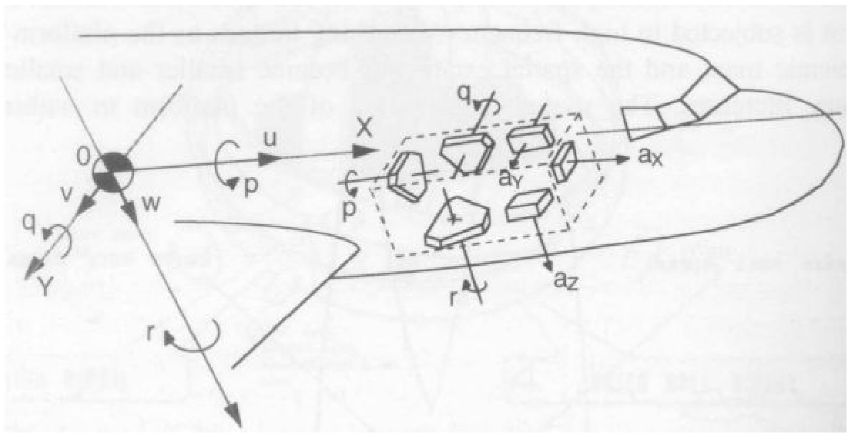


Figure 8: Body Frame (Kumar 2004)

The navigation frame is a fixed coordinate frame. Any destination coordinates passed to the UAV will be given in this frame. This is also a right-handed coordinate frame with the axes oriented as in figure 9, (Kumar, 2001). This is known as a North East Down frame, referring to the direction of the x, y and z axes respectively.

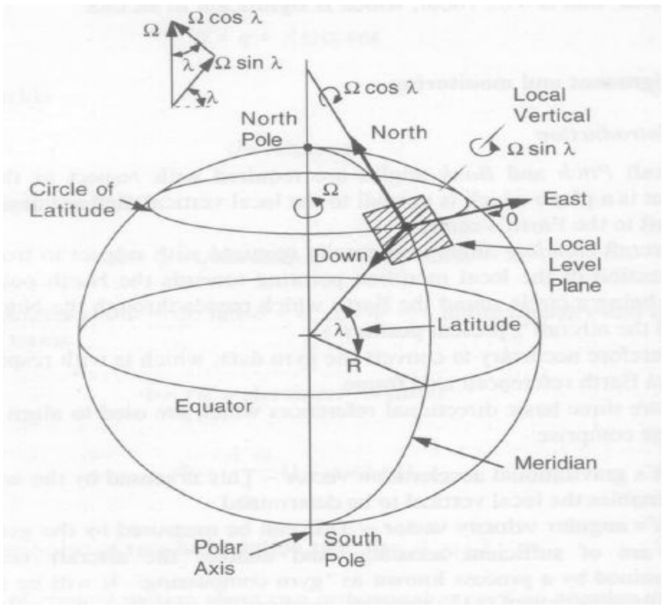


Figure 9: Navigation Frame (Kumar, 2004)

4.2.2 Navigation Equations

As previously stated, the navigation equations take as input the acceleration and angular rates in the body frame provided by the IMU. These measurements are then used to evaluate a set of navigation equations that compute the attitude and position of the UAV. These equations can be divided into a set of equations to calculate attitude and a set of equations to calculate position. The equations presented are particular to the INS described in Kumar 2004.

Attitude

The attitude calculations are carried out using utilizing quaternion algebra. The derivation of the mathematics involved is a non-trivial exercise; however it is not necessary to have an in-depth understanding for this application. A quaternion is made of four components e0, e1, e2 and e3 that we shall refer to as Euler Parameters. The initial values of these parameters are calculated from the initial Euler angles φ, θ, ψ (roll, pitch and yaw respectively).

$$e_0 = \cos\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} + \sin\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2} \tag{4.1}$$

$$e_1 = \cos\frac{\psi}{2}\cos\frac{\theta}{2}\sin\frac{\phi}{2} - \sin\frac{\psi}{2}\sin\frac{\theta}{2}\cos\frac{\phi}{2} \tag{4.2}$$

$$e_2 = \cos\frac{\psi}{2}\sin\frac{\theta}{2}\cos\frac{\phi}{2} + \sin\frac{\psi}{2}\cos\frac{\theta}{2}\sin\frac{\phi}{2} \tag{4.3}$$

$$e_3 = -\cos\frac{\psi}{2}\sin\frac{\theta}{2}\sin\frac{\phi}{2} + \sin\frac{\psi}{2}\cos\frac{\theta}{2}\cos\frac{\phi}{2} \tag{4.4}$$

These parameters must always satisfy the following relation. This requires normalization once these parameters have been updated.

$$e_0{}^2 + e_1{}^2 + e_2{}^2 + e_3{}^2 = 1 \tag{4.5}$$

The rates of change in these parameters are related to the angular rates; these rates are periodically integrated to calculate the value of the Euler parameters at a given time.

$$\dot{e_0} = -\frac{1}{2}(e_1p + e_2q + e_3r) \tag{4.6}$$

$$\dot{e_1} = \frac{1}{2}(e_0p + e_2r - e_3q) \tag{4.7}$$

$$\dot{e_2} = \frac{1}{2}(e_0q + e_3p - e_1r) \tag{4.8}$$

$$\dot{e_3} = \frac{1}{2}(e_0r + e_1q - e_2p) \tag{4.9}$$

The roll, pitch and yaw of the vehicle can then be calculated using the following equations.

$$\theta = \sin^{-1}[-2(e_1e_3 - e_0e_2)] \quad (4.10)$$

$$\phi = \cos^{-1}\left[\frac{{e_0}^2 - {e_1}^2 - {e_2}^2 + {e_3}^2}{\sqrt{1 - 4(e_1e_3 - e_0e_2)^2}}\right]\text{sign}[2(e_2e_3 + e_0e_1)] \quad (4.11)$$

$$\psi = \cos^{-1}\left[\frac{{e_0}^2 + {e_1}^2 - {e_2}^2 - {e_3}^2}{\sqrt{1 - 4(e_1e_3 - e_0e_2)^2}}\right]\text{sign}[2(e_1e_2 + e_0e_3)] \quad (4.12)$$

Position and Velocity

The position of the UAV is determined using both accelerations (*ax*, *ay* and *az*) and angular rates (*p*, *q*, *r*) in the body frame. Again this information is acquired using an Inertial Measurement Unit. *U*, *V*, *W* are the velocities in the body frame, *g* is the acceleration due to gravity (calculated with respect to current latitude, longitude, and height). As these velocities are in the body frame, they are transformed into the navigation frame using the direction cosine matrix from body to navigation (described below) and integrated to obtain the position of the UAV in the navigation frame.

Accelerations in the body frame are calculated using the following equations.

$$\dot{U} = a_X + Vr - Wq + g\sin\theta \quad (4.13)$$

$$\dot{V} = a_Y - Ur + Wp - g\cos\theta\sin\phi \quad (4.14)$$

$$\dot{W} = a_Z + Uq - Vp - g\cos\theta\cos\phi \quad (4.15)$$

These accelerations are integrated to obtain the velocities in the body frame. Velocities in the navigation frame are calculated as shown below.

$$\begin{bmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{bmatrix} = \begin{bmatrix} V_N \\ V_E \\ V_D \end{bmatrix} = \mathbf{C}_N^B \begin{bmatrix} U \\ V \\ W \end{bmatrix} \quad (4.16)$$

$\mathbf{C}_N^B$ is the transformation matrix from body to navigation defined as :

$$\mathbf{C}_N^B = \begin{bmatrix} \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ \cos\theta\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi \\ -\sin\theta & \sin\phi\cos\theta & \cos\phi\cos\theta \end{bmatrix} \quad (4.17)$$

4.2.3 Kalman Filter

Sensor fusion of GPS and INS (which relies on IMU readings) is achieved via a Kalman filter. A Kalman filter is a recursive algorithm that can *"conveniently integrate navigation sensor data to achieve optimal overall system performance"* (Levi J., 1997). It does so using stochastic methods and a state space model of the current system. In this application it is used to estimate the errors of the current INS calculations using GPS readings. The number of errors estimated depends on the order of the Kalman filter; for navigation purposes these can be errors in position, velocity and attitude as well as bias, offset and scaling errors in the accelerometers and gyroscopes.

The Kalman filter consists of computing a series of recursive equations, known as Riccati equations, to appropriately weight a given measurement based on previous information as depicted in figure 10. The recursive nature of this filter makes the filter attractive for real-time systems because no batch processing (of system history) is required. The derivation of the Riccati and other equations, and the theoretical particulars of the filter are beyond the scope of this thesis, and the reader is referred to Minkler, Minkler, 1993.

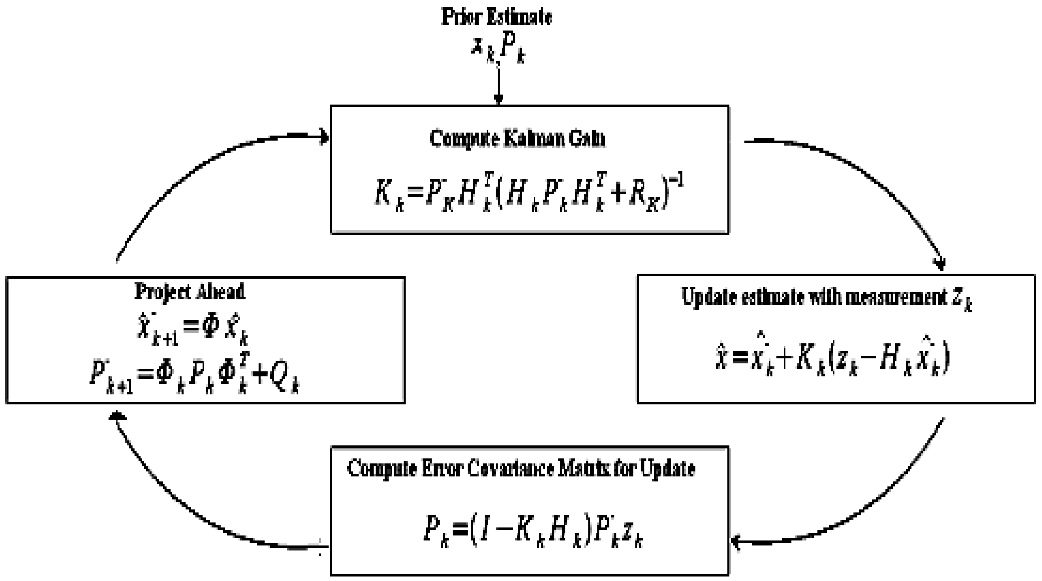

Prior Estimate
x_k, P_k
Compute Kalman Gain
$K_k = P_K^- H_k^T (H_k P_k^- H_k^T + R_K)^{-1}$
Update estimate with measurement z_k
$\hat{x} = \hat{x}_k^- + K_k(z_k - H_k \hat{x}_k^-)$
Compute Error Covariance Matrix for Update
$P_k = (I - K_k H_k) P_k^- z_k$
Project Ahead
$\hat{x}_{k+1}^- = \Phi \hat{x}_k$
$P_{k+1}^- = \Phi_k P_k \Phi_k^T + Q_k$

Figure 10: Recursive calculation of Riccati equations (Kumar, 2004)

$\hat{x}$ is a 9 element state vector containing the estimated errors in the INS calculations and P is the process error covariance matrix . Φ is the state transition matrix, derived from the system dynamics matrix (Kumar, 2004).

$$\Phi \approx I + F\Delta T \tag{4.18}$$

The measurement is the difference between calculated position and the position provided by GPS.

$$z_k = r^n_{INS} - r^n_{GPS} = \begin{bmatrix} \lambda_{INS} - \lambda_{GPS} \\ \mu_{INS} - \mu_{GPS} \\ h_{INS} - h_{GPS} \end{bmatrix} \tag{4.19}$$

Qk is the process noise matrix calculated using a design matrix G, initial covariance Q and the state transition matrix.

$$Q_k = \Phi_k G Q G^T \Phi_k^T \Delta T \tag{4.20}$$

Q contains the error variance of the accelerometers and gyroscopes.

$$Q = diag\,(\sigma^2_{ax}\,\sigma^2_{ay}\,\sigma^2_{az}\,\sigma^2_{p}\,\sigma^2_{q}\,\sigma^2_{r}) \tag{4.28}$$

These errors are then fed back into the system to correct the INS calculations as shown in figure 11.

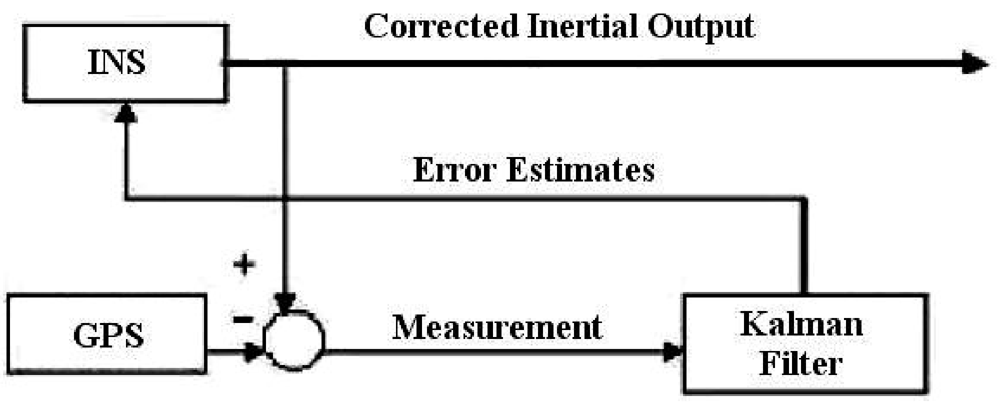


Figure 11: Feedback of Kalman filter error estimates for GPS aided INS

4.3 Implementation

At the commencement of this project there was no sensory hardware available; however a similar project run by the same department involving a rotary wing UAV had purchased a combined GPS/IMU unit. This sensor unit was used to develop a GPS aided INS. This exercise was carried out to determine how appropriate this sensor configuration was in order to decide which sensor components should be purchased for the project. A digital compass was eventually purchased to implement a navigation system combining GPS and digital compass.

All components of the navigation system were implemented as plug-ins and interfaced via the software architecture described in section 2.

Phoenix O-Navi Integrated Global Positioning System and Inertial Measurement Unit

Software modules were developed to read sensor information from the Phoenix O-NAVI GPS/IMU unit. This software obtained inertial sensor readings at a rate of 75Hz and 1Hz for GPS. Testing of the Inertial Measurement Unit led to the eventual conclusion that much of the information provided in the vendor's documentation was not accurate. This included data transmission formats, positive direction of axes and rotation about the axes. O-NAVI was contacted in attempts to acquire more recent documentation however they failed to respond. The difficulties experienced and conclusions arrived at are corroborated by (Yagen, 2005). Software was also developed to read only GPS data form the Phoenix sensor unit in order to provide functionality for a GPS and digital compass navigation system.

Figure 12: Phoenix O-Navi Integrated IMU/GPS (O-Navi, 2006)

Digital Compass

A digital compass was purchased to provide roll, pitch, and heading information. The purchased digital compass is the Ocean-Server OS-3000 (Ocean Server: Compass, 2006). This digital compass integrates two gyroscope/accelerator pairs as well as a flux gate IC. On board filtering provides tilt compensated heading, roll and pitch information with accuracy in the range of ±0.1 degree. It serves as a complete attitude and heading reference system. The OS-3000 interfaces to a computer via a RS-232 connection and provides sensor data at 20Hz. A software plug-in module was created to integrate this sensor into the software architecture.

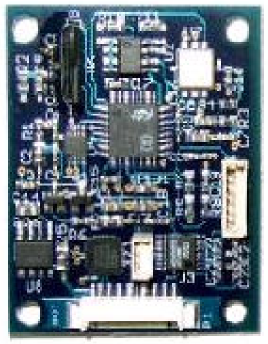


Figure 13: OS-3000 Digital Compass, (Ocean Server: Compass, 2006)

Inertial navigation System and Kalman Filter
A software module was created to implement the inertial navigation equations described in section 4.2.2.

The nine state Kalman filter described in (Kumar, 2004) was implemented into a separate module to interface with Inertial Navigation module. The filter module performs periodic correction of INS calculations upon reception of GPS sensor data. This Kalman filter estimates the error in the INS calculations for position, velocity and attitude.

4.4 Results

Real world data was collected on a trip around Monash University. This path as described by GPS readings from the data collected is shown in Figure 14 (screenshot of base-station software). Short periods of what appears to be GPS outage are marked with red circles. This data also serves to demonstrate that GPS is a reasonably accurate source of position (and hence velocity) data.

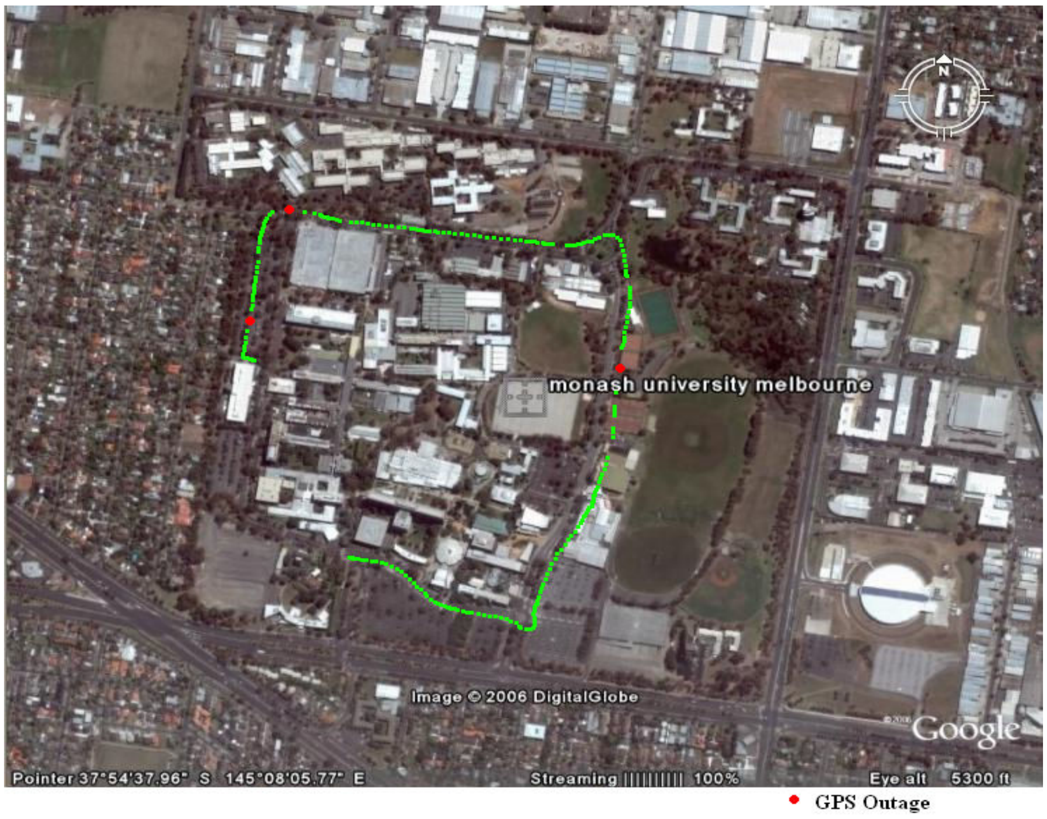

monash university melbourne
Image © 2006 DigitalGlobe
Google
Pointer 37°54'37.96" S 145°08'05.77" E
Streaming |||||||||| 100%
Eye alt 5300 ft
● GPS Outage

Figure 14: Path travelled

An example of running the GPS aided INS on the data collected is shown in figure 15 (information is plotted with respect to latitude and longitude unless otherwise stated). It can be seen that the calculations made in the INS are very inaccurate. It can also be seen that every time the Kalman filter is executed the INS calculations are adjusted. From this it can be deduced that the Filter is working however the noise in the inertial sensor data is too large. If a higher order Kalman filter, estimating offset, bias and scale errors in the IMU was implemented this could be rectified as inertial data can be adjusted upon reception; however the quality of inertial data in this implementation was still too poor to for navigational purposes.

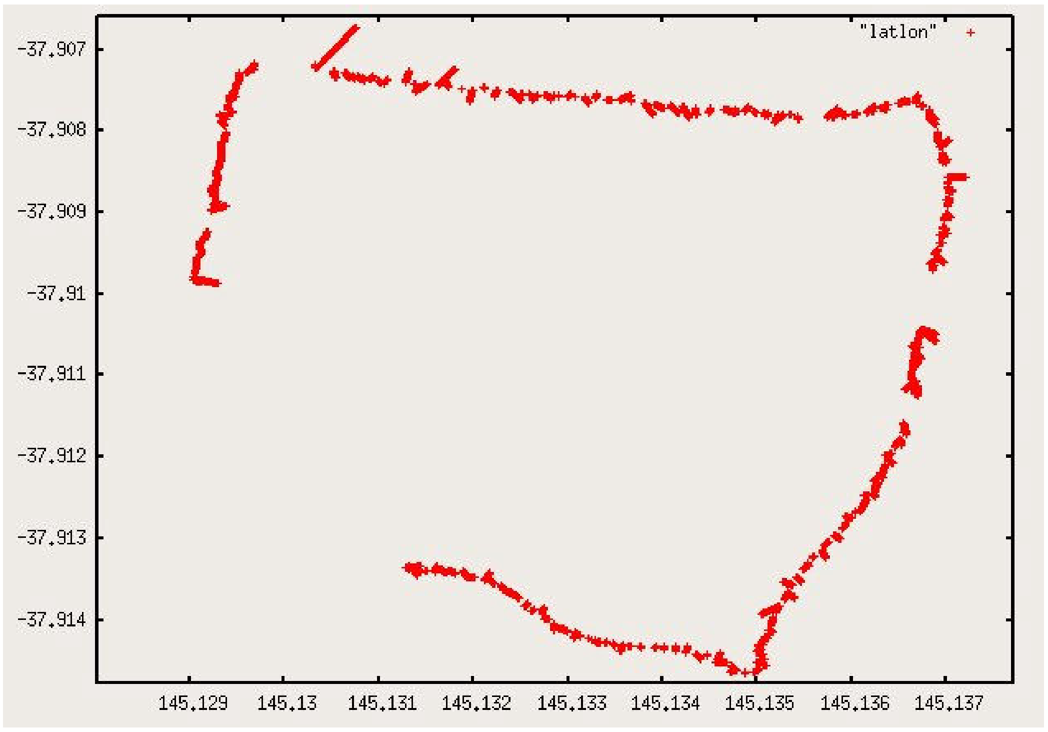


Figure 15: Results of running GPS/INS on real world data

In order to further verify the correct operation of the filter, the system was then run on the same data but this time inertial sensor readings were zeroed. It was assumed that the filter would still correct position, velocity and attitude. The INS would still integrate velocity values to calculate position. The results are depicted in figure 16 below.

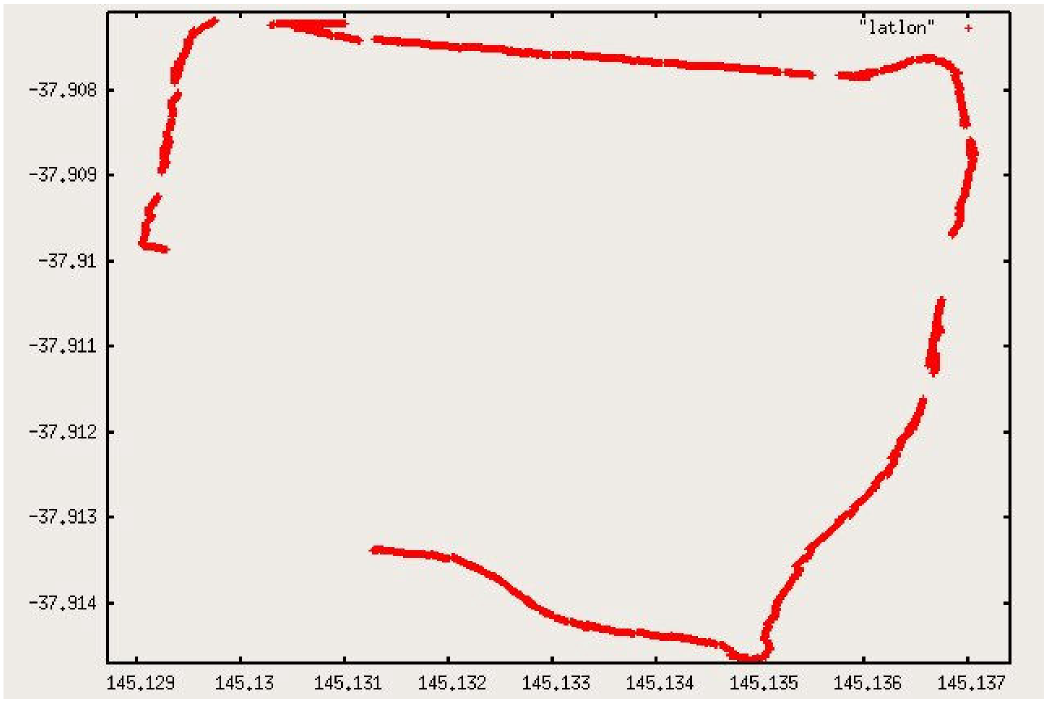


Figure 16: Results of running GPS/INS with inertial sensor data zeroed

From figure 16 it can be seen that the filter accurately estimates the trajectory of the vehicle. However also note however that velocities tend to be exaggerated after periods of GPS outage (top left corner of plot). This is due to the Kalman filter assuming that GPS readings occur every second. After periods of GPS outage the filter assumes that the plane has traveled from its last recorded position to where it is now in one second resulting in high estimates for velocity. This can be rectified by making the time between filter executions a variable stored with the filter module.

These results indicate that GPS is a satisfactory method for fixed-wing UAV navigation, at least as far as the path of the UAV is concerned. However control requires attitude data more frequently and with more accuracy than GPS can provide, especially during periods of GPS outage that typically lasts for a few seconds.

For this reason the OS-3000 digital compass was purchased. As previously stated this chip provides roll, pitch and heading with ±0.1 degree accuracy. The on-chip filtering also results in a great reduction in the computation load imposed by the navigation subsystem. The resulting navigation system accurately provides position and velocity data from GPS and attitude data from the digital compass.

The navigation system developed will satisfy the requirements of a fixed wing UAV. A GPS only sensor can be purchased for under $400 AU and the digital compass was purchased for just under $420 AU. This is a low cost sensor system that is congruent with the goals of creating a low cost UAV solution.

4.5 Future Work

Although the final sensor configuration decided upon provided adequate navigation information to facilitate higher level guidance and control, a more robust navigation system can be designed. This system would involve sensor fusion of all three sensors, GPS, IMU and the digital compass, via a Kalman filter. It has recently come to the author's attention that an open source UAV project (Freshmeat.net: Project Details for Autopilot UAV, 2003) provides Matlab source code for such a filter. This code would be a good starting point for the development of such a system.

5. Control Surface Operation

Aircrafts change their velocity, trajectory and attitude by changing the position of control surfaces. These are ailerons, rudder and elevators. This chapter describes the movement of control surfaces by operating servomotors connected to them.

5.1 Background

An aircraft controls its directions of flight and orientation by changing the position of its control surfaces (ailerons, elevators, rudder), and the amount of thrust delivered by its engine (throttle), (Kermode, 1994). These control surfaces are depicted in figure 17. The exact configuration of these control surfaces varies depending on the flight platform. In order for the UAV to change its velocity, trajectory and attitude, it is essential to implement functionality that enables the on-board computer to position the control surfaces and specify throttle levels. The positions of the control surfaces and the level of thrust are the outputs of control algorithms.

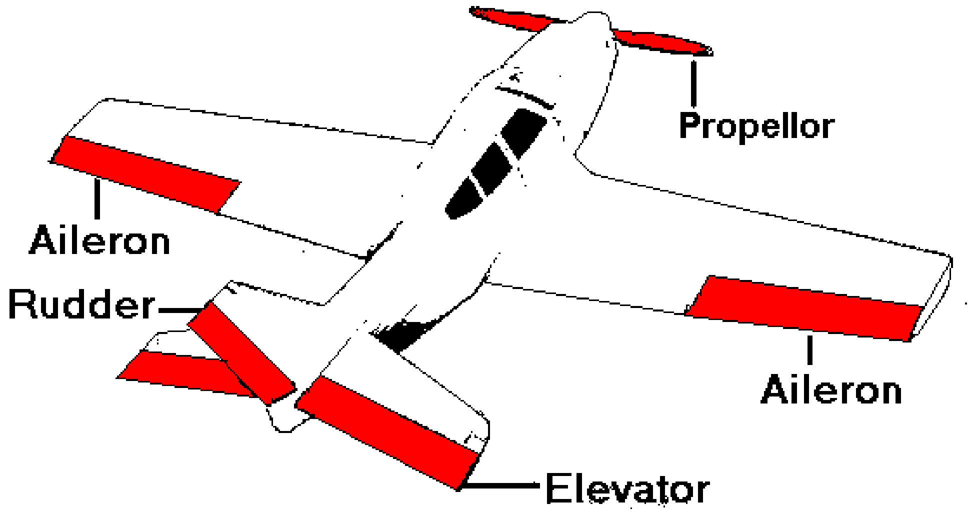


Figure 17: Aero plane control surfaces (Aviation History Online Museum, 2002)

The control surfaces of the UAV and the engine throttle lever are connected to servomotors. These servomotors are typical of those deployed in hobby radio controlled aero planes and helicopters. The position of the control surfaces and throttle levels are specified by the control algorithms outputs. The software responsible for servomotor operation specifies the position of the servomotors to a servo-controller card typically interfaced via a serial connection. A single servo-controller card controls many servomotors.

The servo controller card translates the specified position of a servo to a pulse width modulated square wave. This train of pulses is the command signal to the servomotor. Servo motors contain an internal feedback control which allows the length of positive input pulses, to accurately describe the position of the motor. The internal feed back control loop generates an error signal to move the motor and maintain the commanded position. The pulse width is typically between one and two milliseconds. This train of pulses is illustrated in figure 18 (Dollery, 2001).

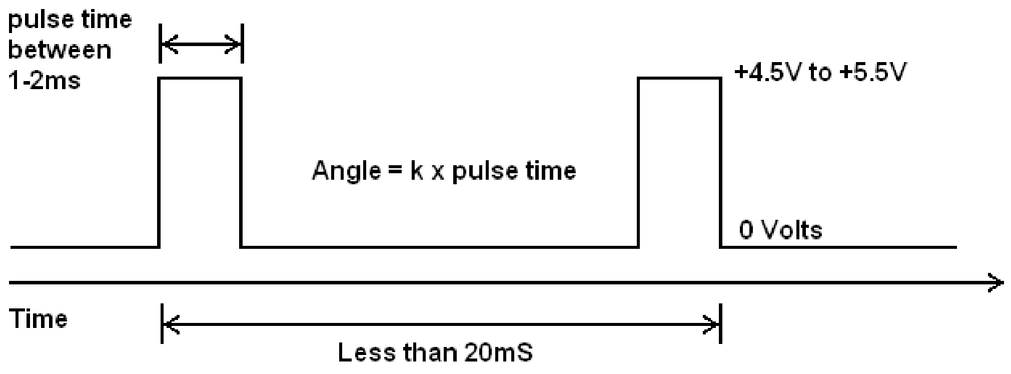


Figure 18: PWM servo command signal, (Dollery, 2001)

5.2 Implementation

Servo control for the UAV was implemented using the Mini SSC II servo controller card from Scott Edwards Electronics, Inc. This servo controller card, figure 19, interfaces to a computer via a serial connection. It communicates at 9600 baud and controls 8 servos. It allows for servo operation over either 90 or 180 degrees of rotation. These servo controllers can be connected in cascade allowing the control of more servos over one serial connection.

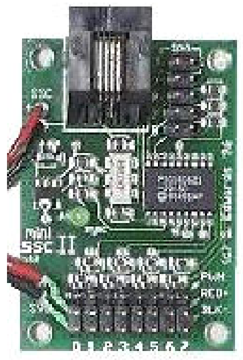


Figure 19: Mini SSC II Servo

Each position written to the servo controller card is as sequence of three bytes in the following format:

First Byte	**Second Byte**	**Third Byte**
sync_byte	servo_id	servo_position

The *sync_byte* always has the value 255 and is used to tell the servo controller card when a new position is being transmitted. The *servo_id* is the identifier of the servo to be positioned. This will be a value between 0-7 if only one servo card is being used. The *servo_position* specifies the position of the servo and can range from 0 to 254. This offers a resolution of 0.35 degrees and 0.71 degrees for servos operating over 90 degrees and 180 degrees of rotation respectively.

The servo operation module was created as a plug-in. It took as input servo positions stored in shared memory. These positions should be generated as outputs from a control module. A mock control module was developed to produce these values to be proportional to the readings from the digital compass sensor (to simulate a control algorithm for steady forward flight).

5.3 Results

The control surface operation software was created as a plug-in and tested. It was found to be functioning as expected and positioned the control surfaces of the test platform successfully. However it was found that if the control surface movement was restricted due to friction between moving parts, too large a current was required to move the servos. This affects the operation of all servos as they all draw current via the servo controller card. All control surfaces should move with minimum friction.

6.0 Ground Station

A UAV system is not complete without some sort of ground control centre to track its progress and issue commands. This chapter describes the development of a simple ground station program.

6.1 Background

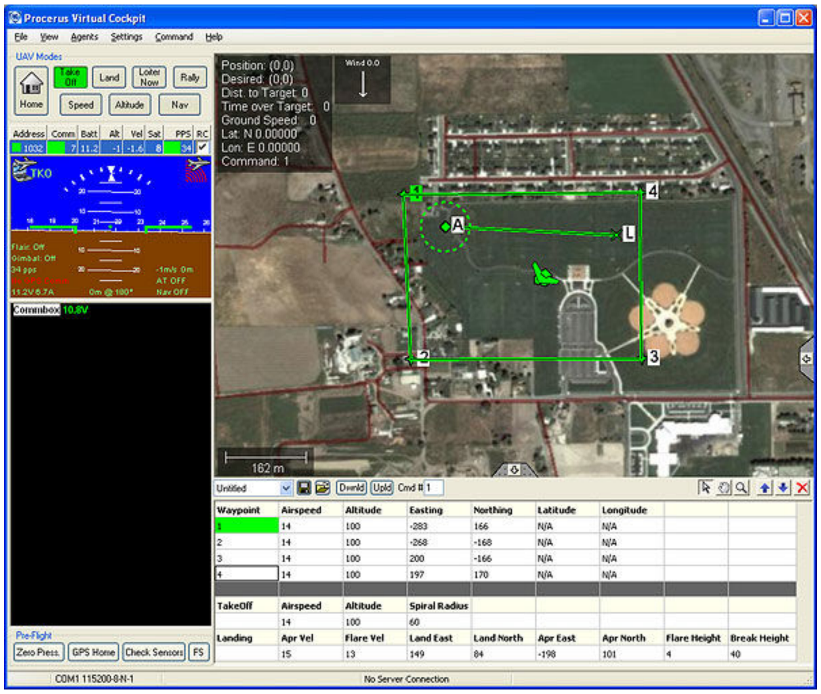
Often the mission objectives of the UAV can change during flight. This is particularly true of military applications. The designated path may be provided pre-flight and may change while the UAV is in operation. Unforeseen circumstances, for example dangerous weather conditions, may arise during flight time. This information may need to be transmitted to the UAV so that it modifies its flight path or returns to base. Also the UAV position should be periodically transmitted to an entity on the ground so as to track its progress. The operational task that UAV is performing may also require that it transmits data collected during flight to the ground station in real-time (Hsiao et al, 2002). Thus, for these reasons, it is necessary to maintain communication with an entity on the ground. This entity is referred to as the ground station.

Many ground stations have been designed for various academic, commercial and military projects. These range from computer programs communicating with the UAV via a radio modem on a laptop, to control centers on multiple consoles communicating via satellites for military purposes.

An example of a commercially sold ground station is Virtual Cockpit package for the Kestrel Autopilot System. It provides functionality for operating several UAV, dynamic path control, transmission of video and still shots, UAV control via a video game controller and many other features. The user interface for this software package is shown in figure 20. The Virtual Cockpit software costs $2995 US and only works with the Kestrel Autopilot System which costs $5000 US (Procerus Technologies, 2006).

The Predator is a US defence UAV described as being a long endurance, medium altitude unmanned aircraft system for surveillance and reconnaissance missions. It represents the absolute state-of-the-art in UAV technology and as such, the ground station unit consist of a *"single 30ft trailer, containing pilot and payload operator consoles, three Boeing data exploitation and mission planning consoles and two synthetic aperture radar workstations together with satellite and line-of-sight ground data terminals*", (Air Force Technology – Predator, 2006).

For this project a simple ground station was developed as a foundation for future work.



Position: (0,0)
Desired: (0,0)
Dist. to Target: 0
Time over Target: 0
Ground Speed: 0
Lat: N 0.00000
Lon: E 0.00000
Command: 1

Waypoint	Airspeed	Altitude	Easting	Northing	Latitude	Longitude		
1	14	100	-283	166	N/A	N/A		
2	14	100	-268	-168	N/A	N/A		
3	14	100	200	-166	N/A	N/A		
4	14	100	197	170	N/A	N/A		
TakeOff	Airspeed	Altitude	Spiral Radius					
	14	100	60					
Landing	Apr Vel	Flare Vel	Land East	Land North	Apr East	Apr North	Flare Height	Break Height
	15	13	149	84	-198	101	4	40

Figure 20: The Virtual Cockpit Package

6.2 Implementation

A simple ground station program was created using Glade-2. The ground station software receives messages over the data link pertaining to the crafts position and altitude. The software then plots the position of the UAV onto a user provided map. It displays the attitude of the UAV on an artificial horizon display similar to that used in aircraft. The ground station program developed also has the ability to start and stop the UAV over the data link. The communications over the data link are described in more detail in chapter 3.

6.2.1 Glade-2

Glade-2 is an open source software package used to create graphical user interfaces. It runs on any Linux desktop provided that Gtk, Gnome and GDK libraries are installed. Gtk, Gnome and GDK are libraries providing widget types to create and manipulate the components of a graphical user interface. They also provide the functionality for image manipulation and other tools necessary for the development of the ground station software. Callback functions are used to handle events, for example a button being clicked. For this project these functions were written in the C programming language, however Glade-2 supports C++ and Python also (Glade User Interface Builder, 2006). Glade-2 was chosen because it was open source, ran on Linux and provided the functionality required for developing the ground station software. In addition it was advised that the techniques used to develop GUI software using Glade-2 can be mastered in a short period of time; this proved to be true.

6.2.1 Ground Station Software

Figure 21 shows a screen shot of the ground control software. The map display window and the attitude display can clearly be seen. These two components make up the greater part of the software so far. Also if a valid map has been loaded then commands to start and stop the UAV can be issued by the operations panel.

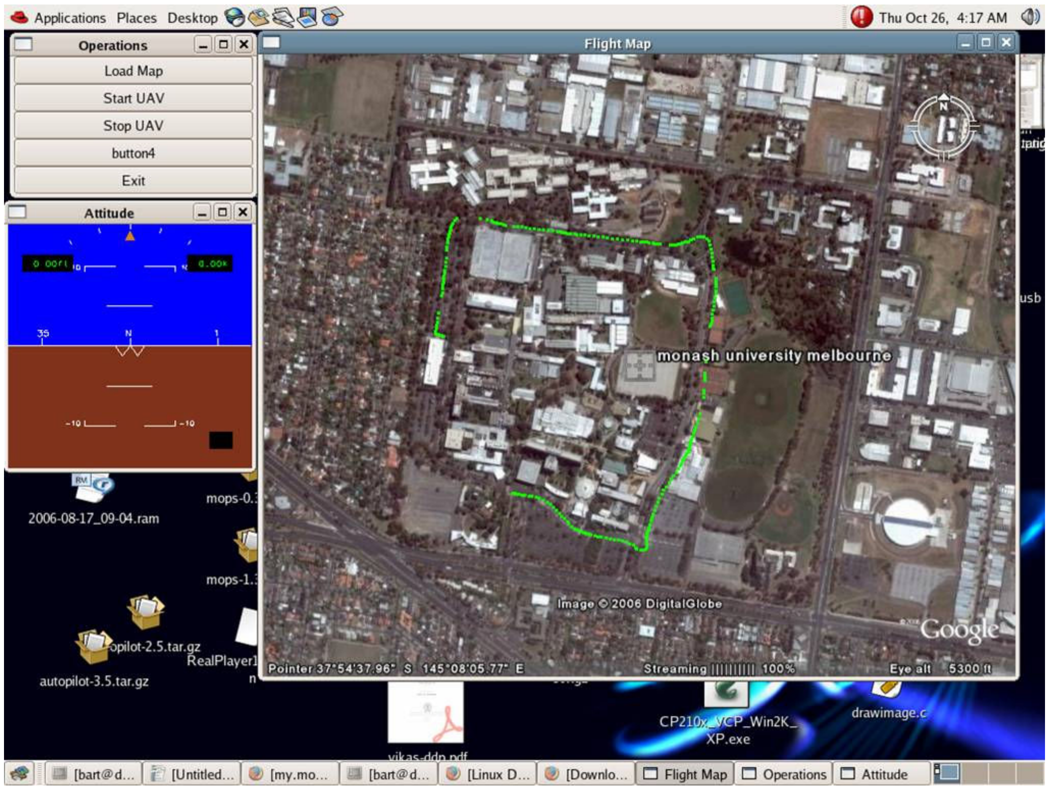


Figure 21: Screenshot of ground station software

Flight Map Display

A map is loaded by clicking the Load Map button. This opens a file chooser window. Once a valid image has been loaded, another window, figure 22, opens so the user can enter the map details. These are the longitude and latitude of the top-left pixel and bottom-right pixel of the map image. With this information the ground station software can plot the received position data accurately on the map.

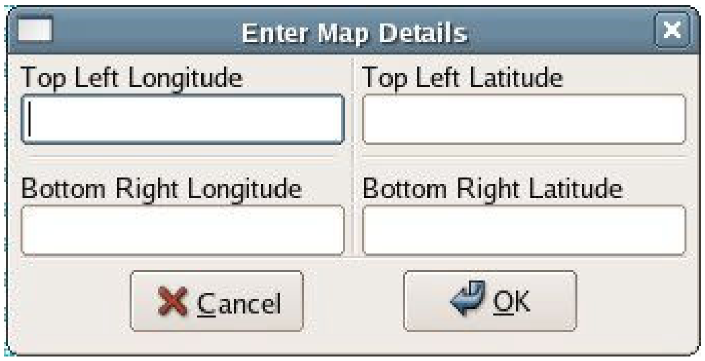


Figure 22: Entering Map Details

Although any image can be used as a map, it is intended that the map image is captured from the Google-Earth program. This program is freely available and provides satellite pictures of almost any location on the planet at varying zoom-levels. It also provides the longitude and latitude of any pixel on the map image, (Google Earth – Home, 2006). An example of a map captured from Google-Earth and used to plot data collected from a path around Monash University is shown in Figure 21.

To use the ground station software the user captures an appropriate map (screenshot) from Google-Earth, records the position data of the top-left and bottom right pixel of the map image, and together the image and the position data can be loaded into the ground station program. Once a valid map and details have been loaded the START command can be issued by clicking on the *Start UAV* button.

Attitude Display

The attitude display was an open source C++ software module created for another UAV project, Autopilot (Freshmeat.net: Project Details for Autopilot UAV, 2003). This software was converted to C and integrated into the ground station software. It displays roll, pitch and heading information transmitted by the UAV. This information comes from the digital compass.

6.3 Results

The simple ground station software was implemented and performed as expected on collected real-world data. This was achieved by running the UAV system and the ground station on two separate computers connected by a serial cable using the data link described in section 3. However the current implementation is not sufficient for a practical ground station solution as it is still in the initial stages of development.

6.4 Future Work

There remains much functionally that is yet to be implemented in the ground station software developed.

Some mission control and path planning must be added to the program for it to be a practical solution. Also it should display various variables such as temperature, fuel levels and altitude. To highlight the shortcomings of the system so far one can easily make a comparison between figures 20 and 21. It is intended that future work will be carried out to bridge the gap between this system and similar commercial packages.

7. Conclusion

This thesis has discussed the development, design and integration of a number of essential subsystems required to provide autonomous functionality for a small fixed wing aircraft. Specifically, these were a software architecture, data link, sensors and navigation, operation of control actuators and ground station software.

The software architecture created proved to be very useful in the design and integration of the subsystems developed. It is at a mature level of development and met its design objectives. The control surface operation module also met its design objectives. A sensor configuration and software modules adequate for the navigation of a fixed wing UAV was implemented, however a more robust navigation subsystem may be investigated. Other sub-systems still require further development and have their limitations. These are discussed in the following sections.

Despite the limitations and further work required, a solid foundation was created for all the subsystems developed.

7.1 Limitations and Future Work

The work carried out in this project is by no means complete. Some subsystems essential to autonomous operation of an UAV were not addressed.

A control subsystem was not developed as it requires knowledge specific to aerospace engineering, and is left as a future exercise for students from engineering. Control is essential for stable flight and the execution of any operational task.

Due to the time constraints, mission control and path planning was not implemented. This is a necessary for the autonomous movement of a UAV. Once implemented a UAV once given a set of coordinates should then be able to navigate to these destinations as required. This functionality is provided through a higher level control module on board the UAV and features in the ground station software. The ground control station also needs to provide new functionality to monitor UAV status variables including but not restricted to velocity, temperatures, fuel levels and power levels.

The data link developed served as a proof of concept however due to the lack of communication hardware was not used for wireless communication. It requires further development for a more practical protocol, encryption and compression of transmitted data and adaptation to a RF communications device. The methodology for these improvements has been provided and can be incorporated into the existing structure of the data link.

The sensor system and navigation modules developed will provide for the control and guidance of a fixed-wing UAV; however it is possible to develop a more robust navigation subsystem. As suggested this will ideally be carried out via the fusion of IMU, GPS and digital compass sensors.

It is also suggested that each sub-system be focused on individually as each requires a significant amount of development to reach a level of maturity, required for real world applications.

8. References

Air Force Technology – Predator, (2006), Retrieved on November 1st 2006, Retrieved from http://www.airforce-technology.com/projects/predator/

AUVSI UAV Student Competition, Retrieved on 29th October 2006, Retrieved from http://www.bowheadsupport.com/paxweb/seafarers/default.htm

Brown, X, Argrow, B., Dixon, R., Doshi, S., Thekkekunel, R.G., Henkel D., (2004), *Ad Hoc UAV Ground Network (AUGNet),* AIAA 3rd "Unmanned Unlimited" Technical Conference, Chicago, IL

Dollery, W., (2001), *Autonomous Flying Robots (Helicopter): Development of Processing Platform and System Design*, School of Computer Science and Software Engineering, Monash University, Melbourne Australia.

Downey J., Michini B., Doherty M., Engel C., Katz J., Kulling K., (2006), *Project AARES*, Massachusetts Institute of Technology Unmanned Aerial Vehicle Team, 2006 AUVSI Student UAV Competition Journal Paper, Submitted June 1, 2006

Fernandez M., Macomber G.R., (1962), *Inertial guidance engineering,* Prentice-Hall

Fontana, R., Ameti, A., Richley, E., Beard, L., Guy, D., (2002), *Recent advances in ultra wideband communications systems,* Ultra Wideband Systems and Technologies, 2002. Digest of Papers. 2002 IEEE Conference on , vol., no.pp. 129- 133,

Freshmeat.net: Project Details for Autopilot UAV, (2003), Retrieved on 1st November 2006, Retrieved from http://freshmeat.net/projects/uav/

Giotto, (2006), Retrieved on 26th October 2006, Retrieved from http://embedded.eecs.berkeley.edu/giotto/

Glade User Interface Builder, (2006), Retrieved on 1st November 2006, Retrieved from http://glade.gnome.org/

Google Earth – Home, (2006), Retrieved on 1st November 2006, Retrieved from http://earth.google.com/

Grewal M. S., Weill L. R., Andrews A. P., (2001), *Global Positioning Systems, Inertial Navigation, and Integration*, John Wiley & Sons, Inc.

Hong W., Lee J., Rai L., Kang S., (2005), *RT-Linux Based Hard Real-Time Software Architecture for Unmanned Autonomous Helicopters*, *rtcsa*, pp. 555-558, 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)

Horowitz B., Liebman J., Ma C., Koo J., Henzinger J., Sangiovanni-Vincentelli A., Hsiao F., Yang C., Wu C., Lee M., Hsu C., Tu T., (2002), *The development of onboard computer system and portable ground station for an autonomous UAV*, 1st Unmanned Aerospace Vehicles, Systems, Technologies, and Operations Conference and Workshop, Portsmouth, VA; UNITED STATES; 20-23 May 2002.

Kermode A, (1994), *Flight Without Formulae*, Sterling Book House, fifth edition-updated by Bill Gunston

Kirsch C., Sanvido M., Henzinger T., Pree W., (2002), *A Giotto based helicopter control system*, EMSOFT 02: Embedded Software (A. Sangiovanni-Vincentelli and J. Sifakis, eds.), LNCS 2491, Springer-Verlag, , pp. 46–60.

Kumar, V., (2004), *Integration of Inertial Navigation System and Global Position System Using Kalman Filtering*, Department of Aerospace Engineering Indian Institute of Technology, Mumbai.

Levy, J., (1997), *The Kalman Filter: navigation's Integration Workhorse*, GPS World, September, pp 65 – 71.

Maxstream Products: Wireless, Radio Modems, Retrieved on 12th October 2006, Retrieved from http://www.bb-elec.com/maxstream.asp

Mckerrow P., (1991), *Introduction to Robotics*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA

Minkler G., Minkler J., (1993), *Theory and Application of Kalman Filtering,* Magellan Book Company

Ocean-Server: Compass. Retrieved on 5th November, 2006. Retrieved from http://www.ocean-server.com/compass.html

O-Navi, (2006), Retrieved on 5th November 2006, Retrieved from http://www.o-navi.com/products.htm

Ortiz G., Lee S., Monacos S., Wright M., Biswas A., 1999, *Design and development of a robust ATP subsystem for the Altair UAV-to-ground lasercomm 2.5-Gbps,* PROCEEDINGS-SPIE THE INTERNATIONAL SOCIETY FOR OPTICAL …, 2003 - International Society for Optical Engineering;

Procerus Technologies, (2006), Retrieved on 1st November 2006, Retrieved from http://www.procerusuav.com/productPricing.php

Saeedipour H., Azlin M., Sathyanarayana P., (2005), *Data Link Functions and Attributes of an Unmanned Aerial Vehicle (UAV) System Using Both Ground Station and Small Satellite*, University of Science Malaysia

Sangiovanni-Vincentelli, A.; Martin, G., (2001), *Platform-based design and software design methodology for embedded systems, Design & Test of Computers, IEEE* , vol.18, no.6pp.23-33

Sastry, S.,(2002), *Embedded-software design and system integration for rotorcraft UAV using platforms,* in Proc. 15th IFAC World Congress: Elsevier

Silberschatz A., Galvin P., Gagne G., (2001)*, Operating System Concepts*, John Wiley & Sons, Inc., New York, NY

The Aviation History On-Line Museum., (2002), Retrieved on October 22nd Retrieved from http://www.aviation-history.com/theory/flt_ctl.htm

Weiss M.. (1997), *Data Structures and Algorithm Analysis in C*. Addison Wesley, 2nd edn

Yagen, A., (2005), "Sabbatical activities at Monash Clayton University - Helicopter Model, Navigation and Control", Monash University Melbourne, Australia