2010

Flight Control System for Small High-Performance UAVs

Jefferson McBride
Virginia Commonwealth University

School of Engineering
Virginia Commonwealth University
This is to certify that the thesis prepared by Jefferson C. McBride entitled

"Flight Control System for Small High-Performance UAVs"

has been approved by his or her committee as satisfactory completion of the thesis requirements for the degree of Master of Science.

_____

Dr. Robert Klenke (Advisor)

_____

Dr. Mike McCollum (Committee Member)

_____

Dr. James Ames (Committee Member)

_____

Dr. Ashok Iyer (Department Chair)

_____

Dr. Russell Jamison (School Dean)

_____

Dr. Rosalyn Hobson (Graduate Dean)

Flight Control System for High-Performance UAVs

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

By

Jefferson Clark McBride

B.S. Electrical Engineering, Virginia Commonwealth University, May 2005

Director: Dr. Robert Klenke, Associate Professor, Virginia Commonwealth University

Virginia Commonwealth University

Richmond, Virginia
May 2010

Table of Contents

Table of Figures....5
Table of Tables....6
Abstract....7
Chapter 1:Background....1
Introduction....1
AirSTAR Program....2
Previous Work at VCU....3
Other Research Efforts....4
Commercial Flight Control Systems....6
MicroPilot MP2128g....6
CloudCap Technologies Piccolo....7
Kestrel....7
Introduction to the VCU Aerial Vehicles....8
High-Level System Design....10
Chapter 2:Hardware....13
Flight Control Computer....13
Expansion Board (T-Board)....16
Intertial Measurement Unit....19
Chapter 3:Software Architecture....21
Software Overview....21
Development Tools....24
Configuration Storage....24
Hardware-in-the-loop Simulation....24
Chapter 4:Control System....27
Flight Stabilization....27
Navigation....42
Chapter 5:Ground Control Station....45
GCS Design Goals....45
GCS Architecture....46
VACS Communications....51
Chapter 6:Results and Performance....54
Flight Test Process....54
Test Data....55

Table of Figures

Figure 1.1: The Procerus Kestral autopilot....8
Figure 1.2: FQM-117 Mig aircraft....8
Figure 1.3: The DV8R jet turbine aircraft....9
Figure 1.4: The Jurassic jet-turbine UAV....10
Figure 1.5: High level system overview....11
Figure 2.1: Suzaku Board from Atmark Techno....14
Figure 2.2: T-Board expansion PCB block diagram 16
Figure 2.3: The T-Board with Suzaku, uBlox GPS Module, and Radio modem mounted to it....17
Figure 2.4: The MIDG2 IMU....20
Figure 3.1: Software periodic loop flow chart 22
Figure 3.2: Sample FCS build configuration output....24
Figure 4.1:Generic PID controller block diagram 28
Figure 4.2: Elevator Multiplier as a function of roll....34
Figure 4.3: Target heading field lines for a loiter....38
Figure 5.1: GCS Architecture Diagram 47
Figure 5.2: FCSControl application screenshot....51
Figure 6.1:Actual and target airspeed. 56
Figure 6.2: Altitude versus a constant target altitude....56
Figure 6.3: Measured vs. target pitch angle....57
Figure 6.4: Measured vs. target airspeed for second flight sequence. 57
Figure 6.5: Measured and target altitude for second flight sequence. 58
Figure 6.6: Measured and target pitch for second flight sequence. 58
Figure 6.7: Actual versus target roll angle....58
Figure 6.8: Aerial view of flightpath....59
Figure 6.9: Aerial view of a clock-wise circuit flying a triangular waypoint pattern....59
Figure 6.10: Aerial view of clockwise circuit flying a trianglular waypoint pattern....61
Figure 6.11: Measured and target roll angle during three autonomous roll maneuvers....62
Figure 6.12: Measured and target altitude during three autonomous roll maneuvers 62
Figure 6.13: Measured and target pitch angle during three autonomous roll maneuvers....62

Table of Tables

Table 2.1: Microblaze system peripheral IP cores used in FCS.....15
Table 2.2: Suzaku Flash Memory Map.....16
Table 3.1: Software Module List.....27
Table 4.1: FCS_CONTROL structure fields.....40
Table 4.2: FCS Control loop parameters.....43
Table 4.3: High level NAV modes.....44
Table 4.4: NAV Override Modes.....44
Table 4.5: Waypoint Options.....45
Table 5.1: IPlaneControl Interface.....49
Table 5.2: VACS Packet Format.....53
Table 5.3: Plane type response data payload format.....54

Abstract

FLIGHT CONTROL SYSTEM FOR SMALL HIGH-PERFORMANCE UAVS

By Jefferson Clark McBride, M.S.

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at Virginia Commonwealth University.

Virginia Commonwealth University, 2010

Director: Dr. Robert Klenke, Associate Professor, Electrical and Computer Engineering

This thesis documents a research project in which an autonomous flight control system (FCS) was designed to control and navigate small, high-speed, unmanned, jet-turbine powered fixed-wing aircraft. The FCS was designed to allow the aircraft to maintain controlled flight, and return to a home location, without any operator intervention. The flight control computer was built with an FPGA, using a Microblaze soft-core microprocessor running the uClinux operating system. The configurable FPGA computing platform allowed flexibility for interfacing quickly with a wide range of sensors and control modules. A commercial inertial measurement unit was used for aircraft state estimation, and the flight control system was able to provide stability and precise flight-path control for multiple turbine-

powered aircraft over the wide flight airspeed envelope these vehicles are capable of. In addition, the custom ground control station which provides an operator control interface for the FCS is discussed.

Chapter 1: Background

Introduction

Unmanned aerial vehicles (UAVs) are quickly finding extensive use in the military sector for performing missions that would be too long, boring, expensive, or dangerous for a full-scale piloted vehicle. Civilian applications -- although currently limited by complications arising from UAVs sharing the national airspace with civilian aircraft -- hold further potential for UAVs. A third area of usefulness for UAVs, and one that is already being explored by many, is as tools in various research disciplines, including meteorology, aerodynamics, and communications. UAVs can be used to cheaply carry sensors over wide ranges for studying atmospheric conditions, or as a more cost effective means to prototype and test new airframe designs, for example.

Partial motivation for this project is provided by the need of one of these research groups -- the Generic Transport Model (GTM) project at NASA Langley Research Center -- for a flight control system capable of controlling the $1/20^{th}$ scale transport jet models used for aerospace research. The flight control system designed for this thesis could provide a backup controller that can safely control the aircraft and return it to a landing location in the event of a failure of the ground based test-pilot control system, or allow certain flight test maneuvers to be programmed into the autopilot allowing more precise control than a pilot on the ground can achieve. Commercially available autopilots are not ideal

for this application, due to the extended requirements of the Turbine model (such as Engine Control Unit (ECU) integration) and limited ability to customize. Further efforts are being pursued at VCU towards a more advanced data collection system to monitor the many sensors on the well-instrumented aircraft, and telemeter this information to the ground and provide data logs. Ultimately, the flight control and full data collection should be combined into one more capable platform.

The intent of this Master's thesis is to document the design and testing of a digital flight control system (FCS) capable of performing autonomous stability and navigational control of small high-speed jet turbine powered fixed-wing aircraft. The work described here is an extension of previous work done at VCU, including autopilots with different computing platforms, different sensor packages, and continually evolving software. This author was responsible for defining the software architecture used in the flight control system and developing the control laws used for fixed-wing aircraft control, developing the GCS architecture, messaging system, server application, along with the FCSControl client, as well as integrating new commercial inertial measurement units (IMUs). This thesis will discuss the Microblaze-based hardware platform used for the embedded computing system, the Microbotics MIDGII IMU based sensor package, the flight control software that runs on this platform, the control laws that are implemented as part of this software, and the ground station software that provides the human operator interface to monitor and command the FCS.

AirSTAR Program

AirSTAR[1] is part of the NASA Aviation Safety Program at Langley Research Center. The goal of the AirSTAR program is to provide a flight testbed capability using a 5.5% dynamically scaled turbine powered generic transport model (GTM) aircraft, as well as a Mobile Operations Station (MOS) and a test range. The combination of these elements will allow a research pilot, located on the ground in a virtual cockpit in the MOS, to operate the vehicle outside of the typical flight envelope of a full scale

transport aircraft. It will allow for less expensive, lower risk experimentation in "off-nominal and loss-of-control flight regimes...and validation of advanced control system algorithms for flight control failures".

The design of the AirSTAR system places the research pilot and the flight control computers on the ground. A high-speed data link to the plane provides telemetry and control command updates at a rate of 200 Hz. In addition, a safety pilot is located outside of the MOS with a remote-control link to the experimental vehicle, who is able to take-over and fly the vehicle visually. However, because experiments will require the vehicle to fly outside of the safety pilots visual range, and because there is potential for communications link failures between the research pilot and the vehicle, an autonomous flight control system is required on-board in order to maintain stability, return the vehicle to the operating area, and in the case that control cannot be regained, automatically land it on the runway. The system described in this thesis is intended to fulfill that role. VCU also developed a research data system to passively monitor human controlled flights and collect telemetry data. This system has already been installed in multiple NASA aircraft. It is likely that the flight control system will initially be installed and tested independently, but ultimately should become an upgraded capability of the research data system.

Previous Work at VCU

Electrical and Computer Engineering students at VCU, both graduate and undergraduate, have been developing computer control systems for unmanned vehicles for several years now. Naturally, these systems have developed over the years and have expanded to fill more roles, and provide improved and extended functionality.

The original VCU flight control system was based on an Atmel FPSLIC processor. This chip includes an 8-bit AVR RISC microprocessor, as well as a 40K gate equivalent FPGA[2]. A GPS receiver was used

for navigation, and the "CoPilot" flight stabilizer made by FMA was used to control roll and pitch angles using its infrared thermal sensors. No direct measurement of the attitude angles was made by the FCS.

Although this system proved capable of controlling small aircraft, and was used successfully to fly the FQM-117 Mig aircraft (discussed below) for test flights and in a student UAV competition, it did have drawbacks. It did not have the computational power to perform very computationally complex flight control and payload control algorithms at very high speeds, it could not accurately measure many of the vehicle state parameters such as attitude angles, and it was overly sensitive to environmental factors (due to the limitations of the CoPilot sensor infrared sensor).

In later designs, the CoPilot controller was removed, the sensors were connected to an analog-to-digital converter to be read by the FCS, and the roll/pitch control that had previously been performed by the CoPilot controller was implemented on the FCS processor. In addition, a third infrared sensor was installed to allow a full three axis solution. This gave the FCS the ability to measure the aircraft attitude (which is useful for monitoring purposes and for controlling payloads), and allowed an improvement in performance because the sensitivity of the sensors could be adjusted on-line as conditions changed.

The FPSLIC based system was used to compete (rather successfully) for two years at the AUVSI student UAV competition at Webster Field in Maryland. The next generation hardware platform is based on a 32-bit soft-core (implemented in FPGA fabric) microprocessor from Xilinx called the Microblaze. This is the processing system used in the system described by this thesis.

Other Research Efforts

Recently, many university groups have been working on various aspects of UAV control systems. This work is widely varied, and includes low-level vehicle control systems, payload control, and

collaborative control of large “swarms” of multiple vehicles. Many of these groups make use of a commercial autopilot for vehicle control, adding their own systems on top for added functionality, however some other universities have created fully custom autopilot solutions.

The Big Blue[3,4] project is a test-bed UAV for Mars airplane technology at the University of Kentucky. The vehicle, with inflatable wings, is launched by hot air balloon from an altitude of around 90,000 feet. A custom set of of avionics was created to control the plane and communicate with ground operations. This system was designed to distribute functions among three small, 8-bit microprocessors (SiLabs C8051 series) with one each for: Mission Control, Chute Control, and Flight Control. These processors communicated via an I2C bus, using a shared EEPROM on the bus for message passing. Long range communication was achieved using amateur radios

A group from the Georgia Institute of Technology developed a custom autopilot to control a statically stable aircraft using a single antenna GPS receiver as the sole sensor[5]. Other states, such as the vehicle's roll angle, are inferred by observation of the GPS velocity. Although this is a novel idea and may provide guidance for a very stable vehicle, the control system bandwidth is necessarily limited by the GPS receiver, and it could not be used for very dynamic flight conditions or less stable aircraft.

Georgia Tech also maintains an unmanned helicopter for UAV system research, which it calls the GTMax[6]. The system uses the Yamaha R-Max industrial helicopter airframe, and has a standard set of avionics as well as the ability to add custom modules for different projects. It contains two embedded PC processors, running at 266MHz and 800MHz, as well as GPS and intertial sensors. In addition to the hardware platform, a software and hardware-in-the-loop simulation platform has been developed for testing experimental control system.

At the University of South Australia, a reconfigurable computing based system has been built to provide the processing power of FPGA hardware acceleration for on-board processing[7]. Primarily

focused on video processing, the system makes use of a Micropilot autopilot for vehicle control, and includes an 800MHz x86 processor, and an FPGA board from Celoxica with video capture hardware. Flight tests were performed with all of the computer hardware on-board, though it seems most of the flight testing was done under manual control. Interestingly, they note that the process of configuring the micropilot "proved to be more difficult than expected". Finally, the authors created an operating system layer to support dynamically reconfiguring the FPGA hardware in order to switch functions during flight.

Commercial Flight Control Systems

Several commercial flight control systems are currently on the market, intended to be customizable for integration with a range of unmanned aerial systems. They typically include integrated sensors such as GPS receivers, accelerometers, rate gyros, and barometric sensors for measuring the state of the vehicle as well as a processing system to run control loops and communicate with ground station software (typically provided with the autopilot). Some of the major commercial systems are discussed below.

MicroPilot MP2128g

The MP2128 is MicroPilot's "flagship" model autopilot, with an increase in processing power over previous generations. Weighing only 28 grams (not including a radio modem or enclosure), on a 4 by 10 centimeter board, it includes a GPS reciever, 3 axis accelerometers and rate gyros, and barometric altitude and airspeed sensors. Ground station software, called Horizon, is included with the system to visualize the telemetry data and to allow operators to modify waypoints, update PID control loop gains, etc. The unit sells for $7000 in single quantities.

The XTENDER development kit, available from Micropilot at an additional charge, can be used to develop plug ins to extend and modify the software on board the autopilot, as well as on the ground

station.

CloudCap Technologies Piccolo

Cloudcap technologies produces the Piccolo series of autopilots. The larger version, the Picollo plus, is contained in a 4.8”x2.4”x1.5” enclosure, and weighs 212 grams. It includes Cloudcap's Christ Inertial Measurement Unit (IMU) with rate gyros, accelerometers, a ground station radio link, GPS receiver, and barometric sensors. The processing component is a 40MHx power PC (MPC555), with 26K of SRAM.

A smaller version of the Piccolo, the Piccolo LT, has recently been introduced. The same processor, and similar sensor suite is available on this system, in a 109 gram, 4.7”x2.25”x0.7” package. The smaller version has less I/O capabilities and a simpler data link[8].

Kestrel

The Kestrel autopilot is produced by Procerus Technologies, and claims to be the smallest, lightest commercial autopilot on the market. It weighs 16.7 grams and fits into a 2”x1.37”x0.47” form factor, and includes 3-axis rate gyros, 3-axis accelerometers, and barometric sensors. A GPS receiver is not included, and must be included externally. Additionally, magnetometers are supported, but must be added externally. A 29MHz Rabbit 3000 processor runs the control software, and has four serial ports for connecting to radio modems, payloads, or other processors, and 4 on-board servo ports.

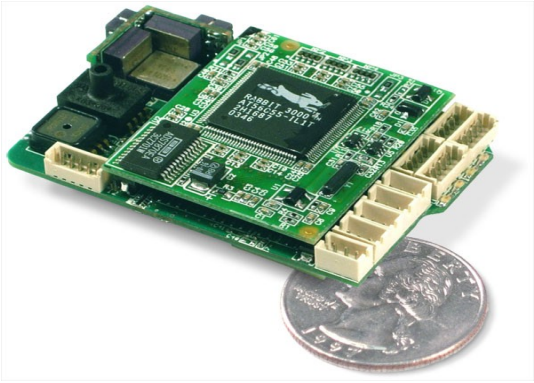


Figure 1.1: The Procerus Kestral autopilot

Introduction to the VCU Aerial Vehicles

Although intended to fly the GTM model, this flight control system and its predecessors have been developed and tested using a range of aircraft.

FQM-117 "Mig"

Surplus Army target drones, originally designed to be RC piloted, these vehicles were the first and primary airframe for VCU UAV development. The Mig airframe is made of foam, originally with a 5.5ft wingspan, and powered by either a nitromethane engine, or an electric motor. In order to decrease wing loading and increase payload capacity and flight endurance, a larger wing was custom built for VCU airfract with a 7ft wingspan. The Mig flight envelope includes airspeeds from roughly 25-70 knots.

Figure 1.2: FQM-117 Mig aircraft

DV8R

The DV8R is a jet turbine powered R/C model kit. It has an 84 inch wingspan, weighs 27 lbs without fuel, and has an operating flight envelope of approximately 50-110 knots, powered by a Jetcat P-80 engine with 18lbs of thrust. The DV8R provides a faster, turbine-powered model, as an initial test-bed.

Figure 1.3: The DV8R jet turbine aircraft

Jurassic

In order to extend the variety of vehicles for evaluation, a second turbine model was acquired and outfit with a flight control system. The Jurassic Jet Trainer Model has a wingspan of 64 inches, a nose-to-tail length of 83 inches, and weighs 22 lbs. without fuel. Powered by a JetCat P-70 engine with 17 lbs of static thrust, it can achieve a top speed of approximately 130 MPH.

Figure 1.4: The Jurassic jet-turbine UAV

High-Level System Design

The complete flight control system consists of the vehicle and it's control servos, the MIDGII IMU, the flight control computer on-board the vehicle to measure its state and provide control signals, a ground control application running on a PC (e.g. Laptop) providing an operator control interface, and typically a safety pilot with separate control link who is able to override the FCS and fly the vehicle manually. The manual pilot override is enabled by a commercial-off-the-shelf (COTS) piece of hardware called the UAV Safety Switch.

The safety switch, and R/C control system are commercial off the shelf components. This thesis will discuss the other control path shown in figure 1.5: The flight control computer, the control system running on it, and the ground control station.

The flight control computer is an FPGA based embedded processing system: A Microblaze soft-core microprocessor running in a Spartan-III FPGA. The FPGA system is hosted on a small FPGA processing board called the Suzaku, made by Atmark-Techno. This PCB includes the FPGA and support circuitry such as Flash and SDRAM. The Suzaku board is mounted as a daughter-board onto a

custom PCB containing support circuitry, sensors, a radio modem, and IO to interface the embedded system on the Suzaku with the rest of the vehicle.

The flight control software uses the uClinux embedded linux distribution as an operating system, and is written entirely in C. It reads data from the vehicle sensors – GPS Receiver, Inertial Measurement Unit, static and pitot pressure – to compute the vehicle state estimate, then performs the control law calculation to determine the required position for the four control outputs: elevators, throttle, rudder, and ailerons. In addition, it communicates with the ground control station using the on-board radio link.

The ground control station is a windows application written in C#. It has been designed to support the variety of unmanned vehicle efforts on-going at VCU, by being easily adaptable to different platforms and maintaining concurrent support of all of these platforms without having different versions of the ground station for different vehicles.

The FCS hardware, FCS software, and GCS software will all be covered in the following chapters.

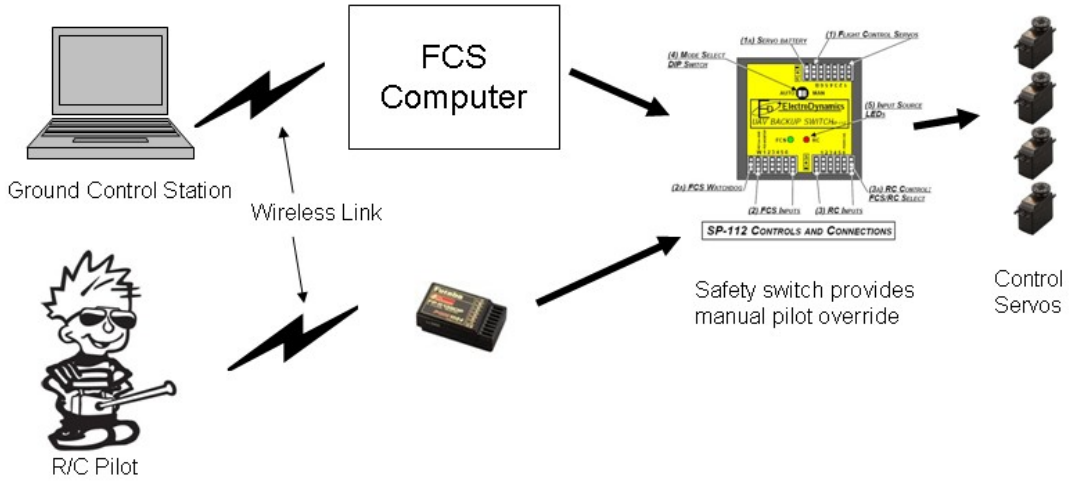


Figure 1.5: High level system overview

Chapter 2: Hardware

This chapter will discuss the hardware component of the UAV flight control system. This includes the Suzaku processing platform, as well as the custom expansion board (the T-Board), sensors, and communications hardware. In addition, the hardware components implemented in the Spartan 3 FPGA will be covered, as they provide the Microblaze-based embedded system on which the flight control software is executed.

Flight Control Computer

The flight control system is based on a single board computer, called the Suzaku, from Atmark-Techno. The board contains a Xilinx Spartan 3 FPGA which can be used to implement a Microblaze 32-bit soft-core processor and required peripheral hardware. The suzaku board contains the all of the resources necessary to run the Microblaze system:

- XC3S1000 FPGA
- 8MB Flash memory
- 16MB SDRAM
- Ethernet MAC/PHY

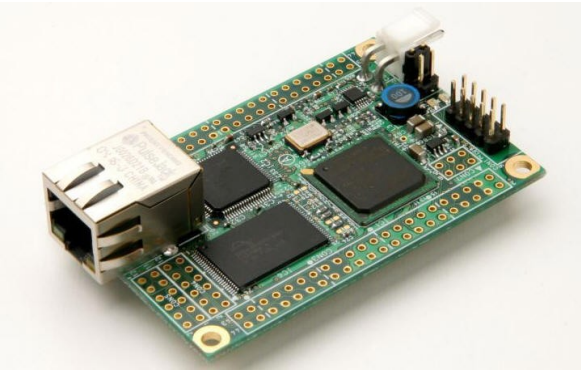


Figure 2.1: Suzaku Board from Atmark Techno

The Microblaze system itself is completely implemented in the fabric of the Spartan3 FPGA. This provides great flexibility for how to configure the processor and its peripherals. For example, extra UARTs or custom peripherals can easily be added to the system, or a floating point unit can optionally be included, or not included to conserve space and power. Xilinx provides the Embedded Development Kit (EDK) to configure and generate the Microblaze hardware. The EDK includes intellectual property (IP) cores for common microprocessor functions that can be easily included in the design. These include memory controllers, UARTs, SPI bus interfaces, and timers, among others. Some IP cores are available only with extra licensing fees, however most are included for royalty-free use with the EDK. In addition, the EDK allows for custom IP cores to be created in a hardware description language (e.g. VHDL), and provides tools to connect these custom cores to the On-chip Peripheral Bus (OPB) used by the microblaze to communicate with most peripherals. This makes integrating your peripherals as memory mapped registers in the Microblaze memory space a simple process.

The primary custom peripherals used by the VCU FCS are 10-bit resolution, multi-channel PWM generation and sampling cores. The *pwm_write* core allows a configurable number of channels of pulse

width modulated servo control signals to be generated by the FPGA. Conversely, the *pwm_read* module samples incoming pulse width signals (e.g. From the receiver receiving manual pilot signals) and measures the width of the pulses. The microblaze can simply perform a memory read or write to read the most recent incoming pulse width for a channel or to change the output pulse width. This proves very useful as the flight control system is required to measure and generate a large number of PWM signals, and the hardware based solutions allows this to be done with nearly zero processor load.

The complete list of the IP cores used by the typical FCS system is shown in table 2.1.

Core Name	Description
opb_timer	32-bit timer counter used by Linux for task scheduling, etc.
lmb_bram_if_cntlr	Memory controller for on-chip BlockRAM memory
opb_uartlite	Several of these are used for serial communications
opb_intc	Interrupt controller
opb_sdram	External SDRAM memory controller
opb_emc	External FLASH memory controller
opb_gpio	Provides general purpose IO pins (Used for diagnostic LEDs, boot control jumpers, etc.)
pwm_read	Pulse width modulation input channels (Five used)
pwm_write	Pulse width modulation output channels (Four used)

Table 2.1: Microblaze system peripheral IP cores used in FCS

Configuration and Code Storage

The Xilinx FPGA is SRAM based, so the configuration data that defines the circuit implementation is volatile, and must be reprogrammed at each power-on. The FPGA configuration bit file, the Hermit Bootloader, and the operating system image are all stored in the flash chip. At power-on, a special configuration IC (the TE7720 from Tokyo Electron Device) reads the bit file data from flash, and configures the FPGA. The bit file configuration also initializes the small block RAM on the FPGA with a simple first stage bootloader. The Microblaze then begins executing this bootloader, which loads the second stage bootloader (Hermit) from flash into the SDRAM, and begins executing it. Hermit is responsible for copying and booting the uClinux operating system kernel from the SDRAM.

There are three critical sections of the flash storage:

- FPGA Configuration
- Hermit Bootloader
- Linux Image (linux kernel, and file system)

Additionally , there is an extra section of flash that can be used to store user data, e.g. Configuration options. The FCS application uses this region to save flight control options such as control loop gains.

Name	**Size**	**Description**
free1	64kB	
free2	448kB	
fpga	512kB	FPGA Configuration Stream
bootloader	128kB	Hermit bootloader
image	6.81MB	Linux kernal and userland applications
config	64kB	Application configuration data area

Table 2.2: Suzaku Flash Memory Map

Expansion Board (T-Board)

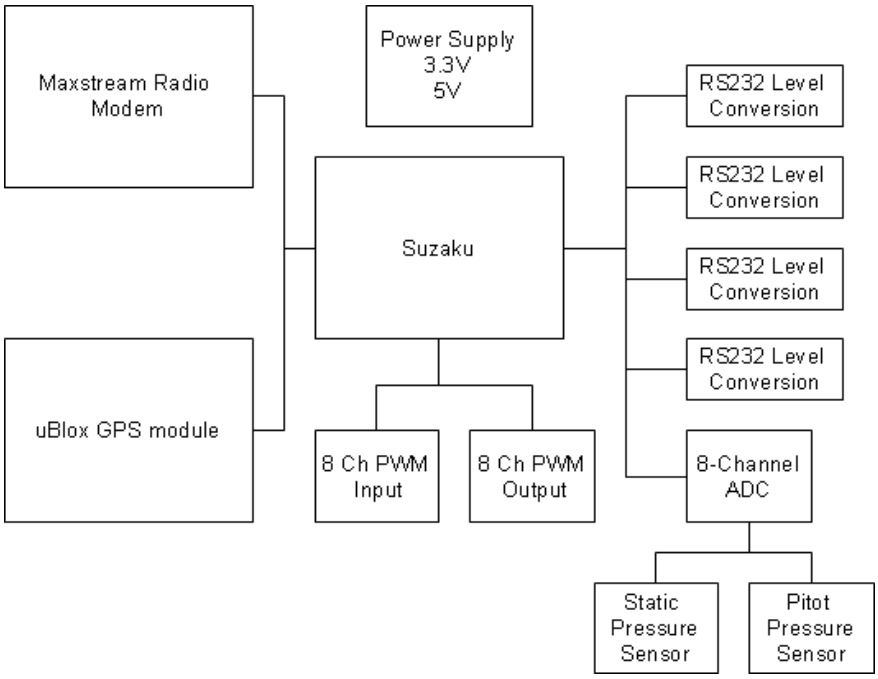


Figure 2.2: T-Board expansion PCB block diagram

A custom expansion board was designed to provide the extra I/O connections and sensors required for the flight control system. This PCB, called the T-Board, is shown in figure 2.3. It provides power supplies, connections for servo control signals, barometric sensors, several channels of analog-to-digital converters, RS232 level conversion, and board to board connection points for directly mounting a GPS receiver module and a Maxstream radio modem.

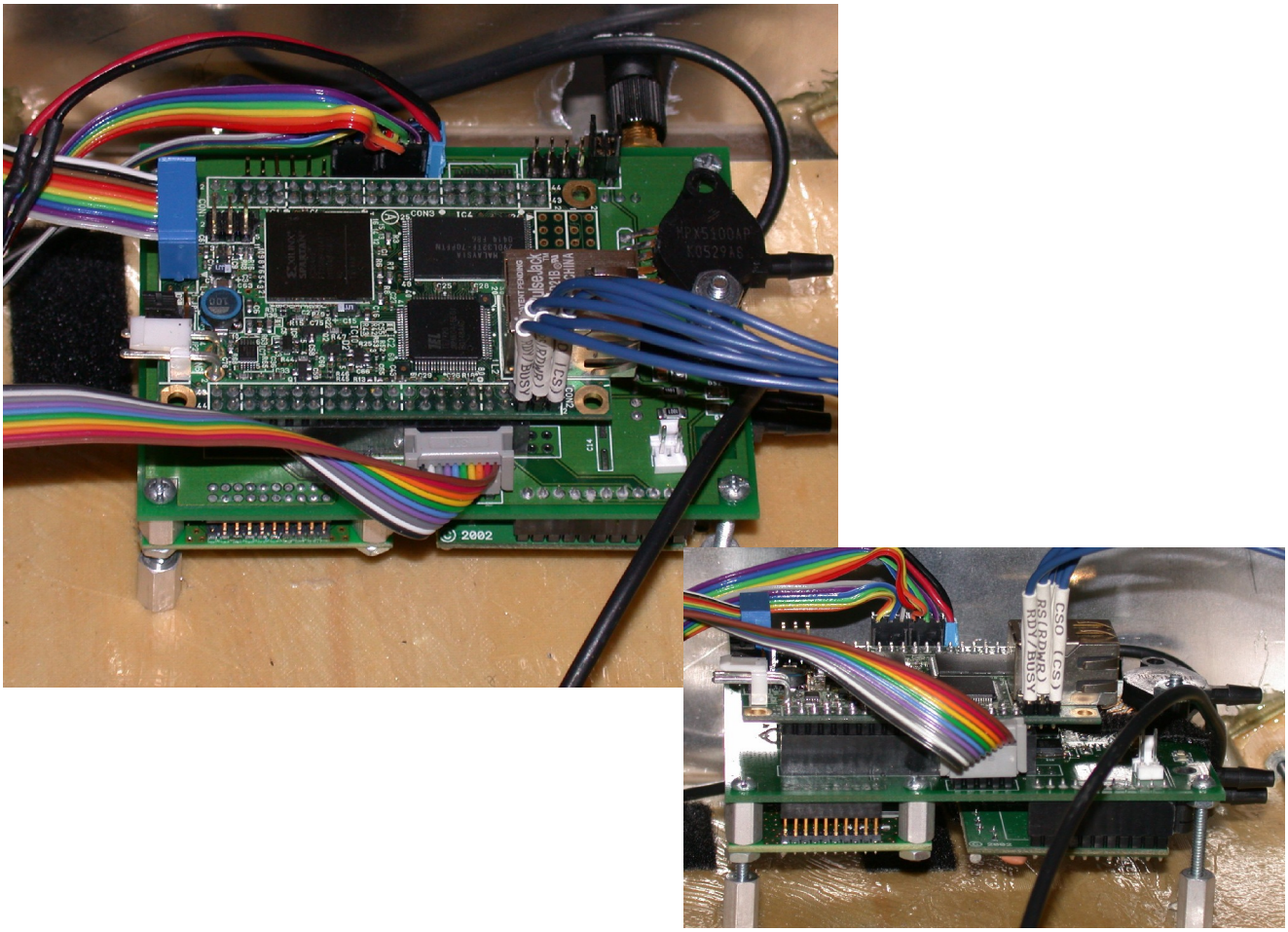


Figure 2.3: The T-Board with Suzaku, uBlox GPS Module, and Radio modem mounted to it

Analog-to-Digital Converters

Eight analog input channels are provided on the T-Board, each with 12-bit resolution and a 0 to 5V input range. The ADC is connected to the Microblaze via an SPI bus. The ADC channels are used for measuring sensor and battery voltages.

GPS Module

The T-Board supports the uBlox Antaris 4 OEM GPS receiver module, with an 71x41mm form factor. It is WAAS enabled for improved accuracy, and provides a 4Hz position update rate via an RS232

serial connection. In typical FCS configurations used at VCU, this GPS receiver is used along with an infra-red attitude sensor for aircraft state estimation. In the configuration used for this project however, this GPS receiver is not populated. Instead, a MIDG II inertial measurement unit (IMU) is used which provides a full sensor suite including the GPS receiver. The MIDG II is discussed further below.

Radio Modem

A Maxstream Xstream radio modem was used to provide the telemetry and control link to the ground station. This provided a plug and play 9600 baud data link with the PC GCS software. The OEM module was used on board the vehicle, and a packaged version (with metal enclosure) is used for the ground station end. Maxstream has both RS232 and USB versions of the modem available. Additionally, the modems can be replaced with a Maxstream Xtend modem, which provides a pin-for-pin compatible link at 115200 baud. However, experience has shown that the Xtend modems cause more interference with the 72MHz manual pilot link, potentially disrupting the backup pilots ability to control the vehicle.

Barometric Sensors

Two barometric sensors, one absolute and one differential, are used to provide altitude and airspeed estimates, respectively. The absolute pressure sensor is a Freescale MPX5100AP. It provides a measurement range of 15 to 115kPa (2.2 to 16.7 psi) with an analog voltage output from 0.2 to 4.7. An amplifier circuit is used to apply an offset and gain to the pressure sensor output voltage in order to increase the resolution in the applicable pressure range. The gain on the T-Board is setup to provide a measurable altitude range up to approximately 5000 feet.

The MPX5010DP is used for the differential sensor to provide an airspeed measurement. A pitot-tube is mounted on the wing and connected to the port of the sensor, so that a pressure will be induced by

forward airspeed. The differential sensor is capable of measuring pressures ranging from 0 to 10 kPa (0 to 1.45 psi). This allows measurements of airspeeds ranging from 0 to 255 knots.

A zero point calibration is required for both sensors, and it is performed at power-on. In the case of airspeed, this calibration is required primarily to adjust for variation in the zero-pressure voltage output of the sensors from part to part. For altitude, the barometric pressure at ground level varies depending on location and weather conditions.

Intertial Measurement Unit

A commercial Inertial Measurement Unit (IMU), the Microbotics MIDG II, can be used instead of the uBlox GPS and thermopile sensors to provide faster, more accurate position and attitude estimates. The MIDG II is a standalone unit with a GPS receiver, 3-axis accelerometers, magnetometers, and rate gyros, and a processor to perform data acquisition and state estimation algorithms. It provides vehicle position at 10Hz, and attitude angles at 50Hz update rates via an RS232 serial port. Although significantly more expensive than the IR/uBlox state estimation sensors, the MIDG II provides a higher performance in a small package. A significant amount of testing done with both this unit and several other commercial IMU units has found that the MIDG II provides the most accurate high speed position updates, and is the least susceptible to being upset by erratic high-G maneuvers. The unit measures 1.50” x 1.58” x 0.88”, weighs 55 grams, and consumes 1.2W max power.

Figure 2.4: The MIDG2 IMU provides complete attitude and position estimation in a small form factor

Chapter 3: Software Architecture

This chapter will discuss the flight control software that runs on the Microblaze processor. The flight control system runs as an application on the uClinux operating system. The architecture of this application will be covered in this chapter, and the control laws implemented by the application will be covered in detail in the following chapter.

Software Overview

The FCS system is based on the Linux operating system, in particular, the Microblaze uClinux distribution[9]. This distribution is a modified linux kernel, designed to support small embedded processors like the microblaze; it provides support for processors without a memory management unit (MMU). Until the most recent release of the Microblaze (v7.00), no hardware memory space management was available. The benefit provided by the linux operating system is ease of development, and easy support for file systems, networking, etc. The primary downside is the extra overhead in processing time (e.g. Context switching, slow kernel driver access), extra space required for the OS, and limited real-time performance.

The FCS application runs in a single thread, at a configurable periodic rate. Figure 3.1 shows the main program flow. First, all available data from all of the incoming serial streams is processed into packets and handled as required. These incoming streams include commands from the GCS, orientation and

position estimates from the GPS/IMU, status from the ECU, or when in HIL simulation mode, "fake" sensor data from the simulation. This data is stored into memory to be used by later processes. Next, analog sensors (e.g. Barometric and thermopiles) are read, and the data is processed to update the state estimate accordingly. Once all of this data has been acquired, and the state stored to memory, the NAV system is run to update the vehicles navigation goals. The NAV system also calls the lower level FCS control routine, which updates the PWM outputs to the servos and ECU. Finally, a series of routines are called to write downlink messages to the GCS, save log data (when enabled), and send control information to the simulator (when in SIM mode).

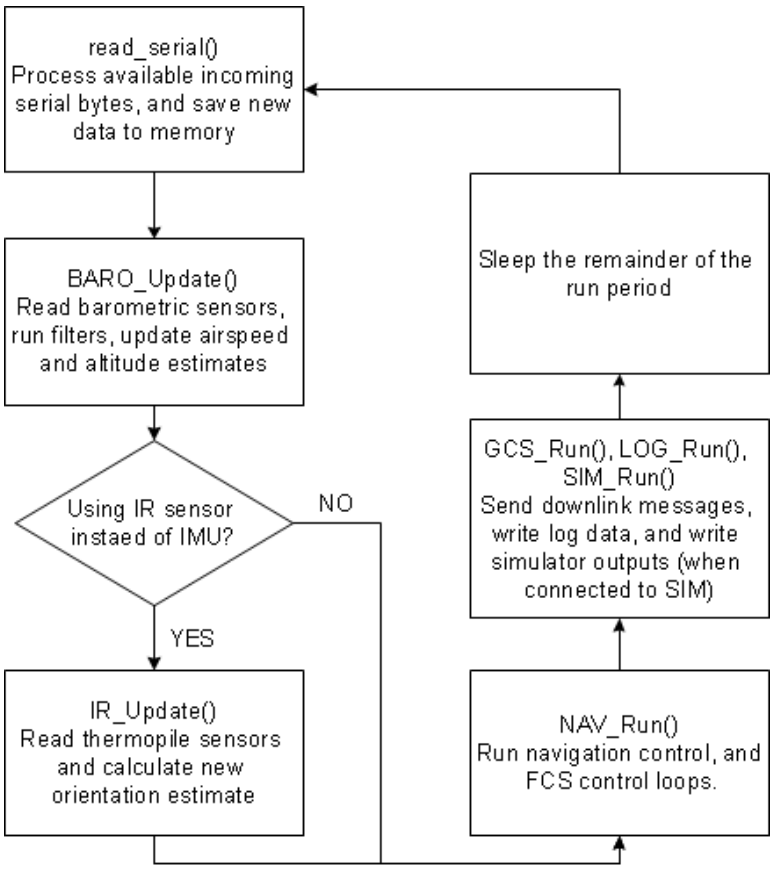


Figure 3.1: Software periodic loop flow chart

The time is checked at the beginning and at the end of execution of each loop period. After execution,

whatever time is remaining until the beginning of the next execution period is used up by a combination of the linux usleep call (for large amounts of time), and busy looping (for high resolution control after usleep). When the execution runs over the allotted period, a overrun counter is incremented. This counter value is downlinked to the GCS in order to diagnose when the processing load becomes too heavy to maintain the specified frequency of execution. Typically, a 50Hz execution frequency is used.

Most of the math used for state estimation and control law calculation is done in single-precision floating point. The floating point unit of the microblaze is enabled so that this is computationally feasible. Although significantly less efficient, use of floating point math greatly eases code development, and ensures sufficient precision in all calculations.

Software Modules

The FCS software is broken into many modules to provide specific functions. In particular, different sensors that interface to the FCS are each wrapped in a software module. Because the use of sensors is configurable at compile time, an extra layer of abstraction is added in the SENSOR module. This provides compile time configuration support for different sensors, such as the uBlox GPS and IR attitude sensors, uBlox GPS and Atair IMU, or the MIDG2 providing both GPS and attitude estimates. Many other options are also configurable at compile time (e.g. Serial port assignments), and these options are set in the build_config.h header file. When built and programmed into the microblaze system, all of the specified options as well as the build location, user, date, and time can be display by logging into uClinux system via telnet and running the FCS program with a command line option.

```
UCU-FCS Build configuration:
Built by: mcbridejc
Build date: Wed Aug  1 07:48:18 EDT 2007
Built in: /home/mcbridejc/cvswork/suzaku_fcs
GPS_SOURCE: MIDG2
IMU_SOURCE: MIDG2
ECU: Disabled
Serial ports:
        GCS: /dev/ttyS1
        GPS: /dev/ttyS2
        IMU: /dev/ttyS2
        ECU: /dev/ttyS3
        SIM: /dev/ttyS5
PLATFORM: DX
ADC_OVERSAMPLE: 25
Plane type 1, Version 3
```

Figure 3.2: Sample FCS build configuration output

Development Tools

The software was written for the uClinux operating system. A modified version of the uClinux code base is maintained by Atmark-Techno with some Suzaku-specific changes. This is the code base used to build the linux kernel and basic userspace applications (such as shell, ftp, telnet server, etc). The flight control system runs as an executable in the linux kernel. It is compiled with the Microblaze GNU C Compiler (mb-gcc), released by Xilinx.

Configuration Storage

The suzaku flash includes a section set aside for user application storage. This flash section is used to store FCS configuration parameters while powered down. A default set of parameters is built into the code at compile time, however this can be changed via commands sent from the GCS over the radio modem link and saved to flash. This section of flash is 64kB in size. The configuration data saved includes servo calibration, vehicle message send rates, and FCS configuration options such as default altitude targets, and control loop gains.

Hardware-in-the-loop Simulation

Support for hardware-in-the-loop (HIL) simulation is built into the FCS software, by means of an extra serial port. When a command is sent over the simulation port to enable HIL simulation mode, data from the real sensors is ignored, and in it's place data received over the simulation port is used. The NAV

and FCS modules are then run as normal, unaltered and indifferent to the fact that the data they are operating on came from the simulator and not from actual sensors. After the control laws are run, the control values are sent to the simulator (they are also output to the vehicle, so that control surface movement can be observed).

The mechanism for simulation mode is built into each of the sensor modules (BARO, IR, UBLOX, and MIDG2). Each of these modules must provide function calls to enable simulation mode, and to accept injected simulation data. When simulation mode is enabled, the sim module calls functions in these sensor modules to provide the simulation data when it is received over the serial port. This way, whatever processing is done in the sensor module (unit conversions, filtering, etc.), can still be done in the same block of code as it would in actual flight.

Flightgear[10], an open source flight simulator, and JSBSim, an open source flight dynamics modeling project, are used as the flight dynamics model (FDM) for the simulation, as well as for visualization (the flightgear display allows the user to see the UAV fly through the virtual landscape). A custom C# application runs on the simulator PC and provides the bridge between the vehicle under simulation and Flightgear. This application interfaces to Flightgear via UDP sockets, and converts the Flightgear state data into the appropriate sensor values (e.g. Barometric sensor voltages) expected by the avionics, possibly after adding a configurable amount of random noise to the sensor values. In addition, it receives control system outputs over the serial port connection to the autopilot hardware, and passes these on to Flightgear.

Module File	Description
adc.c	Routines for accessing the ADC via the SPI bus
baro.c	Provides barometric sensor read and processing
config.c	Functions to handle the saving of configuration data to flash partition
ecu.c	Engine control unit interface
fcs.c	Low-level vehicle control loops
gcs_comm.c	Ground station message handling
ir.c	Infra-red thermopile attitude measurement system
logging.c	Functions for logging flight data to a log file in RAM
main.c	Main entry point, initializes and runs all threads
mBin.c, mMIDG2.c, mQueue.c, midg2_comm.c	MIDG2 IMU interface functions provide message handling to parse and interpret the serial data stream from the sensor
nav.c	Higher level navigation control
pwm_io.c	Interface for PWM read/write cores
sensor.c	Generic sensor abstraction to allow configurable sensor packages. For example, a Sensor_GetAttitude() would provide attitude information from the appropriate source, such as the IR sensors or the MIDG
sim.c	Provides interface to the PC simulator to override sensor inputs with simulated data
ublox.c, ubx.c	Ublox GPS module interface
vacs.c	Message parsing utilities for the VACS protocol used for GCS and simulator connections

Table 3.1: Software Module List

Chapter 4: Control System

This chapter describes the flight control algorithms used in the FCS and their implementation. The navigation and flight control software supports several different modes of operation. In general, the control system can be broken up into two independent control loops. The longitudinal controller actuates the throttle and elevator to control airspeed, pitch, and altitude/climb rate of the vehicle. The lateral controller actuates the ailerons and rudder of the vehicle to control bank angle, turn rate, and heading. It is common to segment aircraft control systems this way for both analyses and controller design because the longitudinal and lateral axes on the aircraft system are only very weakly coupled.

For convenience, the control system is also broken up into two levels of hierarchy: Navigation and flight stabilization. Each level is implemented in it's own software module, with a well-defined interface between them. This has the advantage of separating higher level, mostly aircraft independent navigation control from the low level control that is strongly dependent on the aircraft response. This section will discuss the background of how the PID control loops are implemented, the lower level flight control laws that are used, the interface for providing target course information to the FCS module, and finally the navigation modes and controllers that implement them.

Flight Stabilization

The low-level flight stabilization “inner-loop” controllers are implemented in the FCS module. They

are built primarily out of a set of Proportional-Integral-Differential (PID) controllers, or in some cases a subset of the PID controller, such as a PI controller (in which the differential gain is zero).

PID Controller

The control loops make use of standard Proportional-Integral-Derivative (PID) controllers, or in some cases using a subset of these three factors (e.g. A PI, or P controller). The PID controller is a well-established type of controller, frequently used to control a large range of systems[11]. The classic PID controller is a single-input, single-output controller, meaning one output control signal is generated to zero out a single error signal.

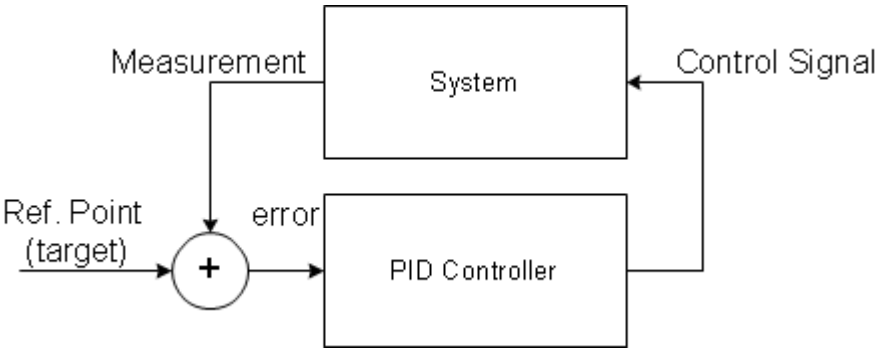


Figure 4.1:Generic PID controller block diagram

The PID controller generates an output signal based on three components of the error signal:

- The proportional error is the instantaneous error value
- The integral error is the integral of the error (The sum of all previous errors)
- The derivative is the instantaneous rate of change in the error

Many different implementations of the PID controller have been realized, with some presenting certain challenges and requiring adjustments to how certain components are handled. A PID controller can be implemented in the analog world using op-amps and discrete RC elements, or it can be implemented in a digital controller. The general form of the PID controller is:

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_d \frac{d}{dt} e(t)$$

In this case there are three tunable parameters, controlling the gains of the three error components. In order to be implemented in a digital processor, this formula must be discretized. With an adequately high sample rate (e.g. Twice the bandwidth of the system to be controlled), discretization of the proportional and integral components is straightforward. However, the derivative component is more difficult to discretize, as it is impossible to calculate an instantaneous rate of change. This can cause instabilities in the system, and makes the controller more susceptible to noise. For this reason, extra filtering is typically required in a digital controller when the derivative control is desired, and the differential gain must be kept smaller, if not eliminated all together.

The PID controller below is the implementation used for this flight control system, with the derivative gain sometimes set to zero to implement a PI controller:

$$u[n] = K_p \left(e[n] + K_i \sum_0^n e[n] + K_d e_f[n] \right)$$

$$e_f[n] = e[n] - e_{accum}[n]$$
$$e_{accum}[n+1] = e_{accum}[n] + \Delta t K_{LAG} e_f[n]$$

In this formula, e_f denotes the filtered derivative error signal, Δt denotes the time period period between control loop iterations, and K_{LAG} is the gain constant for the lag filter applied to the derivative error. This pseudo-differential calculation provides improved noise immunity, at the cost of an artificial lag

on the differential term[12].

In addition, the proportional gain constant is treated differently in this version of the equation. The gains have been redefined so that the proportional gain is applied to the all three output components. This removes a dimension from the integral and derivative gains by removing the units of output (e.g. Deflection angle), and allows the gain of the controller to be adjusted without changing the relative contribution of the time based terms (I and D). Although this does not fundamentally change the controller, it does make the tuning process more intuitive. In this form, Ki has units of 1/seconds for all loops, and Kd is denominated in seconds.

FCS Control Modes

The FCS module implements all low level controls, and allows for several different modes of control and parameters to configure those modes. It is controlled by providing a set of C structures with the operating parameters and control loop gains, and it returns four normailzed (-1 to 1) control positions: ailerons, elevator, rudder, and throttle. It is intended specifically for fixed wing aircraft, however it is capable of controlling a variety of fixed-wing vehicles with different sets of control parameters.

At every iteration of the FCS control system, the *FCS_Run()* function is called. It is passed as an argument a pointer to an *FCS_STATE* structure containing all of the required state variables for the aircraft, including location, orientation, velocity, and airspeed. Note that these are not raw sensor measurements, but are the result of a state estimation process outside the scope of the FCS module.

Prior to calling *FCS_Run()*, the FCS module must have been provided an *FCS_PARAMS* structure containing all of the control loop parameters (gains, limits, etc.) shown in table 4.2. It must also have been provided with a *FCS_CONTROL* structure (see table 4.1) to specify all of the control targets (e.g. Waypoint locations, target altitude, control mode). These structures are provided via the

FCS_SetParams() and *FCS_SetControl()* functions appropriately, and may be changed every iteration, or they may stay constant for long periods of time. .

The FCS module separates the longitudinal and lateral control. They have independent gains, independent targets, and are controlled by independent controllers. Although there is a relationship between lateral (location) and longitudinal (altitude) targets when flying (Changing altitude when navigating to a certain waypoint, for example), this is handled in the NAV module by updating the *FCS_CONTROL* structure. A callback function can be provided to the FCS module, and it will be called anytime the FCS module determines that it has reached a waypoint. This is used by the NAV module to update waypoint based control targets, such as adjusting target altitude or airspeed for a new waypoint.

Longitudinal Modes

The FCS controller is designed to support 4 different longitudinal modes of control:

1. Elevator controls altitude, and throttle controls airspeed
2. Elevator control airspeed, throttle controls altitude
3. Elevator controls pitch, throttle controls airspeed (no altitude target provided)
4. Glide slope mode, in which elevator is used to maintain climb rate to stay on a specified glideslope, and throttle controls airspeed

Lateral Modes

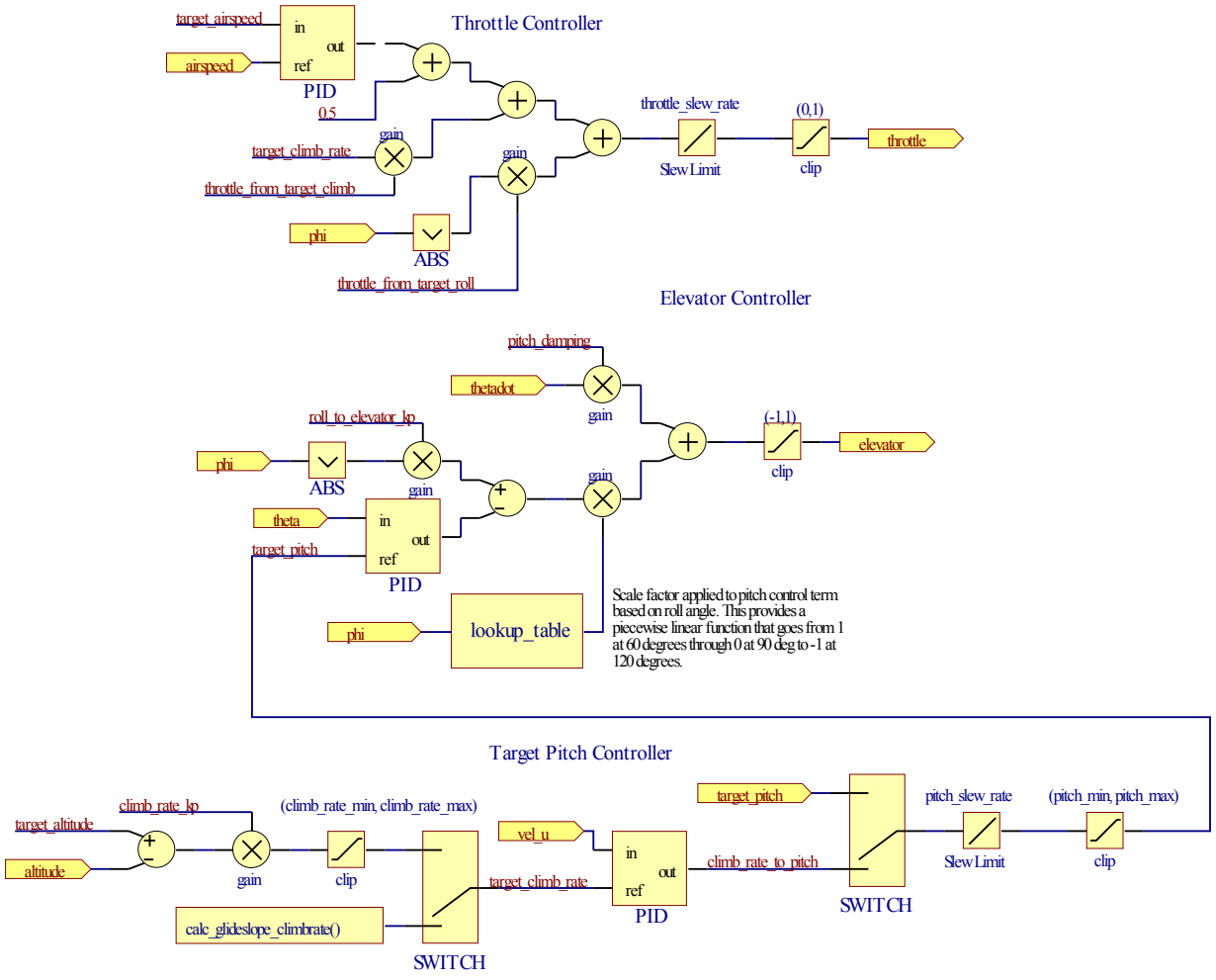
Four different lateral control modes are also supported:

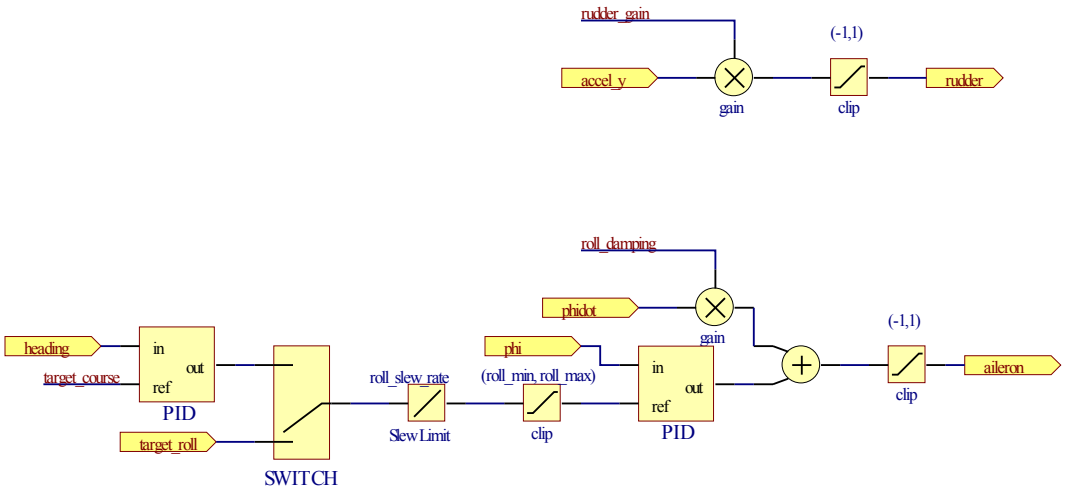
1. Course hold – a specified bearing is maintained
2. Roll angle hold – maintain a target roll angle

3. Waypoint nav – Fly a sequence of specified latitude/longitude points

4. Loiter – Loiter around a specified location

For any of these modes, the FCS also supports an inverted flight mode. When this mode is enabled, the vehicle is flown upside down.





Longitudinal Control

The longitudinal controller controls the aircrafts pitch angle, airspeed, and altitude using the elevator control surface and engine throttle. For this control system, throttle is used to control airspeed, and elevator is used to control aircraft pitch and climb rate. In reality, these two terms are strongly coupled. In fact, an increase in throttle will only cause a transient increase in airspeed. Once a steady-state is again reached, the airspeed will be unchanged; instead the rate-of-climb will have increased by some amount. It is equally possible to design a control system such that throttle is used to control climb rate, and elevator is used to control airspeed. A more advanced approach might use a multi-input, multi-output control system to control for both.

Feed-forward terms are included in the throttle controller in order to alleviate the lag in response that occurs in changing flight conditions. For example, when the target climb rate is increased, additional power will be required from the engine, but the throttle control loop will not provide this until a drop in airspeed is seen. A feed-forward term is included for the target climb rate, and for the target roll angle. The latter is there to account for the extra power required to maintain airspeed and altitude in a steady state turn. The throttle output is slew limited to provide smooth engine control, as sudden jumps in throttle are capable of stalling some engines. The engine response is slow enough in both combustion and jet turbine engines that this slew limit will have a negligible effect on the control system.

In all longitudinal control modes, a PID controller drives the elevator to maintain a target pitch. The PID output is multiplied by a scale factor based on the current roll angle via a lookup table function. The function used is shown in figure 4.3. This is to support the inverted flight mode, by multiplying elevator control by -1 when inverted, and providing a seamless transition when in between (near +/- 90 degrees roll angle). In normal flight regimes, when the roll angle is between +/-60 degrees, the coefficient is 1 and the lookup table has no effect. When flying inverted, when the roll angle is between -120 and +120 degrees, the coefficient is -1. At +/- 90 degrees the coefficient is zero, as the elevator has no effect on pitch at this orientation.

The source of the target pitch angle depends on the active control mode. In target pitch mode, it is provided the user (or a higher level control system). In all other modes, another PID controller sets the target pitch to maintain a target climb rate. In altitude hold mode, the target climb rate is given by the altitude error multiplied by a proportional gain (P controller). In glideslope control mode, the target climb rate is calculated to keep the aircraft on a specified glideslope.

Glideslope Mode

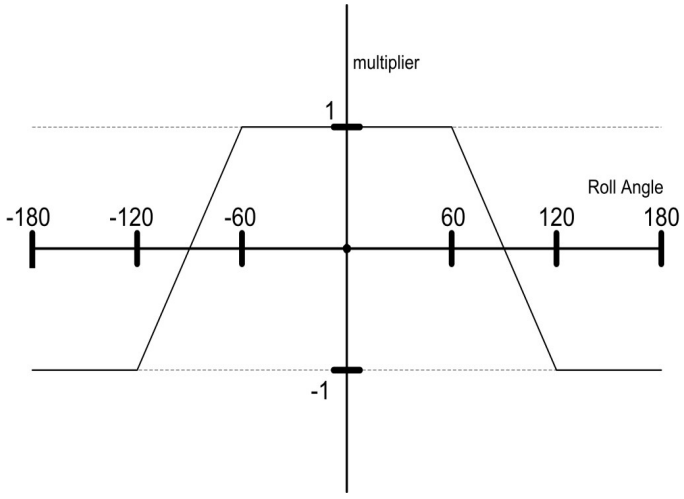


Figure 4.2: Elevator Multiplier as a function of roll. This multiplier is used to correct elevator output for inverted flight, and to smooth the transition as the vehicle rolls through 90 degrees.

A glideslope is specified by a start location and an end location, each with an associated altitude. This can be used to perform a controlled slow descent over a leg of a flightpath, or for example, to perform an automated landing approach. The glideslope controller is a function that generates a target climb rate based on the vehicle's position relative to the glideslope waypoints, and the vehicles velocity component along the bearing between the two waypoints. The glideslope calculation is performed by the *calc_glideslope_climb_rate()* function in fcs.c. The first step is to determine when the vehicle is currently between the two waypoints, or it not, whether it is on the start side (wp1) or the end side (wp2). When outside the glideslope region, the controller tries to maintain either the start or end altitude, depending on which side it is on. This can be determined by comparing the differences of the bearings from the vehicle to each of the glideslope waypoints and the bearing from wp1 to wp2:

$\Psi_{(wp1,wp2)}$ = Bearing from waypoint 1 (start) to waypoint 2 (end)
$\Psi_{(wp1)}$ = Bearing from current location to waypoint 1
$\Psi_{(wp2)}$ = Bearing from current location to waypoint 2

$\Delta_1 = \Psi_{(wp1)} - (\Psi_{(wp1,wp2)} - 90)$ (Normalized to [-180, 180])
$\Delta_2 = \Psi_{(wp2)} - (\Psi_{(wp1,wp2)} - 90)$ (Normalized to [-180, 180])

If both Δ_1 and Δ_2 are positive, then the vehicle is on the start side of the glideslope region. If they are both negative then it is on the end side of the glideslope. If the signs are opposite, it means the vehicle is currently between the two points.

When in between the two glideslope waypoints, the climb rate is calculated from two components: The climb rate required to maintain the glideslope, based on the current vehicle speed, and a correction factor proportional to the current error in altitude from the glideslope.

$$climbrate = \frac{(alt_2 - alt_1)}{(d_{12} * v_{gs})} + (alt_{gs} - alt) * K_p$$

alt_2 = Ending altitude
alt_1 = Start altitude
d_{12} = Distance from wp1 to wp2
v_{gs} = Component of ground speed along glideslope bearing
alt_{gs} = Glideslope altitude for current position
alt = Current vehicle altitude
K_p = Proportional correction gain

Lateral Control

The lateral control loop drives the rudder and ailerons to control the vehicles roll angle and navigate it along the desired lateral trajectory. The ailerons are used to provide a rolling moment to stabilize the vehicle, and to control it's bank angle for turning. The rudder is used to cancel any adverse yaw generated by the ailerons. The rudder control is very simple. Rudder position is proportional to the vehicle's y-axis acceleration; that is, the axis pointed along the spar of the wing.

The primary lateral control is provided by the ailerons. The inner loop consists of a PID controller controlling the vehicle bank angle. In bank-angle-hold mode, the target bank angle is provided to the FCS. In all other modes, the target bank angle is the output of a second PID controller controlling the vehicle heading (direction of travel derived from GPS ground speed).

The target heading comes from different sources, depending on which of the three navigation modes are enabled: Waypoint, waypoint with crosstrack correction, or loiter.

Waypoint Mode

In waypoint mode, the vehicles target heading is simply the bearing from its current position to the current waypoint. When in waypoint mode, each waypoint has an associated flag specifying whether or

not crosstrack correction should be enable when navigating to it. When set, an extra term is added to the heading to guide the plane back onto the line that connected the two waypoints. The crosstrack corrected heading is given by:

$$\Psi_{desired} = \Psi_{waypoint} + e_{ct} * Kp$$

where,

$\Psi_{waypoint}$ = Bearing from current position to current waypoint
e_{ct} = Crosstrack error
K_p = Cross track error gain term

Crosstrack error is defined as the distance from the vehicle's current position to the nearest point on the line connecting the current waypoint to the previous.

Loiter Mode

When placed into loiter mode, the FCS will navigate to the provided location and continue circling it until instructed to do otherwise. A function is defined for loiter mode to generate a target heading based on the position of the aircraft relative to the loiter point. When set with appropriate parameters, this function can provide a smooth transition from transit to the waypoint to loitering about it. The desired heading for loiter is calculated as:

$$\Psi_{adjust} = sign(d_{waypoint} - r_{loiter}) * (d_{waypoint} - r_{loiter})^2$$
$$-30 \leq \Psi_{adjust} \leq 90 \text{ (adjust limit)}$$
$$\Psi_{desired} = (\Psi_{waypoint} + 90) + sign(d_{waypoint} - r_{loiter}) * (d_{waypoint} - r_{loiter})^2 * K_{loiter}$$

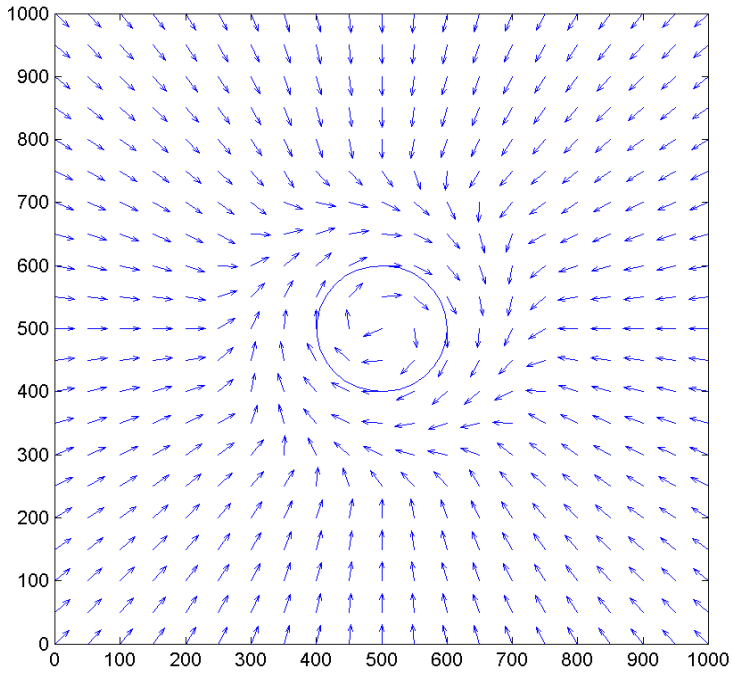


Figure 4.3: Target heading field lines for a loiter radius (r_{loiter}) of 100 meters, and loiter correction gain (K_{loiter}) of 0.0025. Axes denote x and y position in meters.

When placed in loiter mode, the FCS will circle the specified location indefinitely. The NAV module is responsible for determining when to enter loiter, and when to leave. Of course, the specified loiter radius must be appropriate for the vehicle or it will not be achieved. In general, the plane will find a steady state orbiting outside of the specified loiter radius when the radius is too small.

Field Name	Description
Longitudinal mode	One of: 0 - Elevator controls altitude, throttle controls airspeed 1 - Elevator controls airspeed, throttle controls altitude 2 - Elevator controls pitch, throttle controls airspeed 3 - Glideslope
Lateral mode	One of: 0 - Hold course 1 - Hold roll angle 2 - Fly waypoints 3 - Loiter
Invert Flight	Boolean value, true to enable inverted (upside-down) flight mode. It applies to all lateral modes
Longitudinal Targets	
Target airspeed	Desired airspeed, in knots
Target altitude	Desired altitude, in meters (Applies only when long. Mode = 0, 1, or 2
Target pitch	Desired pitch angle, in degrees (applies only when long. Mode = 3)
gs_lon1	Glideslope starting longitude
gs_lat1	Glideslope starting latitude
gs_alt1	Glideslope starting altitude
gs_lon2	Glideslope endpoint longitude
gs_lat2	Glideslope endpoint latitude
gs_alt2	Glideslope endpoint altitude
Lateral Targets	
Target course	Desired ground track heading (Applies when lateral mode = 0)
Target roll	Desired roll angle (Applies when lateral mode = 1)
Loiter Longitude	Longitude of loiter point (Applies when lateral mode = 3)
Loiter Latitude	Latitude of loiter point (Applies when lateral mode = 3)
Loiter Radius	Desired radius of loiter circle, in meters (Applies when lateral mode = 3)
Arrival range	Maximum range to waypoint required to “capture” a waypoint and proceed to next (Applies when lateral mode = 2)
WP Count	The number of waypoints in the following list
WP List	Array of *FCS_WP* structures containing waypoint locations

Table 4.1: FCS_CONTROL structure fields. This structure specifies the control targets for the FCS module control loops.

Control Parameters

The FCS control parameters consist of a set of gains, output limits, etc. for the control loops. These parameters are aircraft specific, and should be tuned on a vehicle by vehicle basis, but do not require adjustment for different stages of a flight. A common problem among flight controller design is that the

vehicle dynamics change over the flight envelope. For this reason, it is difficult or impossible to define a set of gains for a linear controller that will be adequate over all flight conditions. Prior to flying the jet turbine aircraft, the VCU flight control systems had been successful on the Mig aircraft with a single set of gains. It was found that the faster aircraft, having a wider range of airspeeds, could not be reliably controlled at all speeds with a single set of gains. The most common method of overcoming this problem is gain scheduling[13]. The FCS parameters are scheduled by airspeed. During the gain tuning process, two sets of parameters are specified. Each set has a corresponding airspeed value, and each parameter is linearly interpolated between the two points based on the current airspeed at each iteration of the controller.

The airspeeds at which the high and low speed parameters are specified is yet another variable for tuning, and varies depending on the flight envelope of the aircraft. In general, the two sets of gains should be specified close the the ends of the operating spectrum -- near the minimum and maximum airspeeds -- in order to limit the possibility of instabilities outside of the two points. Although the interpolated gains not be optimum over the entire envelope, if they are stable at either end, it is assumed that the will be stable between. Scheduling gains has been shown in flight tests to allow better control over a range of airspeeds, though it does add complexity to the gain tuning process.

At each iteration of the control loop, the parameters to be used are calculated by interpolation based on the currently measured airspeed, as shown below.

$$slope = \frac{(Param_{high} - Param_{low})}{(Airspeed_{high} - Airspeed_{low})}$$

$$Param_{interp} = Param_{low} + slope \times (Airspeed_{current} - Airspeed_{low})$$

Name	Description
Elevator Controller	
pitch_damping	Elevator from pitch-rate gain
pitch_kp	Elevator from pitch proportional gain
pitch_ki	Elevator from pitch integral gain
pitch_imax	Elevator from pitch integral component limit
roll_to_elevator_kp	Elevator from roll angle feed-forward gain
Target Pitch Limits	
pitch_slew_rate	Slew rate limit for target pitch (deg/s)
pitch_max	Target pitch upper limit (deg)
pitch_min	Target pitch lower limit (deg)
Pitch from Airspeed Gains	
airspeed_pitch_kp	Proportional gain
airspeed_pitch_ki	Integral gain
airspeed_pitch_kd	Derivative gain
airspeed_pitch_imax	Integrator limit
Pitch from Target Climb Rate	
climb_pitch_kp	Proportional gain
climb_pitch_ki	Integral gain
climb_pitch_kd	Derivative gain
climb_pitch_imax	Integral limit
Throttle from Climb Rate	
climb_throttle_kp	Proportional gain
climb_throttle_ki	Integral gain
climb_throttle_kd	Derivative gain
climb_throttle_imax	Integral limit
Target Climb Rate Control	
climb_rate_max	Climb/descent rate limit
climb_rate_kp	Proportional gain for climb rate controller
glideslope_kp	Glideslope proportional gain for glideslope altitude mode
Throttle from Airspeed	
airspeed_throttle_kp	Proportional gain
airspeed_throttle_ki	Integral gain

Name	Description
airspeed_throttle_kd	Derivative gain
airspeed_throttle_imax	Integral limit
throttle_slew_rate	Maximum slew rate for throttle
Throttle Feedforward Gains	
throttle_from_target_climb	Target climb to throttle feed-forward gain
throttle_from_target_roll	Target roll angle to throttle feed-forward gain
Rudder Control	
rudder_gain	Y-axis (lateral) acceleration to rudder proportional gain
Roll Control	
roll_damping	Derivative gain, but is applied directly to gyro roll rate, rather than PID
roll_kp	Proportional gain
roll_ki	Integral gain
roll_imax	Integral max
Target Roll from Header Controller	
heading_roll_kp	Proportional gain
heading_roll_ki	Integral gain
heading_roll_kd	Derivative gain
heading_roll_imax	Integral limit
crosstrack_kp	Crosstrack error to target heading p-gain (degrees/meter) used in crosstrack mode only
Roll Limits	
roll_slew_rate	Roll slew limit
roll_max	Maximum roll angle (typically equal to roll_min)
roll_min	Minimum roll angle (typically equal to roll_max)

Table 4.2: FCS Control loop parameters

Navigation

The navigation module provides higher level navigation control. It accepts flight paths from the ground station, interprets them, and manages the FCS control loops to achieve the desired flight path. The

NAV module supports four top-level modes of operation, as listed in table 4.3. However, for a typical flight, the default "Waypoint" mode is all that is required. In this mode, the control is specified by whatever flightpath has been uploaded by the GCS. The NAV module steps sequentially through the waypoints, which can contain a variety of behaviour specifications, including altitude and airspeed targets, or loiter commands.

In additions to following the waypoint, several "Override modes" are supported (see table 4.2). When an override mode is entered (either because of an operator command, or some detected condition such as loss of GCS link), the behaviour of that modes override the current flightpath. However, when the mode is disengaged the NAV module returns to the same point in the flightpath and resumes navigation.

Mode	Description
Waypoint	Standard operating mode, in which the vehicle follows the specified flightplan
Waypoint Inverted	The same as Waypoint mode, except the vehicle flies upside down.
GCS Manual	All controls are uploaded from the GCS. Due to high round-trip latencies, this mode has never been used in flight, but can be useful for test purposes.
Pilot controls Roll	In this mode, the FCS controls the longitudinal axes (Elevator, throttle) to maintain specified altitude and airspeed, but the RC pilot has direct control of the ailerons and rudder.

Table 4.3: High level NAV modes, these modes can be selected by the GCS operator.

Override Mode	Description
Rally	Return immediately to saved home location
Takeoff	Takeoff mode
Land	Land at saved runway
Roll	Perform roll maneuver
Loiter	Loiter at current or specified position

Table 4.4: NAV Override Modes. These modes override the primary control mode either briefly or indefinitely

Flightpath Specification

A flightpath is a sequence of one or more waypoints. Each waypoint at a minimum must specify a location as a longitude and latitude. However, several other parameters can be specified for each waypoint, providing more flexibility in controlling the flight of the vehicle. Upon reaching the last waypoint in the flightpath, the vehicle's default behavior is to return to the first waypoint and repeat the sequence. Alternatively, the final waypoints loiter mode can be set to “Wait”, causing the vehicle to loiter indefinitely at the final waypoint rather than repeating the flightpath.

Field	Description
Longitude	Waypoint location longitude (degrees)
Latitude	Waypoint location latitude (degrees)
Altitude	Waypoint altitude (meters) (Use depends on altitude mode)
Altitude mode	Altitude Specification: None – No altitude specified for this waypoint Leg – Altitude to be maintained for the entire leg to this waypoint Glideslope – Altitude of current and previous waypoint specify endpoints of a constant climbrate glideslope to be followed
Airspeed	Waypoint Airspeed in Knots
Airspeed mode	None – No airspeed specified for this waypoint Enabled – Airspeed should be maintained for the leg preceding this waypoint
Loiter time	A time to loiter at this waypoint when loiter mode “Timed” is specified
Loiter mode	None – Do not loiter, proceed straight to next waypoint Timed – When reached, loiter at this waypoint for the specified time Wait – When reached, loiter at this waypoint until a continue command is received from the GCS
Crosstrack mode	Enable or disable crosstrack navigation mode.

Table 4.5: Waypoint Options

Chapter 5: Ground Control Station

The Ground Control Station provides the human interface to monitor and control the air vehicle, as well as several other unmanned vehicle systems developed at VCU. It consists of several Microsoft .NET applications, written in C#, running on a laptop computer. In general, there is one server application that acts as a gateway between all vehicles in the system, and all control clients. There are several types of control clients, for controlling different vehicles, or for special purposes such as reviewing imagery collected from the video payload. The general GCS architecture and specifically the FCSControl client application which is used to control this FCS is discussed in this section.

GCS Design Goals

- Concurrently support different vehicles with different messaging protocols.
- Identify different types of vehicles and load a corresponding protocol definition from a configuration file at run-time to allow for adjustments in message formats without recompiling of GCS software
- Receive arbitrary data from a vehicle and make it available to all clients. For example, a new telemetry value could be added to a particular vehicle type, and be made accessible clients without the GCS having any knowledge of what that data value is, and without required a

compilation of the GCS software.

- Communicate with multiple UAVs over a single connection using the VACS protocol.
- Support software simulated vehicle running on local computer or on a second computer via a network connection

GCS Architecture

The GCS is setup in a "star" configuration, with one server application at the center, one or more clients connected to it via a TCP link, and one or more vehicles connected to it via radio modem or other means. The server application provides a central place for all data passed between users (client applications) and vehicles. It communicates with the vehicles via radio modems connected to a USB or RS232 serial port on the PC. All vehicle communication is encapsulated in VCU Aerial Communication Standard (VACS) packets. The server can simultaneously support several different types of vehicles, each with a different set of message types being sent and received.

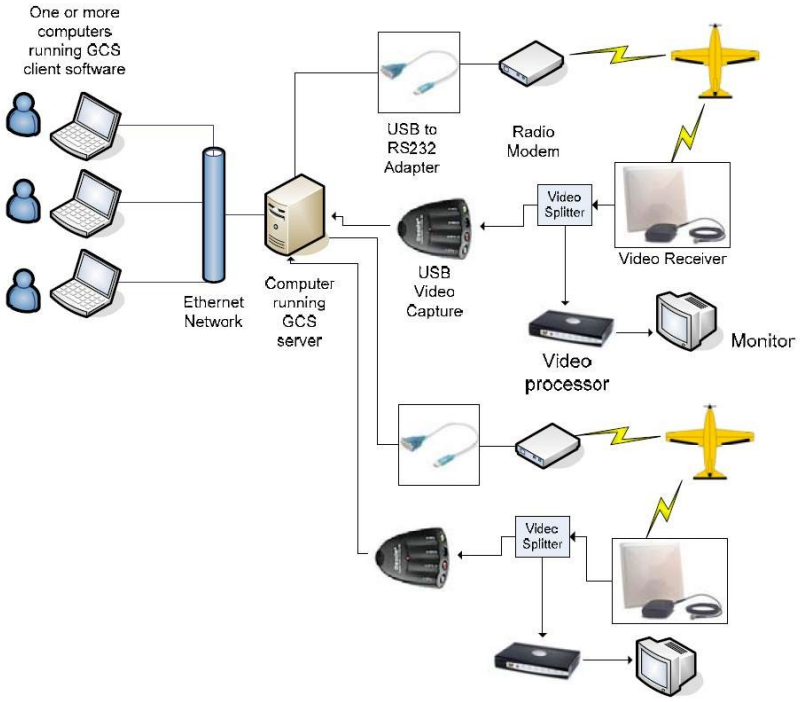


Figure 5.1: GCS Architecture Diagram

Plane Controller

The job of communicating with the vehicles is performed by a set of modules in the GCS server called Plane Controllers. These modules are all C# classes conforming to the IPlaneControl interface. Although not dynamically loadable (e.g., they are compiled into the application), the plane controllers can be thought of as plugins. When setting up the GCS server, the user must add a plane controller for each communication channel that is to be controlled. To date, three different plane controllers have been developed:

1. VACSPlaneController: The standard plane controller that interfaces to one or more VACS conforming vehicles via a serial port.
2. SimplePlaneSim: A simple vehicle simulator, this module does not interface to anything, but instead runs a simple model of an ideal vehicle that will respond to flight path commands. It can be used for testing and demonstration.

3. UDPPlaneController: Similar to the VACSPlaneController, this controller controls VACS conforming vehicles via an IP network. It was designed to support control of distributed simulations of multiple vehicles for a collaborative UAV simulation environmen, where the simulated vehicles could reside on different computers and execute full flightgear based vehicle models.

A C# interface defines a set of methods and properties. A C# class can inherit from one or more interfaces, meaning that it implements the methods and properties specified by those interfaces. The IPlaneControl interface allows the GCS Server application to instantiate any class that inherits from IPlaneControl, and control it via those methods. The methods and properties of the IPlaneControl class are shown in Table X.

Type	**Name**	**Description**
PropertyTreeNode	ControllerNode	The property tree node which this controller is assign to control. This must be set by the server upon instantiation of this controller.
string	DataDirectoryPath	The path to the GCS data directory
string	MissionDataPath	The path to the directory where mission data is saved
string	Name	Read only, the controller should return a descriptive name for itself
Method	ShowControl()	Causes the controller's (custom) control dialog to be displayed
bool	IsActive	Read only, returns true if the controller has active planes

Table 5.1: IPlaneControl Interface

When instantiated by the server, the plane controller class must be setup with some requisite information: It must be assigned a node in the property tree under which it will store data related to the vehicles under its control, and it must be given that paths to the GCS data directory (where plane definitions, map data, etc. are stored) and a path to the current mission data directory (where log files, etc. are stored). Having a link to the property tree node is sufficient for the controller to control the

plane. In addition, a method is provided to display the controller's control form. In this way, each controller can provide a custom windows form (or forms) to monitor, configure, and control it. For example, the VACSPlaneController must provide a dialog to select COM ports and configure baud rates, and the SimplePlaneSim must provide a dialog to position the plane, and control it's model parameters, such as speed.

Property Tree

All of the data associated with the vehicles in the system are stored on the server in the property tree data structure. Modeled after the property tree used by the open-source flight simulator Flightgear, the property tree is a collection of data nodes organized in a tree structure. There is one root node, with many branches underneath. Each vehicle in the system has an associated node, underneath which is stored all of its data. For example, an aircraft's current latitude might be stored in the node specified by the path */controller[0]/uav[0]/position/latitude*. Most of the communication with client applications is performed by sharing the property tree. When a client connects, it gains access to the full property tree system, and communicates with the vehicles through reads and writes to property tree nodes.

Plane Definitions

One of the primary goals of the GCS software was to support the variety of autonomous vehicle projects being performed at VCU, without requiring different ground control software, or maintaining several variants of the software. Also, it is desirable to be able to make changes to the data communicated to and from a vehicle without having to modify and rebuild the ground station software. This is accomplished by creating plane definition files for each type of vehicle, and mapping all communications into the property tree. The plane definition is an XML file that defines what messages can be sent and received by a vehicle, and maps each item of data to a specific node in the property tree on the GCS server. For transmit messages, it also defines certain trigger nodes, so that clients can send

a message by writing the correct value to the trigger node.

Each plane definition is identified by two numbers: a vehicle type and a version. When a new plane is powered up, or is first detected by the GCS, it reports its plane type and version number. The GCS them searches its collection of plane definitions for the appropriate file, and loads it for that plane. A sample plane definition is provided in appendix A.

FCS Control Application

The GCS architecture provides a generic interface for communicating with a variety of vehicles. However, the user interface often must be customized for different applications. This is accomplished by creating a client application. The FCSControl application is the client application designed for controlling the fixed-wing FCS described here. It provides a moving map display with aerial picture overlay for monitoring the planes position, and inputing flight path info. It also provides gauges and text display to display the vehicle's telemetry information. Additionally, dialogs are provided for changing the FCS control parameters, etc.

Figure 5.2: FCSControl application screenshot

VACS Communications

Although each different vehicle can support a different set of messages, defined by the plane definition XML file, there are is a common packet framing protocol, and a set of common messages shared by all. These are defined by the VCU Aerial Communications Standard (VACS).

Packet Framing

The packet format for all VACS packets is shown in Table 5.1. In order to support multi-drop networking, a source address and destination address is specified in each packet. This allows multiple vehicles can communicate with each other and with the ground station using a single radio channel. A 2-byte message type ID specifies the format of the data payload of the message, and a two byte

checksum follows each message so that the validity can be verified by the receiver.

Byte	Field
0	Sync1 Byte (0x76, 'v')
1	Sync2 Byte (0x63, 'c')
2	Destination Address
3	Source Address (address of the sending host)
4	Message ID (High Byte) (defines type of message
5	Message ID (Low Byte)
6	Length N (Gives length, in bytes, of the data payload (0 to 255)
...	Data bytes (N bytes of data)
N+6	Checksum A
N+7	Checksum B

Table 5.2: VACS Packet Format

The checksum is calculated using a 16-bit Fletcher algorithm, similar to the TCP protocol error detection scheme. The checksum calculation includes the data bytes as well as the destination address, source address, message ID, and length. It does not include the two sync bytes. The two bytes of the checksum, ChkA and ChkB, and calculated as follows:

```
ChkA = 0;

ChkB = 0;

// Array bytes contains the data to which the

// check sum is being applied

for(int i=0; i<num_bytes; i++)

{

   ChkA = ChkA + bytes[i];
```

```
    ChkB = ChkB + ChkA;

}
```

Plane Type Query Standard Message (0x0)

Message ID 0x0 is reserved for the Plane Type Query message, which must be implemented by all vehicles. The query is sent from the GCU when it wishes to identify an unknown plane, and the response is sent by the queried vehicle to identify it plane type and version. This information is used by the GCS to load the correct plane definition file. The message from GCS to air vehicle has a zero-length data payload. The response message from the air vehicle is the same message ID, but has a three byte data payload, as described in table 5.2.

Byte	**Type**	**Field**
0	U16	Plane Type
2	U8	Plane Version

Table 5.3: Plane type response data payload format

GCS Ping Standard Message (0xFF)

Message ID 0xFF is reserved for a GCS ping message. This message, when enabled, is sent out every 5 seconds by the ground station to all air vehicles. It can be used by the air vehicles to detect loss-of-link to the GCS. It is enabled by setting the <*enable-ping*> property to “true” in the plane definition file. It contains a zero-byte data payload.

Chapter 6: Results and Performance

Testing and performance evaluation of the FCS has been performed on several platforms. The Flightgear based hardware-in-the-loop simulator was used as a low-cost, safe platform for initial testing of the system, and three different aircraft were used for flight testing.

The HITL simulation does not include a precise model of the actual aircraft which much be controlled, and models only a simple random noise for sensors. However, it allows the majority of the software and processing hardware to be tested for functionality, and provides an environment similar to actual flight for testing of ground control, as well as the FCS. The HITL simulator proved invaluable in increasing the effectiveness and safety of flight testing by finding most bugs prior to leaving the lab.

Initial flight test were performed in the FG-117 Mig propeller aircraft. After confirming operation on this vehicle, the system was moved first to the DV8R jet turbine, and finally to the Jurassic jet turbine model.

Flight Test Process

The goal of flight testing was to evaluate the performance of the control system, and to tune it in order to increase that performance, addressing any problems discovered. All flight tests were performed with a human pilot who could, at any time, either fly the plane manually or allow the FCS to control the plane. During a flight, all of the FCS parameters discussed in the Controls section previously could be changed from the ground control station. Initial values for these were chosen by estimation based on the vehicle, and testing for stability on the HITL simulator.

When first finding parameters after moving to a new vehicle, the longitudinal axes were singled out first. In order to support this, a mode of control was provided where the FCS would control the

longitudinal axes, and allow the pilot to continue to steer the plane. While the pilot steered the plane in a typical race-track pattern, gains were adjusted until the vehicle was stable in pitch, and held altitude with minimal steady state error. Then a series of altitude steps, up and down, are commanded to evaluate stability and ensure reasonable climb/descend rates are achieved.

Once the longitudinal control loops are sufficiently tuned, the FCS is commanded to take over control of the lateral axes, and a similar process is repeated for these control loops. In this case, the parameters of interest are roll stability, achievable turn radius, the cross-track error of the vehicle from the desired flightpath, and the amount of disturbance in pitch/altitude when entering or leaving a turn.

Test Data

The following data was collected during a flight test on October 31, 2007[14], flying the Jurrasic aircraft. Measured data is based on the MIDG-II state estimate in most cases, with the exception of airspeed with is based on the pitot pressure sensor.

Longitudinal Control

The following figures show data collected for the longitudinal controllers (altitude/pitch angle and airspeed).

Figure 6.1, figure 6.2, and figure 6.3 show longitudinal data from a single flight sequence where target airspeed is varied, and target altitude is held constant.

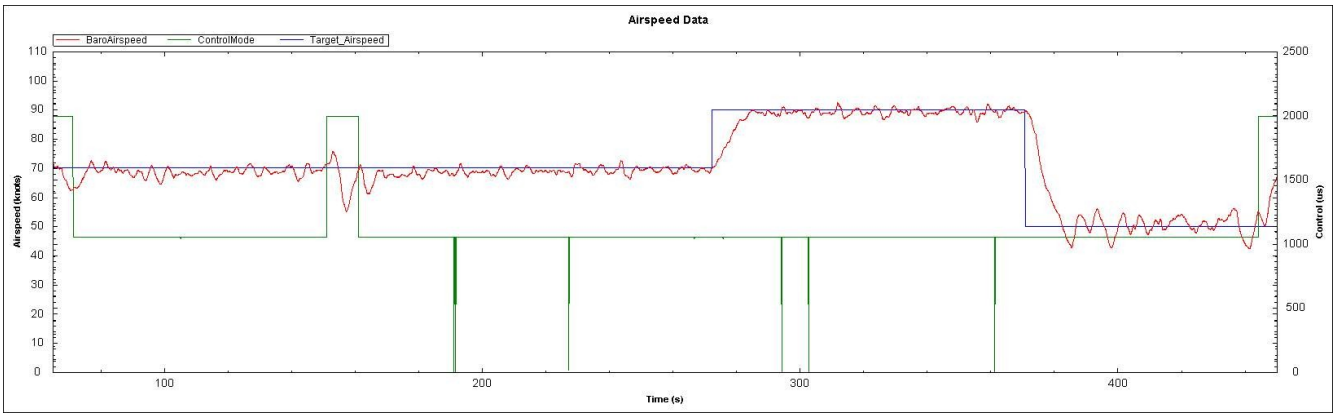


Figure 6.1:Acutal and target airspeed. The green line indicates whether the vehicle was in automous (value less than 1500) or manual mode (value greater than 1500).

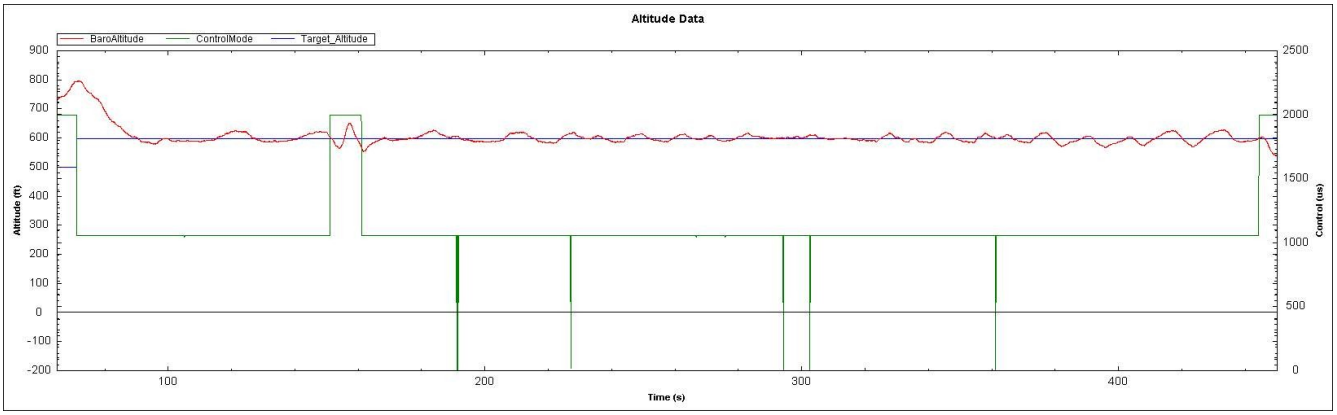


Figure 6.2: Altitude versus a constant target altitude. Maximum deviations are +27/-34 feet during the two lower speed sections, and +/-18 feet during the high speed section. The green line indicates whether the vehicle was in automous (value less than 1500) or manual mode (value greater than 1500).

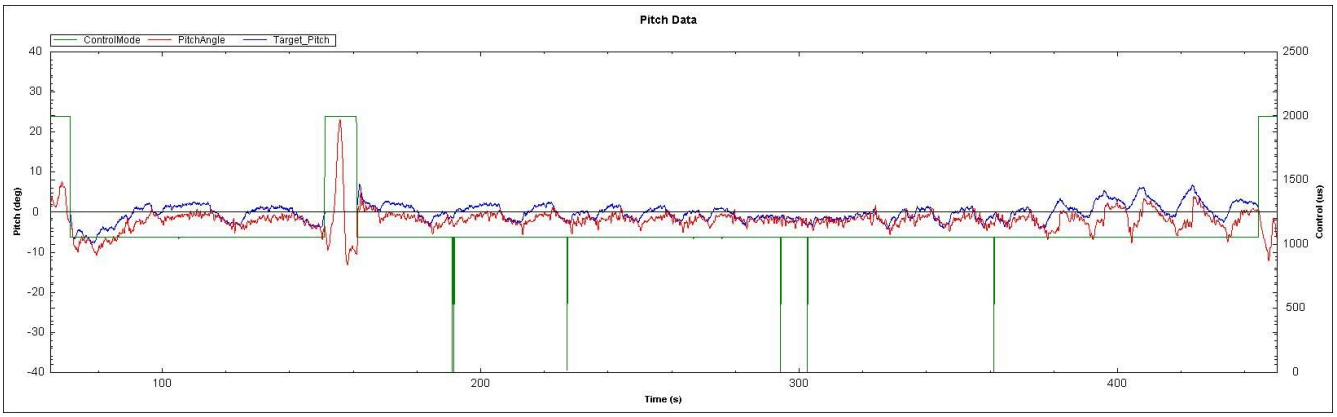


Figure 6.3: Measured vs. target pitch angle. Actual pitch angle appears to lag behind the target when pulling up (increasing target). The green line indicates whether the vehicle was in automous (value less than 1500) or manual mode (value greater than 1500).

Figure ? Show a second flight sequence in which four airspeed target changes are made while holding target altitude constant, followed by three step changes to the altitude target while airspeed is held constant at 70kt.

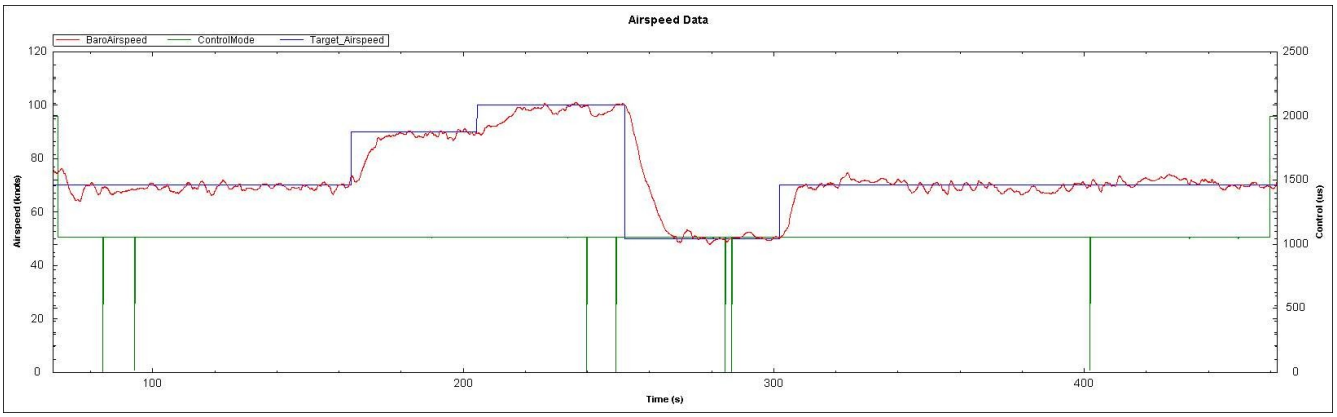


Figure 6.4: Measured vs. target airspeed for second flight sequence.

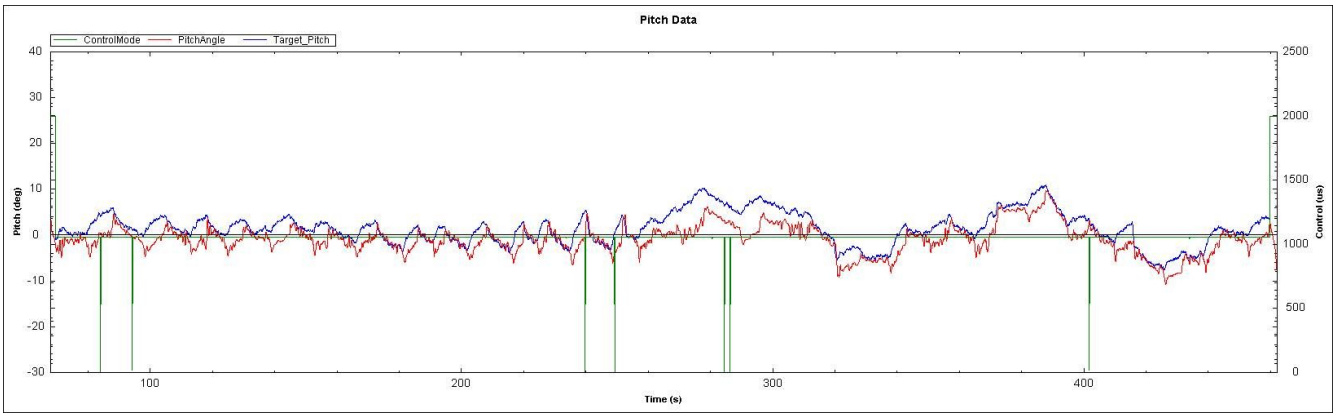


Figure 6.6: Measured and target pitch for second flight sequence. inputs in target altitude while flying at 70 knots airspeed.

Lateral Control

The following figures show data collected for the lateral control (Roll angle, heading).

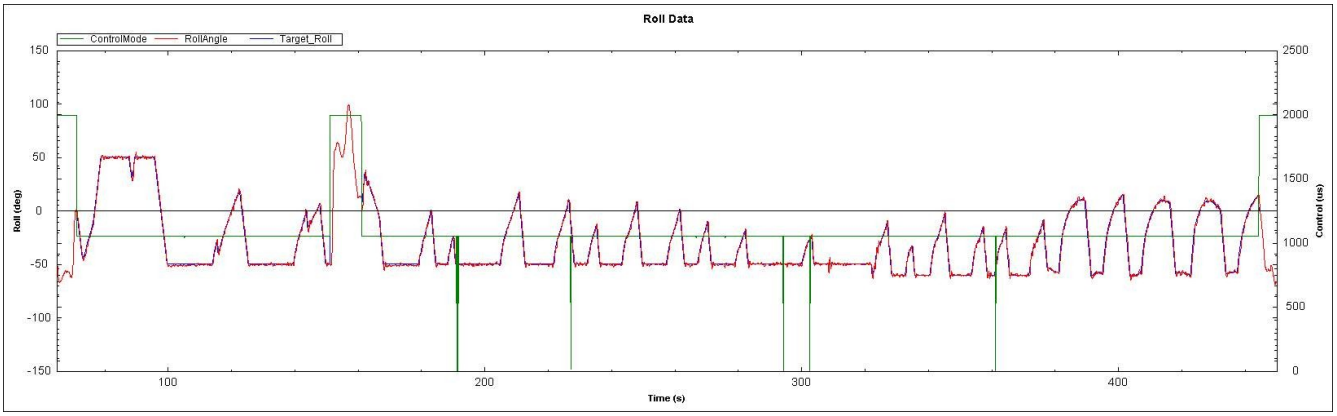


Figure 6.7: Actual versus target roll angle. It is clear that the inner roll controller is performing nearly perfectly, as the measured roll follows the target roll very closely. The green line indicates whether the vehicle was in automous (value less than 1500) or manual mode (value greater than 1500).

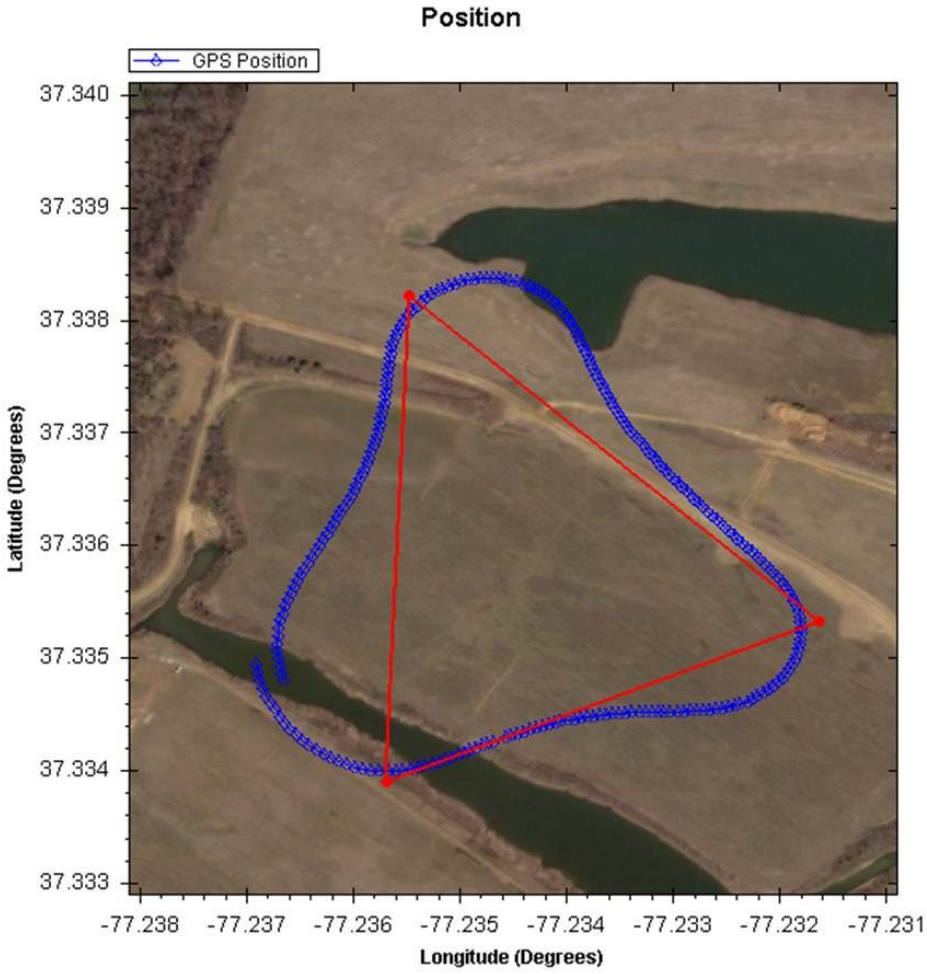


Figure 6.9: Aerial view of a clock-wise circuit flying a triangular waypoint pattern at an airspeed of 50 knots.

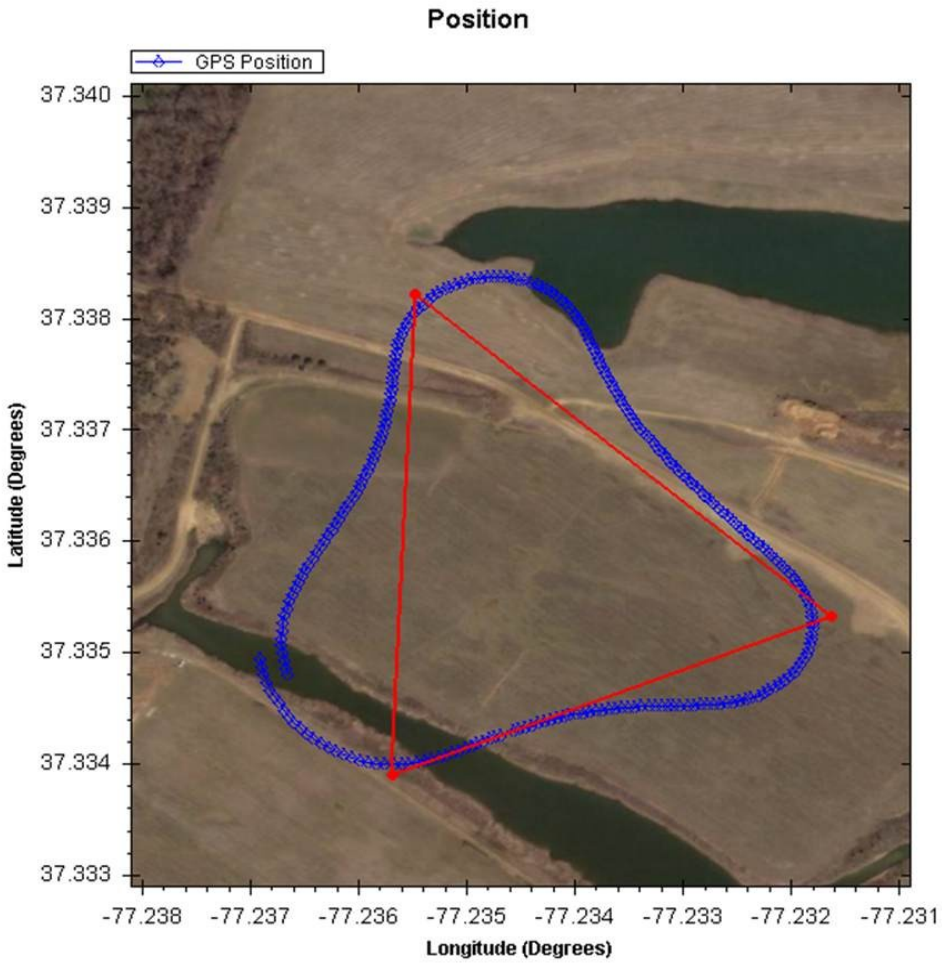


Figure 6.10: Aerial view of clockwise circuit flying a trianglular waypoint pattern at an airspeed of 50 kt.

Autonomous Roll Maneuver

The following sequence of data demonstrates the autonomous roll behaviour. A sequence of three rolls was performed while flying a figure 8 pattern at 80 knots airspeed. The rolls are clearly visible at 358s, 392s, and 426s in the roll angle plot. Again, the roll angle is tracked very accurately. Figures 6.8 and 6.7 show the altitude and pitch disturbance caused by the roll maneuver. This is due to the fact that as the plane transitions through 90 and 270 degrees roll, it has no pitch control authority.

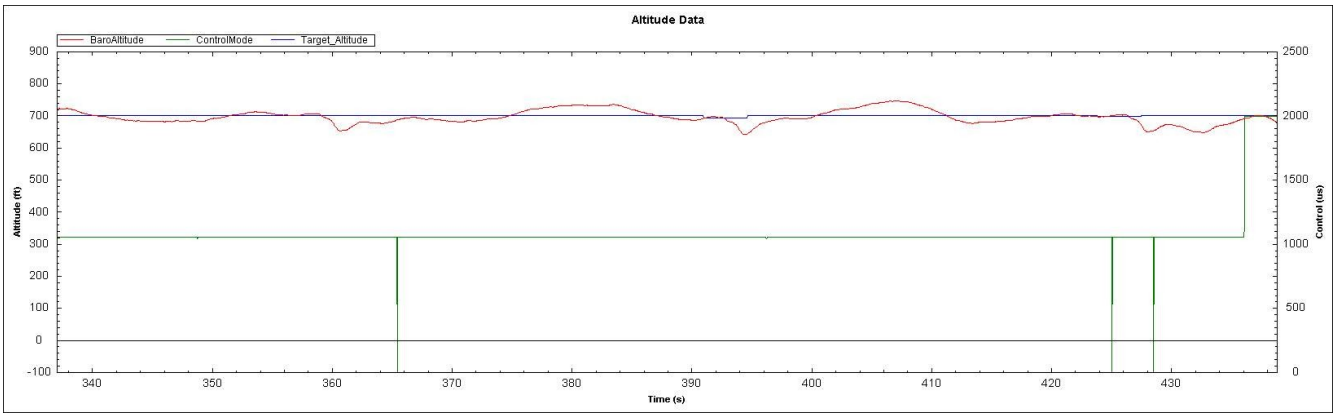


Figure 6.12: Measured and target altitude during three autonomous roll maneuvers

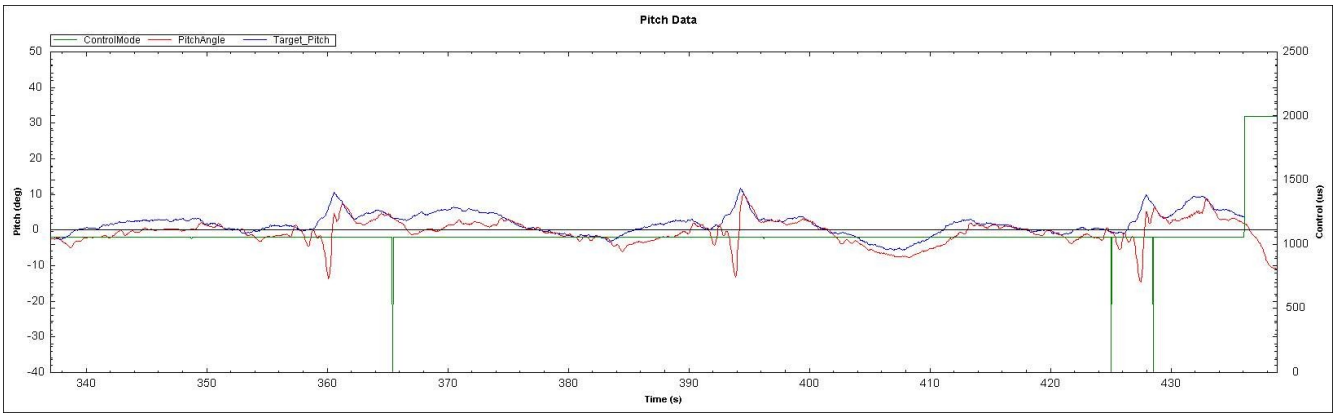


Figure 6.13: Measured and target pitch angle during three autonomous roll maneuvers

References

1 Jordan, Thomas L., Foster, John V., Bailey, Roger M., and Belcastro, Christine M.,"AirSTAR: A UAV Platform for Flight Dynamics and Control System Testing," *25th AIAA Aerodynamics Measurement Technology and Ground Testing Conference*, AIAA, San Francisco, CA, 2006

2 ??, "An FPSLIC-based UAV Control Platform", ??

3 Chandler, G.D., Jackson, D.K., Groves, A.W., Rawashdeh, O.A., Rawashdeh, N.A., Smith, W.T., Jacob, J.D., Lumpp, J.E., "A Low-Cost Control System for a High-Altitude UAV," *Aerospace Conference*, Big Sky, MT, IEEE, 2005

4 Simpson, A.D., Rawashdeh, O.A., Smith, S.W., Jacob, J.D., Smith, W.T., Lumpp, J.E., "BIG BLUE: High-Altitude UAV Demonstrator of Mars Airplane Technology," *IEEE Aerospace Conference*, Big Sky, MT, IEEE, 2005, pg 4461-4471

5 Johnson, E.N., Fontaine, S.G., Kahn, A.D., "Minimum Complexity Uninhabited Air Vehicle Guidance and Flight Control System," *Digital Avionics Systems,* Daytona Beach, FL, IEEE, 2001

6 Johnson, E.N., Schrage, D., "The Georgia Tech Unmanned Aerial Research Vehicle: GTMax", *AIAA Guidance, Navigation, and Control*, Austin, TX, AIAA, 2003

7 Wigley, G., Jasiunas, M., "A Low Cost, High Performance Reconfigurable Computing Based Unmanned Aerial Vehicle," *Aerospace Conference*, Big Sky, MT, IEEE, 2006

8 "Cloudcap Technology – Piccolo Family of Small Integrated Autopilots," http://www.cloudcaptech.com/piccolo_system.shtm

9 Microblaze uClinux Project Homepage, http://www.itee.uq.edu.au/~jwilliams/mblaze-uclinux/

10 The Flightgear Project, C.L. Olson, http://www.flightgear.org

11 Levine, W. S., "The Control Handbook", CRC Press 1996, pg 198

12 Langton, R., "Stability and Control of Aircraft Systems", John Wiley and Sons, 2006 pg 217

13 Oosterom, Marcel and Babuska, Robert, "Fuzzy Gain Scheduling for Flight Control Laws", *IEEE International Fuzzy Systems Conference, 2001, pp. 716-719*

14 "Data Analysis of VCU Jurrasic Flight Testing: Day Six", J. Cooper, R. DeMott, F. Obeidat, Oct 2007