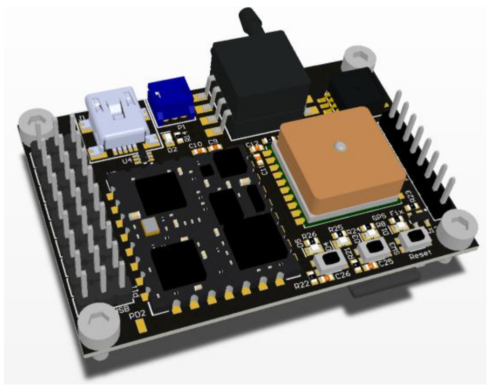
AUTONOMOUS CROSS COUNTRY GLIDER: HARDWARE DESIGN AND LOW LEVEL EMBEDDED SOFTWARE

Alexander Pabouctsidis
Microengineering – Microsystems
hepia - HES·SO



Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

h e p i a
Haute école du paysage, d'ingénierie et d'architecture de Genève

1. Table of Contents

2. INTRODUCTION

The objective of this semester project is to begin the design of an autopilot that would then fit into scale model of a glider. The glider is launched from the top of a cliff or mountain with preprogrammed GPS waypoints. As soon as the glider is launched it follows the programmed trajectory as best as it can, and once the glider reaches its final waypoint, it circles in the air waiting for the user to take command of the glider using a standard RC (i.e. Radio Control) radio in order to land the glider safely.

Considering the size and complexity of the project, only the following points will be covered in this semester project:

- The study of the feasibility of such a project and determine which sort of instrumentation would be needed
- The selection and acquisition of a scale glider that best fulfills the requirements, and if needed, the assembly of the glider
- The design a circuit board with all the required sensors and outputs, with a microcontroller at the heart.
- The low level programming of the microcontroller to access and control all sensors, peripherals, and outputs of the circuit board.
- Protocol definition and implementation, for communication between the autopilot and a PC
- Creating test routines to observe and validate low level functions of the autopilot.

3. Glider selection

There were four main criteria when choosing a suitable glider.

- The first was the size and space available inside the shell for the electronics. The glider had to have enough room to accommodate the autopilot and any other components that would be needed.
- Second, it had to have a decent glide ratio (or Life-to-Drag ratio), as the goal is to have the glider fly as far as possible with a given starting altitude.
- Third, it had to have an electric motor to allow field tests or could be used as a safety net if the glider desperately needs to gain altitude.
- And lastly, it had to be easy to assemble and inexpensive: around $400.

After posting on different RC forums with the criteria asking for recommendations, one glider came heavily recommend by multiple sources. It was the Cularis from Multiplex. The Cularis is made out of Elapor foam, making it light, rigid, and inexpensive. It has plenty of space in the "cockpit" for all the electronics, including the autopilot.

Here are its specifications:

- Wingspan : 2.61 [m]
- Length : 1.26 [m]
- Flying Weight : ~1.7 [kg]
- Wing Loading : 30 [g/dm^2]
- Wing Area : 55 [dm^2]

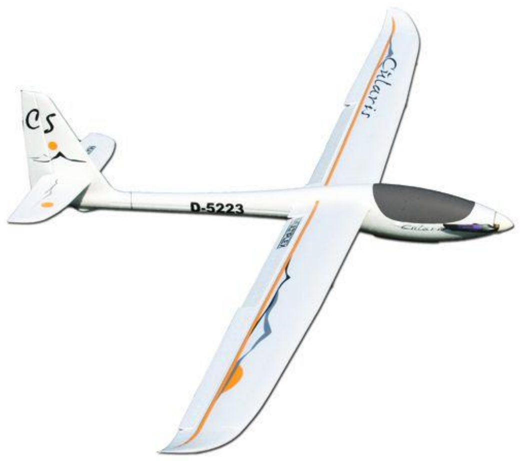


FIGURE 1: MULTIPLEX CULARIS GLIDER

The glider has a large wingspan and a low win load which, theoretically, should give it a decent glide ratio. Real life test will need to be done in order to determine the actual glide ratio and at which speed this is achieved. It has 6 control surfaces: 2x Ailerons, 2x Flaps, an Elevator, and a Rudder, allowing precise control of the aircraft.

It can be mounted with an electric motor, which will be used for field tests as well possibly as a safety thruster if the glider drops too low. The kit was bought on eBay as it was cheaper, as well as the servos. The motor and speed controller were bought from a well-known online RC store in china (www.hobbyking.com), as they are much cheaper and still high quality.

Component	Choice	Unitary price (USD)
Aileron Servos	2x Hitec HS-65	$21
Flap Servos	2x Hitec HS-65 with Metal Gears	$32
Elevator Servo	Hitec HS-85	$31
Rudder Servo	Hitec HS-85	$31
Motor	Turnigy 35-48-C 800Kv Outrunner Brushless Motor	$15
Speed Controller	Turnigy K-Force 40A Brushless ESC	$32

TABLE 1: GLIDER COMPONENT SELECTION

The autopilot can act as a receiver, using a micro satellite receiver from Spektrum. The receiver can communicate the user inputs directly to the autopilot using standard serial communication.

4. Hardware Design

4.1 Glider and Subsystems

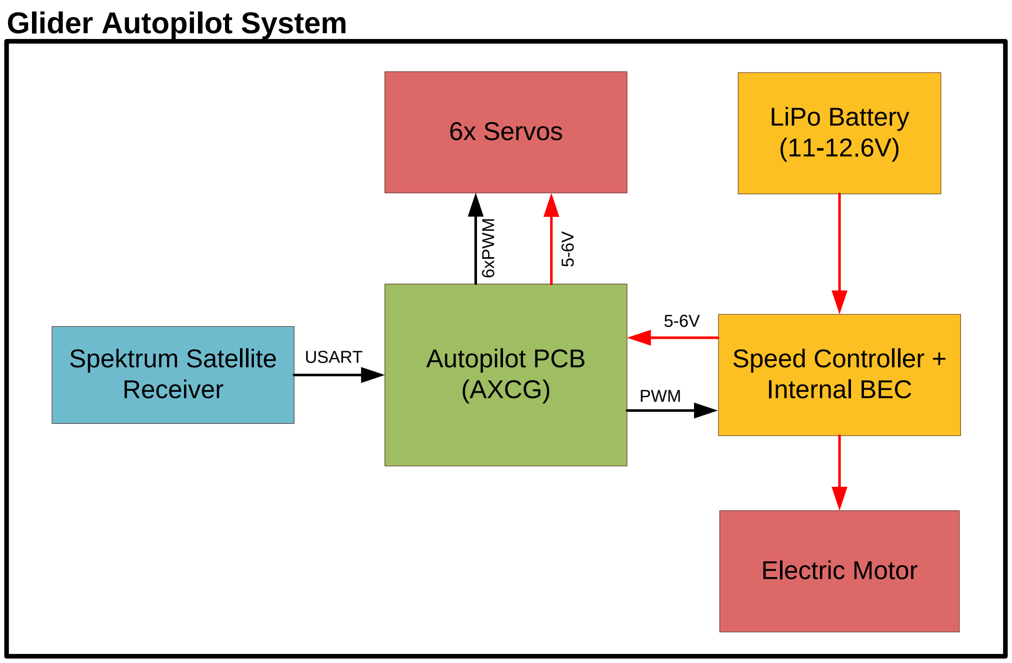


FIGURE 2: GLIDER AUTOPILOT SYSTEM

The glider is controlled using standard industry servos. Servos are controlled using a PWM (Pules Width Modulation) signal with varying pulse width. The refresh rate, or frequency can be anywhere between 30 to 100Hz. The glider can optionally mount an electric motor. If this is the case, a speed controller will be mounted which incorporates an internal battery eliminator circuit (BEC). A battery eliminator circuit is essential a voltage regulator that converts the 12V battery input to ~5-6V which is used by most RC equipment such as servos and radios, removing the need of a second lower voltage battery. The speed controller is controlled via a PWM signal just like the servos.

To control all 6 servos and the motor, the autopilot board will have 7 PWM outputs. To receive commands from a RC radio, a Spektrum satellite receiver is connected directly to the board, where the data is received via a USART bus.

FIGURE 3: SPEKTRUM RECEIVER

The glider is powered by a 3 cell 2.2Ah lithium-polymer battery which is connected to the motor speed controller, which in turn produces a separate output voltage with its internal BEC between 5.5V to 6V. This output is used to power the autopilot as well as the servos.

5. Study of Required Instruments

Besides the obvious fact that a real glider is much bigger than a scale RC glider, not much changes in the way that they fly or how the user controls the aircraft. A quick look at older general aviation aircrafts and you would notice they all have some form or another of the standard 6 pack instruments.

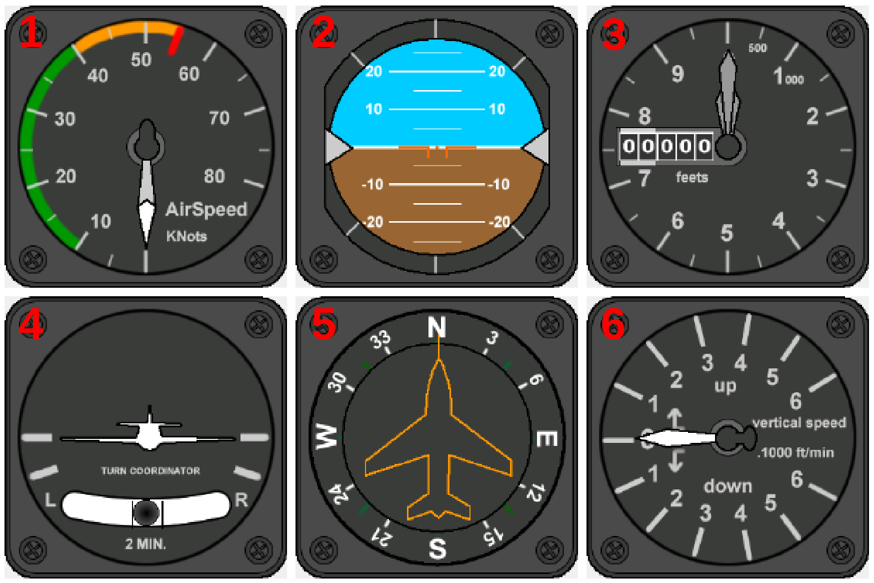


FIGURE 4: 6 PACK INSTRUMENTS

These instruments are:

1. The Air speed
2. Artificial Horizon
3. Altitude
4. Turn Indicator and Sideslip
5. Heading
6. Vertical Speed

Some instruments are not as critical as others such as the turn indicator (4) or vertical speed (6). If a pilot is flying the aircraft during the day with good visibility he can determine his orientation relative to the horizon by simply looking outside. In our case however, the autopilot does not have eyes and the artificial horizon is a very critical instrument, as it informs the autopilot of the current attitude (orientation in space relative to earth) of the aircraft. The airspeed is also a critical instrument, without it the aircraft could easily stall or go too fast and destroy some of its structure. The altitude is needed to make sure that it does not fly into the ground, or does not go too high. The heading is also required in order to notify the autopilot in which direction the aircraft is flying, so that it can correct it accordingly to make sure it stays on course.

In order to navigate, the autopilot needs to accurately know where on the globe the aircraft is, which is usually measured using a GPS.

The information the autopilot will need in order to fly are: the air speed, the attitude of the aircraft, the altitude, and the aircrafts position relative to the earth. We will go into more detail below.

5.1 Sensor Selection

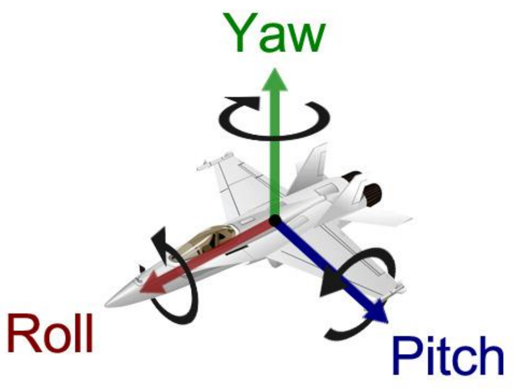


FIGURE 5: YAW, PITCH, AND ROLL ANGLES

The attitude of an aircraft can be represented by many different mathematical models such as quaternions, direction cosine matrices, Euler angles, etc. For now, the autopilot is working with Euler angles as they are intuitive, and easy to interpret.

Euler angles are a set of 3 angles which the aircraft may turn around – in a certain order – to end up in its final orientation.

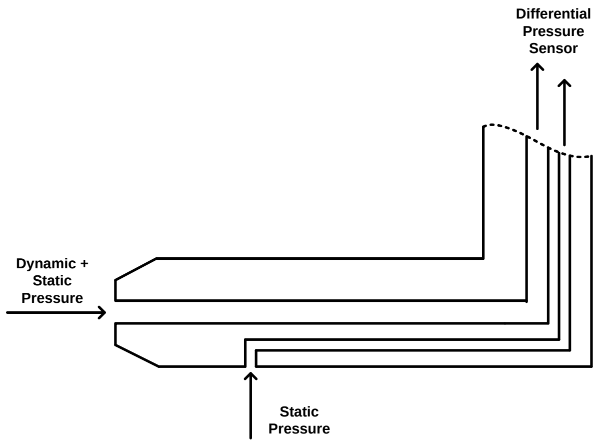
They are called: Yaw, Pitch and Roll, or YPR for short. The autopilot will work with these angles when controlling the aircraft. In the aviation industry, an AHRS (attitude and heading reference system) is used to determine the attitude (Yaw, Pitch, and roll angles) and heading. Until recently, these were very expensive as they required sophisticated sensors, and algorithms. With the fast moving evolution of MEMS (Microelectromechanical Systems) technology, many affordable AHRS's are showing up on the market, which was used to our advantage when choosing an AHRS for the autopilot.

To navigate, the autopilot will need to know its current position in space. The simplest and most accurate way is to use a GPS module. A GPS module would provide the latitude, longitude, altitude, ground speed, and heading of the aircraft.

With the GPS giving us our position and altitude, and an AHRS providing the attitude, only the air speed remains.

Measuring air speed on commercial and general aviation aircrafts is done using a Pitot tube. A Pitot tube is essentially a hollow tube that extends out of the aircraft into the air. The force of the wind applies pressure at the tip of the tube that is proportional to the wind speed. At the side of a tube is a hole with secondary airway that measures the static pressure of the atmosphere.

These two airways then feed onto a differential pressure sensor, which measures the difference in pressure between the two.



Using aerodynamics, we can then determine the airspeed. Bernoulli's equation states:

$$p_t = p_s + \frac{\rho V^2}{2}$$

V : fluid velocity [m/s]
P_t : Total pressure [Pa]
P_s : Static Pressure [Pa]
ρ : Fluid density [Kg/m^3]

Solving for velocity we get:

$$\boxed{V = \sqrt{\frac{2(p_t - p_s)}{\rho}} \quad (1)}$$

The differential pressure sensor measures P_t-P_s. Assuming that the air density is constant (for the altitudes that we will be working in), we can easily calculate V.

To summarize, the state variables that are measured are:

Yaw, Pitch, Roll, and all three angular velocities. - (α, β, γ, α', β', γ') – via AHRS.
Longitude, Latitude, and Altitude. – (X, Y, Z, V_X, V_Y, V_Z) – via GPS.
Air speed – (Vr) – via differential pressure sensor + Pitot tube.

Now that we've determined what where the outputs, the state variables, and how to measure them, we can draw the following block diagram of the control loops:

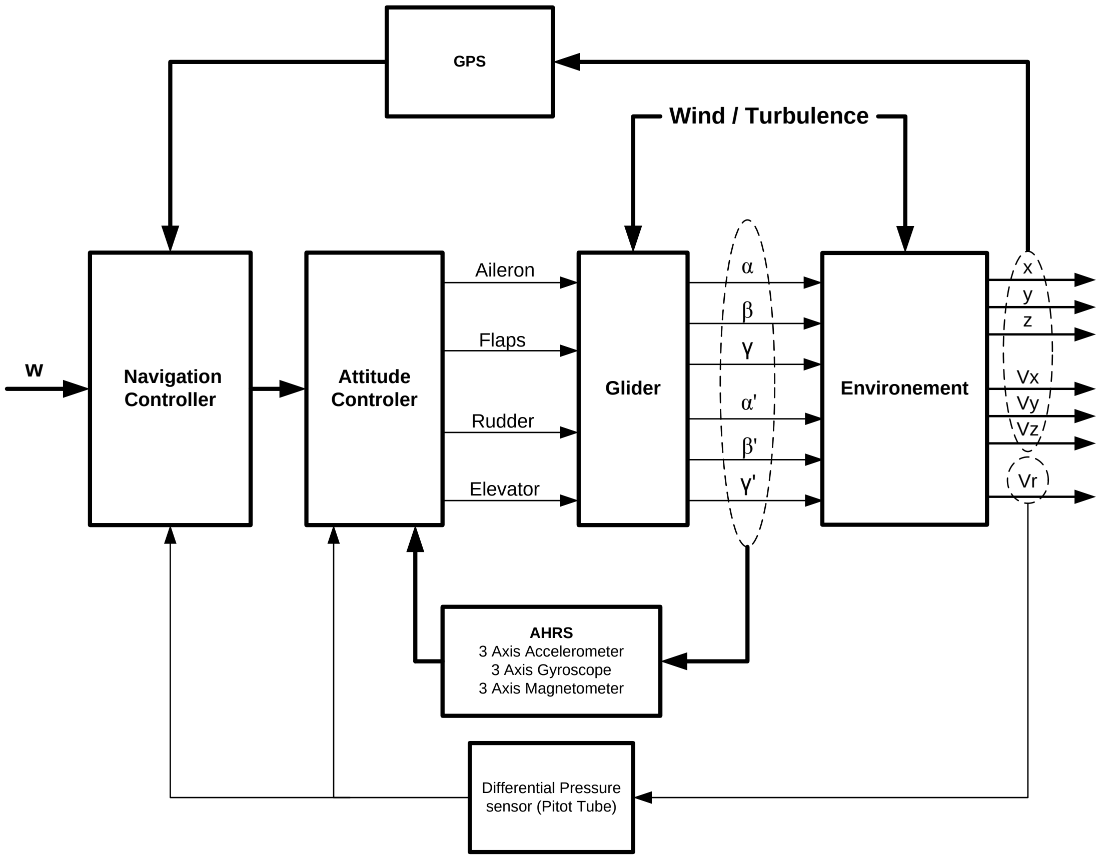


FIGURE 6: CONTROL LOOPS

The control system of the aircraft can be separated into two control loops. The inner loop controls the attitude of the aircraft, while the outer loop is used to navigate the aircraft according to the programmed waypoints. The two controllers are called:

Attitude Controller - Inner loop
Navigation Controller - Outer loop

The attitude controller needs to measure the attitude of the aircraft, or the Euler angles, as well as the angular velocities. As the force exerted by the deflection of the control surfaces is directly proportional to the dynamic pressure, the attitude controller will also need the air speed.

The navigation controller will command the attitude controller, and its set points will be the user programmed GPS waypoints. The navigation controller will need to know the current position of the aircraft (latitude, longitude, altitude) as well as its speed in the earth reference frame (Vx, Vy, Vz).

5.1.1 GPS

The search for GPS module started on digikey, where a multitude of OEM (Original Equipment Manufacturer) GPS modules were found. Some of the manufacturers were Navman Wireless, and DeLorme. One parameter they all had in common was the GPS chipset. It was very apparent that what defines the characteristic of these difference GPS modules was the GPS chipset inside. So in a sense, the search for the GPS module, turned into a search for an appropriate chipset.

The choice of the chipset was based on these important factors:

Refresh Rate: In order to facilitate navigation, the GPS had to have a high refresh rate, the higher the better.
Sensitivity: The GPS had to have a high sensitivity, as to not lose the GPS signal too easily.
SBAS (satellite-based augmentation system): Support of WAAS (Wide Area Augmentation System) and EGNOS (European Geostationary Navigation Overlay Service) to increase accuracy.

After researching many GPS OEM modules, two popular chipsets were found. The first being the SiRF Star III, and the second being the Mediatek 3329. Here is a comparison of the two chipsets:

Feature	Mediatek 3329	SiRF III – GSC3e/LPa
Channels	66	20
Sensitivity	-165 dB	-159 dB
Position Accuracy	<2.5m (with SBAS)	<2m (with SBAS)
SBAS	WAAS, EGNOS, MSAS	WAAS, EGNOS,MSAS, GAGAN
Refresh Rate	10 Hz	1 Hz
Cold Start	<35s	<35s
Warm Start	< 34s	<35s
Hot Start	< 1s	<1s

TABLE 2: MEDIATEK 3329 VS. SIRF III

Both of them have very similar features; they are both very sensitive, have SBAS support, and equivalent Cold/Warm/Hot start times. The Mediatek 3329 has more channels, while the SiRF III has slightly higher horizontal position accuracy. The deciding factor, however, is the refresh rate. The Mediatek 3329 has a very high refresh rate of 10Hz compared to 1 Hz for the SiRF III.

Now that the required chipset had been selected, a GPS module incorporating said chipset had to be found. This proved to be very difficult. The first GPS module found was the Fastrax IT520. The module had a very small size of 10.4x14.0mm. Unfortunately a supplier from which to order the GPS module could not be found.

After spending more time searching, an amateur UAV website (www.diydrones.com) which sold sensors and electronic kits for hobbyists was found. They sold various GPS modules, of which one used the Mediatek 3329 chipset. This is the module that was used for the autopilot.

FIGURE 7: MEDIATEK 3329 GPS MODULE FROM DIY DRONES

5.1.2 AHRS

Much time was spent on deciding what kind of sensor should be used for the aircraft attitude, and whether it should be designed from scratch, or bought. All modern aircrafts use what's called an AHRS (Attitude and heading reference system). An Attitude Heading Reference System consists of sensors on three axes that provide heading, attitude and yaw information of aircraft. They are designed to replace traditional mechanical gyroscopic flight instruments and provide superior reliability and accuracy.

An AHRS consist of either solid-state or MEMS gyroscopes, accelerometers and magnetometers on all three axes. The key difference between an IMU (Inertial Measurement Unit) and an AHRS is the addition of an on-board processing system in an AHRS which provides solved attitude and heading solutions versus an IMU which just delivers sensor data to an additional device that solves the attitude solution.

The on board processing of the sensor data is usually done using an Extended Kalman Filter. An Extended Kalman filter fuses the data from all three sensors, and provides the best possible estimation of the current attitude.

In the past, AHRS were reserved to the aviation industry as they relied on extremely accurate and expensive sensors. However, with the rapid advancement in MEMS technologies and embedded processing power in the last couple years, it is now possible to find Micro AHRS modules at affordable prices.

One of such modules is the VN-100, developed by a startup called VectorNav. The VN-100 is a small 22mm x 24mm x 3mm AHRS module encompassing a 3 axis gyroscope, a 3 axis accelerometer, a 3 axis magnetometer, and a 32-bit processor running an extended Kalman filter.

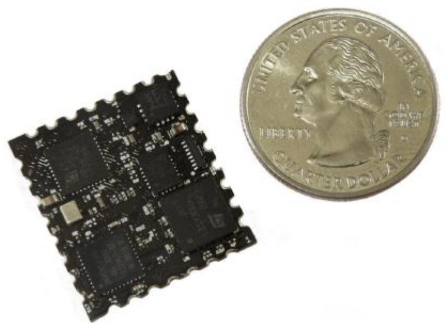


FIGURE 8: VN-100

VN-100 Sensors and Specifications:

Sensors	
Accelerometer	Analog Devices ADXL325
Angular Rate Gyro	Invensense IDG-500 & ISZ-500
Magnetometer	Honeywell HMC6042 & HMC1041Z

Specifications	
Acceleration Range	2g / 6g (User selectable)
Gyro Range	500°/sec
Magnetometer Range	6 Gauss
Orientation Accuracy	<0.5° Static, <2° Dynamic
Orientation Resolution	<0.05°
Output Modes	Euler angles (Yaw, Pitch, Roll) Quaternion Rotation matrix Raw Data (Acceleration, angular rate, and magnetic field)
Interface options	SPI: up to 18Mhz TTL Serial: up to 921,600 Baud
Data Rate	1 to 200Hz

The small size, accuracy, ease of use, and user configurability made the VN-100 a prime choice. The unit price of the module is $500, which might seem expensive (considering the hardware component costs of the module), but most IMU and AHRS found were either in or above that price range.

VectorNav also supplies a graphic PC utility called Sensor Explorer for their development kit to view the sensor output data and configure the sensor via USB. After configuring the AXCG as a USB ↔ Serial adapter between the computer and the module, the autopilot was then able to directly use the Sensor Explorer utility. A lot of time must be spent to tune the parameters of the Kalman filter, and this utility greatly simplifies the process, makes it easy, and intuitive.

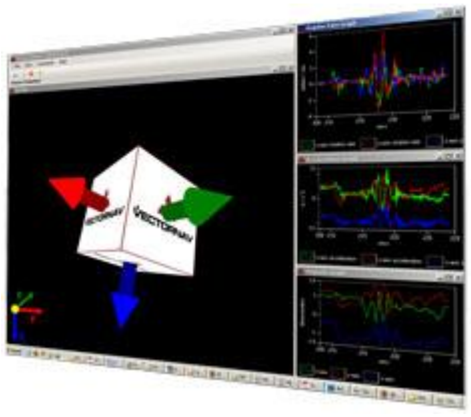


FIGURE 9: VN-100 SENSOR EXPLORER

5.1.3 Pressure Sensors

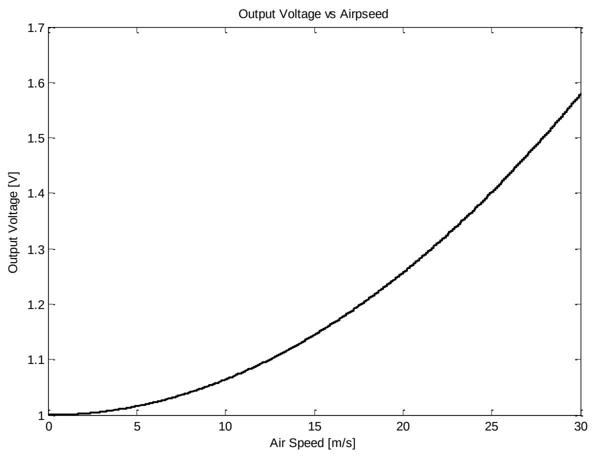
A differential pressure sensor was required for the Pitot tube, as well as an absolute pressure sensor to aid the GPS in determine the altitude, as well as providing the air density to calculate the true air speed. After searching Digikey, and other hobby electronic websites, it became apparent that the Freescale pressure sensors were widely used for Pitot tubes in amateur UAV's (Unmanned Aerial Vehicle).

The pressure sensor that was widely used is the MPXV5004DP, and is capable of measuring up to 3.92 [kPa]. The voltage output is given by:

$$V_{out}(P) = 5 \cdot [(0.2 \cdot P) + 0.2]\ [\frac{V}{kPa}]$$

FIGURE 10: MPXV5004DP

From this equation and using equation (1) from the sensor selection section, we can determine the voltage output and the ADC code vs. the air speed.



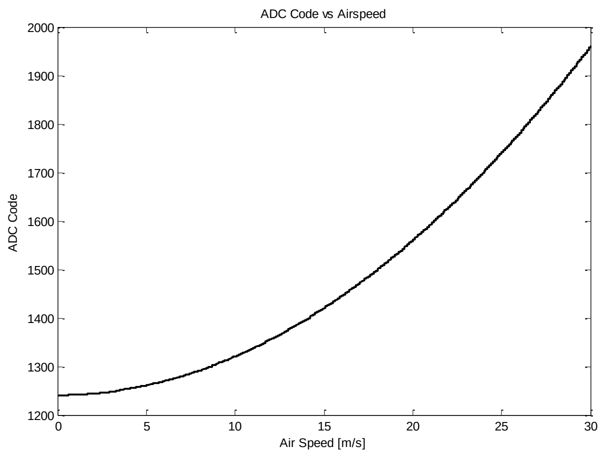


FIGURE 11: PRESSURE SENSOR OUPUT (VOLTAGE AND ADC CODE)

From these two graphs, we can see that at low velocities (below 2 [m/s]) the ADC will trouble measuring the airspeed. However as the airspeed increases, so does the sensitivity (the slope), meaning at higher velocities, the ADC will be able to accurately measure the air speed. The slowest possible speed the autopilot could theoretically measure is about 1.5 [m/s] or 5.4 [km/h]. This is the required speed for an increase of 1 LSB. The maximum speed is well over 50 [m/s] or 200 [km/h], which is when the output voltage goes over 3.3V (maximum voltage the ADC can convert).

5.2 Autopilot Circuit Board

At the heart of the autopilot is a STM32 micro controller. The STM32 microcontroller is a 32 bit ARM cortex-M3 based microcontroller family produced by STMicroelectronics. The reason for choosing a STM32 microcontroller was largely based on past experience. This is the second project on which the STM32 is being used, and being familiar with all their peripherals, saves a lot of time when writing low level hardware functions. Additionally, the STM32 microcontrollers are fast (up to 120 MHz), with sophisticated peripherals for our application needs.

The GPS module and AHRS module are connected to the microcontroller via standard serial interfaces. The pressure sensor output goes through a simple RC filter before connecting to one of the microcontroller's analog inputs. The circuit board also possesses a Micro SD card connector for data logging that is connected to the micro controller via a SPI bus.

User programming of the autopilot will be done via USB, so naturally the PCB has to have a USB connector. The servo outputs are standard 3 pin headers, with one pin connected to 5V from the BEC, the other to GND, and the third to a PWM signal from the microcontroller.

The Spektrum satellite receiver periodically sends radio data via a standard serial bus, and is connected to a USART input pin of the microcontroller. Radio data is sent in bursts of 16 bytes every 20ms, with 2 bytes per channel. The protocol was found by probing the receiver with an oscilloscope, as well as reading RC forums.

In addition, a couple user buttons and LED's are added, which are used for debugging, and user interaction.

Here is a simplified diagram of the autopilot PCB and its sub components:

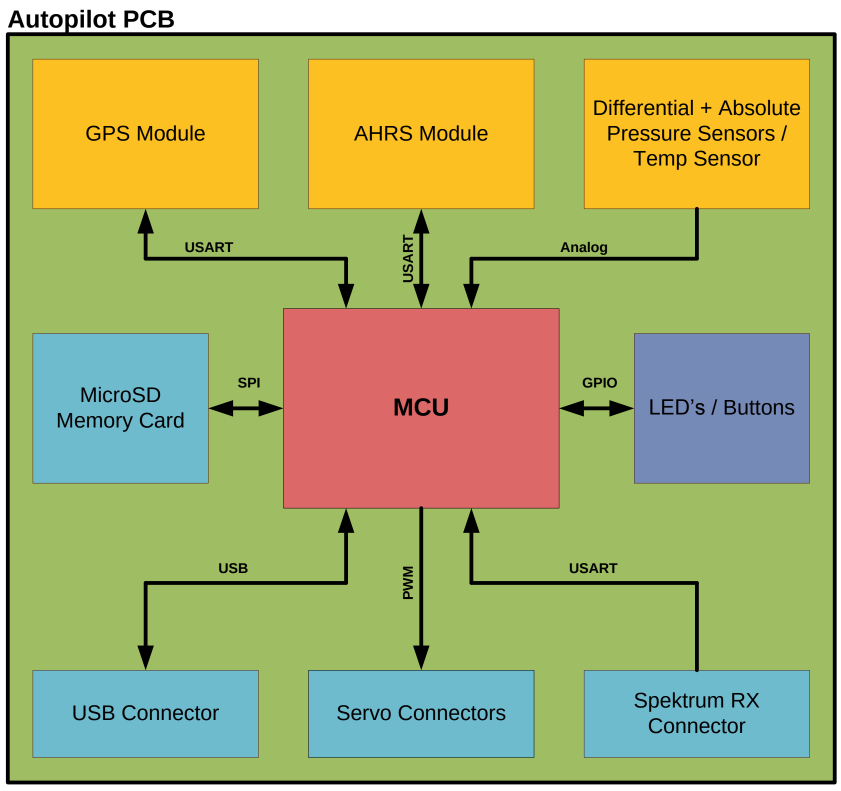


FIGURE 12: AUTOPILOT PCB COMPONENT BLOCKS

6. Schematic Design

Altium Designer 9 was used for the schematic entry, PCB layout and routing. The schematic is drawn hierarchically, and was separated in different blocks largely inspired by the simplified block diagram. The Blocks are:

- Power Supply
- GPS Module
- AHRS Module
- Pressure and Temperature sensors
- Micro SD and EEPROM
- Connectors
- LEDs and Switches
- Control

The central block is the control block. The control block contains the STM32 microcontroller, with all the net connections of the external components.

Here is the top schematic, showing all the blocks, and their interconnections:

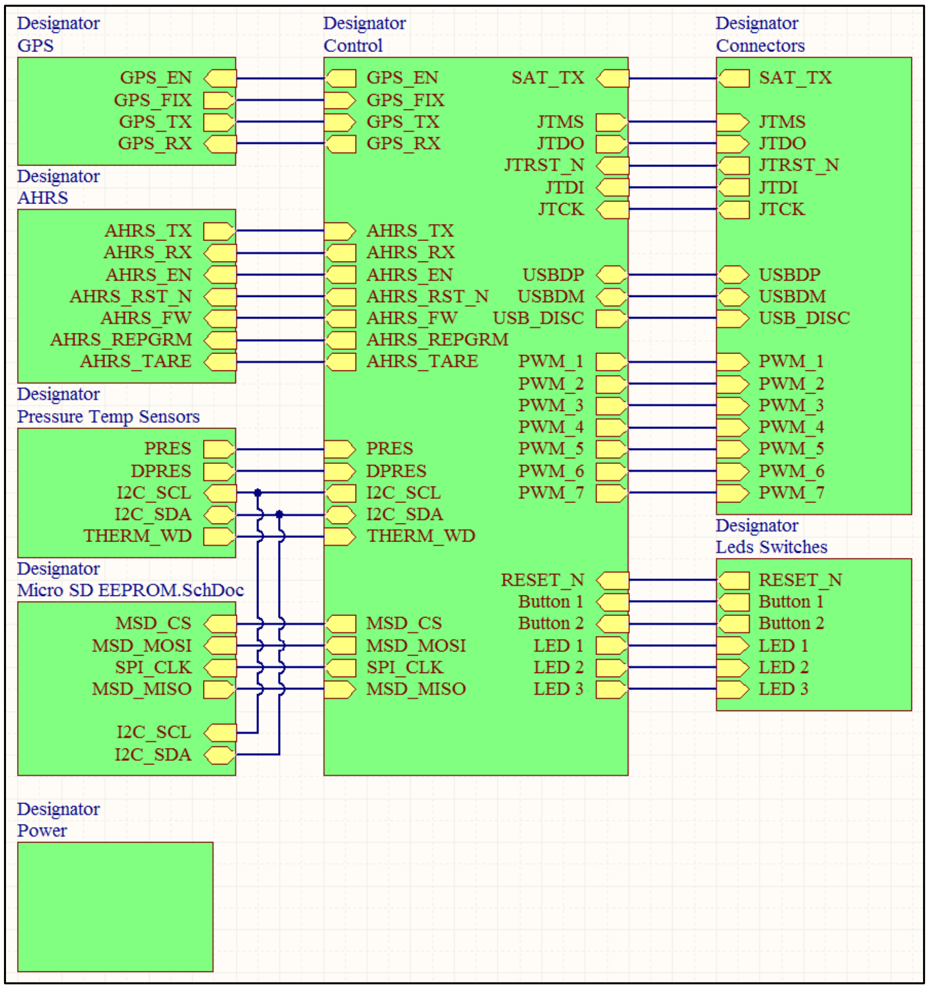


FIGURE 13: TOP SCHEMATIC

6.1.1 Power

From the component selection, the circuit needs two supply voltages, 3.3V and 5V. The only components that require 5V are the pressure sensors (both absolute and differential) and the GPS module. From the start, it was decided the circuit could be powered via 5V USB, or through an external source. The speed controller used on the glider has an integrated BEC (battery eliminator circuit) that converts the ~12V from the battery to ~6V.

To simplify the design, a low dropout regulator and a linear regulator were used for the both the 5V and 3V. To select whether to use the 5V from the LDO (Low-Dropout) regulator or the USB, a 3 pin jumper is used, giving the user the option to power the board either from an external BEC or via USB.

To choose the two voltage regulators, we have to determine the power consumption of the different components. A rough estimate with the highest power components was calculated:

Component	Voltage	Current	Power
MCU	3.3V	30mA	99mW
VN-100	3.3V	65mA	214.5mW
GPS	5V	40mA	200mW
Absolute Pressure	5V	7mA	35mW
Differential Pressure	5V	10mA	50mW

A rough estimate of the total current provided by the 3.3V regulator is about 95mA. For the 5V regulator, it would need to provide at least 57mA plus the current of the 3.3V regulator, bringing it to a total of 152mA. Now these are just rough estimates, and exclude all the extra digital circuits, such as the amplifiers, temperature sensors, etc. In addition, this board will be a prototype, and additional modules or components (such as a Zigbee RF module) might be added in the future. With this in mind, an 800mA 5V LDO regulator, and a 500mA 3.3V linear regulator were selected. They were selected based on their max current output, that they had fixed voltage outputs (3.3V and 5V), and their low prices.

The regulators selected were: for 3.3V - MCP1825 @ $0.56, for 5V - LP3871 @ $3.36

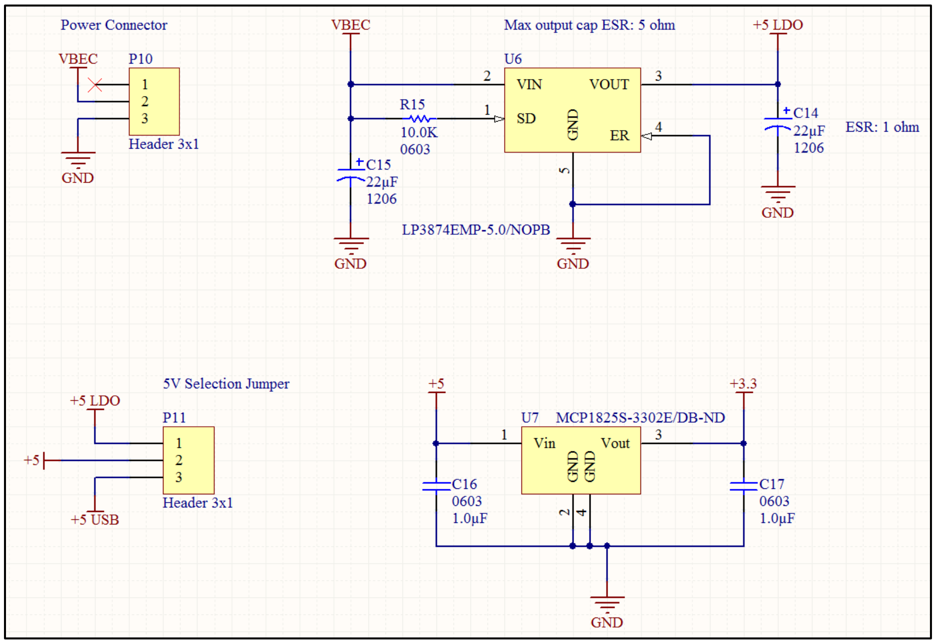


FIGURE 14: POWER BLOCK

6.1.2 GPS

The Mediatek 3329 GPS module schematic is as per the data sheet. The GPS module has both a USB output and a serial output (UART). The serial output was used, and the USB pins were left floating. Additionally, the GPS has 3D fix output pin, which indicates the status of the GPS fix – 0V if 3D fix, flashing if no fix. This output is connected to an LED in order to indicate to the user if the GPS is ready or not.

The GPS is powered at 5V, with a ferrite bead and capacitors to cut off high frequency noise to and from the GPS module.

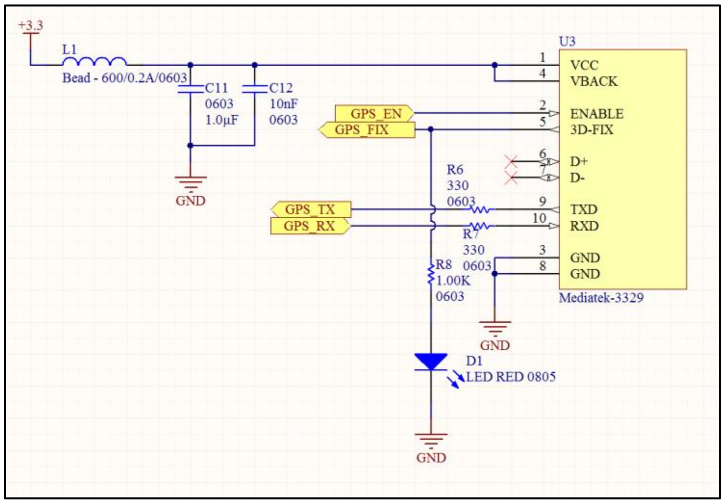


FIGURE 15: GPS BLOCK

6.1.3 AHRS

The VN-100 AHRS circuit was drawn as per the data sheet. The AHRS module has two possible communications modes, UART and SPI, as well as separate signal command lines. A choice was made to use the UART bus to communicate with the AHRS module instead of SPI, for the simple reason that this would allow the AHRS module to continuously send data without the need for the microcontroller to initiate the transfers. The SPI pins were left floating, and the Tx and Rx signals are connected directly to the microcontroller. The separate digital signals (tare, reprogram, firmware, reset, enable) are connected to standard GPIO pins of the microcontroller. As a standard, a 1uF capacitor is added for power surges, and a 10nF decoupling capacitor to filter noise.

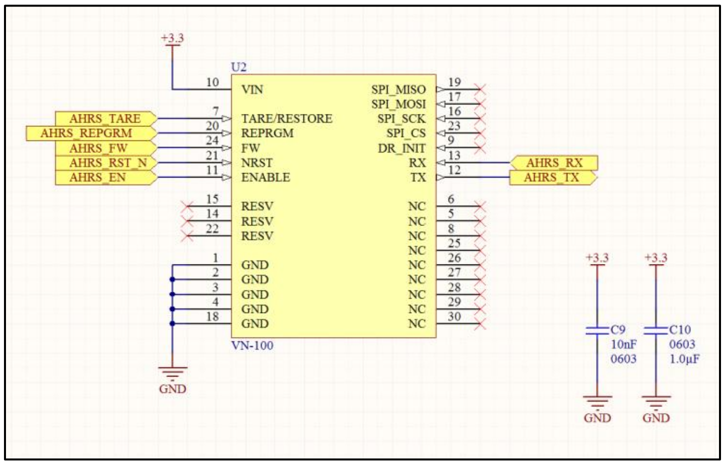


FIGURE 16: AHRS BLOCK

6.1.4 Pressure & Temperature Sensors

One of the problems with the pressure sensors was that they operate at 5V, and their analog outputs can vary between 0 and 5V. Unfortunately, the microcontrollers ADC input can only measure values up to 3.3V. Additionally, during operation, a lot of noise would be generated (air turbulence, etc.), and a low frequency low pass filter had to be added. An operational amplifier configured as a voltage follower is used as to not pull any current from the sensor as it might induce errors. A simple first order RC filter with voltage divider is connected to the output of the first follower operational amplifier. The second voltage follower serves as a high impedance output for the RC filter, as to not be disturbed by the microcontroller's ADC input impedance. The ADC input impedance can vary depending on its configuration, with an average value of about 50 [KΩ].

The transfer function of the RC filter was calculated as (R1 = R17/19 and R2 = R18/20):

$$G(s) = \frac{R_2}{R_1 + R_2} \cdot \frac{1}{\frac{R_1 R_2}{R_1 + R_2} C \cdot s + 1}$$

With the chosen values of R2 = 10K, R1 = 4.7K, and C =1uF, we obtain a static gain of 0.68 and a cutoff frequency of 50Hz. Both the static pressure sensor and the differential pressure sensor have the same analog filters at their outputs. (Although after mounting the board, it was noticed that the differential pressure voltage divider wasn't necessary and R18 was removed)

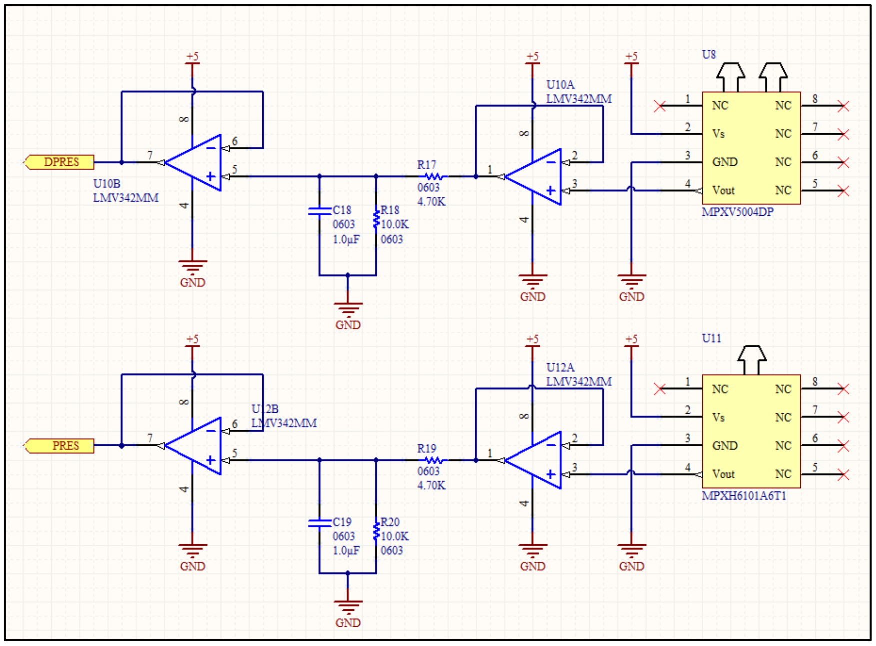


FIGURE 17: PRESSURE SENSORS

Considering the different types of environment the glider might encounter, and how different parameters vary with the temperature, such as sensor errors, air density, etc., a digital temperature sensor was added.

The temperature sensor selected is the STLM75DS2F, as it's easy to implement, and is digital. The temperature sensor communicates to the microcontroller via I^2C, and has a separate watchdog signal that can be used as an alarm (external interrupt) if the temperature crosses a preprogrammed threshold.

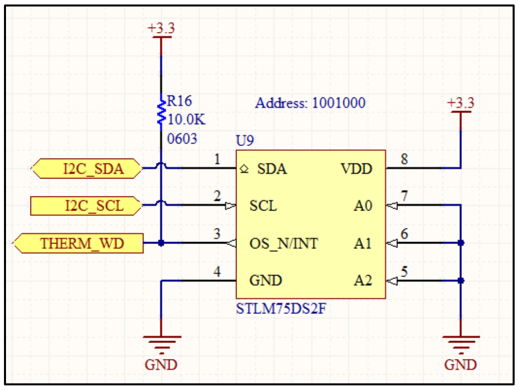


FIGURE 18: TEMPERATURE SENSOR

6.1.5 Micro SD & EEPROM

One important function the autopilot has to have is data logging. Micro SD cards have two communication modes: SD and SPI. The microcontroller has an SD input to communicate with SD cards, but in order to save pins the connector is configured in SPI mode. A standard micro SD connector from Hirose Electric Co Ltd was used. The SPI signals are connected directly to the microcontroller SPI pins, and the extra pins used during SD mode were left floating. A 1uF capacitor was added between the 3.3V and GND near the connector.

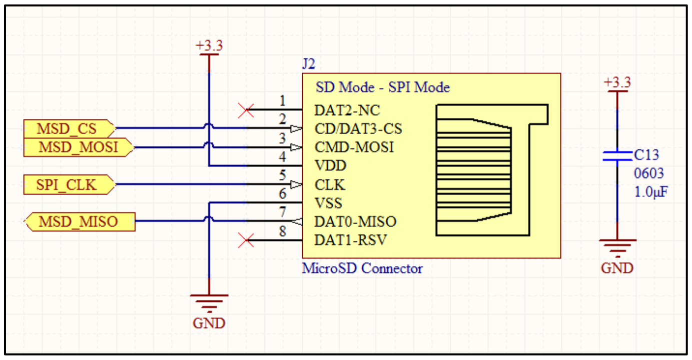


FIGURE 19: MICRO SD CONNECTOR

An EEPROM is always useful to easily store and access small amounts of information, such as configuration settings, etc. the STM32 microcontroller doesn't have an integrated EEPROM, and since an I^2C bus was already used for the temperature sensor, it didn't cost any extra pins to add a small I^2C EEPROM. A small (SOT-23-5) 2K x 8 EEPROM from Microchip Technology was selected for its small size and low price.

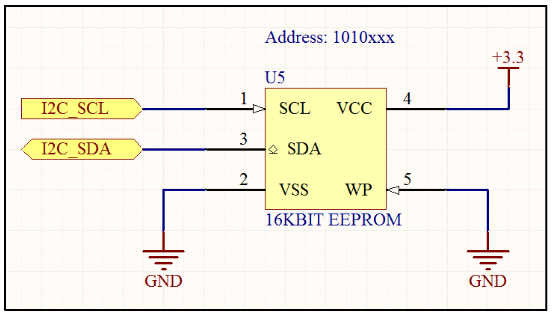


FIGURE 20: 16KBIT EEPROM

6.1.6 Connectors

The autopilot PCB has 4 groups or type of connectors:

- USB connector
- Spektrum Satellite receiver connector
- Servo Connectors
- JTAG header

The cheapest Micro USB connector available was used, coupled with a USB line termination IC. The line termination has an internal 1.5k resistor required for USB discovery, as well as zenner diodes for ESD (Electrostatic discharge) protection. It also has a USB Disconnect signal that either connects or disconnects the USB. This signal is controlled by the microcontroller, and is connected to a led to notify the user of the USB communication status. The USB line filter is the USBUF02W6 from STMicroelectronics.

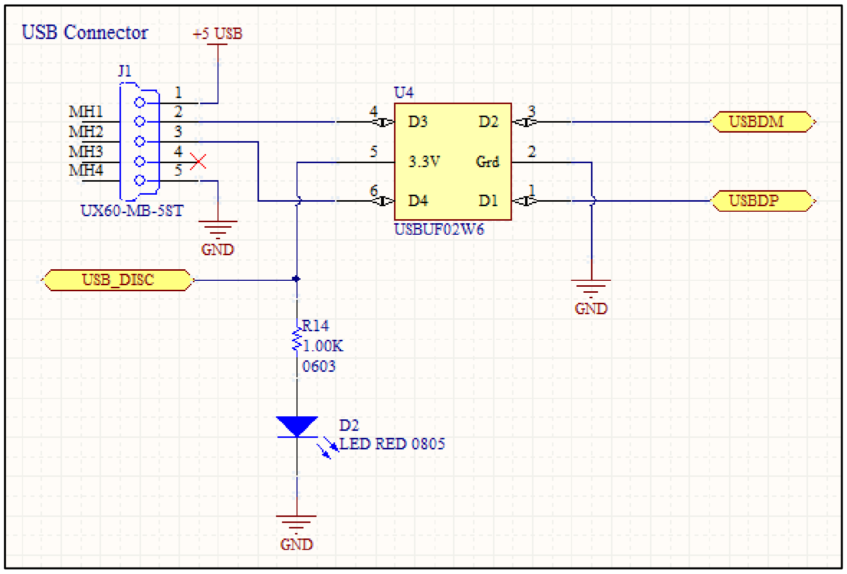


FIGURE 21: USB CONNECTOR + FILTER

The Spektrum satellite receiver uses a specialized connector. The exact model of the male connector that Spektrum use for their radio receivers is the S3B-ZR, so naturally this was the connector used. After probing with an oscilloscope the 3 pins of the reciever, It was determined that the reciver has a signal serial output, and a 3.3V (plus GND) supply voltage input. The serial output is directly connected to the microcontroller.

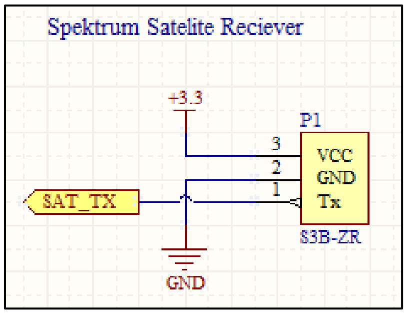


FIGURE 22: SPEKTRUM CONNECTOR

To command the aircraft, the autopilot uses standard RC servos that are commanded with PWM signals. Standard 3 pin 0.1" male headers are used. In order to operate complex aircraft such as a glider, the autopilot has to have sufficient output channels. The Cularis glider needs 6 channels (+1 if motor is mounted) so seven connectors are used. The servos are powered by an external battery eliminator circuit on the glider.

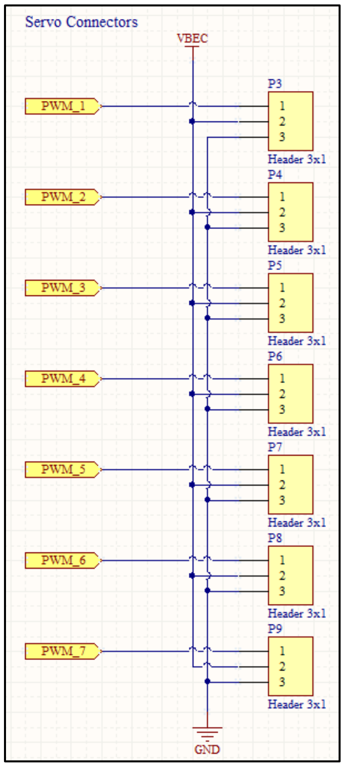


FIGURE 23: SERVO CONNECTORS

Programming and debugging of the STM32 microcontroller is done via JTAG. There is no standard JTAG connector, so a simple 1x8 0.1" male header was used.

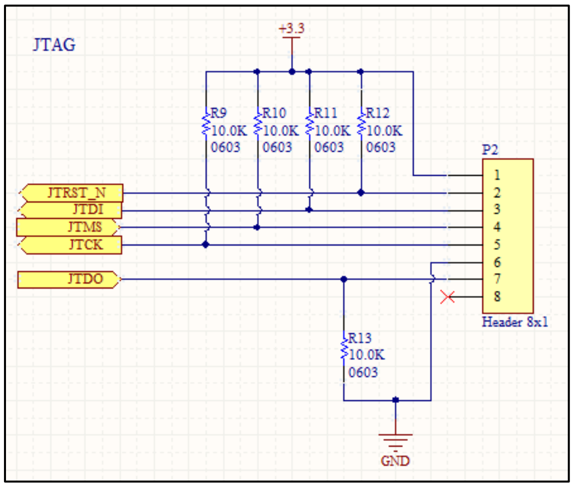


FIGURE 24: JTAG HEADER

6.1.7 LEDs AND SWITCHES

Three LEDs were added without any specific purpose in mind. During the software development they will be used for debugging purposes, and for final release they may be used to indicate the different configurations of the autopilot, whether it's running, calibrating, etc.

The LEDs used are low power 2mA red LEDs with a 0805 footprint. A 1K resistor limits the current to ~2mA. The LEDs are powered by the GPIO outputs of the microcontroller, and thus, are connected directly to the microcontroller.

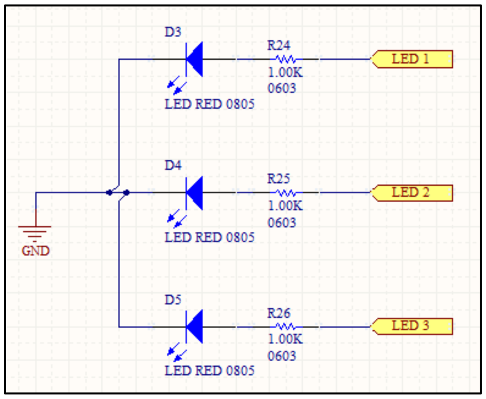


FIGURE 25: USER LEDS

In the field, the user needs to be able to initiate certain functions without the need of a computer. Two push buttons are used for this exact purpose. Their functions are not yet determined, but they will be used for debugging purposes (just like the LEDs), and for the final release, they can be used to start/stop the data logger, start/stop the autopilot, etc. The output of the push button is at 3.3V at rest, and is pulled down when the user pushes it. To filter out rebounds, a 0.1uF capacitor is added in parallel to the button.

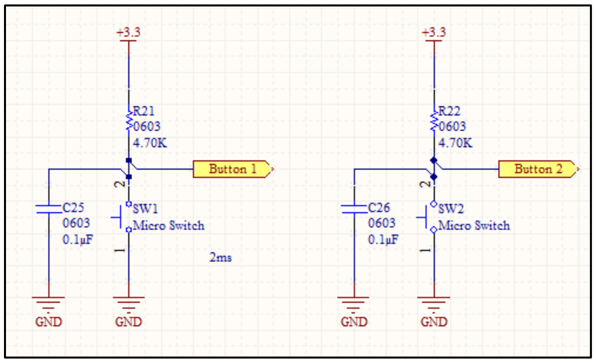


FIGURE 26: USER BUTTONS

Additionally, a third button is used to reset the microcontroller. A special reset IC that filters out rebounds was used. The output signal routes to the microcontroller's reset input. The schematic is as per the datasheet. The reset circuit is the STM6315SDW13F.

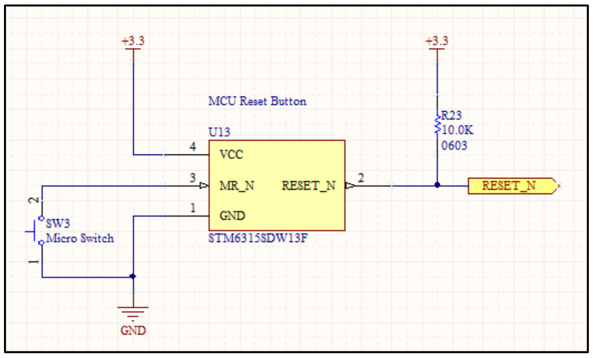


FIGURE 27: RESET BUTTON + CIRCUIT

6.1.8 Control

The control block is the heart of the PCB, with the STM32 microcontroller. The Microcontroller used is STM32F103RCT6. It is packed in a 64 pin LQFP package. It's based on the ARM Cortex-M3 architecture running at 72 MHz and has many sophisticated peripherals that reply to all the autopilot's needs. The variant used has 256Kb of Flash giving ample room for all the autopilot code, and 48K x 8 RAM. The microcontroller relies on an external 8 MHz resonator that is then multiplied internally by a PLL to 72 MHz. All signal outputs of the different blocks explained previously are directly connected to the microcontroller. Of course, the pins of the microcontroller each have specifics functions, and much time was spent on the pin assignments. Pins assignments were a back and forth process between the schematic and PCB layout. The microcontroller has a lot of flexibility on the pins functions, allowing us to swap nets to uncross them on the physical layout. Not all pins were used and all unused pins were connected to a small pad giving them easy access in case they may be needed in the future.

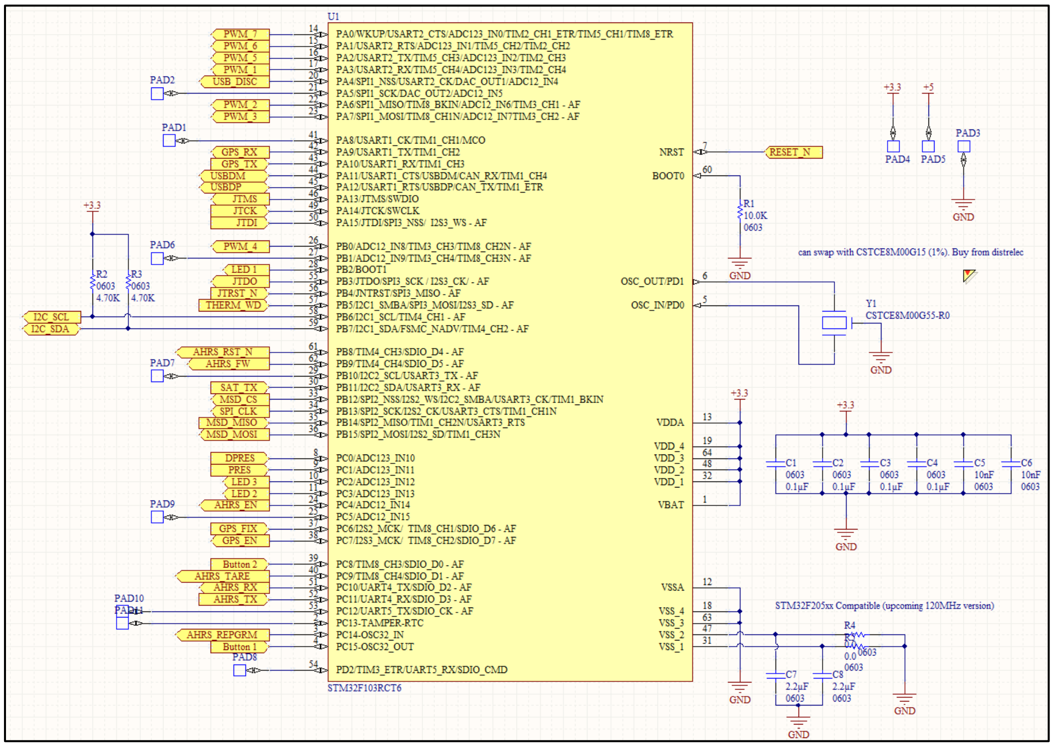


FIGURE 28: CONTROL

6.2 PCB Design

6.2.1 Component Placement

After the schematic entry was completed, double and triple checked, then validated, the physical layout of the PCB could start. The most important aspect of any PCB design is the component placement. The components must be placed as to:

- Be as compact as possible
- Limit the number of crossed nets, to facilitate the routing afterwards
- Separate analog sensors/components from digital and/or noise producing components.

A high priority was placed on the PCB size. The space inside the glider being limited, the PCB had to be as compact as possible, which is why the PCB is double mounted (components on both sides of the board: Top and bottom).

The top side contains all the connectors: seven radio outputs, JTAG Header, USB, and Spektrum. In addition, all the large modules and sensors were put on top such as the GPS, AHRS, and both pressure sensors. The user LEDs and button are also on top for obvious ergonomic reasons. After deciding which components were on top, it was a puzzle to fit them in a highly compact and efficient way. Effort was made to separate the analog (pressure sensors + filter) and the digital components. The pressure sensors were placed at the top right of the board, with the AHRS module, and servo PWM outputs on the bottom left of the board. After many hours of tinkering with different variations, here is the final top placement:

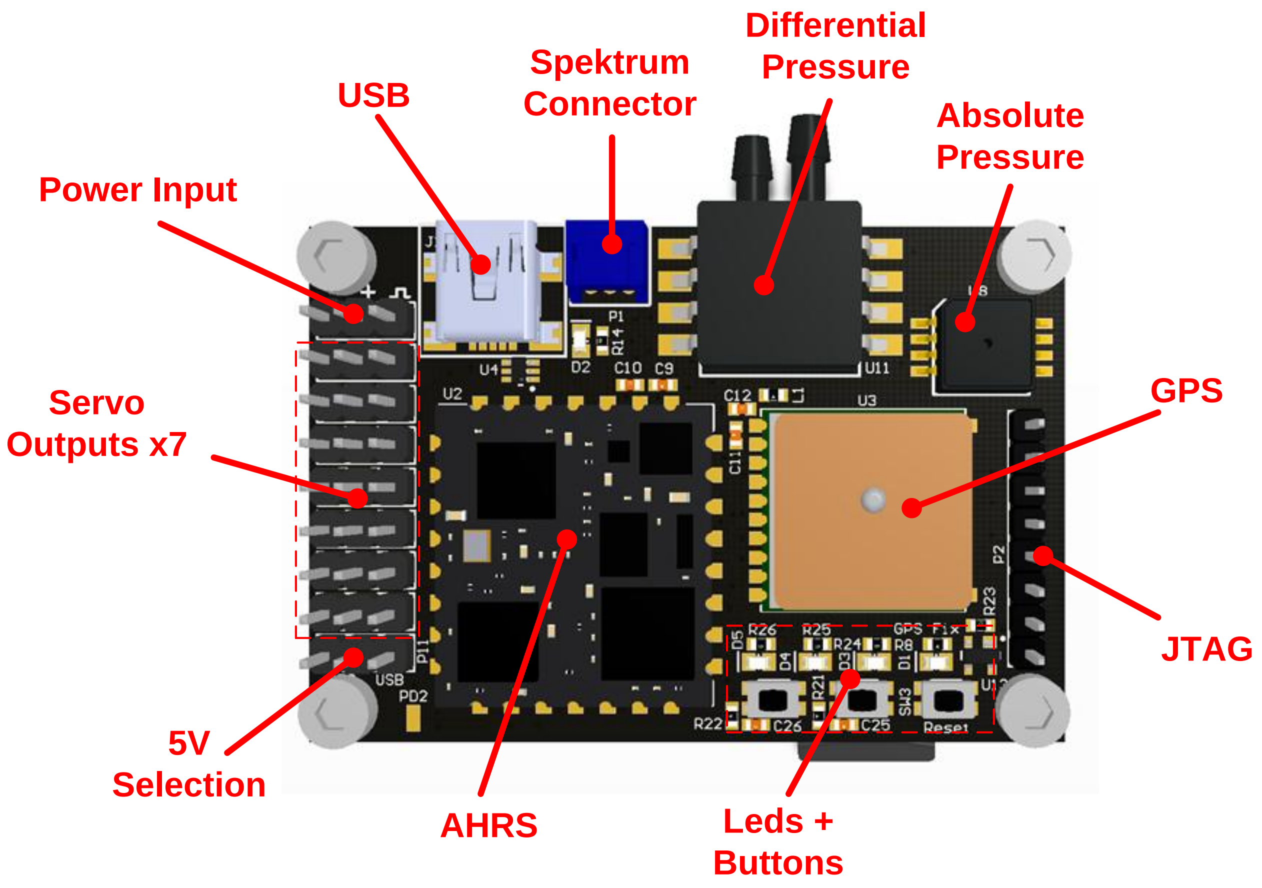


FIGURE 29: AXCG TOP PCB PLACEMENT

With all the sensors, modules, and connectors on top, only the microcontroller, Micro SD connector, power supply, and discrete components were left to place. The microcontroller was placed at the center, which was the most optimum with nets going in all directions. The analog filters of the pressure sensors were placed directly underneath the pressure sensors in order to reduce the trace lengths, increasing noise immunity. The SD card connector was then placed where ever there was room left. The decoupling capacitors are placed as close as possible to the integrated circuits. Here is the final bottom placement:

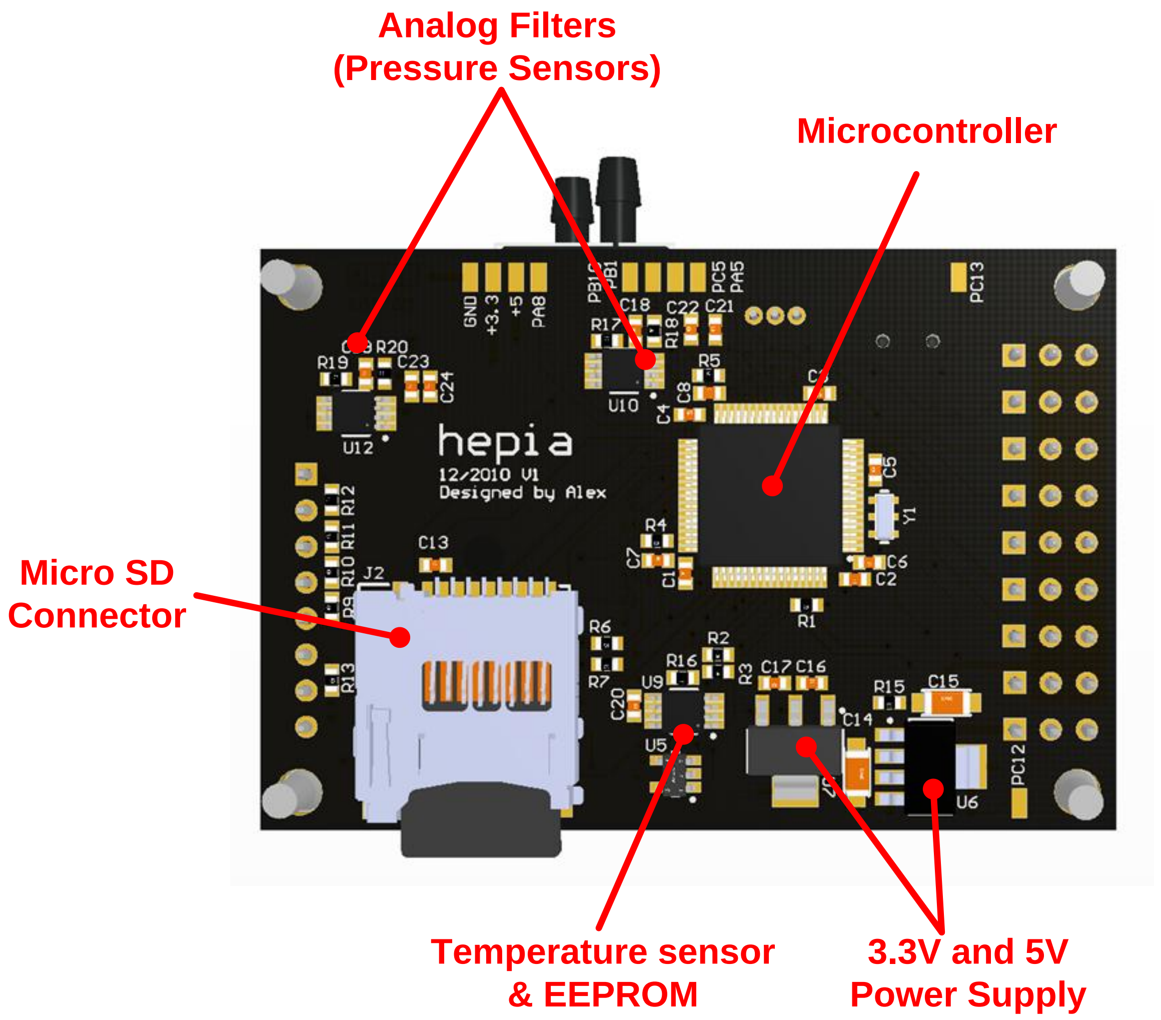


FIGURE 30: AXCG BOTTOM PCB PLACEMENT

The final size of the PCB is 41x60mm. Four screw passage holes were positioned on all four corners for attaching the board to the glider in the final application. The holes were calculated for M3 screws, giving them a diameter of 3.2mm.

6.2.2 Routing

Only after validating the component placement does the routing actually start. The PCB was routed using only 4 layers, with two power planes, and two routing layers. The two internal layers were used as the power planes, with one layer with a solid ground (GND) copper plane, and the other a large 3.3V copper plane, with large 5V traces cutting through going to specific 5V components.

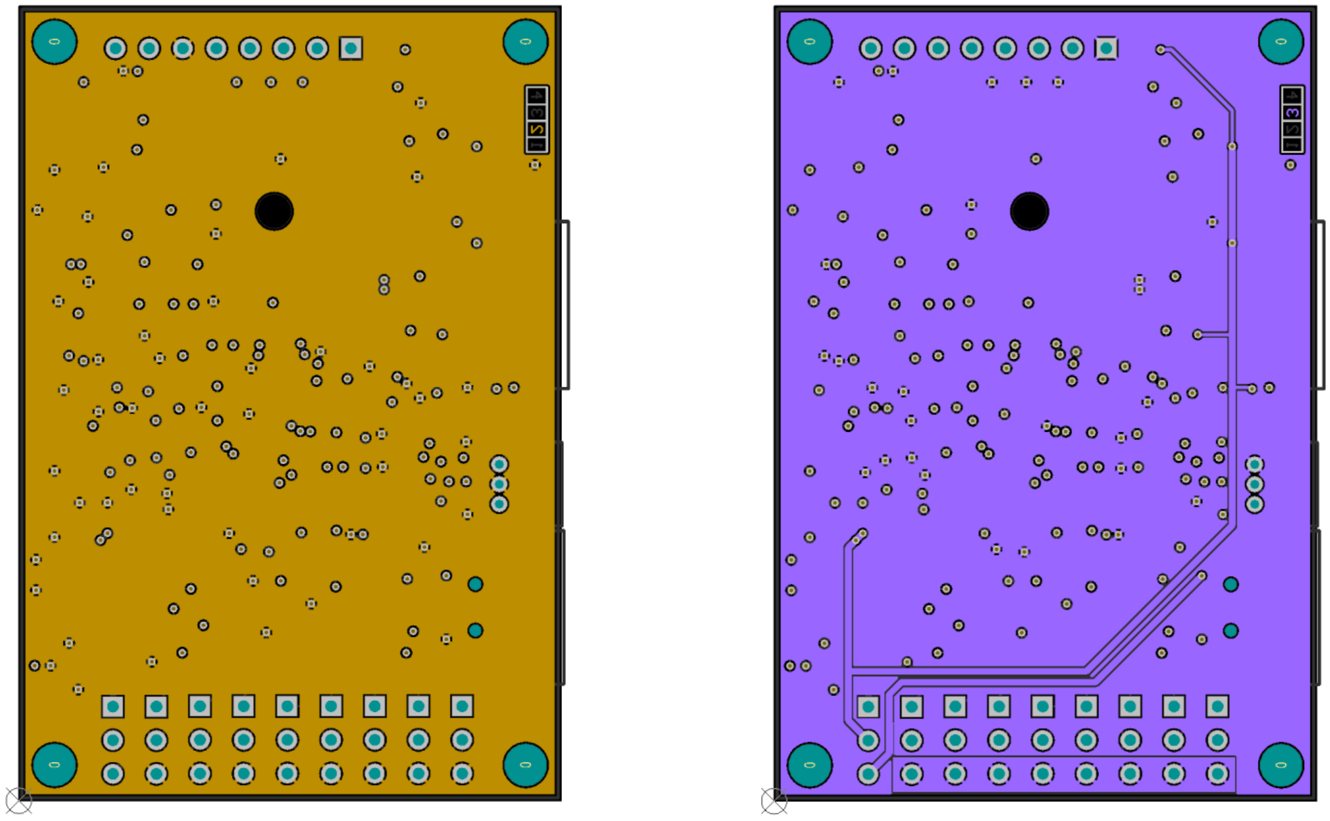


FIGURE 31: LAYER 2 - GND (LEFT) & LAYER 3 - POWER (RIGHT)

The GPS Data sheet states that no conductive traces or copper pours may pass underneath the module at a specific location 3mm in diameter. It's that 3mm hole that can be seen on both copper pours.

Once the power planes were completed, all individual signals were routed by hand. Nets were routed from the most critical, such as analog signals, communications bus (USB, USART, SPI, etc.) to less critical lines such as the user LEDs and buttons. This ensured that all critical nets were short, and used as little vias as possible, while the less important nets had to zigzag through more layers.

Standard signal traces are all 150um wide, with power traces going up to 500um wide. As with the power plane copper pours, there's a no trace zone under the GPS keep out zone. Vias are all 0.3mm in diameter.

Here are the final top and bottom layers of the PCB.

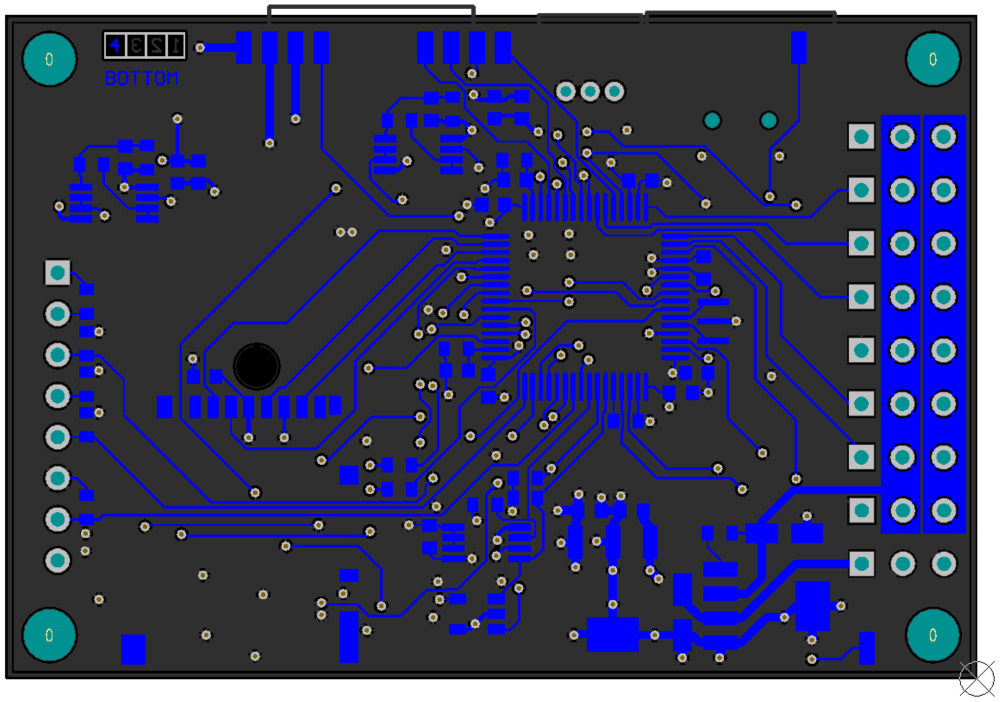


FIGURE 32: PCB TOP LAYER

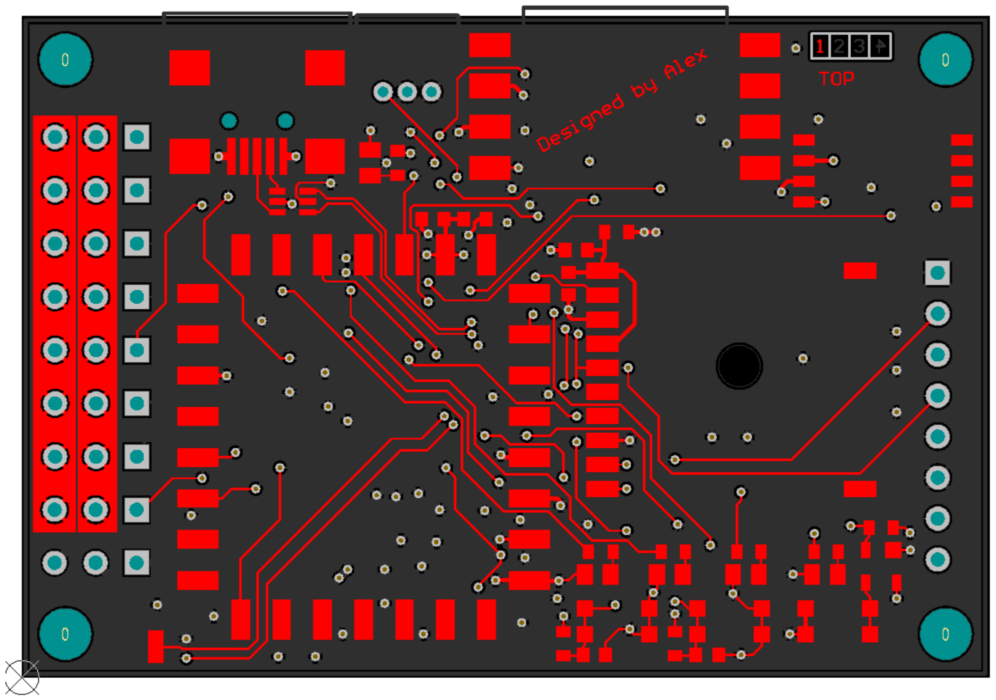


FIGURE 33: PCB BOTTOM LAYER

Top and bottom Layer with lithography (no ground or power planes)

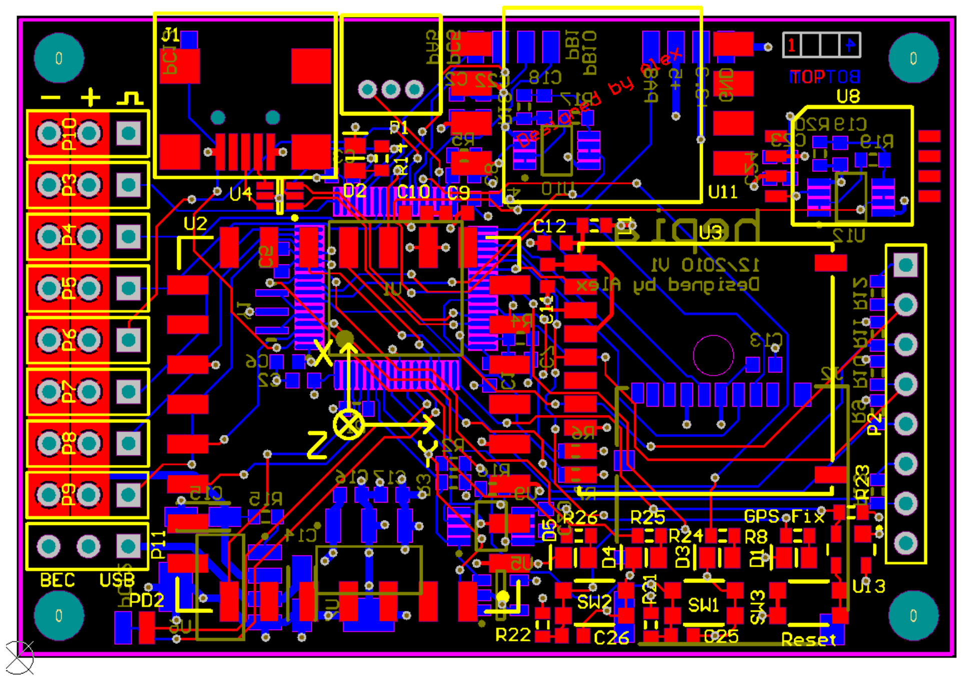


FIGURE 34: PCB TOP AND BOTTOM

See appendix for the entire schematic and all PCB layers.

6.2.3 MOUNTING THE PCB

All components were purchased from digikey, except for the GPS and AHRS. The total unitary price comes in at around \$590, with the AHRS making up most of the budget. For a complete bill of materials see appendix.

The first components mounted were both power supplies. After testing the voltage regulators, it was quickly noticed that the 5V regulator wasn't working while the 3.3V was working as expected. It was only after reviewing the datasheet of the 5V regulator that the problem was discovered; the 5V regulator datasheet has two variants, and the wrong component was ordered from digikey (LM3874 instead of LM3871). After ordering and swapping for the correct voltage regulator, the board power supplies were validated.

Since all components only interact with the microcontroller, there is very little risk that one component could induce failure on the others, unless it short circuits the power supply. For this reason, it was decided to mount everything else except for the AHRS (it is the most expensive component, and was the last one mounted). Once everything, except for the AHRS, was mounted, the power supplies and current consumption were checked again making sure that they still operated as expected and no short circuits were present. Only after validating the entire board, was the AHRS mounted.

A second error was discovered regarding the push buttons. Unfortunately, an error was made when drawing the foot print of the push button. The order of the pins were crossed which meant that the push button was as if it was always being pushed. This was discovered because the microcontroller wouldn't boot. It turned out the reset signal was always active, because the reset button was always activated. To resolve this issue, small push buttons, with only two pads, were used to replace the ones bought. The footprints weren't exact matches, but were close enough to create a solder bridge. The end result is still clean regardless of the wrong footprint. After replacing the buttons the microcontroller could boot, and the programming could begin.

7. EMBEDDED SOFTWARE DESIGN

The IDE of choice was Atollic Lite. It's a commercial variation of the Eclipse environment, and is made to work with STM32 microcontrollers. When creating a new project, it generates all the required startup files and low level functions to initialize the clock and buses. In addition, STMicroelectronics supply a peripheral library with low level functions to access and control all peripherals, as well as demo code explaining how to use said functions.

To simplify the implementation of low level functions, the ST peripheral library was used opposed to writing directly to the registered as done in earlier projects. This translated into slightly larger code, but much easier to read and debug.

The very first function coded was to initialize all the microcontroller pins to their corresponding type (Analog, Digital input / output, USART, SPI, etc.) and enabling all peripherals that will be used.

After the general initialization, low level driver functions were coded to communicated with and control all the sensors and components on the PCB, as well as implement USB as a virtual com port to communicate to and from a computer.

The code is structured hierarchically with low level driver files that control and read the sensors, control servo outputs, etc. These functions use the standard peripheral library published by ST. On top of this layer, are the API's such as the FAT file system and the RTOS. The FAT API relies on low level SPI functions to access the hardware. The top layer is composed of tasks that are created using the FreeRTOS API which in turn use all low driver functions and API's.

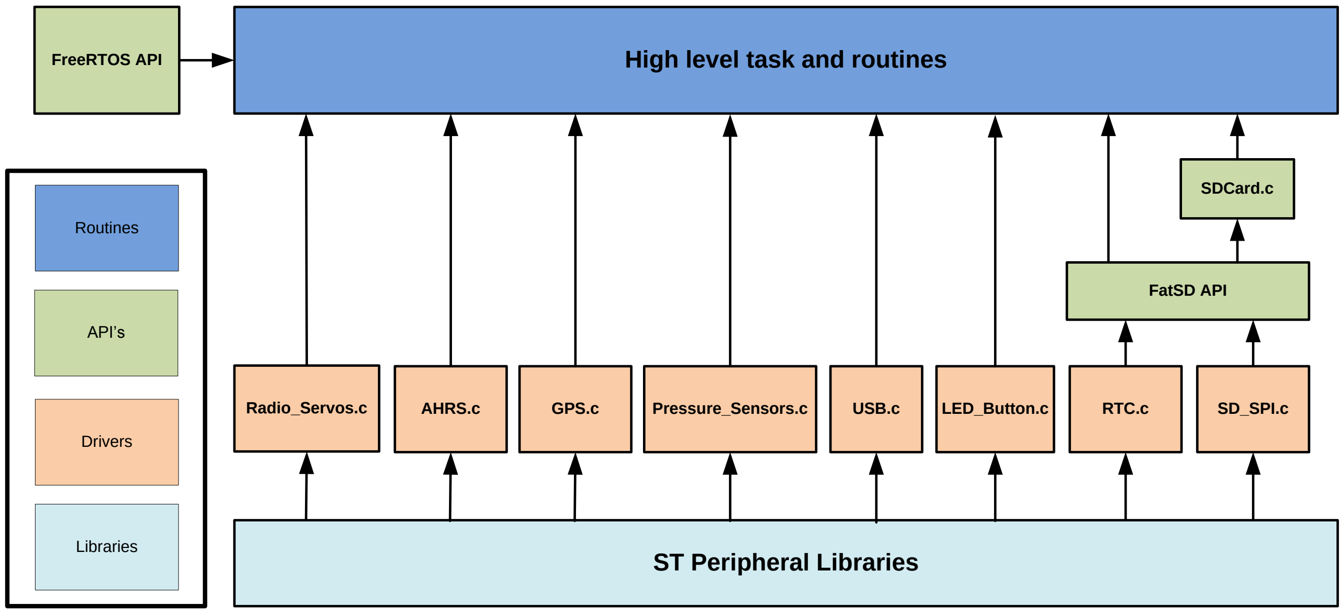


FIGURE 35: CODE STRUCTURE

7.1 AHRS

The AHRS operates using the USART peripheral with a user programmed serial baud rate. The AHRS outputs data at a user programmed refresh rate up to 200 Hz. The data is sent as ASCII strings, with a format that greatly resembles the NMEA protocol.

Here is an example of a data string with the Yaw Pitch and Roll angles:

$VNYPR,-027.33,-005.33,+002.63*65

The string starts with the '$' sign, followed by a 5 letter code describing the type of the data, followed by the values, separated by a comma. At the end of the string is a '*' followed by a two byte checksum. The string is terminated with a new line character '\n'.

There are two methods to read asynchronous data. The first uses an interrupt routine that is called every time a byte is received. The byte is then decoded and/or stored and parsed as soon as a '\n' character is received. The problem with this method is that that processor is continuously interrupted when it can be doing other operations. The second method consists of use a direct memory access (or DMA) peripheral. Instead of interrupting the processor every time a byte is received, the USART peripheral calls the DMA peripheral, which in turn reads the received byte and stores it in memory. The memory can then be read by the processor from time to time to read and decode (or parse) the strings.

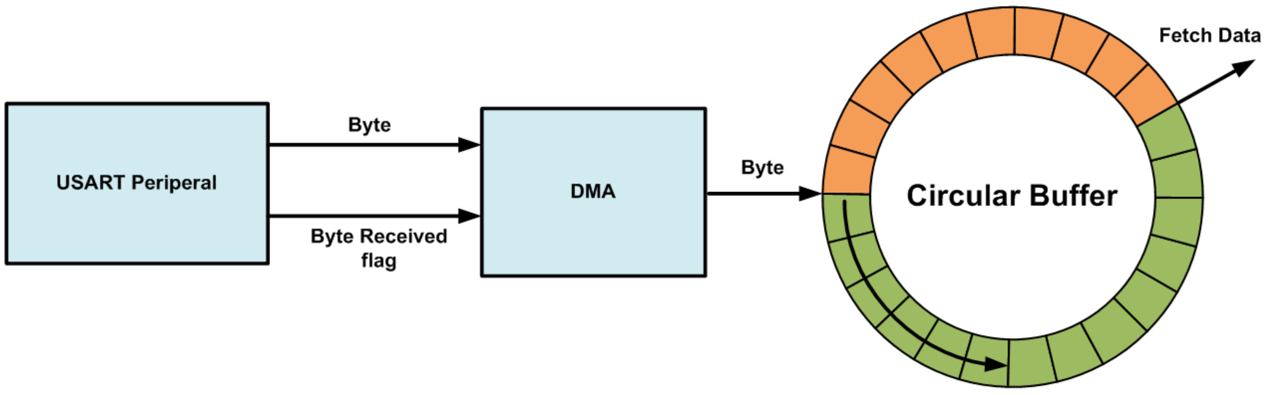


FIGURE 36: DMA - CIRCULAR BUFFER

The AHRS specific functions are:

- AHRS_USART_Initialize
- AHRS_DMA_Initialize
- AHRS_Read_Buffer
- AHRS_Message_Parse
- AHRS_Send_Message

The AHRS USART initialize functions configure the USART in receive and transmit mode, with specific serial baud rate (function input), 8 bits no parity.

The AHRS DMA initialize functions configures the DMA to read all received data from the USART peripheral and store it in a character array 256 long, in circular buffer mode. In circular buffer mode, once the DMA reaches the end of the allocated memory, it starts over.

The AHRS read buffer function is called periodically and reads all the new characters that were stored in the buffer. The function looks for the special characters that indicate the start and end of the string, which are the ‘$’ and the ‘\n’ characters. Once the function has found an entire string, it is then sent to the AHRS Message Parse function that decodes the string to extract the data.

The AHRS Message Parse function decodes entire strings sent from the AHRS. The first part of function identifies what kind of data it’s working with depending on the 5 character code in the beginning (remember VNYPR). After identifying the data type, it goes through the string converting the ASCII decimal value into a binary value, and stores the results in global variables. The converting from ASCII to decimal is done by subtracting 48 to the character and multiplying by a multiple of 10, depending on its position in the string.

Additionally, to send data to the AHRS, an AHRS Send Message function was created. The send message function works using the DMA. The string to be sent is stored in memory, and the address of the string is given to the DMA which automatically sends byte by byte to the USART peripheral until it reaches the end of the string.

7.2 GPS

Like the AHRS, the GPS module periodically sends data which is received using the USART peripheral. The classic protocol used worldwide for GPS modules is the NMEA protocol. This GPS module, however, can output the data in a condensed binary format, removing the need of finding the relevant data within the string and then decoding/converting the ASCII strings to decimal values. For simplicity reasons, the condensed, binary format was used, which greatly simplified the message parsing.

See appendix for specific binary protocol.

Since AHRS and GPS communication protocol are very similar, their communication functions are too. Initially, the DMA was used to store received bytes in memory which were then read periodically, much like the AHRS. Unfortunately due to the limitations of the DMA, it could not be used for both the USART1 (USART used for GPS), and the SPI2 peripheral which was used for the SD card (described in detail below). Since the GPS data output is at a much low frequency than the SD card, the GPS USART uses an interrupt service routine (ISR) to read the incoming bytes. The ISR does not parse the message, but stores the received byte in a 256 byte circle buffer, imitating the function of the DMA and limiting the time spent in the ISR.

The GPS specific functions are:

- GPS_USART_Initialize
- GPS_Read_Buffer
- GPS_Message_Parse
- GPS_Send_Message

The functions are almost the same as the AHRS, except for the lack of the DMA initialize function. Like the AHRS functions, they work the same way.

The GPS USART initialize functions enable the USART in receive and transmit mode, with specific serial baud rate (function input), 8 bits no parity. It’s essentially the same function used for the AHRS except it configures USART1 instead of UART4.

The GPS Read Buffer function looks for the 4 header bytes that indicate the start of the message, and then counts the remaining 28 bytes left to be read (refer to binary protocol).

The GPS Message Parses function combines the bytes together (using bit shifting and adding), and then stores the data in global variables so that they can be used by other routines.

The GPS Send Message function works the same way as the AHRS Send Message, with the exception of using a different USART peripheral.

7.3 Pressure Sensors

The pressure sensors are connected to the Analog input pins of the microcontroller. The sensors are filtered at about 50Hz, meaning that to respect Shannon's theorem, the sampling frequency needs to be at least 100Hz. The microcontroller possesses 3 internal ADC's with multiple channels. To save a DMA channel, the same ADC was used to convert both analog inputs, by alternating one after the other. The ADC is used in continuous mode, meaning that as soon as it completes a conversion, it starts a new one right after.

The ADC clock and the settling time before each new conversion can be programmed. To reduce the sample frequency, the ADC clock was set as low as possible to 9 MHz, and the settling time was set to the maximum of 239.5 clock cycles. This gives us a conversion time of (equation from STM32 manual):

$$T_{conv} = \frac{239.5 + 12.5}{9 \cdot 10^6} = 31.5\ [us]$$

The sample frequency F_s is then $1/T_{conv}$ which equals 31,746 [Hz]. This sample frequency is very high for our application, but it's as slow as it can go without using timers and ADC conversion triggers.

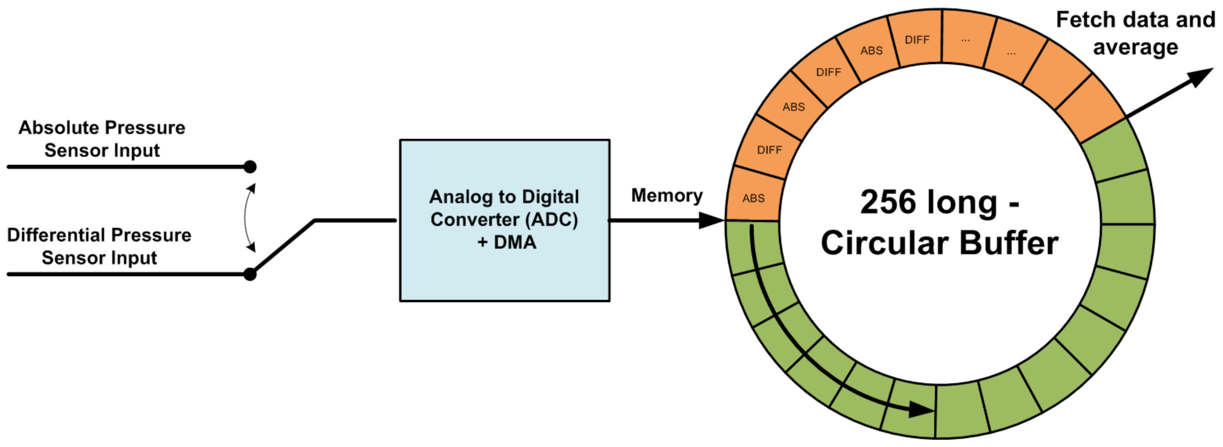

Absolute Pressure Sensor Input
Differential Pressure Sensor Input
Analog to Digital Converter (ADC) + DMA
Memory
ABS
DIFF
ABS
DIFF
ABS
DIFF
...
...
Fetch data and average
256 long - Circular Buffer

FIGURE 37: PRESSURE SENSORS - ADC + DMA

The DMA is used to automatically read and store the ADC conversion results in a 256 long circular buffer. The buffer is then read at a much lower frequency (~50-200 Hz), and the data is converted into millivolts, and averaged. With such a high sample frequency, and a small circle buffer, much of the conversion results are simply not used.

The Pressure Sensors specific functions are:

- Pressure_ADC_Initialize
- Pressure_ADC_DMA_Intialize
- Pressure_Read_Data

The Pressure ADC Initialize function configures the ADC as described above: continuous conversion mode, 2 channel input, 239.5 cycles. Additionally, it goes through an initial calibration to significantly reduce accuracy errors due to internal capacitor bank variations.

The Pressure ADC DMA Initialize functions configure the DMA to read every ADC conversion value and store it in a 256 long circular buffer.

The Pressure Read Data function reads the entire buffer, and averages the 128 values (256 split between two input channels = 128 per channel) and then converts the ADC code to millivolts. It then stores these final results in global variables.

It was later discovered that ARM supply a standard library called CMSIS (Cortex Microcontroller Software Interface Standard) including a DSP library with many functions such as IIR and FIR filters, with fixed and floating point variations. Using Matlab, one could easily calculate the filter coefficients and then simply use the DSP library.

The DSP library was added to the source code and compiled, but not actually used. The functions are all available, so that in the future, a FIR or IIR filter could easily be implemented to filter out high frequency variations (such as turbulence and noise) from the Pitot tube.

7.4 SPEKTRUM RADIO INPUT

The spectrum satellite receiver is connected to a USART input of the microcontroller. The Spektrum radio used has seven channels. After probing with an oscilloscope in the data outputs of the satellite receiver and researching various RC forums, the spectrum protocol was identified. The serial baud rate is 115200, with 8 bits data size, no parity.

The Satellite receiver sends bursts of data at a fixed frequency of 50Hz, with each burst containing 16 bytes. The radio stick positions are coded on 10 bits, and two bytes are used to code each stick position with the extra six bits ignored.

The first two bytes sent at the beginning of each burst transfer are 0x3 and 0x1. These two never change and are used to synchronize the beginning of the transfer. The following 14 bytes are the channel values with 2 bytes per channel.

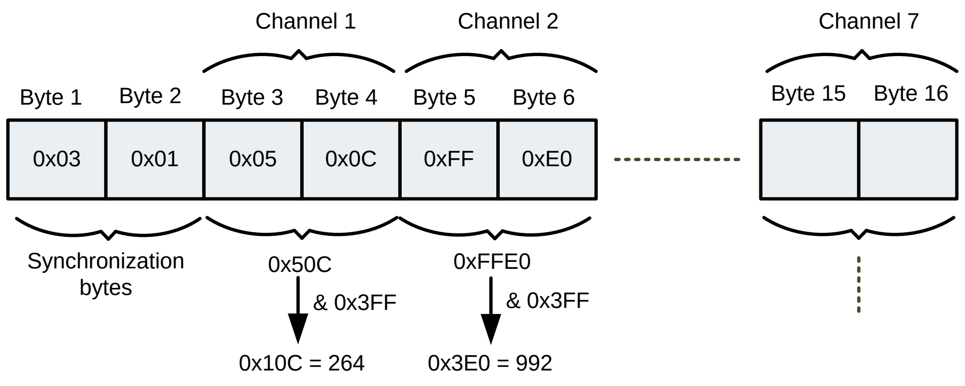


FIGURE 38: SPEKTRUM DATA PROTOCOL

The only Spektrum Radio function is the Spektrum_USART_Initialize function. This function configured the USART in receive mode, 115200 baud rate, 8 bit, no parity, with interrupts enabled.

The decoding of the data input is done in an interrupt service routine. The ISR functions as a simple state machine. As soon both synchronize bytes (0x3 and 0x1) are received, it stores the following bytes in an array. Once all 16 bytes were received, it combines the bytes together (bit shifting, adding and masking) and stores the results in a global array that contains the radio channel inputs.

7.5 Servo Outputs

The servo positions are controlled using a simple PWM signal. The width of the pulse defines the position of the servo angle with 1.5ms being at center, 2ms full right, and 1ms full left. There are no specific requirements for the refresh rate, but some servos can support higher refresh rates whiles other may not. A standard refresh rate is 50 Hz.

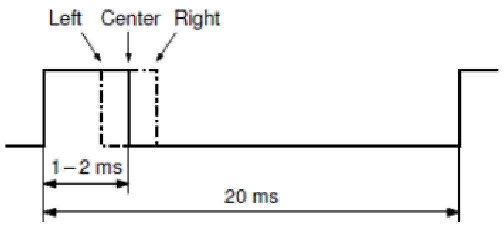


FIGURE 39: SERVO PWM

The Radio channel is codded on 10 bits giving a resolution of 1024, so the base resolution of the PWM outputs was rounded down to 1000 to simplify.

The PWM outputs are generated using Timers. A base clock of 1 MHz was used (T =1 µs), so that a 1000 point resolution would translate in a 1ms variation on the pulse width. When a timer is configured in PWM mode, it needs two parameters. The period which is defined in the Auto Reload Register, and the pulse width which is defined in the Compare/Capture register.

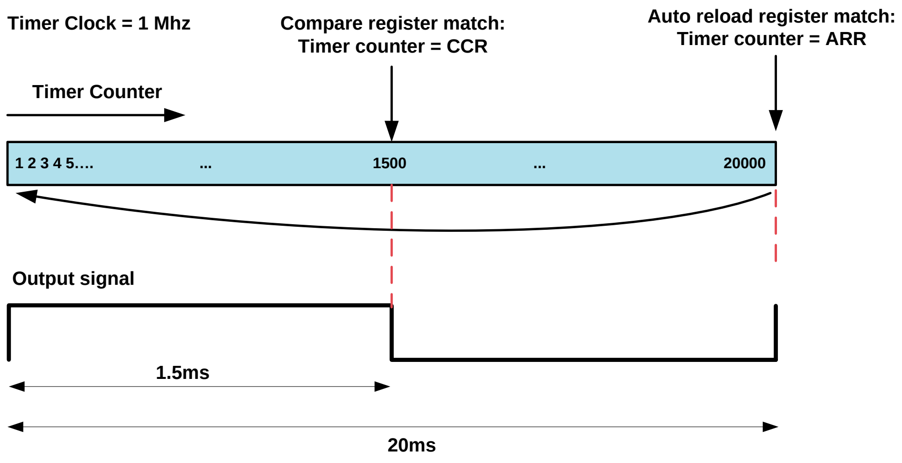

Timer Clock = 1 Mhz
Compare register match: Timer counter = CCR
Auto reload register match: Timer counter = ARR
Timer Counter
1 2 3 4 5....
...
1500
...
20000
Output signal
1.5ms
20ms

FIGURE 40: TIMER - PWM MODE

The timer is basically a counter that increments on every clock rising edge. When it starts, the outputs are set to high. Once the timer counter reaches the value in the Compare/Capture register, the output is brought back to low. Once the Counter reaches the value in the Auto Reload Register, the counter starts over and the output is set to high again. This process continues on indefinitely.

By changing the Auto Reload Register, we can change the refresh rate of the PWM output signal, and by changing the Compare/Capture register we can change the pulse width. The Auto Reload Register value can be calculated for the required refresh rate with:

$$ARR = \frac{Timer\ Clock}{Refresh\ Rate - 1}$$

There are only two functions for the PWM outputs:

- Servo_PWM_Intialize
- Servo_Set_Position

The Servo PWM Initialize function configures the timers (timer3 and 5 were used for all 7 outputs) in PWM mode with a timer prescaler of 71 (to divide the processor clock by 71+1=72, generating a 1 MHz clock), and an auto reload value calculated with the equation given above.

The Servo Set Position function changes the values of the Compare/Capture registers. The CCR register value directly translates into the pulse width in microseconds (if a timer clock of 1 MHz is used).

7.6 SD CARD AND FAT FILE SYSTEM

It seemed necessary from the start that the SD card should work with the FAT file system. This would allow the end user to remove the SD card and plug it in his or her computer to directly read the data logs. Unfortunately, the FAT file system is very complex and would not have been possible to write from scratch in the given time frame. Fortunately, a generic FAT system module for small embedded systems called FatFs was found. However, it still requires the user to write low level functions to send and receive raw data to and from the storage device, as well as provide a Real Time Clock (RTC) for the data and time, which is a requirement for FAT.

The functions that need to be written are

- disk_initialize - Initialize disk drive
- disk_status - Get disk status
- disk_read - Read sector(s)
- disk_write - Write sector(s)
- disk_ioctl - Control device dependent features
- get_fattime - Get current time

Luckily, on the FatFs website, they provide links to various demo applications for a wide range of microcontrollers. One such demo was for a STM32 controller for a micro SD card using SPI, which is exactly what was needed. The demo code was very well written, with all the defines at the top of the file, allowing the possibility to quickly modify the SPI peripheral and pins used, the DMA channel, the Chip select pin, etc.

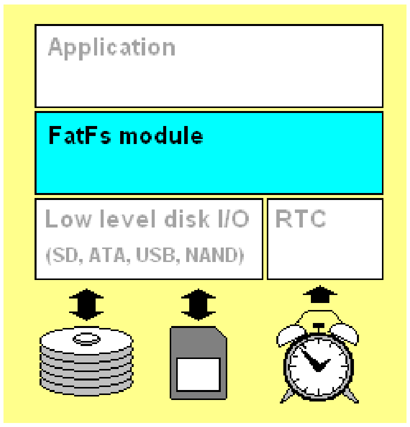


FIGURE 41: FATFS MODULE

After modifying the definitions with the correct values for our application, the FatFs was compiled and tested. A simple test of creating a text file and saving it on the micro SD was implemented, and it worked. After that, writing inside the file was tried, and it worked as well. The autopilot was now capable of creating, reading, and writing files on the micro SD card.

7.7 USB

In the ST peripheral library they provide demo code for a USB virtual comport. The demo application is a USB to USART converter, sending data received from the USART to the USB and vice versa. The demo works using four circular buffers to send and receive data. Data received from the USB is stored in a USB In circular buffer, and data that needs to be sent is stored in a USB out circular buffer. This demo code was directly copied into the AXCG source, and then was modified to work within the application.

Instead reading the USB in circle buffer and copying the data to the USART, the data is parsed based on the protocol defined in chapter 8 below. To send data to the computer, the USB_Send_String function was created that copies a string to the USB out circle buffer that is then automatically sent by the copied demo code.

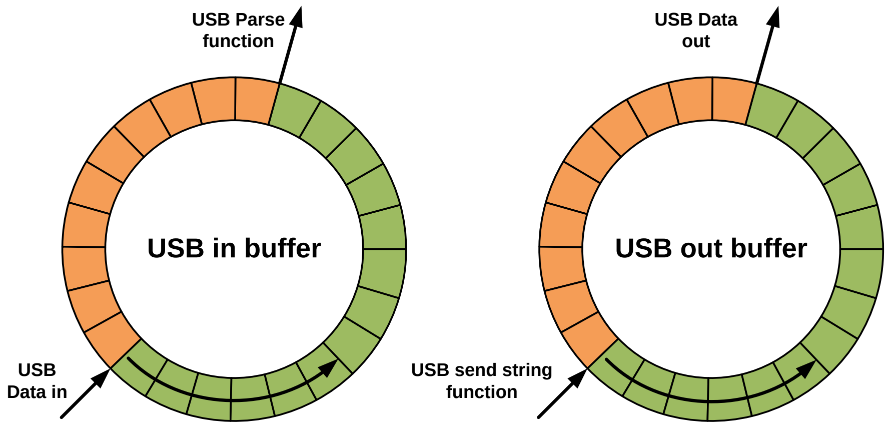


FIGURE 42: USB DATA BUFFERS

The USB communication was a great source of frustration for a long time, even though it was quickly implemented and working, it would periodically miss bytes while sending data from the autopilot to the computer. The source of this problem was not found, and it was not until changing the ST demo code with a newer example that the problem finally disappeared.

7.8 Implementing FreeRTOS Real Time Operating System

After writing all low level functions, search for a real time operating system that was free and easy to use began. The advantage of a RTOS really shines as the code grows in size. It allows to quickly and easily create tasks and routines that all run at different time bases, that work independently or with each other. A quick search for a free to use RTOS returns the FreeRTOS. The FreeRTOS is an open source RTOS with a very large community and support. They have many examples of how to implement the RTOS for a large selection of microcontrollers and compilers, including the STM32 with a GCC compiler. It seemed to be very popular and widely used for amateur to professional applications, and seemed like the ideal choice of an RTOS. They also offer eBooks on how to use FreeRTOS with practical applications and examples as well as a guide on how to implement the RTOS.

The actual core of the FreeRTOS is quiet small, and is contained in four source files: croutine.c, list.c, queue.c, and tasks.c.

In addition to the core, target specific port files need to be added (in our case it was the STM32 ARM Cortex-M3): port.c, heap_2.c, portmacro.h, and FreeRTOSConfig.h

The FreeRTOSConfig.h file is used to define all the parameters (such as the tick frequency) and which API functions are enabled of the RTOS.

After adding all the necessary files, the project was compiled successfully. Thanks to the well written eBook, a couple of test tasks, such as blinking LEDs were quickly implemented. The FreeRTOS was up and running and now any number of tasks with different priorities could be added. The following tasks were then created:

- Read_Parse_Buffers_Task
- Pressure_Data_Task
- Record_Data_Task
- Mission_Control_Task
- Cycle_Register_Read_Task

Read_Parse_Buffers_Tasks runs at 400Hz, reading the GPS, AHRS, and USB buffers. If a complete message was received, it parses it, and the data is then stored in global variables. All this is done using the low level driver functions written previously.

Pressure_Data_Task runs at 200Hz reading the pressure sensor data, using the low level driver functions, and then stores the data in global variables.

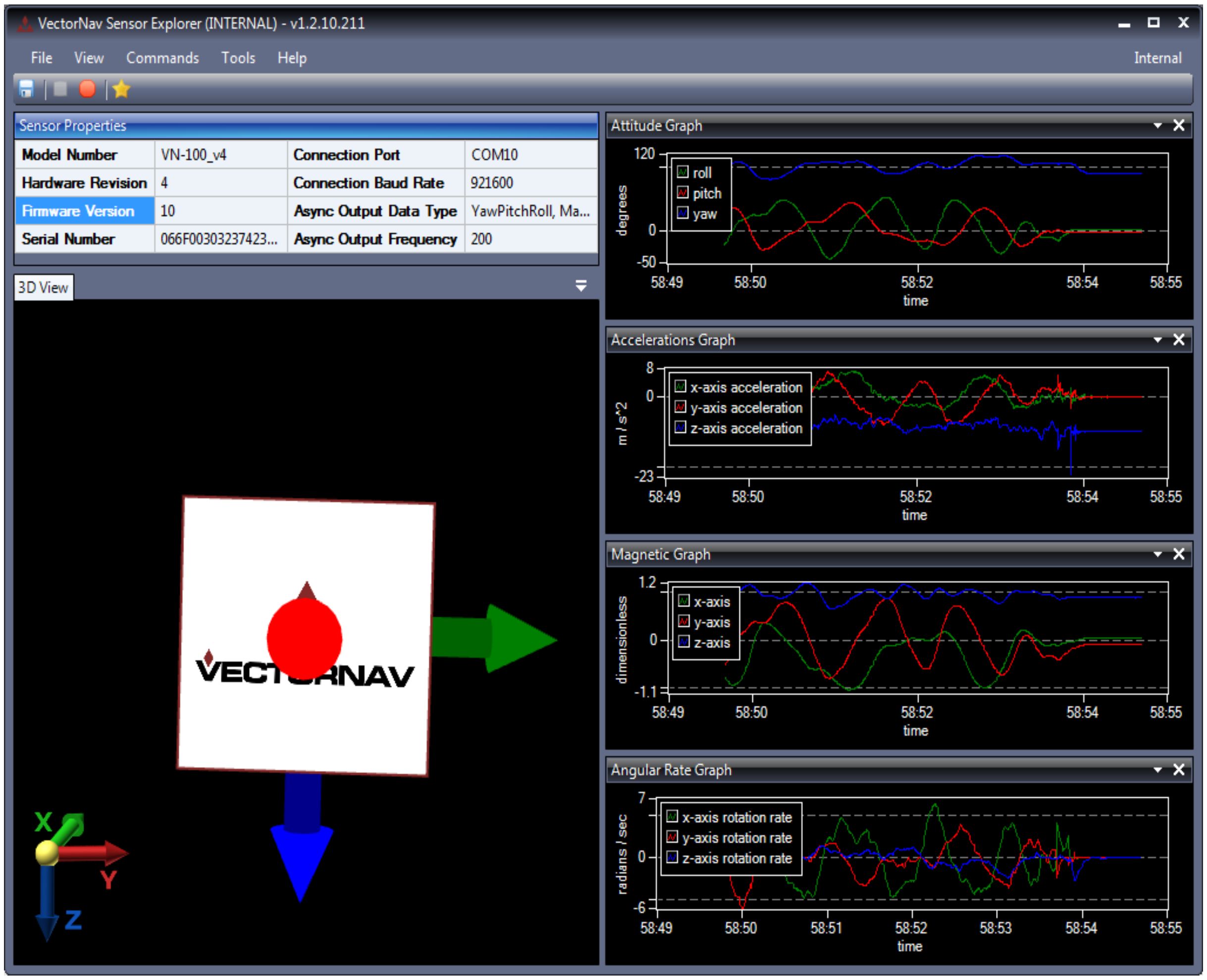
Record_Data_Task runs at a user defined frequency equal the SD Log Frequency register. This task reads the status of a push button, and if it detects a rising edge, it either starts or stops recording. When it starts recording, it creates a new numbered data log file, and starts filling the file with the data present in the global variables. At the same time, a LED is blinked indicating that the autopilot is currently recording. When it stops recording, the file is closed, the LED is turned off.

Mission_Control_Task runs at 100Hz and is in charge of all PC ↔ Autopilot communication. Communication is based the command protocol defined in 8 below. The task parses messages from the USB and either reads or writes to the registers in question.

The Cycle_Register_Read_Tasks is periodically sends the value of the global variables to the computer via USB. The data sent and at what frequency are defined by registers of which the user can access. See 8 for more details on the command protocol and registers.

7.9 Test Routines

7.9.1 Sensor Explorer



Model Number	VN-100_v4	Connection Port	COM10
Hardware Revision	4	Connection Baud Rate	921600
Firmware Version	10	Async Output Data Type	YawPitchRoll, Ma...
Serial Number	066F00303237423...	Async Output Frequency	200

FIGURE 43: VECTORNAV SENSOR EXPLORER

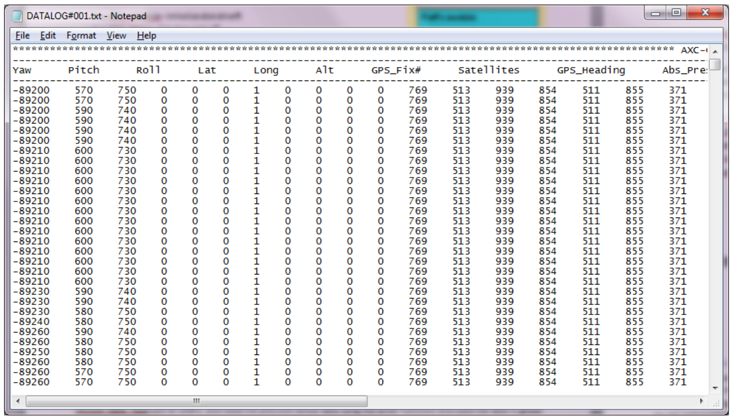
VectorNav supply a very sophisticated PC interface to configure and tune the AHRS. Unfortunately the software is meant to be used with their development board which runs on USB, and can't directly be used with the AXCG. However, a closer look at the development board will show that it's just a simple USB ↔ USART Converter. With the USB and USART working, it was very simple to implement a routine that does the exact same thing; all data received from the USB is directly sent to the AHRS, and all data received from the AHRS is directly sent to the USB. As soon as this routine was implemented, the Sensor-Explorer application was able to function with the AXCG.

Although simple to create, it was a great achievement as the Sensor explorer applications allows you to quickly and easily tune the Kalman filter parameters and other various settings such as the soft/hard iron compensation, dynamic disturbance rejection, etc. It's also very useful to view the data output of the sensors, and get a feel of how well the AHRS works. Accurately tuning the Kalman filter without the sensor explorer would have been almost impossible without this tool.

7.9.2 Data Logging

The second test routine implemented was data logging. This routine creates a data log text file on the SD card, and writes all the values from the different sensors in the text file. This made use of all the low level functions written previously, from reading and parsing the GPS, AHRS, and spectrum USART inputs, and reading the pressure sensor Inputs, as well as using the SD card and FAT file system.

To start the data logging the user presses the button, and a LED starts to blink indicating the autopilot is recording. Pressing the button again stops recording and closes the data log file (as explained for Record_Data_Task). The text file can then be opened in notepad to view all recorded data.



```
*********************************************************************************************** AXC-
-------------------------------------------------------------------------------------------------------
Yaw      Pitch     Roll      Lat      Long      Alt      GPS_Fix#     Satellites     GPS_Heading     Abs_Pre
-------------------------------------------------------------------------------------------------------
-89200   570   750   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89200   570   750   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89200   590   740   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89200   590   740   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89200   590   740   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89200   590   740   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89210   600   730   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89230   590   740   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89230   590   740   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89230   580   750   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89240   580   750   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89260   590   740   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89260   580   750   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89250   580   750   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89260   580   750   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89260   570   750   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
-89260   570   750   0   0   0   1   0   0   0   0   769   513   939   854   511   855   371
```

FIGURE 44: DATALOG TEXT FILE

Additionally, the file can be imported in Matlab, where the data can be processed, such as the system identification tool box, or plotting the different curves.

Here are examples of plots done with Matlab using data gathered by the data logger. On the left we have the yaw pitch and roll angles, and on the right 4 channels of the radio input.

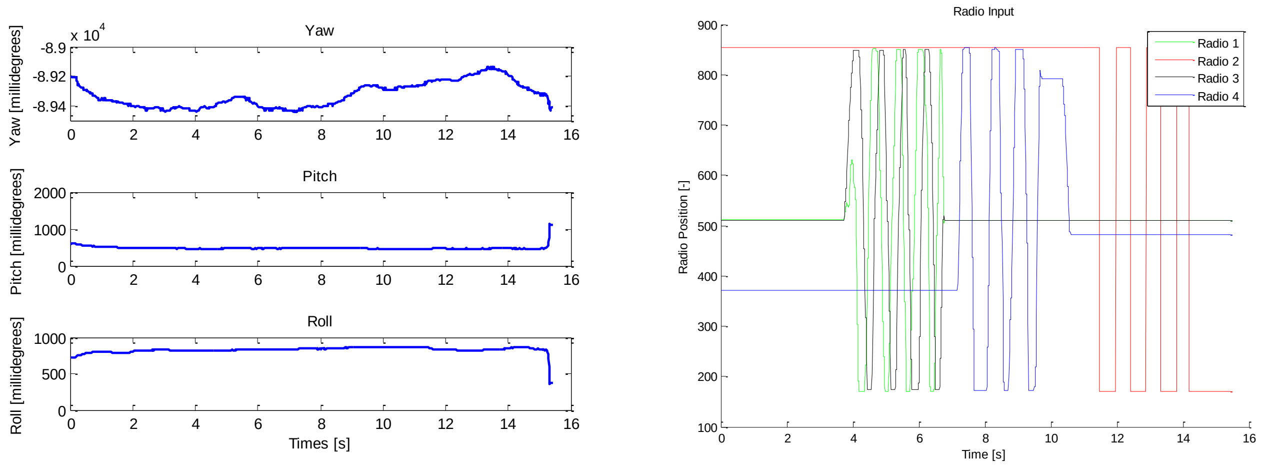


FIGURE 45: DATALOG GRAPHS

8. PC TO AUTOPILOT COMMAND PROTOCOL

To facilitate the communication between the computer and the autopilot, a robust and easy to use protocol had to be defined. Having worked with the GPS module and the VN-100 AHRS module before creating the protocol, the AXCG protocol was greatly inspired by their communication protocol. The GPS uses the standard NMEA protocol, and the VN-100 uses a similar protocol. Naturally, the AXCG command protocol follows this trend.

8.1.1 COMMANDS

Description	Command	Response
Register Read	$AXRRG,XX *CS\n	$AXRRG,XX ,D1,D2,Dx*CS\n
Register Write	$AXWRG,XX,D1,D2,Dx*CS\n	$AXWRG*CS\n

Data fields are represented by Dx. The number of Data fields and their width in bytes depend on the register being written or read. Consult register map for details. 'CS' contains a 2 byte checksum of the string starting after the '$' until the '*'. All commands must be terminated with an end of line character '\n'. If this character is not present the AXCG will not recognize the command string.

8.1.2 ERROR CODES

If there's any error regarding the command sent, the ACXG will return an error code indicating the source of the error.
The string format is the following:

$AXERR,X*CS
Where X can take a value between 1 and 9. Every value corresponds to a specific type of error described in the table below.

Error Name	Error Code	Description
Invalid Register Number	1	Read / Write to an invalid register number
Bad Checksum	2	Checksum doesn't match string received
Invalid Command	3	Cannot recognize command

8.1.3 REGISTERS

Register Name	Register Access ID	Read/Write	Width Bytes	Default Value
YPR Data	1	R	7,7,7	N/A
GPS Data	2	R	9,9,5,4,1	N/A
Pressure Voltage Data	3	R	4,4	N/A
Cycle Read Registers	4	R/W	1,1,1	0,0,0
Cycle Read Frequency	5	R/W	3	20
Mode – Mission Control / Sensor Explorer	6	R/W	1	Mission Control
SD Log Frequency	7	R/W	3	20
Servo Positions	8	R/W	4,4,4,4,4,4,4	N/A

8.1.4 YPR Data

Number	Name	Width	Field Description
1	Yaw	7	Yaw angle in milli degrees. Contains a + or – sign. 1 digit is used for the sign the 6 others for the value.
2	Pitch	7	Pitch angle in milli degrees. Contains a + or – sign. 1 digit is used for the sign the 6 others for the value.
3	Roll	7	Roll angle in milli degrees. Contains a + or – sign. 1 digit is used for the sign the 6 others for the value.

Read Example

Command	Response
$AXRRG,01*73	$AXRRG,01,+102520,+010500,-057320*71\n

Field	Value
Yaw	+102520*10^{-3} = 102.52 [Deg]
Pitch	+010500*10^{-3} = 10.5 [Deg]
Roll	-057320*10^{-3} = -57.32 [Deg]

8.1.5 GPS Data

Number	Name	Width	Field Description
1	Latitude	9	GPS latitude position given in micro degrees
2	Longitude	9	GPS longitude position given in micro degrees
3	Altitude	5	GPS Altitude given in centimeters
4	Ground Speed	4	Speed relative to the ground (earth reference frame) in centimeters per second
5	Satellites	1	Number of satellites in view
6	GPS Fix	1	Type of GPS fix: 1 : No Fix 2: 2D Fix 3: 3D Fix

Read Example

Command	Response
$AXRRG,02*70	$AXRRG,02,046284383,120284383,42574,1223,3*6C\n

Field	Value
Latitude	046284383*10^{-6} = 46. 284383 [Deg]
Longitude	120284383*10$^{-6 =}$ 120.284383 [Deg]
Altitude	42574 *10^{-2} = 425.57 [m]
Ground Speed	1223*10^{-2} = 12.23 [m/s]
GPS Fix	3= 3D Fix

8.1.6 Pressure Voltage Data

Number	Name	Width	Field Description
1	Differential Pressure Voltage	4	Voltage output from the differential pressure sensor. Given in millivolts
2	Absolute Pressure Voltage	4	Voltage output from the absolute pressure sensor. Given in millivolts

Read Example

Command	Response
$AXRRG,03*70	$AXRRG,03,1246,3090*6C\n

Field	Value
Differential Pressure Voltage	1246 [mV]
Absolute Pressure Voltage	3090 [mV]

8.1.7 Cycle Read Registers

The AXCG is capable of sending data via USB automatically at a fixed frequency defined in the Cycle Read Frequency register. “The Cycle Read Register” is used to define what data the AXCG should send. The three registers of choice are the “YPR Data register”, the “GPS Data register”, and the “Pressure Voltage Data” register. Writing a ‘1’ in the corresponding field will enable the cyclic read for that register.

Number	Name	Width	Field Description
1	YPR	1	If = 1, the YPR Data register will be read cyclically at the frequency defined in Cycle Read Frequency register
2	GPS	1	If = 1, the GPS Data register will be read cyclically at the frequency defined in Cycle Read Frequency register
3	Pressure Voltage	1	If = 1, the Pressure Voltage Data register will be read cyclically at the frequency defined in Cycle Read Frequency register

Write Example

Command	Response
$AXWRG,04,1,1,0*4F\n	$AXRRG*5B\n

Field	Value
YPR	True
GPS	True
Pressure Voltage	False

8.1.8 Cycle Read Frequency

To define at which frequency the AXCG should send the register data (defined in Cycle Read Registers), the user has access to the Cycle Read Frequency register which dictates at what frequency the registers should be read.

Number	Name	Width	Field Description
1	Frequency	3	Frequency at which the register defined in the “Cycle Read Registers” Register should be read. Defined in Hertz, with a range from 0Hz to 999Hz

Write Example

Command	Response
$AXWRG,05,050*4F\n	$AXRRG*5B\n

Field	Value
Frequency	50Hz

8.1.9 Mode

The AXCG is capable of operating with two different PC applications. The first is the custom made Mission Control used to program waypoints and other various settings of the autopilot. The other is the Sensor Explorer application from VectorNav that is used for the VN-100 development board, which was adapted so it can be used with the AXCG. The Sensor Explorer application is used to tune the Kalman filter and change the various settings of the AHRS module. In order to use one application or the other, the user has to define in which mode the AXCG needs to operate in, by writing a 1 or 0 in the Mode register.

Number	Name	Width	Field Description
1	Mode	1	0 : Mission Control Mode 1 : Sensor Explorer Mode

Write Example

Command	Response
$AXWRG,06,0*5D\n	$AXRRG*5B\n

Field	Value
Mode	0 = Mission Control

8.1.10 SD Log Frequency

The AXCG has a micro SD card slot that is used for data logging of flight control data. The user chooses at which frequency data is stored by writing in the SD Log Frequency register. Any input between 0 to 999Hz is accepted. But anything over 200Hz would have little use, as the on board sensors have a maximum data output of 200Hz or less.

Number	Name	Width	Field Description
1	Frequency	3	Micro SD card data log frequency. Defined in Hertz, with a range from 0Hz to 999Hz

Write Example

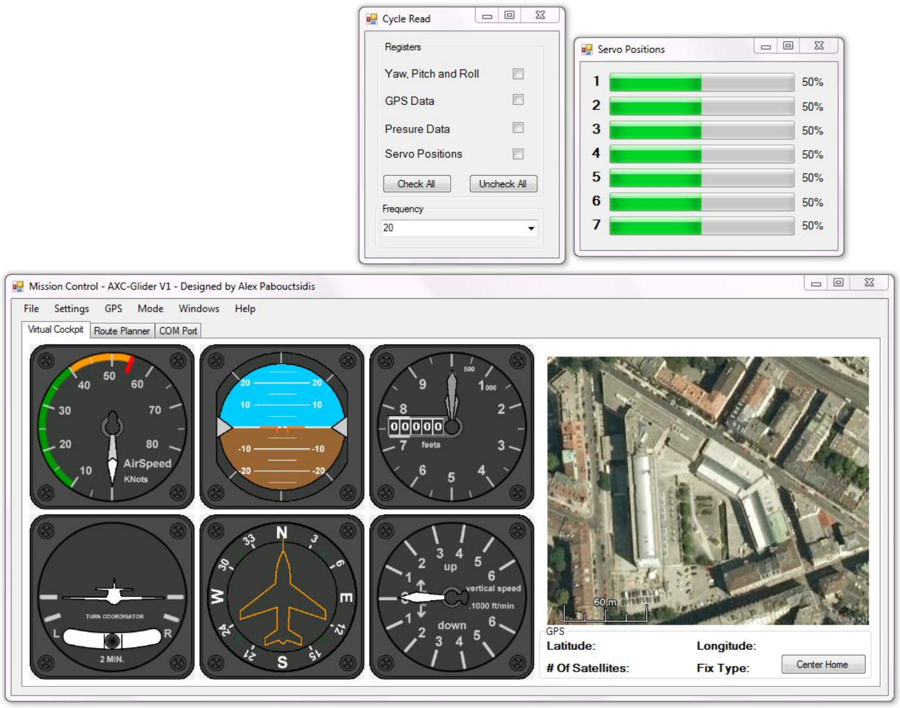
Command	Response
$AXWRG,07,100*5D\n	$AXRRG*5B\n

Field	Value
Frequency	100 Hz

9. PC Interface development

A PC user interface also needs to be created in parallel in order to configure and upload GPS way points to the autopilot board. Development of the application already started, but is still missing the majority of the functionality. It was designed in C# with Microsoft Visual Studios based on the .net framework. It's able to send and receive data via the com port, display Google maps in an embedded web browser, and display various sensor data such as an artificial horizon, air speed, altitude, and heading. The instrument objects were obtained from a similar open source project from www.DIYdrones.com, and then implemented in the Mission Control application. The name of the PC interface is Mission Control.

Using an embedded web browser, it loads a custom made html file with Google maps. To use Google maps via the web browser in the C# application, JavaScript functions (implemented in the html file) will be needed to add lines, markers, etc. as well as extract data using the Google API. These JavaScript functions can then be called from the C# code.



Frequency: 20

1	50%
2	50%
3	50%
4	50%
5	50%
6	50%
7	50%

FIGURE 46: MISSION CONTROL MAIN WINDOW WITH SERVO POSITIONS AND CYCLE READ SETTINGS

To complete the Mission Control program, the JavaScript functions to interact with Google maps need to be written, as well as route planner, to add GPS waypoints, holding patterns, etc. Once the route has been created by the user, it then needs to upload it to the autopilot, with security checks to make sure that a correct route was correctly uploaded. Secondary functions that can be added are play backs: using a data log file, it could play back a flight giving the user a view of all the instruments and data while it was flying, as well plotting the course on Google maps.

10. Next Step

10.1 Plant Identification

Now that we have a solid hardware foundation with the AXCG, the next step is to create the actually navigation software. To design robust and accurate attitude and navigation controllers, we need an accurate module of the glider. Different methods to design the model were evaluated. The Matlab Aerospace tool box has a couple examples of airplane models, which can serve as a starting point in creating the Cularis glider model.

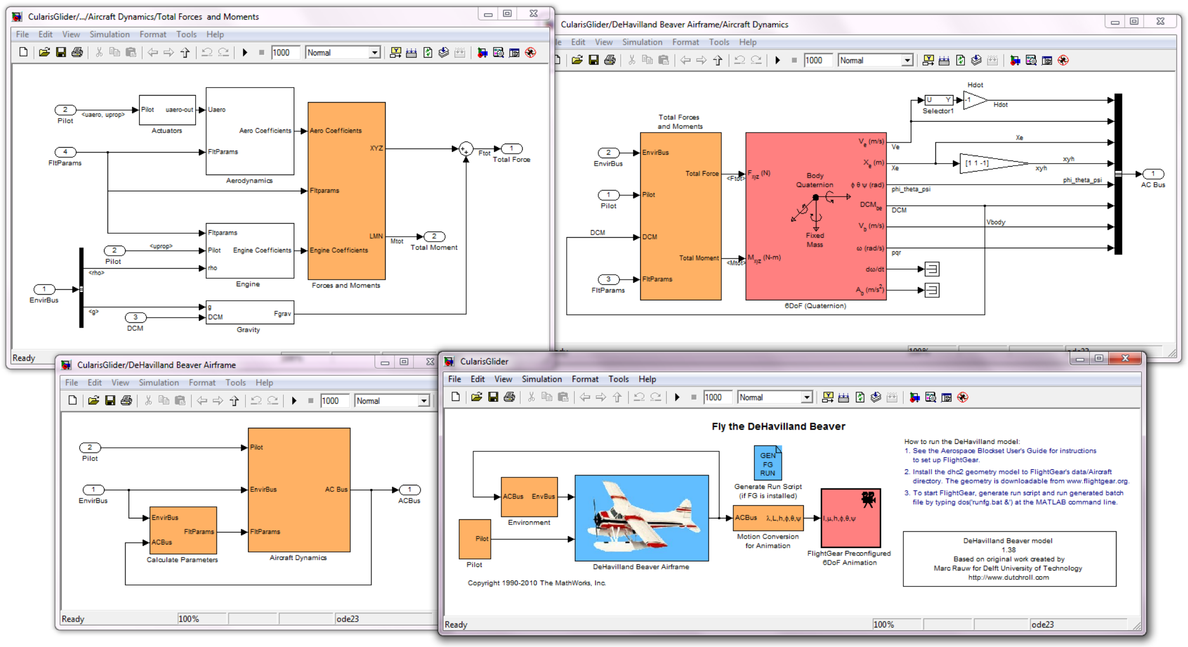

CularisGlider/../Aircraft Dynamics/Total Forces and Moments
CularisGlider/DeHavilland Beaver Airframe/Aircraft Dynamics
CularisGlider/DeHavilland Beaver Airframe
CularisGlider
Fly the DeHavilland Beaver
Actuators
Aerodynamics
Engine
Gravity
Forces and Moments
Total Force
Total Moment
6DoF (Quaternion)
AC Bus
Environment
DeHavilland Beaver Airframe
Motion Conversion for Animation
FlightGear Preconfigured 6DoF Animation
Calculate Parameters
Aircraft Dynamics
Ready

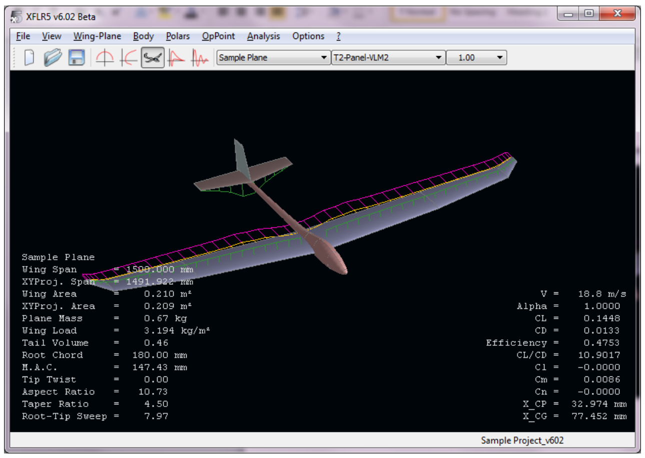
FIGURE 47: SIMULINK AIRPLANE MODEL

After studying the Matlab Simulink model, it was revealed that it relied on airplane specific characteristic curves. These curves are the forces and moments for all three axes acting on the aircraft vs. the angle of attack, side slip of the aircraft, and dynamic pressure of the aircraft. Unfortunately, these curves are difficult to measure.

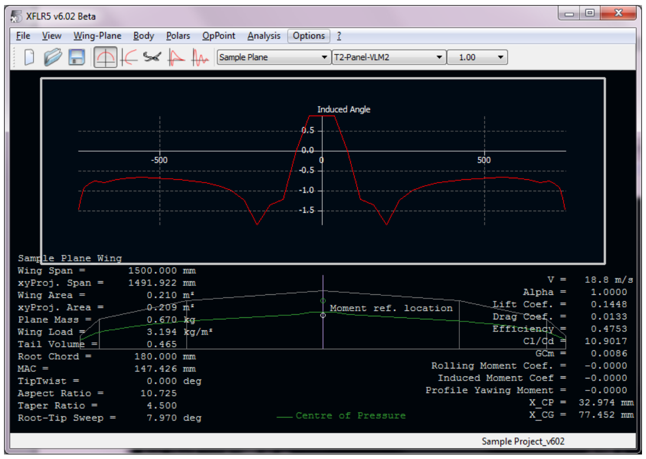
The major problem that remains is extracting these data curves from the Cularis glider. To do this, two tools are at our disposal, the first is the data logger. With the data logger we can measure the angular velocities and attitude of the aircraft, and see how they vary over time in function of servo inputs. This data, coupled with the System Identification tool box from Matlab, could provide an accurate model of how the plane reacts to servo inputs.

The second tool was discovered while browsing DIY Drones called XFLR5. It's an open source analysis tool for airfoils, wings and planes operating at low Reynolds numbers. It includes XFoil's direct and inverse analysis capabilities, and wing design analysis capacities based on the lifting line theory.

Entering the wing shape and airfoils of the glider, XFLR5 computes theoretical force and moments curves. It can also perform stability analysis providing longitudinal and latitude modes pole locations



```
Sample Plane
Wing Span        = 1500.000 mm
XYProj. Span     = 1491.922 mm
Wing Area        =    0.210 m²                      V =   18.8 m/s
XYProj. Area     =    0.209 m²                  Alpha =   1.0000
Plane Mass       =    0.67 kg                      CL =   0.1448
Wing Load        =    3.194 kg/m²                  CD =   0.0133
Tail Volume      =    0.46                 Efficiency =   0.4753
Root Chord       =  180.00 mm                   CL/CD =  10.9017
M.A.C.           =  147.43 mm                      Cl =  -0.0000
Tip Twist        =    0.00                         Cm =   0.0086
Aspect Ratio     =   10.73                         Cn =  -0.0000
Taper Ratio      =    4.50                       X_CP =  32.974 mm
Root-Tip Sweep   =    7.97                       X_CG =  77.452 mm
```



```
Sample Plane Wing
Wing Span =         1500.000 mm
xyProj. Span =      1491.922 mm                          V =   18.8 m/s
Wing Area =            0.210 m²                      Alpha =   1.0000
xyProj. Area =         0.209 m²                 Lift Coef. =   0.1448
Plane Mass =           0.670 kg                 Drag Coef. =   0.0133
Wing Load =            3.194 kg/m²              Efficiency =   0.4753
Tail Volume =          0.465                         Cl/Cd =  10.9017
Root Chord =         180.000 mm                        GCm =   0.0086
MAC =                147.426 mm        Rolling Moment Coef. =  -0.0000
TipTwist =             0.000 deg       Induced Moment Coef =  -0.0000
Aspect Ratio =        10.725          Profile Yawing Moment =  -0.0000
Taper Ratio =          4.500                          X_CP =  32.974 mm
Root-Tip Sweep =       7.970 deg                      X_CG =  77.452 mm
```

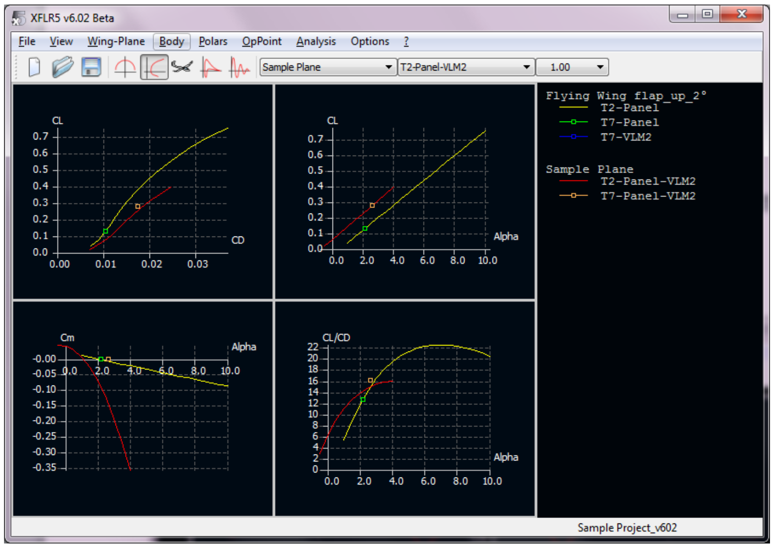


FIGURE 48: XFLR5

In the end, it will be a combination of all these methods that will provide the best possible model. Starting from the Simulink aircraft example, it can be modified using data obtained from XFLR5. After real life flight tests with the data logger, using the system identification tool box, the model can then be refined to reflect the real life dynamics of the glider.

10.2 Controller Design and Implementation

Once an accurate model of the aircraft is built, it can be used to design robust controllers. Without knowing the exact nature of the system, it is hard to guess what form the controllers will take, and what type will be used.

At first glance, a state space controllers might seem the most suited, as they work well with multiple inputs multiple outputs (or MIMO) systems, and are easy to implement mathematically as they are presented under the form of matrices.

Another possible solution would be a robust controlled design using robust technics studied in class. Needless to say, without a valid model of the aircraft, designing a robust and accurate attitude controller will prove to be difficult.

After designing a suitable (or what we think is a suitable) controller, It can easily be implemented into a task with the RTOS once we know it's equation in the Z space.

11. CONCLUSION

During the course of the last 16 weeks, all the hardware and low level functions of an autopilot PCB for an autonomous glider has been developed. The PCB has all the required sensors and components to accurately fly a glider or any other model aircraft through GPS waypoints. It is capable of storing all flight data to an external micro SD card using the FAT file system which can be used as a black box, or to study the dynamics of the aircraft after a flight. The autopilot is running on a real time operating system giving it great flexibility, so that all future code can easily be incorporated. A solid foundation of low level functions provide efficient and quick access to all sensor data in a structured manner, as well as functions to configure sensors and command servo outputs. The autopilot is capable of communicating to a computer via USB configured as a virtual COM port, using a standard 'NMEA like' protocol.

A suitable glider that fulfills the requirement was purchased and assembled, as well as smaller test RC airplanes that can be used for in field testing before moving up to the larger glider.

The development of a computer user interface coded in C# using Microsoft Visual Studio was started, with the basic framework in place. The computer program has Google maps running in an embedded web browser with basic JavaScript function examples.

Multiple tools that can be used to create a model of the aircraft, which in turn will be used to design the attitude and navigation controllers, were found (XFLR5 and Matlab Aerospace toolbox).

12. LIST OF FIGURES

13. Digital Appendix Files

13.1 Datasheets

- 3.3V Regulator - MCP1825S-3302EDB.pdf
- 5V LDO - LP3871 & LP3874.pdf
- 8Mhz Resonator - CSTCE8M00G55-R0.pdf
- 22uF capacitor - TR3A226K010C1000.pdf
- Absolute Pressure - MPXH6101A.pdf
- AHRS - VN-100.pdf
- Button1 - KMR221GLFS.pdf
- Differential Pressure - MPXV5004G.pdf
- EEPROM - 24AA16.pdf
- GPS - Mediatek_3329.pdf
- LED - SML-211UTT86.pdf
- Microprocessor Reset - STM6315SDW13F.pdf
- Micro SD Connector - DM3BT-DSF-PEJS.pdf
- Spektrum Connector - S3B-ZR.pdf
- STM32 Microcontroller - STM32F103RCT6.pdf
- USB Connector - UX60-MB-5ST.pdf
- USB Line Termination - USBUF02W6.pdf

13.2 Firmware Code

- Drivers – low level driver source and header files
- FreeRTOS – FreeRTOS C and header files
- Libraries – Peripheral, FatSD, CMSIS, and DSP_Lib source and header files
- src - main source files
- inc - main header files

13.3 Firmware References

- Mediatek_3329_Custom_Binary_Protocol.pdf
- STM32F1xxx Manual.pdf
- Using the FreeRTOS Real Time Kernel - A Practical Guide - Cortex-M3 Edition.pdf

13.4 Schematic + PCB

- AXC-Glider V1.pdf