The Design and Fabrication of a Quadrotor Platform.

Alexander Stevens
110170206

Electrical Engineering [MEng]

The University of Sheffield
Department of Electronic and Electrical Engineering

May 2014

Supervisor: Dr Luke Seed
Second Marker: Dr Andrew Maiden

Word Count: 6,000

Abstract

A quadrotor is a 4 axis robot used frequently today for numerous aerial applications. Its design combines multiple different areas of engineering including electronic, mechanic and aeronautical disciplines.

The overall objective of the project was to create a safety orientated quadrotor that could be introduced into a school environment; that students are able to physically build, program and modify. Thus a genuine insight into how engineering knowledge can be applied theoretically and physically to real-world project will be achieved through this learning tool.

This report follows the continued development of a Raspberry Pi controlled quadrotor with the aim to simplify the current electrical design whilst considering safety aspects throughout.

This was achieved by introducing a single control board used to implement a flight controller as well as drive four brushless DC motors. A serial program loader was developed enabling a method of writing to the processors memory space by communicating with the boot loader that is included within the Texas Instruments Tiva C series range of processors. This enables the end user a mechanism to download programs wirelessly, without any specialist equipment or software.

1. Introduction

1.1 Background

A Quadrotor is a 4-axis robot, which can be built from readily available materials due to their relative mechanical simplicity. Quadrotors are used extensively for both military and hobbyist purposes; ranging from drones used within search and destroy missions to taking aerial photographs of landscapes [1].

Currently in the UK there is a shortage of engineers due to students choosing against taking STEM (Science Technology Engineering Mathematical) subjects [2]. Research has found that students are becoming frustrated of being taught purely from literature, craving more 'hands on' learning [3]. By introducing a quadrotor to schools, which students could physical build, program and modify, it would provide a useful insight into a real engineering project that may encourage students to continue with STEM education.

1.2 Context

This project continues from a previous student's work which developed a custom AHRS (Altitude and Heading Reference System) for a Raspberry Pi controlled quadrotor by implementing a number control loops using real time data provided by on-board sensors [4]. This quadrotor design included four individual electronic speed controllers, to drive the brushless DC (BLDC) motors, alongside a separate control board containing a microcontroller for sending and receiving pulse width modulation (PWM) channels used for motor control and a sensor board collecting real time data.

A single motor control board has been designed to replace all of the peripheral boards used in the previous design. By including components that have been individually selected for a quadrotor application, it will enable the control software further scope to be improved, thus yielding a high performance quadrotor whilst reducing the complexity of the overall design to the end user.

2. Specification

2.1 Aims and Objectives

The overall objective is to produce a fully functioning quadrotor that could be introduced to schools as a learning tool. Students should be able to build the mechanical frame and install the electrical components. Users will also have the ability to write and modify lines of code and then download it to the robot.

As the final product may be operated by teenagers there is a strong emphasis on safety.

2.1.1 Mechanical Frame

- Each of the rotating blades should be enclosed, preventing any part of the user's body coming into contact with moving parts.
- The frame should be durable and tough; enabling it to withstand a reasonably hard crash without fracturing.
- All of the components are easily accessible, allowing for maintenance.
- The frame should be easily manufactured, using readily available materials and equipment.

2.1.2 Motor Control Board

- Be able to collect real time data and manipulate it in order to fly autonomously.
- The board should be able to drive four BLDC motors without hall-effect speed sensors.

The control board needs to be fully tested, this should include:

- Communications with and between processors confirmed.
- Quiescent levels measured to prevent problems when adding power components.

2.1.3 Programming

- Raspberry Pi running a version of Debian Linux will be used.
- All code will be written and/or modified in C programming language.
- A tool to write and update programs to each of the processors on the board needs to be designed and implemented.
- A BLDC driver, which provides voltage across two of the phases whilst calculating the speed from the back-EMF produced from the third phase, should be developed.

2.2 Gantt Chart

Component	Week																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Frame Design	▨	▨	▨	▨	▨	▨								▨	▨	▨	▨	
Lit. Review						▨	▨	▨	▨	▨	▨	▨	▨					
Programming						▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨		
Overall Project																▨	▨	▨

Table [1] – Project Gantt chart

Table [1] displays the original project timescale over an 18 week period. Literature review accounts for time needed to study the Texas Instruments material released for development with their processors, an overview of which will be included where necessary in this report. The final row, overall project, will be used to develop the project into a package that could be introduced into schools.

3. Frame

3.1 Material Selection

A quadrotor, being an airborne vehicle, has obvious desirable properties of being lightweight, tough and durable allowing for a high performance product that isn't rendered useless as a result of damage; such as being subjected to a relatively harsh crash.

Depron, which is an extruded polystyrene sheet, is used extensively in model aircraft due to its lightweight and moisture resistant properties [5]. Originally developed for insulation applications, it is reasonably ridged allowing for non-specialist, conventional machinery to be used in order to shape it. Other materials such as expanded polypropylene (EPP) were discounted as they require specialist hot wire tools to shape the material.

3.2 Design

The frame design consists of two parts, an external flexible outer frame included for safety purposes and an internal, ridged metal or composite frame used to hold the electronic components in place.

3.2.1 External Frame

The outer frame is constructed of 8 layered 9mm Depron sheets. An extruded slot in the centre is used to sit the battery and electronic circuitry. All four rotor blades are enclosed by the Depron with a fine mesh to be attached on both top and rear faces to prevent users fingers coming into contact with the moving parts.

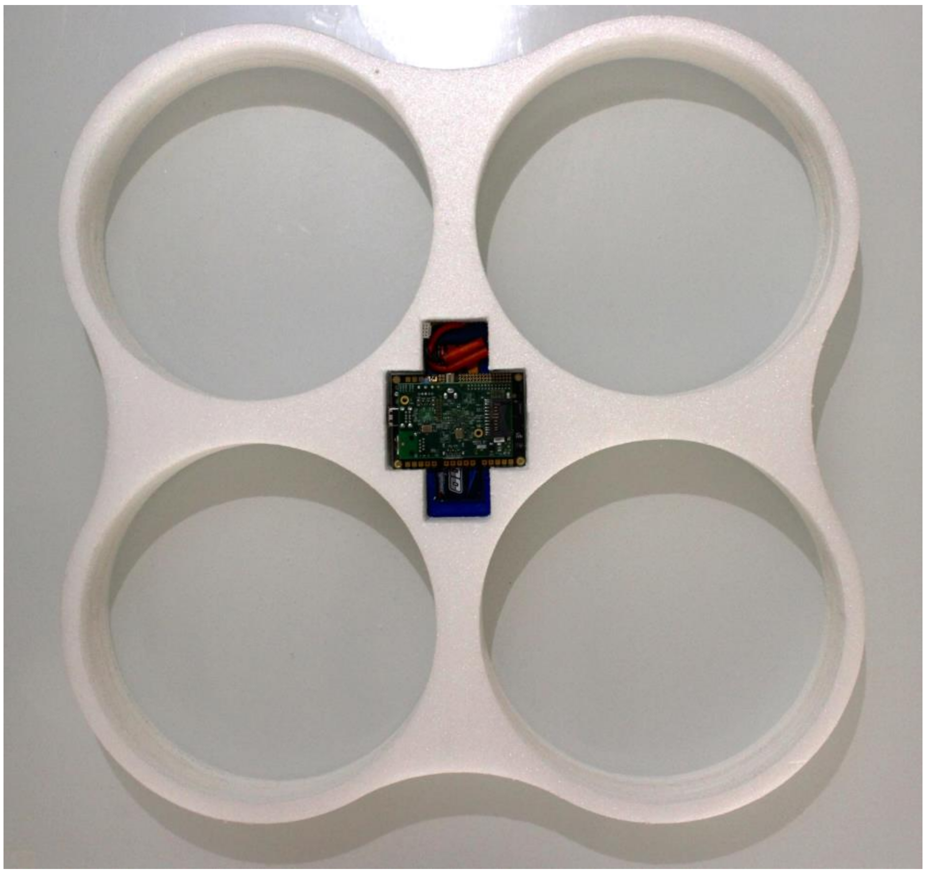


Figure [1] – External Depron Frame with the electronic circuitry and battery housed in the centre

3.2.2 Internal Frame

The four motors will be held in place using an aluminium cross frame that includes a central plate, used for fixing the electronic circuitry and battery in place. The external frame will sit above the internal frame enclosing the motors and rotating blades.

4. Motor Control Board

4.1 Components

The motor control board replaces all of the separate electronic boards used in the previous design. The board must be therefore capable of collecting sensor data, manipulating it and driving the four BLDC motors accordingly, taking into account pilot input.

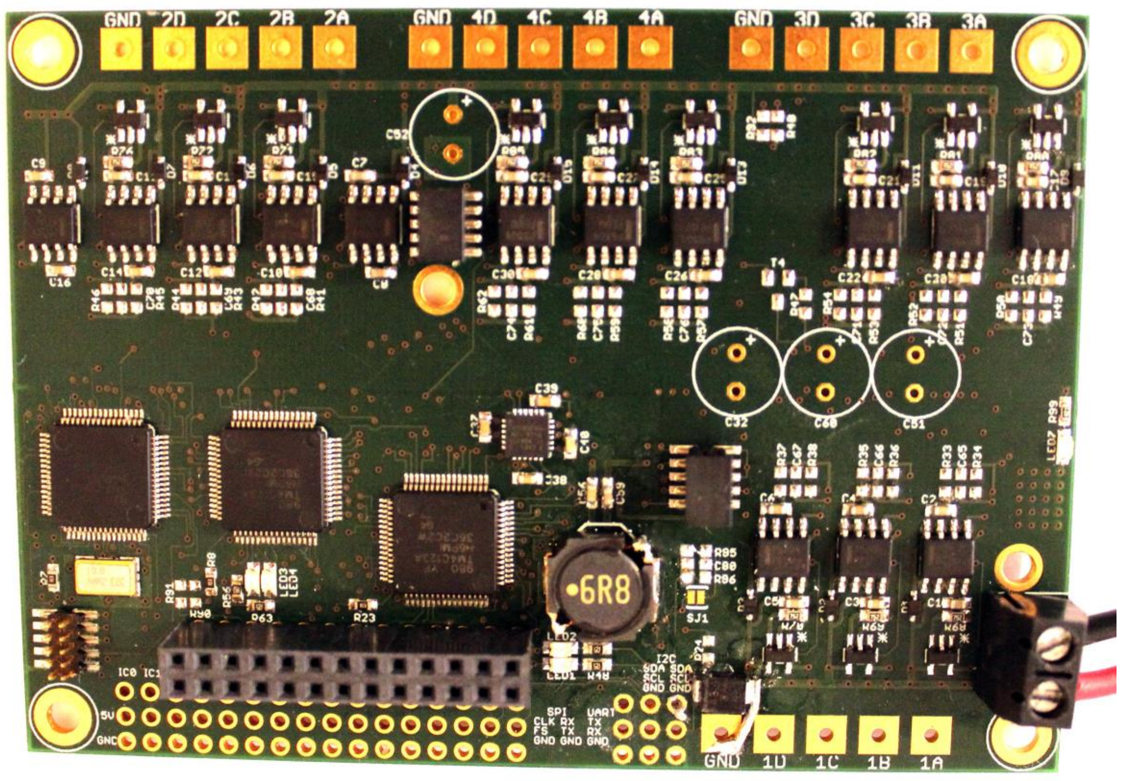


Figure [2] – The Motor Control Board. The three Tiva C-Series Processors are situated in the bottom left, the InvenSense sensor chip sits central, the 8 pin rectangular components are the FET IC drivers with the 3 pin current sensors lying close to the top and bottom edge. The 26-pin GPIO header is located below the three processors.

4.1.1 Tiva C-Series Processors

At the heart of the control board are three Texas Instruments Tiva C-series ARM Cortex-M4 processors. The TM4C123AH6PM processor is designed specifically for low power, hand-held smart devices incorporating motion control [6] thus a good fit for a quadrotor application. They contain a Floating Point Unit (FPU) allowing for further capacity to be integrated into the existing AHRS, alongside I^2C and Serial Peripherals Interfaces (SPI) that enable communication between multiple separate processors.

Tiva C-series processors also contain the TivaWare Boot Loader, stored in the ROM (Read Only Memory) portion of their memory space. This allows the processors to be accessed via serial interfaces, enabling a method to download and update programs without use of specialist equipment or cables.

4.1.2 InvenSense MPU-9150

A single, 9-axis InvenSense MPU replaces the 6-axis sensor board on the previous project. It contains a Tri-axis angular rate sensor (gyro), tri-axis accelerometer and tri-axis compass. It also contains a Digital Motion Processor (DMP) that enables low power processing of complex 9-axis algorithms [7].

4.1.3 On Board Power Supply

The board is supplied from a 12v, 5AH lithium polymer battery [8]. For the purposes of development and testing, a 12v bench power supply was used in place of this. On the circuit board there is a 5v step down non-synchronus regulator and a 3.3v regulator.

4.1.4 Peripheral Electronic Components

The rest of the board is populated with components needed to control the BLDC motors. This includes complementary FET pairs on the reverse side of the board with their individual high power gate drivers on the topside. Current monitors are used for the measurement of back EMF generated in the third phase of the BLDC motors, whilst the other two phases are conducting. This enables the timing of the next commutation from the zero-crossing point to be calculated [9]. The gate drivers will be supplied with a PWM signal from a processor.

4.2 Communications

The board includes three serial interfaces for communication between the Cortex-M4 processors, sensor chip and Raspberry Pi, which will be used for loading programs initially. The SPI (or SSI) bus is the fastest out of the three and is implemented as shown in figure [3], with I^2C and UART busses also included.

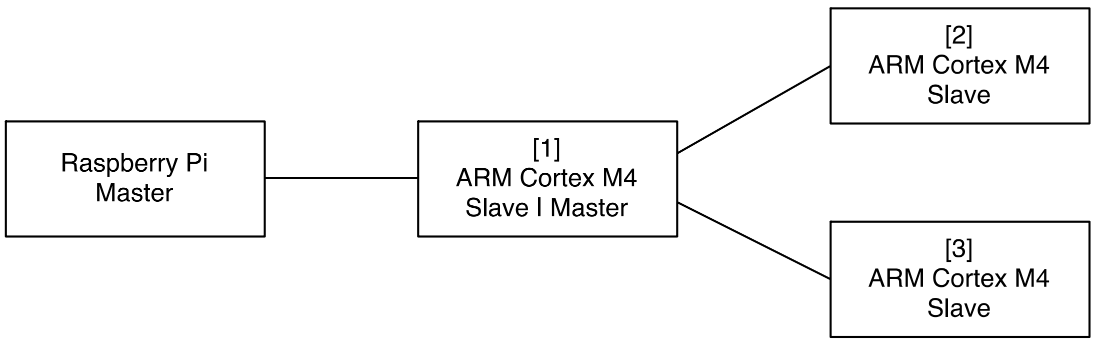


Figure [3] - SPI Master / Slave hierarchy

Processor [1] handles data received from the external Raspberry Pi via all three serial interfaces as well as the sensor chip via an I^2C bus, a two wire serial interface.

Each of the Cortex-M4 MPU's has a JTAG (Joint Test Action Group) header, allowing for direct debugging including single stepping and break pointing.

4.3 Motor Control

The 1st processor (see figure [3]) is tasked with applying the control loops and sensor fusion, which governs the speed at which the motors will run. Whilst the 2nd and 3rd processors will directly control two BLDC motors each by outputting a PWM signal to the gate drivers for each motor.

Processors 2 and 3 should in theory run exactly the same program that includes the BLDC motor driver software.

5. TivaWare

5.1 Overview

TivaWare refers to a large amount of software that Texas Instruments release for use with their embedded processors. It contains the peripheral driver libraries alongside existing example code written for specific applications. Included within the TivaWare term are the boot loader stored in ROM on the Tiva C series processors and driver libraries for motor control to be used in this project. All of the software is written entirely in C and are royalty free [10].

5.2 Peripheral Libraries

Included in the TivaWare are driver libraries. These contain source code for SPI communications, timers, interrupts and PWM channels. When compiling a project you are able to link to these libraries into a project before downloading and running it on the processor. A number of functions are stored in the ROM to enable the boot loader to execute whilst reducing the overall size of the program file that is to be stored in the flash memory space.

5.3 Boot loader

The Boot loader enables the processors to communicate via serial interfaces to external devices and will execute whenever the flash memory is empty.

The processor will operate at the internal 16 MHz clock, as it has no knowledge of the external 10MHz processor on the control board.

The boot loader operates by using defined packets on the serial interface busses to call specific functions that are also stored within the ROM. Reliable communications are maintained by the use of an acknowledgement packet that follows every transmission of data.

There is a set structure that has to be followed in order to send a packet correctly:

1. Send the total number of bytes to be transmitted (including itself).
2. Send the checksum, the sum of the data bytes to be sent.
3. Send the data bytes.
4. Wait for the acknowledgement returned from the boot loader.

A similar process must be applied in order to successfully receive a packet:

1. Wait for the first non-zero packet to be received (the processor will output zero's when not sending data). This will be the byte containing the size of the packet to be received.
2. Read the checksum.
3. Receive the data.
4. Compare the calculated checksum and the checksum received.
5. Send acknowledgement or non-acknowledgement byte accordingly.

6. Programming of the Control Board

6.1 JTAG Programming

The first stage of testing the prototype circuit board was to download a small piece of code to each of the processors, ensuring that they are functioning as expected.

By using a JTAG debug emulator probe it is possible to directly download code to the embedded processor from a PC. The first, simple, program downloaded returned data from processor via a debug console on the PC. This was completed successfully for all three processors.

A program that includes the libraries used for outputting PWM signals was then downloaded. This is used to test the external components on the control board. When downloading this program via the JTAG interface an error was encountered preventing communication between PC and processor. The error code (-1063) suggested that the device was not recognised and not supported by the download driver. This was improbable as the simple program, downloaded previously, debugged as expected without any problems.

A faulty processor was discounted by downloading the same program to a second processor; as a result the second processor also became locked with the same error code. Initially software was believed to be the cause of the error; with either an incorrect configuration file or a problem with the code itself that was causing the serial communications to fail during the download sequence.

A simple explanation for this error occurring is that whilst adding the driver libraries to the processor, some of the pins could have been remapped, most importantly the JTAG pins to GPIO thus preventing communication via the JTAG header. The processor's flash memory is able to be erased by performing a series of JTAG-SWD and SWD-JTAG sequences and as a result remove the defective pin mapping. These unlock sequences proved un-successful and the processor remained locked.

The code, configuration files and compiler settings were confirmed to be correct by the support team at Texas Instruments thus software was discounted as the reason for the error occurring.

With the processor remaining locked attention turned to the circuit around each of the processors. The TM4C123AH6PM user manual includes how each pin should be connected to external components. There was a small difference in the way in which the VDDC pins, the positive supply used for most of the logic function, are connected. The user manual suggests that the two VDDC pins should be only connected to each other and a single capacitor in the range of 2.5-4μF. Currently each pin is connected through a separate capacitor, and then to ground, creating an effective capacitance of 6.6μF. This however is negligible and proved not to be the underlying problem.

Texas Instruments were further contacted alongside interacting on the TI E2E (Engineer to Engineer) forums. Neither of which proved successful in diagnosing the underlying fault on the board, although Texas Instruments did attempt to recreate the problem unsuccessfully suggesting that there was a problem with the control board circuit or components.

The final area of the board that could cause the processor to be locked in such way was the power supply. As a result both 5v and 3.3v regulators were removed, and replaced with a bench power supply providing 3.3v for the MPU's. The processors were then able to be recognised via the JTAG interface allowing the flash memory to be erased and programs to be downloaded, including those of which include the driver libraries.

If the voltage supplied to the processors is not 3.3v it could cause the processors to execute through the power-up sequence incorrectly, producing an internal error and not release the CPU resulting in no communications. During the loading of the larger files, which include the peripheral libraries, a reset signal is transmitted which may draw a larger current causing a momentary lapse in the quality of the power supplied, triggering the processor to not release the CPU. The components were analysed and recommendations for replacement are detailed in Section 10.

6.2 Serial Peripheral Interface

SPI, also referred to as SSI (Synchronous Serial Interface), is a four wire serial bus used extensively for communication within embedded systems. It consists of MISO (Master In Slave Out), MOSI (Master Out Slave In), CLK (serial clock) and SS (slave select) lines. When a slave processor's select pin is low, it will receive data placed on the MOSI line from the master, whilst when it is high, it will ignore data on that line; this therefore enables multiple devices to use the same MISO, MOSI and CLK lines. The master defines the clock frequency, dependant on what the slave processor is able to support.

SPI operates much like a synchronous 16-bit shift register, with an 8-bit half on both the transmitter and receiver, where the clocking cycle is controlled by the master [11]. Each clock cycle will output a packet from the master on the MOSI line to be received by the slave; in return the slave will output a packet on the MISO line to the master. Each data transmission is controlled by the master; if the master doesn't need to send data it will not output a clock. Thus the slave processor can only send a packet when it receives a packet from the master.

6.3 Raspberry Pi

A Raspberry Pi (RPi) is a single board computer which runs the open-source operating system Linux. The RPi contains no internal storage, so requires an external SD card in order to boot. The RPi includes a 26 pin GPIO header, enabling I^2C, UART and SPI communication capability. It should be noted that there is only two pins for the slave select line, only allowing for two slave processors on the SPI bus.

6.3.1 Configuration

The RPi can be operated either by a command line utility or a GUI which initially accessed using a keyboard, mouse and monitor hardware. The RPi is able connect to a local network both by a WiFi adaptor and LAN; this enables the RPi to be accessed remotely using an alternative host computer's hardware. A Secure Shell (SSH) provides secure data communication over a network between a server, in this case a PC and a client, the RPi. This provides access to command line interface. Taking this one step further is to introduce a VNC (virtual network computer) system, which enables the user to view and control the GUI of the client RPi.

In this project the RPi was configured such that it was accessible both by SSH and VNC via a USB WiFi module [12].

6.3.2 SPI Initialisation

Recently the Debian port of Linux (Raspian) was updated to include a SPI controller entitled spidev. Spidev by default is disabled on the RPi and needs to be manually enabled. For this project spidev0.0 is used as the CS0 pin is interfaced to the slave MPU.

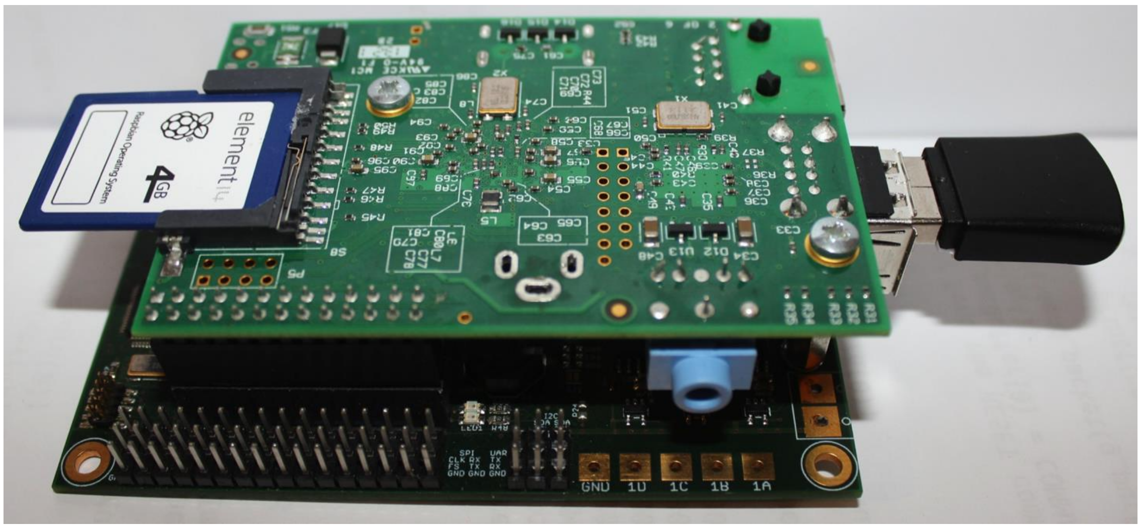


Figure [4] – Raspberry Pi connected to the motor control board through the 26 pin GPIO header visible in the bottom left of the board. The WiFi module is on the right connected via USB to the RPi, with the 4GB SD card installed on the left.

7 Download Utility

7.1 Overview

One of the original objectives was to create a mechanism that enables programs to be loaded onto each of the processors on the control board. Ideally, as the quadrotor is a stand-alone unit, the loading should be done wirelessly without having to modify the quadrotor in any way.

The RPi's GPIO header is directly interfaced to the control board, allowing the SPI bus to be used. Then by accessing the RPi via SSH or VNC, data can be passed remotely from a host PC to be downloaded to the board via the proxy RPi.

Texas Instruments currently only supports the download of programs via a serial interface on a PC running windows operating system, using the LM Flash Programmer tool [13]. There is currently no support for Linux. As a result, a custom serial loader was written to pass the outputted program data to the Tiva C-series boot loader in the correct format using the RPi. This serial loader will use the boot loader as the mechanism to write to the processors' memory space.

7.2 Implementation

Code Composer Studio (CSS) integrated development environment (IDE) was used for writing and compiling the executable code for the control board processors. This is the IDE which Texas Instruments release for development with the Tiva Series processors; however in theory any IDE that can compile projects for the ARM Cortex M4 family could be used.

For the executable output files to be written to the processors a binary (.bin) file is needed. This can be obtained in CSS by configuring a post build step that will produce a .bin file during the compilation of a project.

The user will then log in to the RPi remotely, store the program file locally on the SD card and run the serial loader program.

As the RPi is only connected directly with the first processor on the board, when the user needs to download to either the second or third processor first a "pass through" program, that simply reads the data on both the RPi MOSI & MISO lines and outputs it onto the desired SPI bus, should be loaded onto the first processor. The boot loader program will then work in exactly the same way as if it were communicating with that processor directly.

Adafruit WebIDE allows the user to log into the RPi that is connected to the same local network as the host computer and upload files to the RPi. Code is able to be compiled and executed within an internet browser client [14]. This is an elegant solution as it requires the user not to have to download or install any specific software.

7.3 spidev

A free test program is available to ensure that the RPi & spidev driver are functioning correctly released under the GNU General Public License [15]. This file was modified to enable the RPi to communicate with the Tiva processors. An arbitrary speed of 1kHz was selected that, if a faster download was required, could be increased later.

The spi_handler.c program takes data passed from the main loader program, opens the SPI port, transfers the data in the correct format for the spidev driver and finally closes the port.

7.4 SPI Data Transmission Functions

Within the main loader program (read_file.c) are two functions; read_packet and send_packet, which are called each time a packet is to be sent and/or received. As mentioned in Section 5.3 a specific structure must be followed in order to send or receive a packet on any of the serial buses in accordance with the Tiva C series boot loader. As a result the send function calculates both the number of bytes to be transmitted and the checksum, whilst the read function decodes the first two bytes and compares them with the actual number of bytes and checksum received to ensure no errors have occurred. As all packets must be either acknowledged or non-acknowledged each time a packed is sent or received it is followed by an acknowledgement byte.

When the program expects to receive data, bytes with the value zero are sent out from the master allowing the slave to place data onto the MISO line without causing confusion.

7.5 Importing Program File

The user will specify the address where they have stored the .bin file on the RPi SD card, within the loader program. The loader will then open the file and place it into a local buffer.

7.6 Boot loader Commands

As mentioned previously, the Tiva C series boot loader functions by using well defined packets in order to call certain functions stored in the ROM. Their values are defined in the spi_handler.h header file and their role described below. The following structures are used to download a program and check that the slave processor has accepted the data.

7.6.1 COMMAND_PING

The ping command is a single byte command that returns an acknowledgement byte from the slave processor, ensuring that communications are established.

7.6.2 COMMAND_DOWNLOAD

The command download structure provides the boot loader with the address where it should place the data it is to receive as well as the amount of data it is to expect. This is implemented by two 32-bit values transmitted most significant bit (MSB) first, where packets 1-4 contain the program address and packets 5-8 containing the program size.

Packet [0] – COMMAND_DOWNLOAD
Packet [1] – Program Address [31-24]
Packet [2] – Program Address [23-16]
Packet [3] – Program Address [15-8]
Packet [4] – Program Address [7-0]
Packet [5] – Program Size [31-24]
Packet [6] – Program Size [23-16]
Packet [7] – Program Size [15-8]
Packet [8] – Program Size [7-0]

7.6.3 COMMAND_SEND_DATA

This command follows the COMMAND_DOWNLOAD, or another COMMAND_SEND_DATA. This also is a 9 packet structure where the defined send data byte is followed by 8 packets of data, with a maximum of 251 data bytes to be sent per structure. The loader program recalls this function until all of the program data from the buffer has been transmitted.

7.6.4 COMAND_GET_STATUS

This, like the ping command, is used to confirm that the communications are working as expected. The get status command is a single byte structure and returns the status of the previous command received from the boot loader. It is used after every download and send data command to confirm that the boot loader has understood and processed the data received.

7.6.5 COMMAND_RUN

The run command sends the 32-bit value which contains the address at which the processor should execute from. This enables the processor to execute the program without resetting.

7.5.6 COMMAND_RESET

This is used to tell the boot loader to initiate a reset after downloading a new image to the processor, causing the program to execute from a reset state.

7.7 Download Utility

The way in that the main download sequence executes is as follows:

1. Import the specified .bin file from the on-board SD card to a local buffer.
2. Ping the slave processor to ensure that communications are established.
3. Send the COMMAND_DOWNLOAD structure; one specifies the address where the program should be stored, whilst the size of the program is calculated.
4. Perform a COMMAND_GET_STATUS to ensure the processor has received and processed the command successfully.
5. Providing a successful acknowledgement from the command get status, using the COMMAND_SEND_DATA the utility will begin to send data 251 bytes at a time, until all of the data from the buffer has been sent to the processor.
6. The program will then send the reset command, forcing the processor to execute from a start-up state.

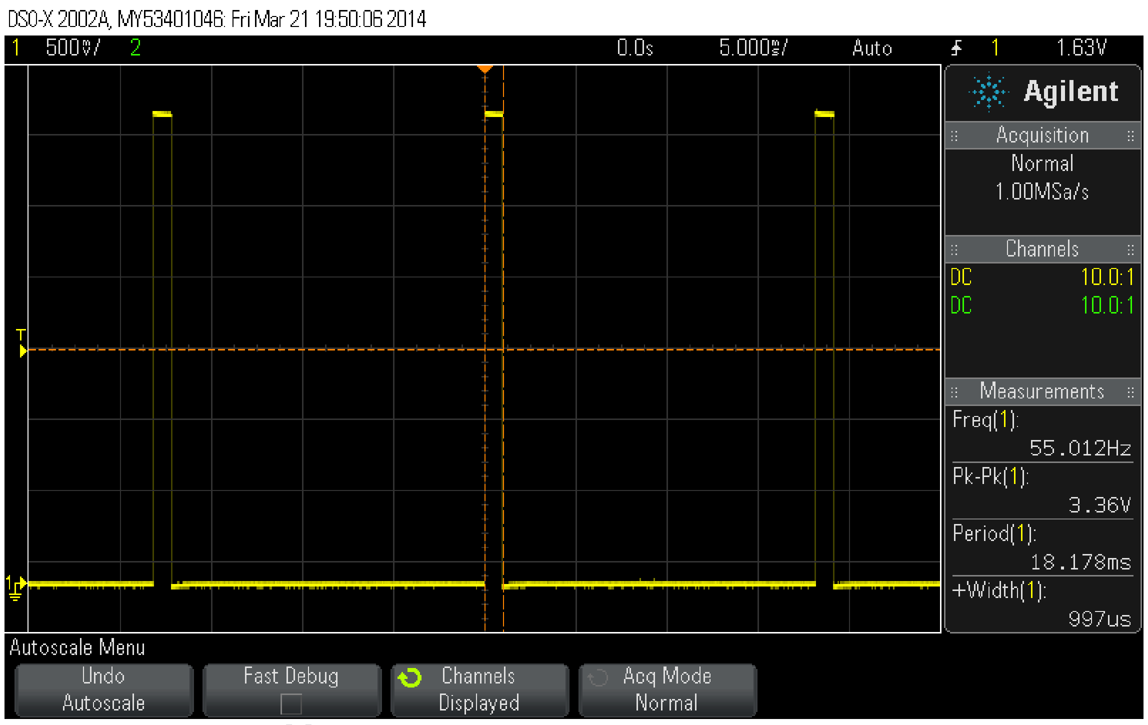
Throughout the main loader applications operation, checks are performed, outputting specific error codes to the console providing the ability to debug a problem if one were to occur.

8 Further Testing of Control Board

8.1 PWM Channels

As processors 2 & 3 will both be used for driving the BLDC motors, it is important that their PWM channels function correctly. As programs are now able be loaded onto the processors without communication difficulties, a program including the TivaWare driver libraries was downloaded to demonstrate that the PWM channels are working.

This program set an arbitrary PWM frequency of 55 Hz, and period of 18.2ms. An oscilloscope was used to probe the signal on the input of the first motor gate driver. The pulse width was varied from 1ms to 2ms as shown in figures [5] and [6]. The accuracy of the pulse width produced from the processor was very accurate, typically ±4µs.


DSO-X 2002A, MY53401046: Fri Mar 21 19:50:06 2014

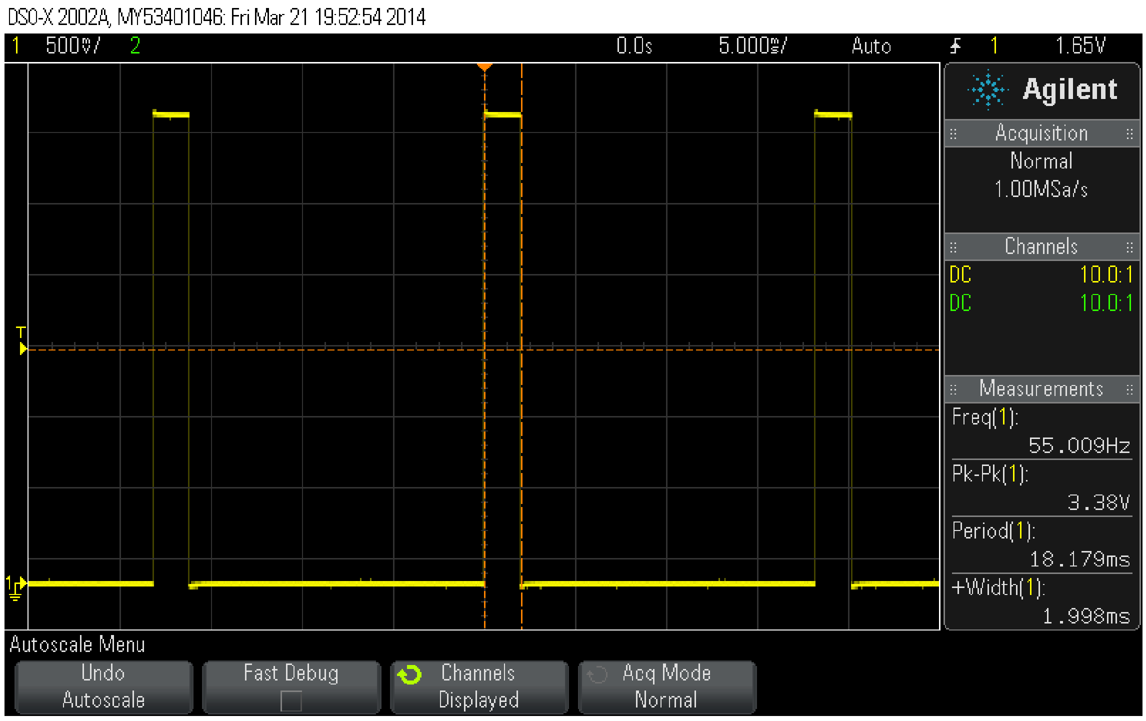
1 500mV/ 2 0.0s 5.000ms/ Auto 1 1.63V

Acquisition: Normal 1.00MSa/s

Channels: DC 10.0:1, DC 10.0:1

Measurements:
Freq(1): 55.012Hz
Pk-Pk(1): 3.36V
Period(1): 18.178ms
+Width(1): 997us

Figure [5] – PWM IC3 channel 1, 55Hz, 18.2ms Period, 1ms Pulse Width


DSO-X 2002A, MY53401046: Fri Mar 21 19:52:54 2014

1 500mV/ 2 0.0s 5.000ms/ Auto 1 1.65V

Acquisition
Normal
1.00MSa/s

Channels
DC 10.0:1
DC 10.0:1

Measurements
Freq(1): 55.009Hz
Pk-Pk(1): 3.38V
Period(1): 18.179ms
+Width(1): 1.998ms

Figure [6] – PWM IC3 channel 1, 55Hz, 18.2ms Period, 2ms Pulse Width

8.2 Quiescent Levels

As the board stands, there are no power devices installed, such as transistors or capacitors. This is purely for the purpose of testing, to ensure that the circuit board functions correctly without causing any unnecessary damage. Before power components can be added, it must be known that the power transistors will be in an off state during all stages of the program loading sequence and boot.

Pins on the TMC123AH6PM processors are configured as GPIO on reset with a value of 0 [16]. This ensures that no signal is applied to the transistor gate drivers; this was confirmed by analysing the logic input to the gate drivers.

9 Conclusions

The original objective was to fully functioning quadrotor that could be introduced into schools which students have the ability to build program and modify. Although this has not been completely met, many of the original objectives have:

9.1 Frame

The outer frame has been fully designed and built with only the wire mesh to be added. The frame fully encloses each rotor blade, in a tough, lightweight and durable material. The electrical components are easily accessible at the centre of the frame, whilst the weight and location of the battery adds to the stability of the overall design.

9.2 Motor Control Board

Testing the motor control board became a significant part of this project, which was initially looked over in the original time plan. Faults with the power supply were identified and solutions are presented in the Recommendations section of this report.

The board is ready for power devices to be added as quiescent levels are known; each pin of the Tiva MPU's is held in a high impedance state by default thus no signal is applied to the FET drivers.

9.3 Programming

The Tiva C series boot loader that runs on each of the processors is used to load programs onto each of the processors on the control board. A Raspberry Pi is used to communicate with the boot loader via a serial interface using a custom loader application written in C. Due to time constraints, a BLDC driver hasn't been developed.

Now that there is a mechanism where users can easily download code to each of the processors on the control board wirelessly, I believe that the development of this project into a fully functioning quadrotor should be relatively straight forward.

10 Recommendations

10.1 Modifications to the Motor Control Board

The main modification that needs to be made to the control board regards the on-board power supply. Currently an ADP2303 nonsynchronous Step-Down regulator is used to supply the 5v rail on the board. The data sheet recommends a TDK VLF10040T-100M3R8, 10μH inductor as well as a Vishay SSB43L, 30V, 4A Schottky Diode [17] which conducts the inductor current throughout the off time of the inductors internal MOSFET. These components are different to the ones originally selected to populate the board. The inductor is able to be replaced easily since both have the same SMT (Surface Mount Technology) footprint; however the new diode's footprint differs due to the increased current rating.

The output capacitors on the two VDDC pins (56 & 25) could also be reduced to a single 3.3μF by first connecting the two pins before earth through the capacitor as recommended in the user guide.

10.2 Serial Loader Application

The loader application is able to function whenever the boot loader on the processor is running, thus either when the flash memory is empty or the boot loader has been called within the program. Currently, the flash is erased manually in order to download programs via the serial interface.

There are two possibilities; either each time the board loses power it triggers an erase of the flash memory. Alternatively a command could be sent to the processors on a serial interface calling the boot loader stored in ROM. I believe that the first solution, that every time that the processor resets due to a loss of power, would be the better choice; the second option would fail if for any reason the processors' serial communications were to become locked, thus the command to call the boot loader would not be received, creating a difficult problem for the end user.

The loader application would have to be modified slightly for this to be implemented, with the COMMAND_RESET replaced with a COMMAND_RUN to prevent the processor erasing its memory every time a program is loaded.

With more time I would further develop the application where all three processors would be loaded in a single sequence, removing the need to manually write the pass through application to processor 1.

10.3 Further Work

In order to produce a fully functioning quadrotor, a BLDC driver to control the four motors needs to be developed using the components employed on the control board. Once a driver has been produced the existing AHRS needs to be ported to run on the Tiva processors including collecting the data provided by the 9-axis sensor chip.

As this project will be introduced to schools, the current serial loader may be modified in order to simplify the application to the end user. The application could be further developed allowing the user to select the three program files on their PC which will be automatically be loaded to the correct processor, without the current need for manual modification of the loader program file.

Due to the powerful electronic components used in this project, it leaves much scope left to be explored. The Raspberry Pi has the potential to be used for much more than just a program loader, with the ability to be accessed via a GSM (Global System for Mobile Communications) gateway. This would enable the RPi and thus quadrotor to be accessed and controlled remotely from a computer or even using an app on a handheld smart device running iOS or android software. Waypoint tracking could be introduced with the addition of a GPS module, whilst a camera could be used to further improve the stability of the AHRS.

This further potential capability could be introduced with the quadrotor to schools, where students would follow tutorials implementing further hardware or software modifications whilst teachers would explain the theory behind the new mechanisms. This would enable teachers to use the robot as a learning tool for larger parts of the educational syllabus.

11 References

[1] Z. Sarris, “Survey of UAV Applications In Civil Markets,” *STN ATLAS-3Sigma AE and Technical University of Crete,* 2001.

[2] J. Gurney-Read, “UK universities facing tough global competition in STEM subjects,” The Telegraph, 26 Febuary 2014. [Online]. Available: http://www.telegraph.co.uk/education/educationnews/10661330/UK-universities-facing-tough-global-competition-in-STEM-subjects.html. [Accessed 18 April 2014].

[3] J. Germanos, “Promoting STEM in Schools, Budget Hearings,” Connection Newspapers, 16 April 2014. [Online]. Available: http://www.connectionnewspapers.com/news/2014/apr/16/promoting-stem-schools-budget-hearings/. [Accessed 18 April 2014].

[4] M. Watson, “The Design and Implementation of a Robust AHRS for Integration into a Quadrotor Platform,” Sheffield, 2013.

[5] Depron Foam, “Technical Details - Depron,” 2009. [Online]. Available: http://www.depron.co.uk/technical.htm. [Accessed 16 04 2014].

[6] Texas Instruments Incorporated, “Tiva TM4C123AH6PM Microcontroller,” 2013.

[7] InvenSense Inc., “MPU-9150 Product Specification Revision 4.3,” 2013.

[8] HobbyKing, “Flightmax 500mAh 20C Li Pro,” Zippy, [Online]. Available: https://www.hobbyking.com/hobbyking/store/__8580__ZIPPY_Flightmax_5000mAh_4S1P_20C.html. [Accessed 16 04 2014].

[9] H. U. M. K. T. E. A. K. M. Kenichi Iizuka, “Microcomputer Control for Sensorless Brushless Motor,” *IEEE Transactions on Industry Applicatons,* Vols. IA-21, No.4, pp. 595-601, 1985.

[10] Texas Instruments, “TivaWare™ for C Series (Complete),” 2014. [Online]. Available: http://www.ti.com/tool/sw-tm4c. [Accessed 16 04 2014].

[11] S. F. B. a. D. J. Pack, “Atmel AVR Microcontroller,” Morgan & Claypool, 2008, pp. 34-35.

[12] element14, “ELEMENT14 WIPI MODULE, WIFI, USB, FOR RASPBERRY PI,”

element14, 2014. [Online]. Available: http://uk.farnell.com/element14/wipi/dongle-wifi-usb-for-raspberry-pi/dp/2133900?ref=lookahead. [Accessed 19 April 2014].

[13] Texas Instuments, “TI E2E Community - CSS Uniflash 3.0,” 2014 Febuary 14. [Online]. Available: http://e2e.ti.com/support/development_tools/code_composer_studio/f/81/t/321508.aspx. [Accessed 24 April 2014].

[14] T. Cooper, “Adafruit WebIDE,” Adafruit Industries, New York, 2014.

[15] MontaVista Software, Inc, “SPI testing utility (using spidev driver),” 2007. [Online]. Available: https://www.kernel.org/doc/Documentation/spi/spidev_test.c. [Accessed 16 04 2014].

[16] Texas Instruments, “Tiva TM4C123AH6PM Microcontroller Data Sheet,” Austin, TX, 2013, p. 1163.

[17] Analog Devices, “ADP2302/ADP2303,” in *Nonsynchronous Step-Down Regulators*, Norwood, MA, 2010, p. 18.

12 Appendices

12.1 spi_handler.h

```
//
//  spi_handler.h
//  bootloader
//
//  Created by Alex Stevens on 17/02/2014.
//

#ifndef bootloader_spi_handler_h
#define bootloader_spi_handler_h
#include "stdint.h"

#define COMMAND_PING                0x20
#define COMMAND_DOWNLOAD            0x21
#define COMMAND_RUN                 0x22
#define COMMAND_GET_STATUS          0x23
#define COMMAND_SEND_DATA           0x24
#define COMMAND_RESET               0x25

#define COMMAND_RET_SUCCESS         0x40
#define COMMAND_RET_UNKNOWN_CMD     0x41
#define COMMAND_RET_INVALID_CMD     0x42
#define COMMAND_RET_INVALID_ADDR    0x43
#define COMMAND_RET_FLASH_FAIL      0x44
#define COMMAND_ACK                 0xcc
#define COMMAND_NAK                 0x33


uint32_t SpiClosePort(int);
uint32_t SpiOpenPort(int);
uint32_t SpiWriteAndRead(uint8_t *pui8DataOut, uint8_t *pui8DataIn, uint8_t ui8Size);



#endif
```

12.2 spi_handler.c

```
//
//  spi_handler.c
//  bootloader
//
//  Modified by Alex Stevens on 17/02/2014.
//
//
 * SPI testing utility (using spidev driver)
 *
 * Copyright (c) 2007  MontaVista Software, Inc.
 * Copyright (c) 2007  Anton Vorontsov <avorontsov@ru.mvista.com>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License.
 *
 * Cross-compile with cross-gcc -I/path/to/cross-kernel/include
 //

#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/spi/spidev.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int spi_cs0_fd;
int spi_cs1_fd;
unsigned char spi_mode;
unsigned char spi_bitsPerWord;
unsigned int spi_speed;



//
// Open Spi Port
// uses the spi_dev library to initialise communications
//

uint8_t SpiOpenPort (int spi_device)
{
        int status_value = -1;
    int *spi_cs_fd;

    spi_mode = SPI_MODE_0;

    // 8 Bits per word
    spi_bitsPerWord = 8;

    // SPI Speed
    spi_speed = 100000;                    //1kHz
```

```
   if (spi_device)
       spi_cs_fd = &spi_cs1_fd;
   else
       spi_cs_fd = &spi_cs0_fd;


   if (spi_device)
       *spi_cs_fd = open("/dev/spidev0.1", O_RDWR);
   else
       *spi_cs_fd = open("/dev/spidev0.0", O_RDWR);

   if (*spi_cs_fd < 0)
   {
      perror("Error - Could not open SPI device");
      exit(1);
   }

   status_value = ioctl(*spi_cs_fd, SPI_IOC_WR_MODE, &spi_mode);
   if(status_value < 0)
   {
      perror("Could not set SPIMode (WR)...ioctl fail");
      exit(1);
   }

   status_value = ioctl(*spi_cs_fd, SPI_IOC_RD_MODE, &spi_mode);
   if(status_value < 0)
   {
      perror("Could not set SPIMode (RD)...ioctl fail");
      exit(1);
   }

   status_value        =        ioctl(*spi_cs_fd,        SPI_IOC_WR_BITS_PER_WORD,
&spi_bitsPerWord);
   if(status_value < 0)
   {
      perror("Could not set SPI bitsPerWord (WR)...ioctl fail");
      exit(1);
   }

   status_value        =        ioctl(*spi_cs_fd,        SPI_IOC_RD_BITS_PER_WORD,
&spi_bitsPerWord);
   if(status_value < 0)
   {
      perror("Could not set SPI bitsPerWord(RD)...ioctl fail");
      exit(1);
   }

   status_value = ioctl(*spi_cs_fd, SPI_IOC_WR_MAX_SPEED_HZ, &spi_speed);
   if(status_value < 0)
   {
      perror("Could not set SPI speed (WR)...ioctl fail");
      exit(1);
   }
```

```
    status_value = ioctl(*spi_cs_fd, SPI_IOC_RD_MAX_SPEED_HZ, &spi_speed);
    if(status_value < 0)
    {
        perror("Could not set SPI speed (RD)...ioctl fail");
        exit(1);
    }
    return(status_value);
}

//
// Close Port
//

uint8_t SpiClosePort (int spi_device)
{
         int status_value = -1;
    int *spi_cs_fd;

    if (spi_device)
         spi_cs_fd = &spi_cs1_fd;
    else
         spi_cs_fd = &spi_cs0_fd;


    status_value = close(*spi_cs_fd);
    if(status_value < 0)
    {
         perror("Error - Could not close SPI device");
         exit(1);
    }
    return(status_value);
}


//
// Read / Write - Reads from DataOut to DataIn
//

int32_t SpiWriteAndRead (unsigned char *pui8DataOUT, unsigned char *pui8DataIN,
unsigned int ui8Size)
{
         struct spi_ioc_transfer spi[ui8Size];
         int i = 0;
         int retVal = -1;
    int *spi_cs_fd;

    spi_cs_fd = &spi_cs0_fd;

         //one spi transfer for each byte

         for (i = 0 ; i < ui8Size ; i++)
         {
            spi[i].tx_buf       = (unsigned long)(pui8DataOUT + i); // transmit from "DataOut"
            spi[i].rx_buf       = (unsigned long)(pui8DataIN + i) ; // receive into "DataIn"
            spi[i].len          = sizeof(*(pui8DataOUT + i)) ;
        spi[i].delay_usecs   = 0 ;
        spi[i].speed_hz      = spi_speed ;
        spi[i].bits_per_word = spi_bitsPerWord ;
        spi[i].cs_change = 0;


    }

    retVal = ioctl(*spi_cs_fd, SPI_IOC_MESSAGE(ui8Size), &spi) ;

    if(retVal < 0)
    {
        perror("Error - Problem transmitting spi data..ioctl");
        exit(1);
    }

    return retVal;

}
```

12.3 file_util.c

```
//
//  read_file.c
//  bootloader
//
//  Created by Alex Stevens on 01/03/2014.
//

#include <stdio.h>
#include <stdlib.h>
#include "spi_handler.h"

//
// Buffer Declarations
//

#define BUFFER_SIZE 2048

static unsigned char g_pucBuffer[256];
static unsigned char ucDataIN[BUFFER_SIZE];


unsigned char ucImage[BUFFER_SIZE*2];
unsigned long ulImage_size;



//
// Delay Function
// (about 10ms)
//

void delay(void){
    static volatile unsigned int val;
    val = 10000000;
    while (val--);
}

//
// Read File
// reads input file and places it into image buffer
//

void read_file(char *filename){

    FILE *file;

    int trans_no = 0;
    unsigned char *pucImage;

    ulImage_size = 0;
    pucImage = &ucImage[0];
    // Open the file
    file = fopen(filename, "rb");
    if (!file) {
        printf("Import Failed '%s'\n", filename);
        return;
    }

    // While not at the end of the file
    while (!feof(file)) {

        int write_no;
        int i;

        // Read from file
        write_no = fread((void *)pucImage, 1, BUFFER_SIZE, file);
        if (!write_no) break;

        // Set Global Variable
        ulImage_size += write_no;
        pucImage += write_no;
        ++trans_no;

        delay();

    }

cleanup:
    fclose(file);


}

//
// Checksum Buffer
// Returns the calculated value of the checksum
//

unsigned char Checksum_buffer (unsigned char *pucBuffer, unsigned short ulSize)
{
    unsigned int i;
    unsigned int sum;
    unsigned char checksum;

    // For loop, increment by i & add value to sum
    for (i=0, sum=0; i < ulSize; i++) {
        sum += pucBuffer[i];
    }

    checksum = ((unsigned char)sum & 0xFF);

    return (checksum);

}

//
```

```
// Send acknowledgement packet
//

void Ack(void)
{

   unsigned char pucBuffer[2] = {0, COMMAND_ACK};
   unsigned int cout;

   cout = SpiWriteAndRead(pucBuffer, ucDataIN, 2);


}

//
// Send non-acknowledgement packet
//

void NAck(void)
{

   unsigned char pucBuffer[2] = {0, COMMAND_NAK};
   unsigned int cout;

   cout = SpiWriteAndRead(pucBuffer, ucDataIN, 2);


}

//
// Read from the device
//

unsigned int read_packet (unsigned char *pucBuffer, unsigned short *pulSize)
{
   unsigned char ulCheckSum;
   unsigned int cout;
   unsigned char zero[16] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

   *pulSize = 0;

   while (*pulSize == 0)
   {
      // Send out a zero, write to pulSize
      SpiWriteAndRead(zero, (unsigned char *)pulSize, 1);
   }

   *pulSize -= 2; //Reduce by 2 for size & checksum

   // Read Checksum
   // Send zero, receive value into ulCheckSum
   cout = SpiWriteAndRead(zero, (unsigned char *)&ulCheckSum, 1);

   // Read Data
```

```
    // Send pulSize no of zeros and receive into pucBuffer
    cout = SpiWriteAndRead(zero, pucBuffer, *pulSize);

    // Check that the checksum of the received data & compart to sent checksum
    if(Checksum_buffer(pucBuffer, *pulSize) != ulCheckSum)
    {
        NAck(); // Send non-ack packet
    }
    else
    {
        Ack(); // Send ack packet
    }


    return (1);

}

//
// Send to the device
//

unsigned int send_packet (unsigned char *pucBuffer, unsigned short ulSize)
{

    unsigned char ucCheckSum;
    unsigned short i;
    unsigned short length;
    int count;
    unsigned char zero[8] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};


    // Calculate checksum
    ucCheckSum = Checksum_buffer(pucBuffer, ulSize);


    // Increment packet size by 2 to account for packet size & checksum bytes
    ulSize += 2;

    printf("\n ulSize: %02x ", ulSize);

    printf("\nData sent:");
    // Send the first byte (size)
    count = SpiWriteAndRead((unsigned char *)&ulSize, ucDataIN, 1);
    printf("\n %02x ", ulSize);

    // Send the second byte (checksum)
    count = SpiWriteAndRead((unsigned char *)&ucCheckSum, ucDataIN, 1);
    printf("\n %02x ", ucCheckSum);

    // Send the rest of the packet data
    ulSize -= 2;

    count = SpiWriteAndRead(pucBuffer, ucDataIN, ulSize);
    for (i=0; i < ulSize; ++i) {
        printf("\n %02x ", pucBuffer[i] & 0xff);
    }
    printf("\n");

    length = 1;

    while (ucDataIN[0] == 0)
    {
        count = SpiWriteAndRead(zero, ucDataIN, length);

        // Check correct amount of bytes has been recieved
        //
        if (count != length) {
            printf("recieved %d, expected %d)\n", count, length);
        }


    }


    if (ucDataIN[0] != COMMAND_ACK) {
        return(0);
    }


    return (1);

}

//
// Download File util
//

unsigned int DownloadUtil(unsigned char *pucBuffer, unsigned int ulAddress, unsigned
int ulLength)
{
    unsigned short ulOffset;
    unsigned char i;

    printf("\n ulLength: %02x ", ulLength);

    // Download Command Structure
    //
    g_pucBuffer[0] = COMMAND_DOWNLOAD;
    g_pucBuffer[1] = (ulAddress >> 24) & 0xff;
    g_pucBuffer[2] = (ulAddress >> 16) & 0xff;
    g_pucBuffer[3] = (ulAddress >> 8) & 0xff;
    g_pucBuffer[4] = ulAddress & 0xff;
    g_pucBuffer[5] = (ulLength >> 24) & 0xff;
    g_pucBuffer[6] = (ulLength >> 16) & 0xff;
    g_pucBuffer[7] = (ulLength >> 8) & 0xff;
    g_pucBuffer[8] = ulLength & 0xff;

    printf("\nBuffer: \n");
```

```
for (i=0; i < 9; ++i) {
    printf("\n %02x ", g_pucBuffer[i] & 0xff);
}
printf("\n");


// Check for errors
//
if (!send_packet(g_pucBuffer, 9)) {
    printf("Download Command Failed \n");
    return(0);
}

delay();
g_pucBuffer[0] = COMMAND_GET_STATUS;
if (!send_packet(g_pucBuffer, 1)) {
    printf("Get Status Command Failed \n");
    return(0);
}

if (!read_packet(g_pucBuffer, &ulOffset)) {
    printf("Status packet not recieved \n");
    return(0);
}

if((ulOffset != 1) || (g_pucBuffer[0] != COMMAND_RET_SUCCESS))
{
    printf("Status isnt Sucess\n");
    return(0);
}

// Send Data Command which follows the Command Download
// allows up to 241 bytes to be transmitted at a time

for(ulOffset = 0; ulOffset < ulLength; ulOffset += 240)
{
    g_pucBuffer[0] = COMMAND_SEND_DATA;
    for(i = 0; i < 240; i++)
    {
        g_pucBuffer[i + 1] = pucBuffer[ulOffset + i];
    }

    // Check for valid data

    if((ulOffset + i) > ulLength)
    {

        printf("Final packet: ");
     printf("%04x ", ulLength - ulOffset);
                    delay();
                    if(!send_packet(g_pucBuffer, (ulLength - ulOffset + 1)))
        {
            printf(" Send Data Command Failed\n");
            return(0);
        }
        }
        else
        {
            // Send the command with rest of Data
            delay();
            if(!send_packet(g_pucBuffer, 241))
            {
                printf(" Send Data Command Failed\n");
                return(0);
            }
        }
    }


    return(1);
}

int main(void){

    SpiOpenPort (0);


    unsigned int error;
        unsigned long ulAddress;

    char *filename;
    filename = "IC2.bin";
    read_file(filename);

    // Run Adress
        ulAddress = 0x00000;


    g_pucBuffer[0] = COMMAND_PING;

    // PING the MPU
    printf("\nSending the ping command...\n");
    if(!send_packet(g_pucBuffer, 1))
    {
        printf("Ping failed \n");
        return(0);
    }

    // Send DOWNLOAD Structure & Data
    delay();
    printf("\nSending the download command...\n");
    if(!DownloadUtil(ucImage, ulAddress, ulImage_size))
    {
        printf("Download failure \n");
        return(0);
    }

    delay();
    // Send RUN structure
    printf("\nSending the run command...\n");
    g_pucBuffer[0] = COMMAND_RUN;
    g_pucBuffer[1] = 0x00;
    g_pucBuffer[2] = 0x00;
    g_pucBuffer[3] = 0x00;
    g_pucBuffer[4] = 0x00;
    delay();
    if(!send_packet(g_pucBuffer, 5))
    {
        printf("Run failure \n");
        return(0);
    }


    // Send the RESET command.
    g_pucBuffer[0] = COMMAND_RESET;
    delay();
    if(!send_packet(g_pucBuffer, 1))
    {
        printf("Reset failure \n");
        return(0);
    }


    SpiClosePort (0);
    return (0);

}
```

12.4 Serial Loader Code

The above code is also included on a CD with this report.

12.4 Interim Report

This is a design project with the overall objective of constructing a fully operational quad-rotor platform, which can be controlled remotely. The project consists of three main parts: design and fabrication of the mechanical frame, the electrical control hardware and the programming. The project has a large focus on safety, as the final product will be aimed to be introduced to schools and thus be used by children of a relatively young age. Therefore at every stage of design safety will be carefully considered.

This project builds on a previous students design of a functioning quadrotor that used individual motor control and sensor boards connected to a Raspberry Pi acting as the master processor running Debian Linux. This project will combine all of these peripheral boards and sensors used previously onto a single PCB, designed specifically for this 4 axis motor application. This PCB board contains 3 ARM Cortex-M4 MPU's; two of which will be used to drive the four brushless DC motors whilst the third will act as the master and control the overall communications. A 9-axis MPU that contains both a 3-axis gyroscope and a 3-axis accelerometer is used in order to provide real time data on location, speed and direction communicating via the I^2C bus to the master processor. The Raspberry Pi will run Raspbian OS and connect to the motor control board via a Serial Peripheral Interface (SPI) bus and will be used to load executable program files onto the Cortex M4 processors enabling; a easy mechanism of programming and updating code.

The two MPU's controlling the Brushless DC (BLDC) motors will be identical apart from which motors they govern. They are connected by I^2C and SPI busses to the master processor. Each has 8 PWM channels, three of which will be initially used in this project to drive the three phases of a single BLDC motor. The motors work by applying a voltage across two of the phases whilst using the third phase to measure back EMF allowing the zero crossing point to be calculated and thus the timing of the next commutation cycle determined. Current sensors are used to monitor back EMF and are connected to the slave processors as an analogue input.

This project, once complete, will ideally be used as a tool for children to be introduced to electrical & electronic engineering during school. They will have the opportunity to write small segments of code, load it onto the processors and see the immediate results, hopefully encouraging interest in electronics.

Specification:

Electronic & Programming

- Executable files can be directly loaded to the MPU's on the control board by using the Raspberry Pi running raspbian OS.
- The control PCB will use real time measurements and apply user input enabling the quadrotor to be directed.
- Two of the MPU's will drive four motors
- Brushless DC motors without hall sensors will be used to power the rotor blades
- MPU's will communicate with one and another using the SPI bus

Frame

- The frame should be easily to manufacture, using readily available materials and equipment
- The frame should be as lightweight and durable as possible
- The frame should be able to sustain a reasonably harsh crash without failing
- The frame should enclose all moving parts whilst allowing access for maintenance easily.

Safety

- All motors should be able to be stopped at an instance using a kill switch
- If a crash were to occur, all motors should be stopped and not restart.

The overall frame of the quadrotor has been designed. Multiple materials were extensively researched to find the ones that exhibited a high durability whilst being lightweight. These properties are extremely important in RC aircraft such as this quadrotor, so that its weight doesn't affect the overall flight characteristics. If a crash were to happen (extremely probable when being used by children) the frame isn't rendered useless as a result of damage. Taking this into consideration I reduced the selection to two materials: Expanded Polypropylene (EPP) and Depron; a closed cell polystyrene sheet, both of which are currently used considerably in the model aircraft industry.

Depron is lighter than EPP but is more ridged and as a result more susceptible to failure if the aircraft were to crash. EPP however is extremely difficult to machine and requires a hot wire cutter alongside other specialist materials to apply a simple finish whereas the rigidity of Depron allows it to be machined simply. For these reason Depron was selected as the material for the frame of the quadrotor.

The frame was designed such that it enclosed all four rotor blades whilst allowing for the electronic boards and equipment (such as batteries) to be positioned in the centre creating an aesthetically pleasing product, unlike a lot of the products currently in the industry. This specification resulted to an outer frame thickness of 70mm to cover the motor and blades. The motors will be held in place by an aluminium cross frame where the centre will be a plate on which the electronics will be mounted. The motors selected are the same as the previous project (Turnigy L2215J-900 motors), these are brushless dc, 200w drawing a peak of 18A. Depron is only commercially available in thin thicknesses so the frame was built by layering 9mm sheets.

The top side of the PCB board components were populated; this included all three processors, and motor drivers. The power regulators were also added to the underside of the board. A large amount of time has been spent learning how to communicate with the processors. The TM4C MPU's are relatively new processors alongside the Texas Instrument (TI) IDE Code Composer Studio (CSS).

The TM4C processors have a peripheral driver library written for them included in "TivaWare" software available from TI. Thus far I have been communicating with the processor via JTAG using a XDS100 emulator. Problems have been occurring when writing reasonable size files to the RAM resulting in the processor becoming locked up and preventing all communications with a computer.
Initially I believed this to be a result of the JTAG pins somehow being incorrectly mapped in the code and as a result the debug probe not recognizing the JTAG signals of one that is a processor. After reading through the TI manual for the processor it became apparent that a SWD/JTAG switching command needed to be transmitted for at least 50 cycles in order to 'Unlock' the JTAG port. TI has a

separate utility that includes a tool to perform this sequence (CSS Uniflash) where one asserts the reset pin on the processor low. This however did not restore communications with the processors and my attention turned to other parts of the board.

Whilst going through the overall schematic of the processors, the VDDC pin connections did not match those in the TI user manual, where it states that the VDDC lines should be tied together and then connected through an external capacitor in the range of 2.5-4 μF where in the original design they are connected through two 3.3μF capacitors resulting in a 6.6μF capacitance. It is an easy fix as one of the capacitors can be removed; however believe this to be negligible and not the real cause of the processor communication failure.

When attempting to reset the internal flash memory by performing a “power on reset” one of the processors was supplied with a high voltage and as a result blown. This also took out the power regulators. The processor & regulators were removed from the board and an external bench supply (3.3v) was used to power the board whilst trying to further investigate the processor communication problems.

The flash memory was able to be reset using external power supply restoring communications with both remaining processors on the board. This implies that there is a problem with the switch mode power supply and/or the 3.3v regulator design, which is where the project is at currently.

Now that the communications have been restored with two processors this enables me to write and debug code using CSS incorporating the TivaWare libraries. Thus far I have managed to set up the clocks, interrupts and timers and began the SPI communications between processors.

Below is a revised Gantt chart. I have included row for literature review, which was initially looked over. This is to account for time needed to read about the TI software and TivawWare functions etc.

Component	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Frame Design	▨	▨	▨	▨	▨	▨								▨	▨	▨	▨	
Literature Review						▨	▨	▨	▨	▨	▨	▨	▨					
Programming						▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨		
Overall Project																▨	▨	▨