Queen's University Belfast

School of Electronics, Electricals Engineering
and Computer Science

Development of an Unmanned Marine Vehicle

BEng Final Year Project – Final Report

William Parr

16th April 2018

Statement of Academic Integrity

By submitting this report electronically or physically to the School of EEECS for assessment I declare that:

- I have read the University regulations on plagiarism, and that the attached submission is my own original work except where clearly identified.
- I have acknowledged all written and electronic sources used.
- I agree that the School may scan the work with plagiarism detection software.

Abstract

The following report, details the progress of the final year project *'Development of an Unmanned Marine Vehicle*' and aims to provide information regarding the history of the project at Queen's University Belfast, a broader scope of information regarding the history of submersible vehicle development and possible future technologies that may find application in this field. Additionally, the report aims to overview the planning regarding the project, it's involved risks as well as provide extensive documentation of the hardware and software designed in the development cycle for a prototype submersible.

The end goal of the project itself, is the creation of an **Autonomous Underwater Vehicle** (AUV) implemented on a provided submarine frame (supplied and created by the University). The frame possesses four 'Rule Bilge Pumps' which have been repurposed into four shrouded propellers (with matched propeller types installed to achieve a state of neutral thrust).

This frame will be controlled by a Raspberry Pi 3B, with power supplied via a lithium polymer battery – with additional hardware created to allow for modularity. Features included are: wireless control, a suite of inertial sensors, pressure (and thus depth) sensing and a dedicated user interface to enable simplified control of the unit.

The software for the controller is written in Python, to enable the software to be easily understood by a third party – and thus ensure the continuality of the project, as it would be possible to pass the software onto future project owners with little hassle. Especially when considering the object-orientated nature of python, supporting features such as class inheritance, which would be of great assistance in future versions – thus allowing for much more functionality to be implemented on the AUV than there has been previously, since the software and hardware design require little revision.

Specification

The specification outlined, by the university, for the titular project is as follows [1]:

Unmanned marine vehicles are now being used in a variety of tasks such as pollution tracking, surveillance operations and mines clearing to name a few. In this project, a robotic submarine which has been developed earlier as part of a previous final year project will be further developed, evaluated and tested. The vehicle, consisting of an embedded controller and a host of sensor suite will be made available for the student. A model of the vehicle is first formulated followed by the development of an advanced autopilot which will be implemented on the vessel. Matlab/Simulink package may be used for modelling and simulating the dynamics of the vessel whilst C++ may be used for real-time implementation.

Expected Procedure:

- To gain an understanding of unmanned systems
- To further develop and test a prototype robotic submarine. This includes both the physical system and related electronics.
- To develop a mathematical model of the provided robotic submarine.
- To develop and test control algorithms (autopilots) on the vehicle.

It should additionally be noted that in agreement with the previous owner of the project, the author intends on developing continuality on the project through the usage of the Python programming language; which is inherently simple to read and understand, enabling the software to be passed to the next iteration of the project without much confusion.

Acknowledgements

It should be noted that much of the progress of this project could not have been achieved without the assistance of many persons: Firstly, Wasif Naeem for setting up the project and thus enabling students such as the author to undertake such a project (which combines aspects of hardware, software and control systems); secondly the members of the Ashby 9th floor workshop staff: Gerry Rafferty, Jim Norney and Ryan McAliskey, who all were of great assistance in the development of the project; and finally Ché Cameron (as well as the other previous holders of the project) whose previous work was of great assistance in moving forwards, as well as his personal assistance in brainstorming and advice when developing the hardware and software of this project.

Table of Contents

1 Introduction

1.1 Project Motivation

Within the modern world, the author believes it to be essential robust and functional Autonomous Underwater Vehicles (or AUVs) exist. Given the nature of our planet, being mostly water-covered (with roughly 71% of the earth's surface being covered in water [2]). However, it is not which is on the surface of the earth that is of interest; given this has already been explored many times over, but rather it is what remains below the surface which is of value (given that in the year 2000, the National Oceanic and Atmospheric Administration (or NOAA) estimated that 95% of the planet's ocean, and 99% of the ocean floor are still unexplored [3]). Thus, it is essential to those of a scientific background that a method of exploring these areas be developed; especially when one considers what mysterious lifeforms and natural wonder's lie in the dark below our planet's vast bodies of water.

Yet alas, given the nature of the ocean; exploration at great depths is no simple task, as many factors are at play against human exploration below the surface: namely, the large pressure values which human bodies cannot operate at, the difficulty of communications underwater and the difficultly of navigating underwater.

One of the largest issues with deep sea exploration is of course the fragility of the average human body (being unable to dive greater than 30 meters of depth for an extended period without adverse effects [4]) and alas the solution to this problem, is the development of an alternative vessel – a submarine – to enable humans to explore at great depths. However, what happens when this vessel fails? Due to the nature of the human body, if the submarine were to be damaged (as it could be easily in combat or exploratory scenario) then a great loss of life could be experienced. Thus, the obvious solution, would be to make these vessels autonomous; to be able to function without a human present to control the vessel.

The next issue lies within the difficulty of underwater communications, which is the result of the most common form of communication (radio waves) having difficulty propagating through a medium of water [5]. Thus, alternative forms of communication must be implemented, most of which are more primitive in terms of bandwidth and speed of transfer. Some examples of these include: Acoustic Communication, Optical Communication (using green and blue wavelengths due to the water penetration of these types of light [6]), Low Frequency Communication (which requires a very large antenna [7]) and Float Antenna Communication (which merely connects the submerged vehicle to an antenna on the surface, where communication can occur normally).

The final issue outlined, difficulty of navigating underwater can pose a significant issue as underwater there is little that can be done to determine one's whereabouts' outside of the usage of 'Dead Reckoning [8]' – which is the usage of a previously recorded location, to determine the present

location. Thus, Inertial Navigation Units (IMU's) are essential to a submarine, as it would enable an AUV to determine its current position, it's start position, and its desired position; which would enable the vehicle to operate entirely autonomously.

1.2 Previous Development at QUB

Previous development of AUV's at Queen's University Belfast has occurred previously over a period of several years. The initial project was based upon an MBED controller, which drove the AUV's motors via an array of solid-state relays. The controller was later updated to an Intel Edison, but the hardware used to control the vehicle remained the same. Afterwards, the project was later redesigned by Ché Cameron (currently a PhD student at QUB), who ultimately adopted the Raspberry Pi 3B as controller and introduced several new features to the AUV (including PWM driven motors, a much more accurate compass and wireless control). However, this iteration was not without its limitation; the most notable of which being the coherence of the software – given it was written in both Python and C++ languages, and formed an extensive codebase. Thus, the author observed that this would present an issue the continuality of the project; the software must be written in a style which is coherent – which is short, simple and easy for a third party to understand. Further, there were some issues observed within the hardware previously designed, thus this had to be redesigned (and done so in an E-CAD software which is freely available, to enable the hardware designs to be passed forward with ease).

1.3 Current Development

In this iteration of the project, the AUV hardware was redesigned – utilising Eagle, a freely available software and based upon the previous work of Ché Cameron. As such, the Raspberry Pi 3B was also selected as the controller for the AUV unit (as the hardware was originally designed for Raspberry Pi platform, and due to its prominence with 'maker' culture – which by extension provides a vast amount of community support).The sensors selected to interface the controller were the Adafruit BNO055 nine degrees of freedom (NDOF) chip and a Honeywell absolute pressure sensor, which were capable of providing control of the AUVs direction and depth. The system is powered by a 14.4 V, four cells, Lithium Polymer (Li-Po) battery; manufactured by 'Zippy' – a brand known within Li-Po for premium grade batteries for a low budget. Software for the AUV was designed in pure Python 3 – which although presents a slightly slower execution time than a compiled alternative (a drop off which is barely noticeable with well written software). However, this one downside was largely outweighed by the many merits of Python; given its status as an interpreted language, with high abstraction. Thus, it is possible to write a software in a far shorter number of lines (when compared to a compiled programme with the same function) and additionally a high level of readability with be retained throughout – thus making python ideal for short term projects, with multiple owners, as it is very much a prototyping language that requires very short development cycles. Moreover, even poorly

written code still retains a high level of coherence – making Python an ideal language for those who struggle with software, and thus the perfect choice for a project which may be adopted by an individual with poor software skills in future.

2 Literature Review

2.1 Historical Submarines

Regarding the history of Submarines, and by extension AUVs, the concept is argued to predate the first successful submergible vehicle – evidence of this concept is first historically recorded, in a legend, from 12th century Egypt, which claimed that Alexander the Great [10] performed reconnaissance in a simple submergible object: A glass diving bell. As depicted below in 2.1.1 – which displays an Islamic painting describing such a feat.

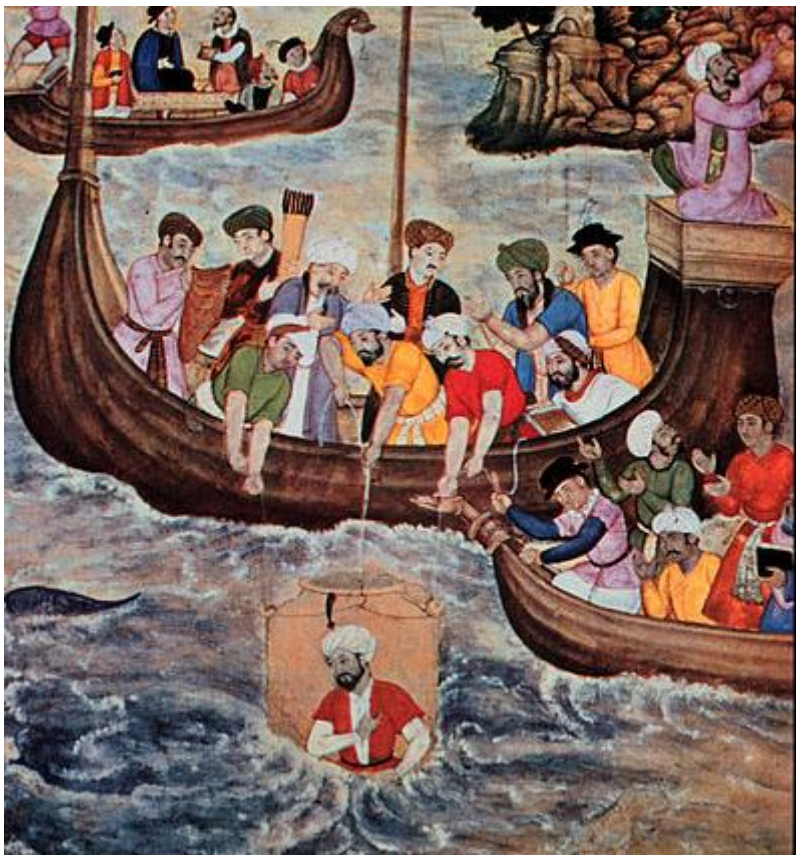


Figure 2.1.1, Painting of Alexander the great, being submerged in a glass diving bell

However, such a primitive method cannot be claimed as equivalent to modern Submarines – as it lacks the ability to freely move within the medium of water; it possesses no means of thrust, or the means to dive and submerge without the user's weight and the force exerted via ropes of those at the surface. To this end, it may be questioned when was the first design of a vehicle possessing such features designed? A question to which the answer may be obtained in the work of William Bourne [11], dated in the year *1578*, in which evidence may be found of an early submarine prototype – operated by crank which adjusted the capacity of two chambers, which filled with water, to determine the submersion depth of the craft. This design is displayed in 2.1.2.

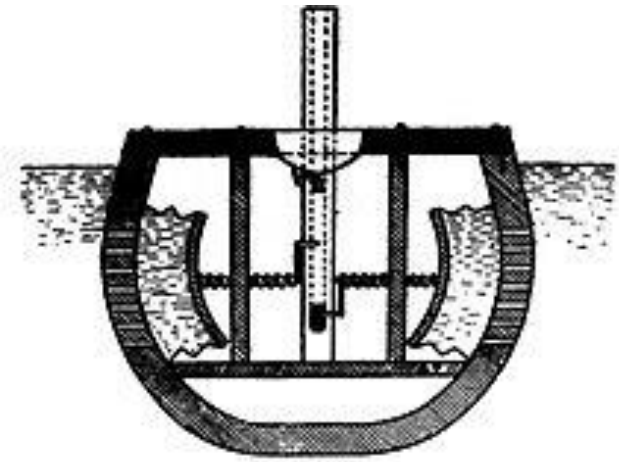


Figure 2.1.2, William Bourne's prototype submarine design

In continuation of this proposed design, the first submergible vehicle was invented by Magus Pegel [12] – a craft to which unfortunately little credit was attributed due to it's lack of feasibility. Regarding the first successful submergible vehicle, this is accredited to Cornelis Drebbel [13]; who built not one, but three submarines – each larger than the previous – under the patronage of James 1st of England. The largest of which could carry 16 individuals, 12 of whom were tasked with propelling the vehicle via oars. The vehicle was capable of submerging using leather 'bladders' contained within the vehicle, which were initially tied off with role (to prevent diving) – in the case of a dive, it was possible to do so, by removing the rope, allowing the 'bladders' to fill. And on the contrary, in the case of surfacing, this was possible by commanding the oarsmen to squash the bladders – a convenient task given, their placement directly under the seats of these men. Additionally, it should be noted that there are some records which mention that Drebbel possessed the means of creating oxygen while submerged; which is largely discredited, due to the period (being in the 1600s), however some believe the plausibility of such claims due to his knowledgeable background with chemicals and their reactions. Of course, recording of this may be observed within Drebbel's own work – in which he states that using saltpetre and fire, it was possible to create fresh air. Additionally, due to the nature of a submergible vehicle, it was required that Drebbel could determine the depth – else the vehicle could possibly become damaged and sink. To this end, it was recorded that Drebbel used a quicksilver barometer (a long tube of quicksilver, in this case) to achieve this. In 2.1.3, a reconstruction of one of Drebbel's vessels is displayed (in this case, one possessing only four oarsmen.)

Figure 2.1.3, the world's first submarine – the Drebbel (recreation)

Of course, due to the nature of the submarine; and great potential for combat, it was not long until the first military grade submarine was invented – roughly 100 years after the completion of the Drebbel, by a carpenter Yefim Nikonov [14] (under the command of Peter the Great, of Russia). It was armed with long tubes capable of being directed towards the surface, and acting as a flamethrower; additionally, it possessed an airlock, to enable soldiers to board enemy vessels from the vehicle. However, the vehicle proved to be a failure – at first. The first test, in 1724, proved to disastrous with the vessel sinking and almost claiming the lives of the four crewmembers. Two further tests were performed, before Nikonov was relieved of his job – after which the vessel was continued to be developed, until it's first successful combat implementation much later in 1775.

Figure 2.1.4, recreation of the Nikonov's submarine

Of course, in all cases thus far there has been no vehicle capable of independent motion within water – both the Drebbel and Nikonov's submarine relied heavily oarsmen to provide the necessary propulsion, something entirely different from the modern submarine. However, this was all changed, in 1776, when David Brushnell [15] created the Turtle; a submarine operated by a single user – utilising a crankshaft and two screw propellers to provide thrust and dive capabilities. This said, there is some stigma regarding the existence of the vehicle – with some naval historians discrediting both the ability of the craft, due to issues regarding natural buoyancy (when using the vertical propeller) and the credibility of its existence, as no British records exist of an attack via submarine at the time – leading to the believe that the craft was the fictitious result of propaganda. However, under the assumption that the vehicle did in fact exist – then it may be accredited the first military submarine independent of oarsmen. Below in 2.1.5, a recreation of the vessel is displayed

Figure 2.1.5, the Turtle replica, as displayed by the Submarine Force Library and Museum

2.2 Recent Advancements

Between the development of the Turtle, several advances in submarine technology emerged – in 1863, the Plongeur [16] was developed: the first instance of a submarine possessing mechanical power; in 1888, the Peral [17] was constructed: one of the first submarines to utilise an electric propulsion system, which is argued to be the first fully capable military vessel.

By the end of the 19th century, Diesel Electric propulsion became the most prominent across submarine prototypes; as it enabled electric batteries to propel the vessel while submerged, and diesel (in some cases petrol) engines were implemented to propel the vessel while surfaced – which enabled both propulsion and recharging of the onboard batteries. Of course, this did limit the speed and range of the vessel while submerged, regardless of the size of the batteries utilised. Thus, it was required that another more appropriate solution could be found, one which enabled the vehicle to possess a large range without a speed limitation.

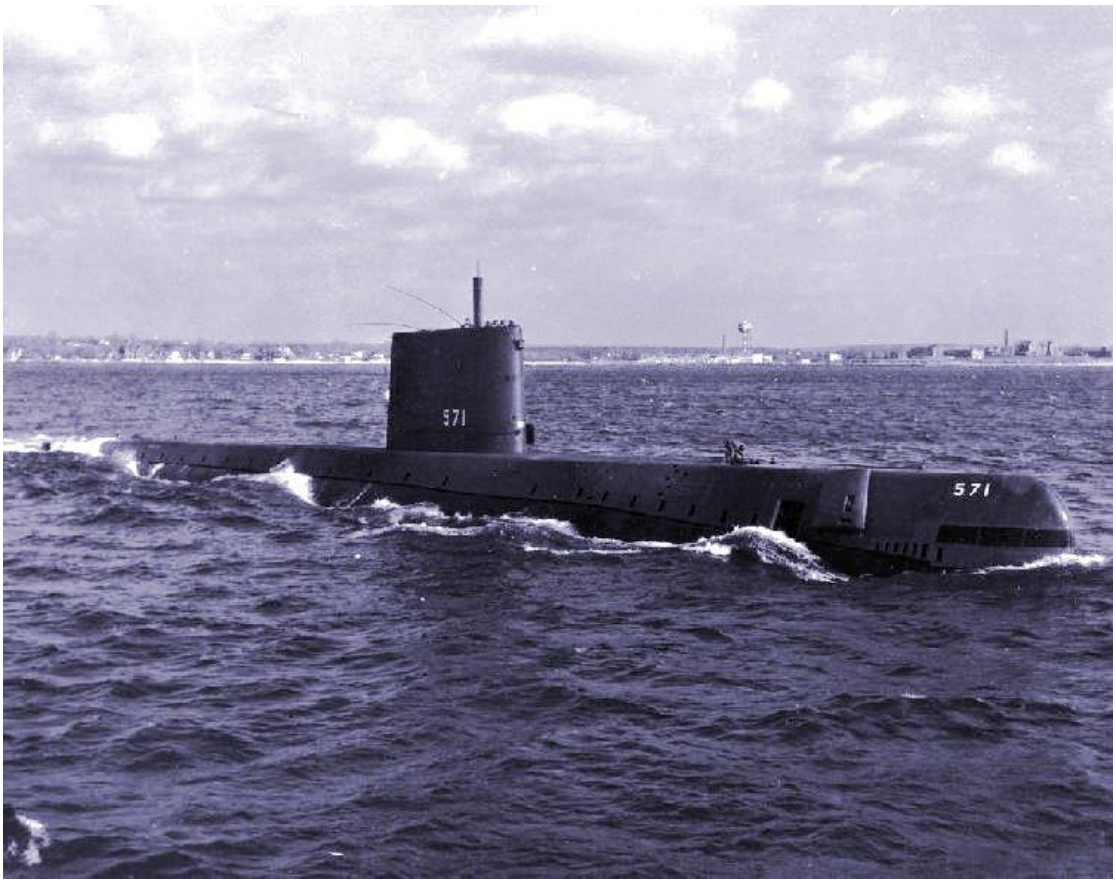


Figure 2.2.1, the USS Nautilus, the first Nuclear powered submarine

In the 1950s, a solution was found – the implementation of Nuclear propulsion, rather than Diesel-Electric power. The first craft to fully utilise this type of power was the USS Nautilus [18], as displayed in 2.2.1 – which was re-launched in 1955 (having been previously launched in 1954, without nuclear power). The Nautilus was a craft fully capable of remaining submerged for weeks or months, as the result of several technologies working in tandem; namely the limitless power supplied by the Nautilus'

reactor, as well as water extraction technology and inertial navigation which enabled the Nautilus to resupply its oxygen from seawater and accurately determine its location without requiring surfacing, respectively.

Of course, the Nautilus did not possess the ability to remain underwater for an unlimited duration – as would be desirable of submarine vessel – due to it's obvious limitations: the crew. As human beings require fresh air, food and water (two of which the submarine could provide: air and water, as it could recycle the air and distill seawater). Consequently, the vessel was limited by the stores of food available onboard: an issue that has posed a problem throughout history of naval vessels. To this end, it's obvious that the human component of such vessels require replacement; considering humans are not built for living deep underwater for a long duration. The solution to this problem is provided through Autonomy: which enables the vehicles to be operated from the surface, remotely – enabling the human component to comfortably remain above sea-level where fresh air, water and food are abundant. One such example of autonomy is the SPURV (Special Purpose Underwater Research Vehicle) [19], displayed in 2.2.2, the first autonomous submarine, developed in 1957. A battery powered remote vehicle which could operate at a maximum depth of 3.6 kilometres, and a speed of 5 knots (for a duration of 5.5 hours). The vessel was powered by a 24 volt, 200 amperes, main battery; and possessed redundancy power, in case of the main battery failing.

Figure 2.2.2, the SPURV, the first autonomous submarine

As for a much more modern implementation of AUV technology, the REMUS 100 platform [20] is considered a significant achievement – which has seen a variety of modifications, for example the Swordfish MK 18 Mod 1 [21], in service of the US navy. A vehicle with the purpose to SCM (Scan, Classify and Map) regions of 'Very Shallow Water' (1-40 ft). The US military also possess several other AUVs, based on a variety of platforms, for different purposes – including the Littoral Battlespace Sensing unit [22] (LBS UUV), based on the REMUS 600 platform, intended to complete environmental surveys of ocean, costal and inshore waters. This platform is capable of operating for 70 hours, at a speed of 5 knots up to a depth of roughly 2000 ft [23]. A vast improvement over precursors such as the SPURV.

Figure 2.2.3, the Swordfish MK18 Mod 1 (based on REMUS 100)

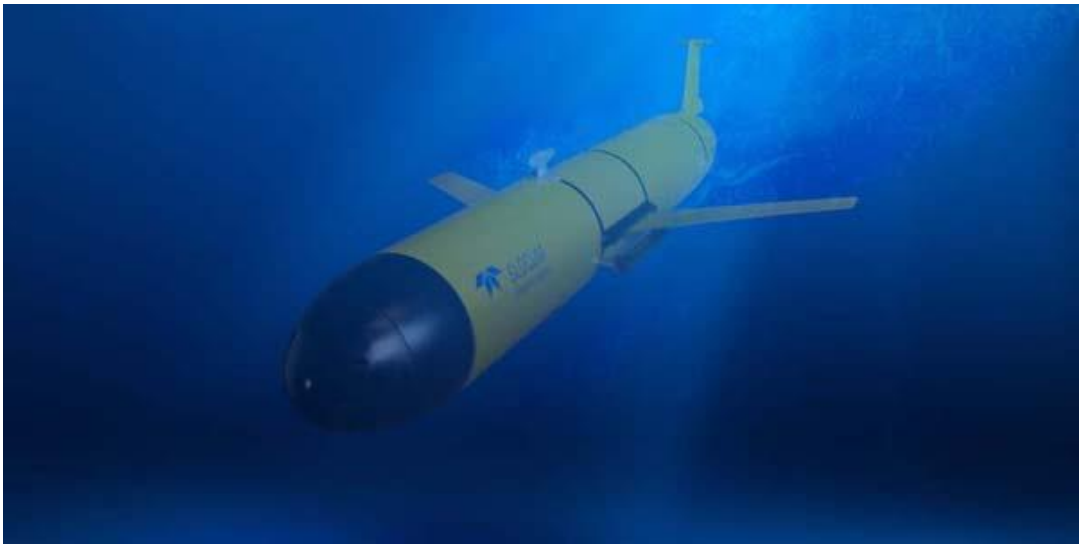


Figure 2.2.4, the LBS-G (glider) variant AUV (based on REMUS 600)

2.3 Future Technology

In terms of power, with the advent of Nuclear Fusion being just outside humanities reach, it is very likely that the future of submarine power lies within onboard fusion reactors – for which a deuterium-tritium reaction [24], seems most viable. This reaction is especially appealing given the abundance of deuterium contained within the earth's oceans; the very location where submarine activity will be concentrated, thus it would be entirely possible to implement the means of filtering deuterium from the surrounding water. Additionally, tritium may be easily produced by feeding lithium, contained within ceramic pebbles, to the reactor – which, when irradiated, produce tritium. This type of power would prove to be far superior, due to the vastly greater energy produced in a fusion reaction compared to fission and additionally due to the safety involved with such a reaction – fission proves too be far more dangerous due to the irradiated by-products. It should also be noted, that despite fusion not yet being a viable power source, MIT have proven that it is possible to create a very small-scale fusion reactor [25] (in attempts to advance towards an ARC – Affordable, Robust, Compact – reactor [26]) with a diameter of 0.68 m; a scale that greatly appeals to industries such as marine vehicles. As this would enable not just manned submarines (which are inherently large vehicles) but also AUV's (which are designed to be small and compact) to be able to source power from fusion reactions – enabling AUV's to operate independently for a significantly longer period than the current battery powered crafts.

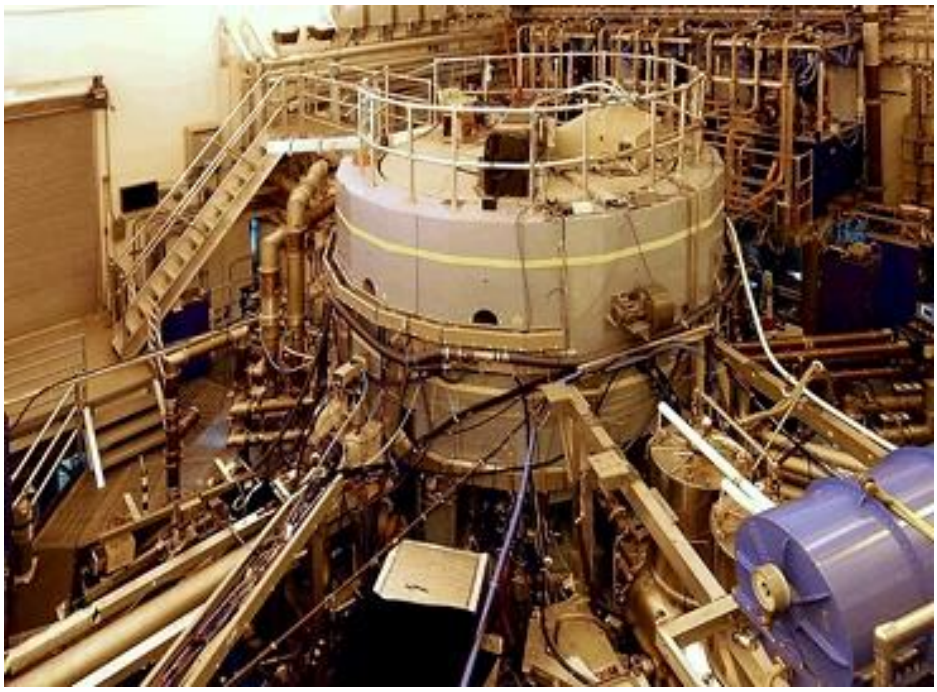


Figure 2.3.1, MITs Alcator C-Mod – encased in a vast amount of experimental equipment

In the case of AUV's, another technology which would be suitable for application would be the idea of supercavitation – which application is currently limited to supercavitating torpedoes. This technology

utilises the concept of cavitation [27] – which states that as propellers cut through a liquid, small bubbles of gas are created. Supercavitation expands on cavitation, through creating a much larger bubble (using a cavitator, which ejects gas from the vehicles nose), capable of surrounding the vehicle in question; greatly reducing the drag experienced while propelling through liquid and enabling much higher speeds. This technology would find great application in an AUV due to their small size, largely alike that of a torpedo, and current reliance on battery power. Thus, supercavitation is a principle that could easily be transferred from torpedo technology and provide a great deal of merit – through increasing the maximum speed of a given vehicle (at a supersonic level), using rocket-based motors, rather than propellers.

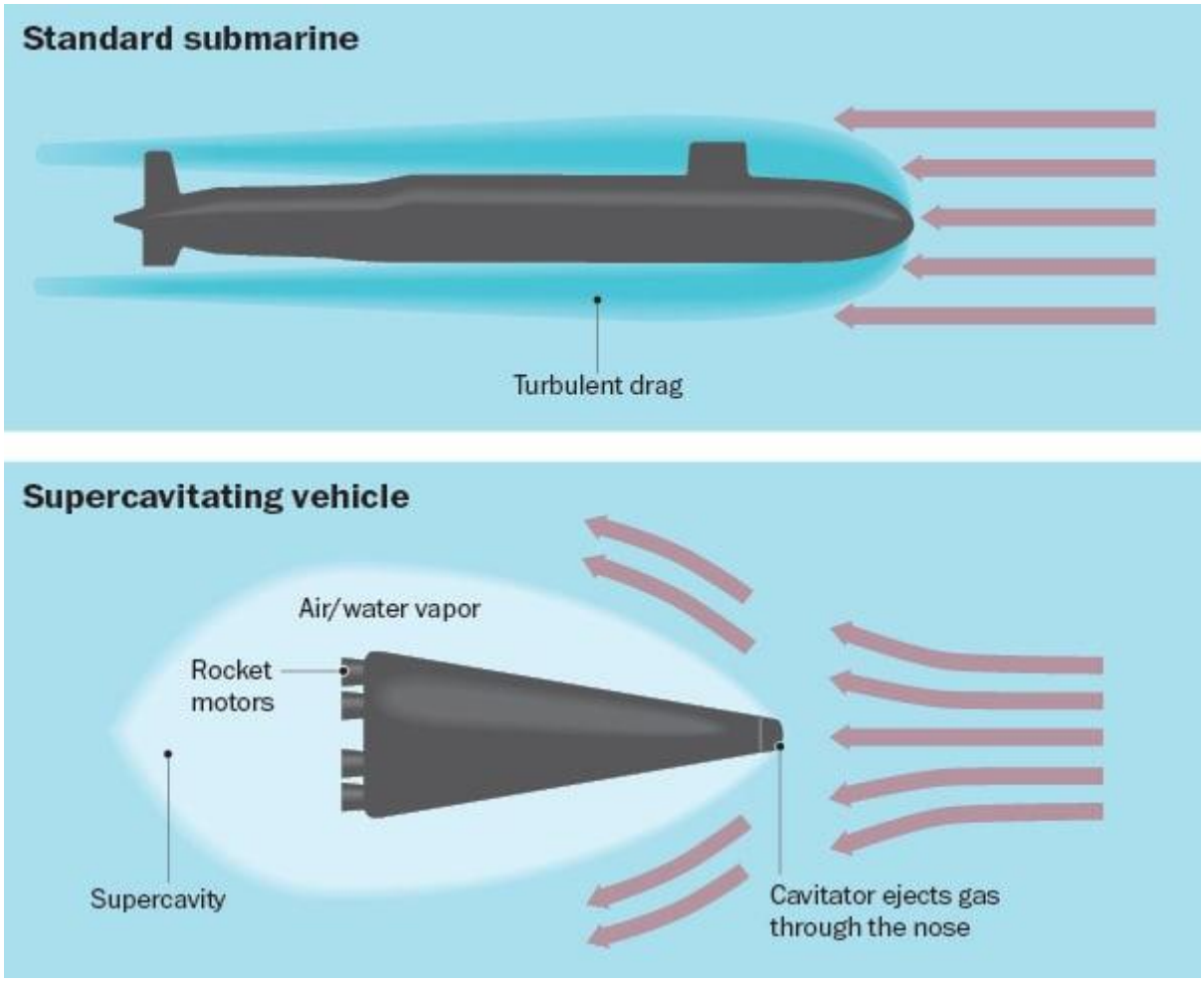


Figure 2.3.2, graphical representation of supercavitation

3 Project Plan

3.1 Projected Timeline

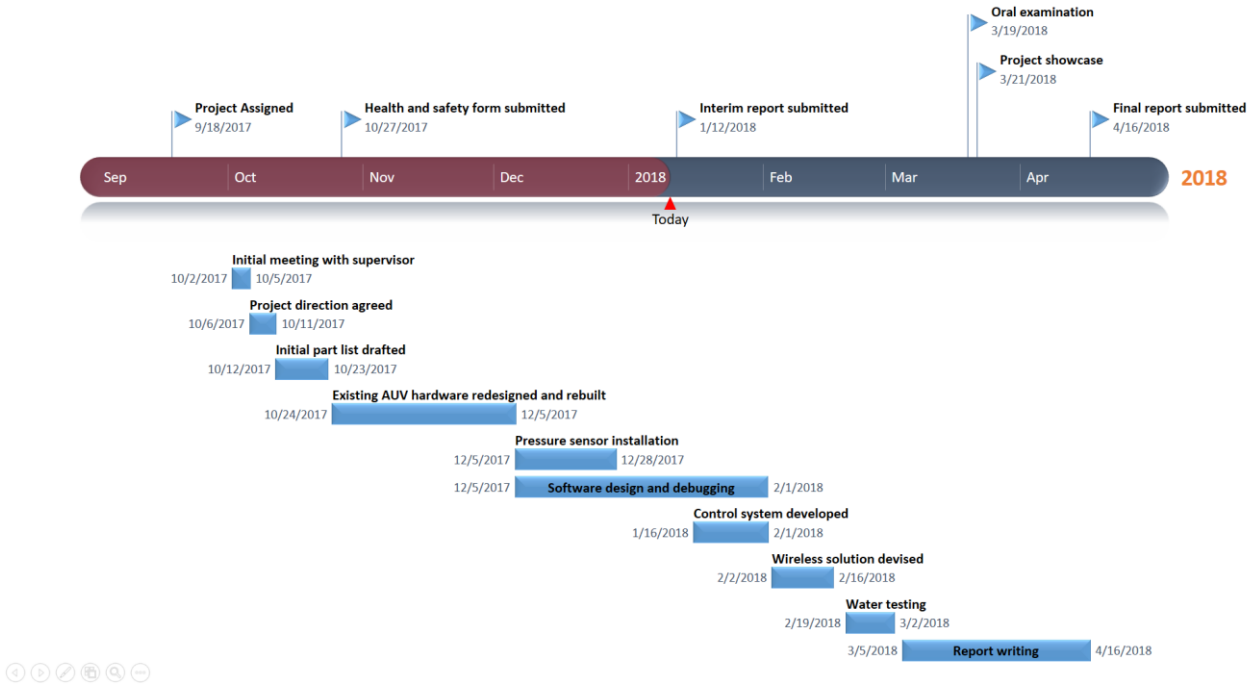


Figure 3.1.1, Projected timeline for project

3.2 Gantt Chart

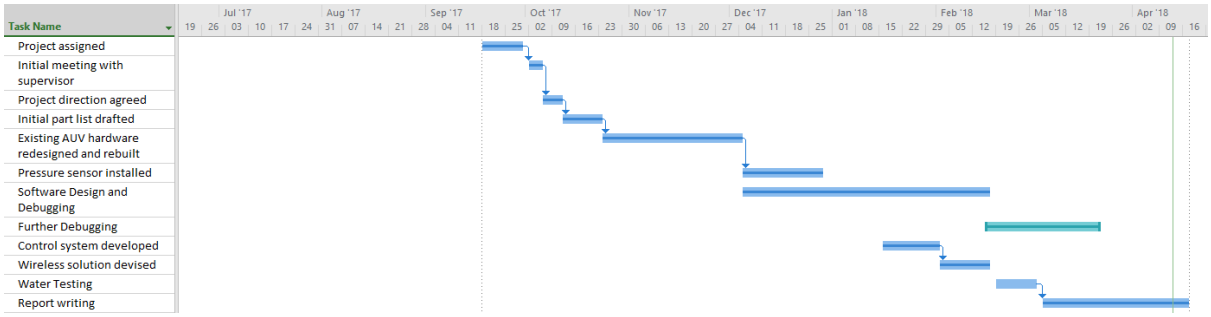


Figure 3.2.1, Gantt chart contrasting actual project milestones against predicted milestones

From above, in 3.2.1, it may be observed that several project milestones were completed as initially planned (initial plan displayed in blue, whereas unexpected issues displayed in green). Evidentially, hardware design went smoothly with all required parts being purchased early in the design cycle and PCB designs quickly developed over several design iterations. Likewise, can be said for the manufacturing of the hardware – with credit owed to Gerry Rafferty, for the swift PCB printing and countless pieces of advice regarding soldering and passive component selection. The hardware was quickly assembled, and subsequently tested – at which point all systems were operating. The pressure sensor was quickly installed to the AUV frame successfully, after a slight modification to the frame – with credit due to the CNC workshop – and secured in place via CT1 glue and sealant (at the advice of

Gerry Rafferty, who claimed it provided an excellent submergible seal). This was performed simultaneously to the modification of the AUVs internal mounting system, in which the original mount was removed and replaced with curtain tracks, bonded to the frame with CT1; which was implemented as rail to which hardware could be mounted. Software design for the AUV, in which despite several unexpected revisions were required, went according to plan. Some issues arose regarding compatibility between the PIGPIO daemon and pieces of hardware, however it was relatively easy to amend this (through the implementation of alternative packages, such as 'SMBus' and 'Serial').

However, following this, issues began to arise in a multitude – which required a great deal of debugging to reach the root of the problem. It should additionally be noted that several ICs required replacement during this period – and a great deal of time sank into software modifications, to determine if the issues were hardware or software related. Eventually the main source of the issue was determined to be the hardware – specifically the PWM generator, which – the author concluded – is extremely vulnerable to static electricity. Additionally, during this hardware debugging phase, a PWM output rail was shorted to the 5 V rail; resulting in the authors' own hardware becoming 'bricked' – as an I2C rail was shorted to ground, within the PWM IC. Thus, with the showcase swiftly approaching, and the inability to provide further time to debug this new problem (at the time), the hardware of the previous iteration of the project was adopted – and software revised to ensure compatibility (it should be noted that this hardware did not operate fully at first, however after a short period of debugging – and the author's previous experience – it was quickly determined to be an issue involving the PWM generator).

Of course, although the issues were eventually overcome, they had put a great constraint on the time remaining until the completion of the project – as such it was advised that water testing would no longer be a possibility, as university term adjustment proved to be unforgiving if delays were experienced. Which the author believes to be a major problem, for titular project in particular, as bugs and issues will always arise in the development of a system – and when the timescale is limited for a hypothetical flawless system design, it does not prove appropriate in the case of a practical system; In which issues are inevitable.

In conclusion, the project unravelled excellently through the early and middle stages of the design; however, as the end stage approached, it became far less smooth and concluded by the author that new development lifecycle adopted (as the result of term changes) severely limits the project – as the month lost, could have provided enough time to ensure further revisions of the codebase as well as ensure the possibility of water testing the AUV.

3.3 Budget

At project competition, it was possible to calculate the total cost of the developed prototype AUV – neglecting any replacement components, which were covered at the authors expense. Additionally, it should be noted that some items listed have been supplied by author and thus will not be subject to the project budget of £200. Furthermore, assuming a water testing period was performed– there would be an additional fee, which is negligible in this case given it uses university premises, of £200 per day of testing at the wave tank. Also, of note is the cost of additional components (such as resistors and transistors), and manufacturing of the PCB's. Considering both factors, it was important to ensure that the PCB's were functional on the first prototype (to reduce additional costs from further printing) and that the water testing period should be optimised given how much inefficiency could cost (where testing is performed, and this cost is considered).

Item	Supplier	Referance Code	Quantity	Min. Order	Price	Total Price
Controller						
Raspberry Pi 3	RS UK	896-8660	1	1	£ 29.99	£ 29.99
Breakout Board						
Innoline R-745 - 5V 4A Switching Regulator*	RS UK	828-9273	1	1	£ 19.13	£ 19.99
MC33269DTRK-3.3G - 3.3V 1A Linear Regulator	RS UK	688-9231	1	2	£ 0.56	£ 1.12
LTC2990CMS - ADC (4ch)	RS UK	823-0076	1	2	£ 3.66	£ 7.32
MCP3221A5T-I-OT - ADC (1ch)	RS UK	772-8758	1	5	£ 1.18	£ 5.88
PCA9685PW - PWM Generator (16ch)	RS UK	727-5649	1	2	£ 1.61	£ 3.22
Motor Driver						
TLE8209-2SA - 9A H-Bridge Motor Driver	RS UK	110-7473	4	4	£ 3.23	£ 12.90
4 Terminal Block - Screw Head / PCB Mount	RS UK	361-7689	2	5	£ 0.75	£ 3.76
XT60 Battery Connectors (5 pair)	Blade Hobbies via Amazo	B00V3F7A9K	1	1	£ 4.99	£ 4.99
Sensors						
BNO055 - 9 DoF Fusion IMU	Cool Components (UK)	1929	1	1	£ 34.96	£ 34.96
Heavy duty PX3 pressure sensors	RS UK	111-5922	1	1	£ 25.65	£ 25.65
Power						
ZIPPY FLIGHTMAX 5000MAH 4S1P	HobbyKing	9067000171-0	1	1	£ 29.13	£ 29.13
iMax B6 LiPo Balance Charger	homeoutlet	N/A	1	1	£ 12.35	£ 12.35
Wireless						
USB repeater	Amazon	N/A	1	1	£ 8.49	£ 8.49
USB wireless dongle	Amazon	N/A	1	1	£ 12.99	£ 12.99
Misc.						
Graupner Marine prop 2308.65L	Model Shop Leeds	N/A	2	1	£ 3.99	£ 7.98
					Total	£ 220.72

Figure 3.3.1, total project expenditure

4 Risk Analysis

4.1 Overview of Risks

As the project is one which uses a minimal amount of dangerous equipment and does not take place, at any time, in a dangerous environment; most of the risks outlined at the initiation of the project were unlikely to occur, as they may be easily avoided, but alas the possibility of their occurrence remained throughout the duration of the project.

Some of the risks agreed are as follow in figure 4.1.1. A blank copy of the risk analysis sheet based on which the following were agreed upon, may be found in *Appendix 1*

Risk	Risk Category	Level	Description
Soldering	Electrical	Low	Possibility of receiving burns, and hazardous to breath the toxic fumes of flux
Rotating Propellers	Mechanical	Medium	It is possible to sustain injury whilst the propellors are in rotation
Lithium-Polymer	Electrical/Chemical	Medium	Lithium-Polymer battieries are known for being fire hazards, care should be taken while charging them
Wave Tank Testing	Electrical/Mechanical	High	It is possible injuries could be received during testing, or equipment could be damaged

Figure 4.1.1, potential hazards

From 4.1.1, it can be observed that 'Soldering' was chosen as a 'Low' level risk – Although it is somewhat hazardous to the person involved, this danger is easily avoided by being aware of the location of the soldering iron in respect to oneself. Additionally, is it simple to avoid breathing the toxic fumes produced by heating flux through stationing oneself in a well-ventilated area.

It can also be seen that 'Rotating Propellers' and 'Lithium-Polymer' were chosen as 'Medium' level risks, the former is a mechanical risk – one which may be avoided by keeping appendages away from the shrouded propellers when in operation – the latter on the other hand is a chemical hazard due to the volatile nature of Lithium-Polymer cells [29]; of note it that a rise of temperature while during charging may present a fire hazard – especially if the battery has exceeded critical discharge (if a single cell falls below 3.0 V), as the internal resistance of that cell exponentially increases for every 0.5 V a cell falls below its critical value. It should additionally be noted that if the battery in question experiences swelling, it should be discharged safely and disposed of immediately; as a swollen Li-Po has a very high chance of combusting when in use. Alternatively, the battery could be salvaged by electrically disconnecting the damaged (swollen) cell and converting the Li-Po to a lower cell count battery (for example, a four-cell battery may be salvaged to three-cell battery).

Additionally, it is apparent that 'Wave Tank Testing' was selected as a 'High' level risk – due to the nature of electronics in water, as water is a great conductor, and since the facility is of high value; thus, no equipment must experience damage during a period of testing.

5 Hardware

5.1 Hardware Utilised

Given the fact, that the developed AUV should be remote; this implies that there should be a method of providing stored power to the Raspberry Pi 3 and Motors. Thus, it is essential that a high-voltage, high-capacity battery supply should be selected. Due to the similarities between an AUV and a hobbyist drones such as Quadcopters; batteries used to provide power for these devices were explored. Ultimately a Lithium-Polymer battery was selected, or a Li-Po; Specifically, the Zippy Flightmax four cell battery was selected (which possesses the following characteristics: a nominal voltage of 14.8 V, 5000mAh capacity – with discharged rated at 30C, or a maximum continuous discharge rate of 150 A, at which rate the battery would experience full discharge in roughly two minutes. As for the reasoning behind the selection of this battery, that was due to the positive reviews of the Zippy brand in the RC Drone industry; most of which boasted of the batteries possessing premium grade quality at a budget price. Additionally, the large maximum discharge rate proves to be desirable as it enables the AUV to operate at full power without being limited by the battery (it should be noted that the AUV would operate in the region of 10-20 A, rather than the rated maximum discharge of 150 A).

Figure 5.1.1, Zippy Flightmax Li-Po battery

However, clearly the battery voltage provided by the Zippy Li-Po greatly exceeds the operational voltage of a Raspberry PI 3 (which operates at 3.3 V across the GPIO and may be externally powered by applying 5 V across pins two and four). Thus, the means of reducing the supply voltage to 5 V is required, and further reducing this voltage to 3.3 V to provide a 5 V and 3.3 V rail for various components. This may be achieved by firstly utilising a buck converter to reduce the battery voltage from 14.8 V to 5 V, this 5 V supply may then be regulated to provide a 3.3 V supply. For these purposes

the following IC (integrated circuits) were selected: the Innoline R-745 [30] and the MC33269DTRK-3.3G [31].

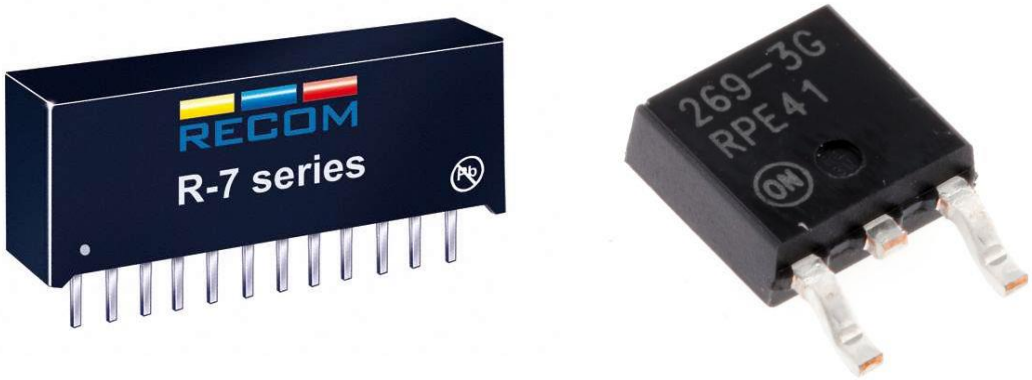


Figure 5.1.2, power conversion ICs

Regarding the control of the actual motors, and propellers, which consists of Rule bilge pumps – from which the brushed DC motors have been extracted and repurposed as shrouded propellers – with the actual propellers consisting of matched Grauper 2308.65's, of 65mm diameter, possessing three blades. The propellers type and direction of rotation being arranged according to the following diagram:

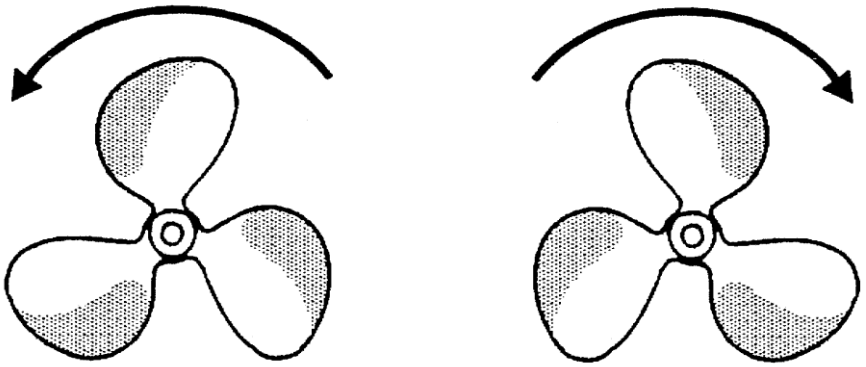


Figure 5.1.3, propeller direction and type

Figure 5.1.4, Rule bilge pump and Grauper 2308.65 (L and R)

Given the Rule DC motor is of a brushed type (as evident by the motor requiring only +VE and -VE inputs), a method of controlling a brushed DC motor was required: to this end, H-bridge motor drivers for brushed DC motors was selected – specifically the TLE8209-2SA [32] which possesses the option of controlling the motors via two PWM (pulse width modulation) channels per chip (and by extension per motor), for forward and reverse configurations.

Figure 5.1.5, H-Bridge motor driver IC

However, one issue which becomes immediately apparent is the fact that the Raspberry Pi does not natively support PWM signals – as all GPIO output pins are digital in nature, providing only a High or Low logic level. Thus, it is required that a separate IC is added to provide PWM signals which may be controlled simply from the Raspberry Pi: the solution to this, was to implement the PCA9685PW [33] IC – a chip which provides 16 channels of 12-bit PWM signals (may provide analogue voltage between 0 V and 3.3 V, with 4095 steps in-between) and is controlled via the I2C bus; a feature which the Raspberry Pi does possess. Of these 16 channels, 8 may be utilised to implement full control (forward and reverse directions) across all four of the AUV's propellers.

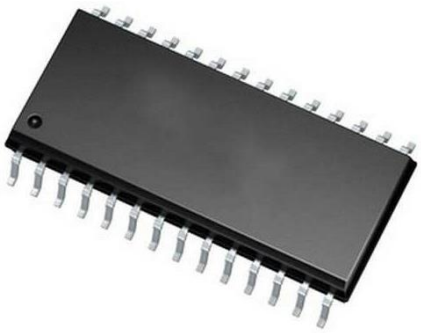


Figure 5.1.6, PWM generator IC

Additionally, another issue, which arises from the fact that the Raspberry Pi does not possess any analogue pins natively, is that the Raspberry Pi board is unable to read voltages which vary between High and Low logic levels: it cannot read analogue voltages. Thus, for the Raspberry Pi to read a variety of sensors, which output non-digital values external Analogue to Digital (ADC) Conversion ICs were required. As such, two ICs were selected to be implemented on the hardware: the LTC2990CMS [34] and the MCP3221AST [35], which provide four channels and one channel of analogue to digital

conversion respectively. Similarly, to the PWM generator, they communicate utilising a feature which the Raspberry Pi does possess – I2C.

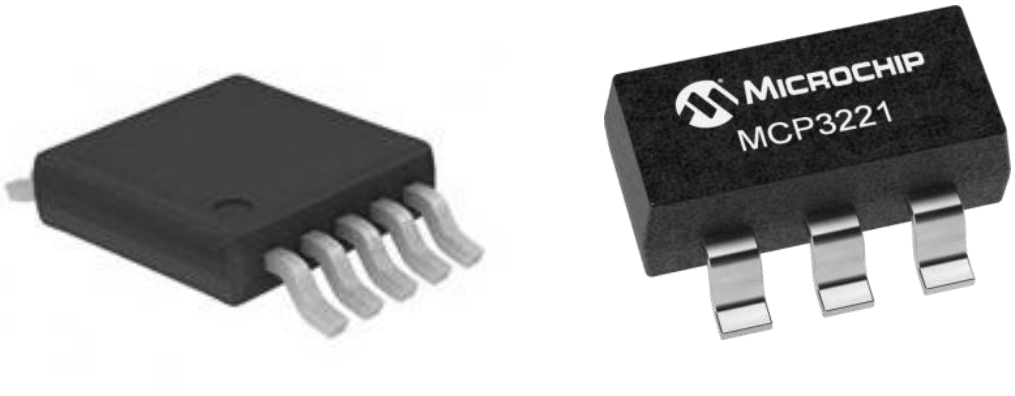


Figure 5.1.7, four channel and single channel ADC ICs

To provide the AUV with a sense of direction, it was required that a compass be featured in the system; a compass which could provide Euler directions (Heading, Roll and Pitch). These were provided, as well as a variety of other sensor capabilities via the Adafruit BNO055 [36] – a Nine Degree of Freedom (NDOF) compass which may communicate with the Raspberry Pi via either I2C or Serial. As the manufacturer, Adafruit, advised against communication between a Raspberry Pi and the BNO055 via I2C (due to the clock-stretching performed by the unit) [37], it was decided to implement communication between the devices via the Raspberry Pi's serial port (*tty/AMA0*, the hardware defined UART port, rather than the mini-UART port *tty/S0* – which is limited by the clock speed of the Raspberry Pi).

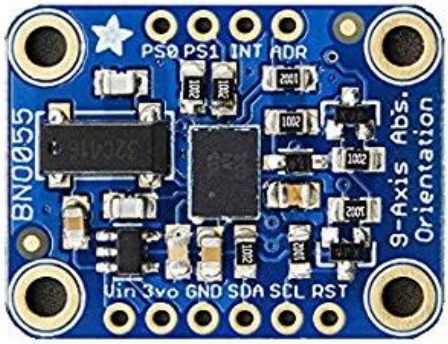


Figure 5.1.7, Adafruit BNO055 NDOF sensor

Additionally, given the intended purpose of the AUV is as a seaworthy vessel; it is essential that a method of determining the current depth is implemented thus a second sensor was included to determine the pressure (and accepts water as a medium for detection). As such, the Honeywell PX3 [38] sensor was selected for several reasons: namely, it's advertisement as a "heavy duty" sensor and

IP rating of 67 (protection from dust and water ingress), it's compatibly with the operating voltages provided by the power circuit (requires an input of 5 V) and its ability to determine pressure in a medium of water. The output of the PX3 is an analogue voltage between 0.5 V and 4.5 V relative to 0 and 25 bars of pressure, which must then be passed through a voltage divider to reduce the output to the operating voltage of the Raspberry Pi and connected hardware (3.3 V). Thus as 25 bars may be converted to approximately 250 m of depth, the AUV can operate up till this threshold.

Figure 5.1.8, PX3 pressure sensor

Additionally, due to fact that radio waves do not propagate desirably through a medium of water, an alternative for communication had to be decided upon to enable the AUV to operate at greater depths without dropping the communication link to the control centre. After some consideration, looking into various methods from Ultrasonic Communication, to Optical communication and even simply communication via ethernet. In the end, the author ultimately decided to adopt an "extended antenna" inspired design by utilising a USB repeater coupled with a USB wireless dongle to provide a wireless solution in which the essential component to the communication link – the wireless antenna is never submerged underwater – thus enable the AUV to operate at various depths whilst still communicating with the controller.

Figure 5.1.9, components of the devised wireless solution

Figure 5.1.10, example of an ultrasonic modem, provided by Teledyne Marine

Figure 5.1.11, example of an optical modem product, provided by Sonardyne

Regarding the operation of the two previous products, displayed in Figure 5.1.10 and 5.1.11: the ultrasonic modem outlined the in former operates through sending and receiving radio waves to a buoy which exists above sea level, these waves are modulated as such that the information may be contained in packets of ultra-sonic data for communication between the buoy and the companion device which is submerged. On the other hand, the latter figure displays an optical modem which again sends and receives radio waves however instead these radio waves are modulated to packets of data carried by Blue and Green light (the members of the visible light spectrum which experience the greatest water penetration). On receival all data is demodulated and transmitted to the connected device.

This concludes the reasoning behind selection of various pieces of hardware for the AUV, within the following section it will be outlined how these components were implemented in circuit board design and interface with the Raspberry Pi.

5.2 Hardware Design

The initial phase of the project consisted of porting the existing PCB blueprints to an opensource E-CAD software package – Eagle was selected due to the size of the existing community, and that it is available as freeware to students [39].

In the first half of this section, the hardware of the Raspberry Pi Breakout Board (see *Appendix 2*, for the full version) will be detailed – as it is the most important of the PCB's given it provides both the power for the controller, as well as the interfacing.

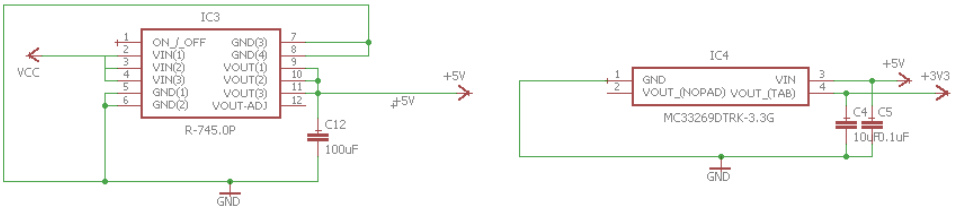


Figure 5.2.1, power IC's to provide 5V and 3V3 power rails

In the above graphic, the integrated circuits (IC's) used to provide the 5V and 3V3 power rails are displayed. IC3, is a 5V 4A buck converter – to convert the voltage supplied by the battery, to a voltage which is compatible with hardware selected. Additionally, IC4 is a 3V3 voltage regulator which reduces the voltage further to a level which is compatible with the many IC's and the GPIO of the Raspberry Pi.

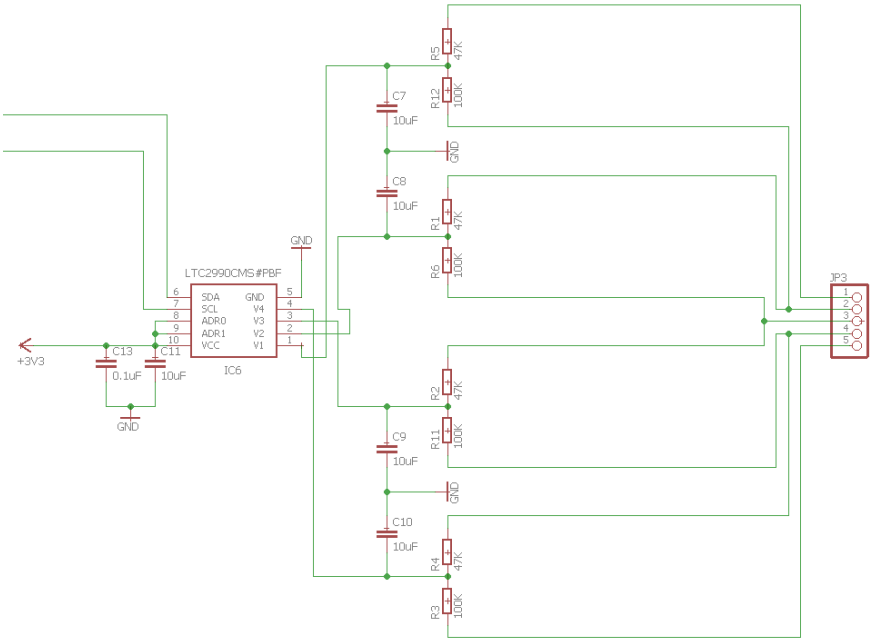


Figure 5.2.2, battery monitoring ADC

In Figure 5.2.2, the hardware used to determine the current charge of the battery (per cell, given the nature of a Li-Po battery – no single cell should fall below 3V, as it would result in permanent damage to the battery). Large values of resistors are used as the voltage dividers to ensure a minimal amount of current is drawn from this segment of the hardware – as it shall be active constantly during the AUVs operation. These divided voltages are received by a four-channel analogue to digital converter (ADC), to enable the Raspberry PI to read the per cell voltage of the battery.

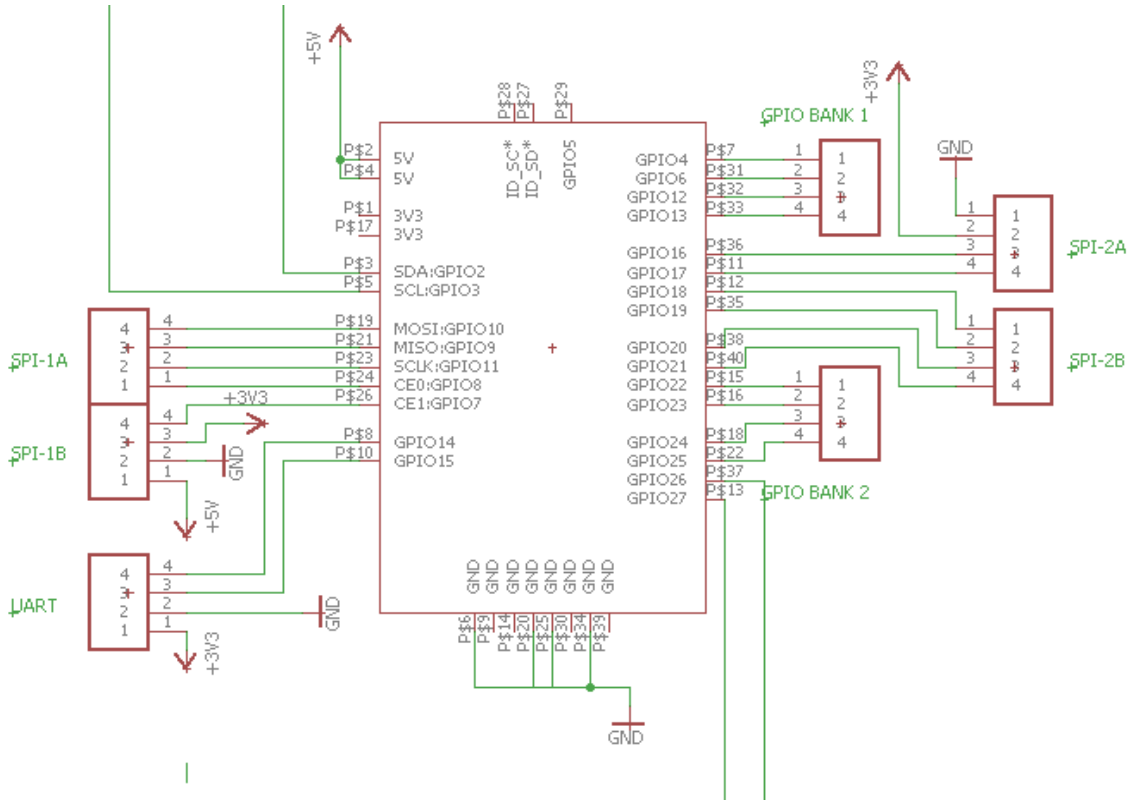


Figure 5.2.3, pins of the Raspberry Pi which are 'broken out'

Above shows the mapping of the Raspberry Pi's GPIO pins, to their new location within the breakout board. With power being supplied directly to the Raspberry Pi, and the different interfaces being separated to enable simple access to various features (such as SPI and UART interfaces, as well as a selection of GPIO pins).

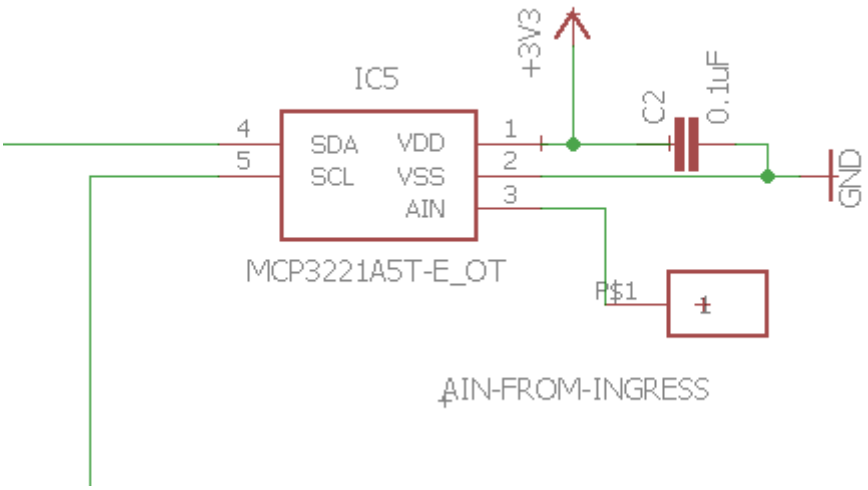


Figure 5.2.4, water ingress detection ADC

Within this graphic, hardware implemented to determine if water ingress has occurred is outlined – simply consisting of single-channel ADC to read the voltage level across the input pin.

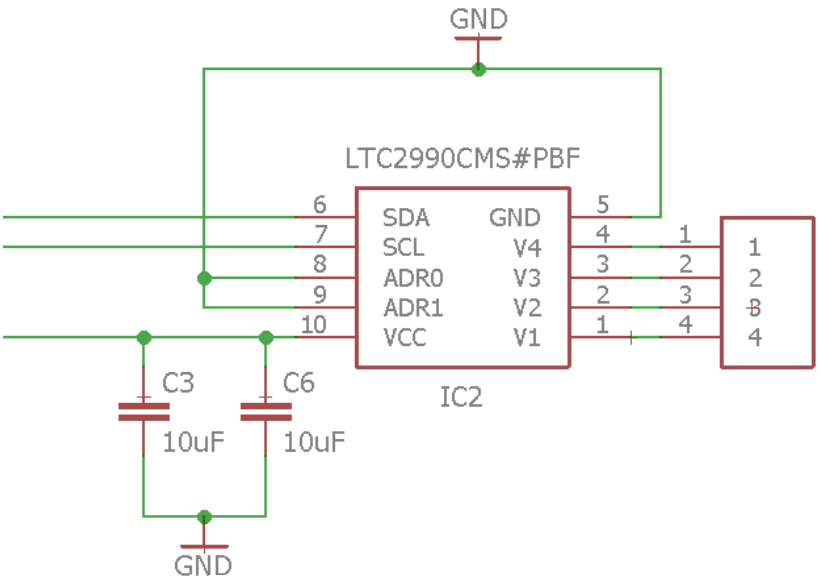


Figure 5.2.5, general-purpose ADC

The above hardware is required to enable future hardware to interface with the Raspberry Pi, given the lack of analogue GPIO pins – thus a general-purpose ADC is required to ensure there are a selection of pins which support analogue input. Two things are of note: firstly, given the same ADC IC has been previously used (see figure 5.2.2) the ADR pins must possess different levels to ensure different addresses on the I2C bus are used for each. Secondly, one of the channels already has an intended use – to read the output of the pressure sensor; thus, a further three analogue inputs are available.

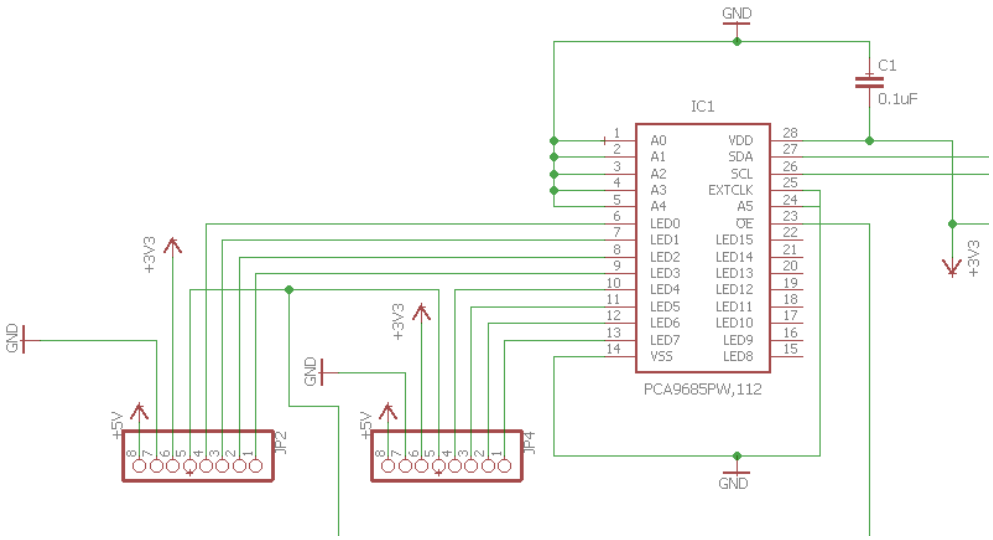


Figure 5.2.5, PWM generator to drive motors

The hardware displayed within this graphic, are required to produce the pulse width modulation (PWM) signals to enable motor control. These signals, as well as power signals, are then transmitted to the motor driver boards via two eight-pin connections.

Thus, concludes the hardware segments contained within the Raspberry Pi Breakout Board, however further hardware was additionally designed for the Motor Driver Boards.

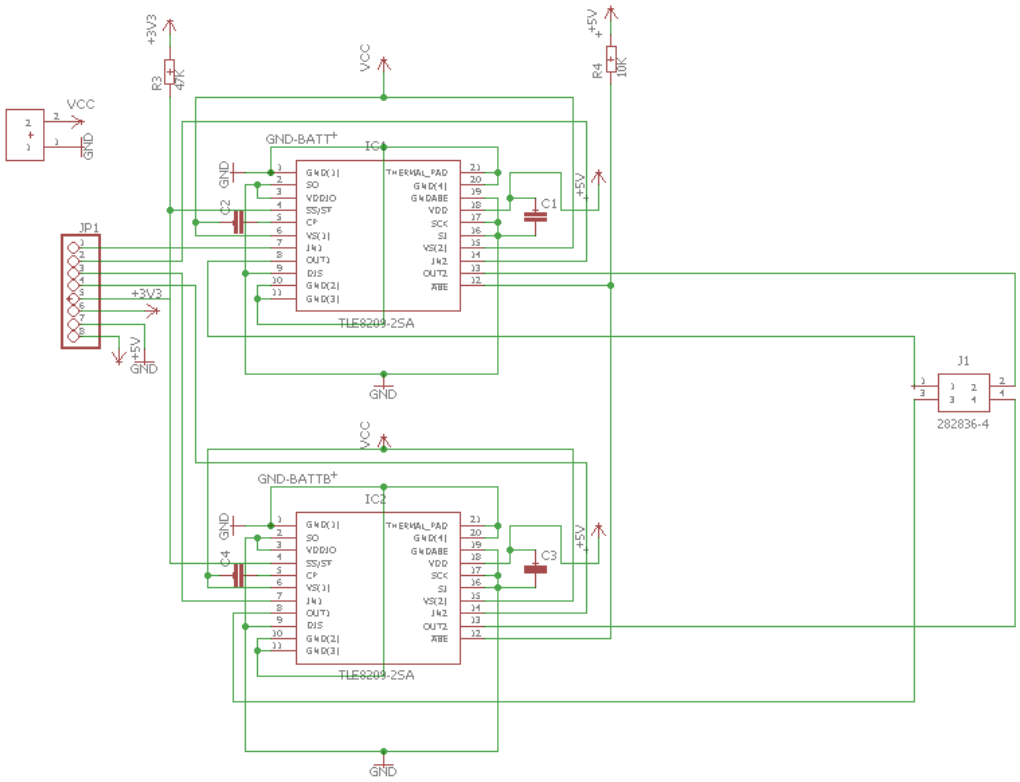


Figure 5.2.6, motor driver board

Within the Motor Driver Boards, there are two H-bridge motor driver ICs to provide the output necessary to drive the propellers of the AUV. The input for these are obtained from the eight-pin connection mentioned previously (see figure 5.2.5), as well as additional power being distributed from the battery supply.

5.3 Hardware Manufacture

Once the hardware was successfully mapped on Eagle, it was required that it was converted to a PCB layout (rather than a schematic) – so the hardware could be printed and then constructed.

Several steps were taken to ensure clean signals along the copper tracks of the PCBs, firstly by ensuring that power was distributed along the bottom on the board and data signals were distributed amongst the top of the board (where possible) and secondly by ensuring that noisy components were separated from sensitive components (for example, the power supplies were on the opposite side of the board from the integrated circuits used).

Below, the final PCB designs for the Breakout Board and Motor Driver Boards will be displayed.

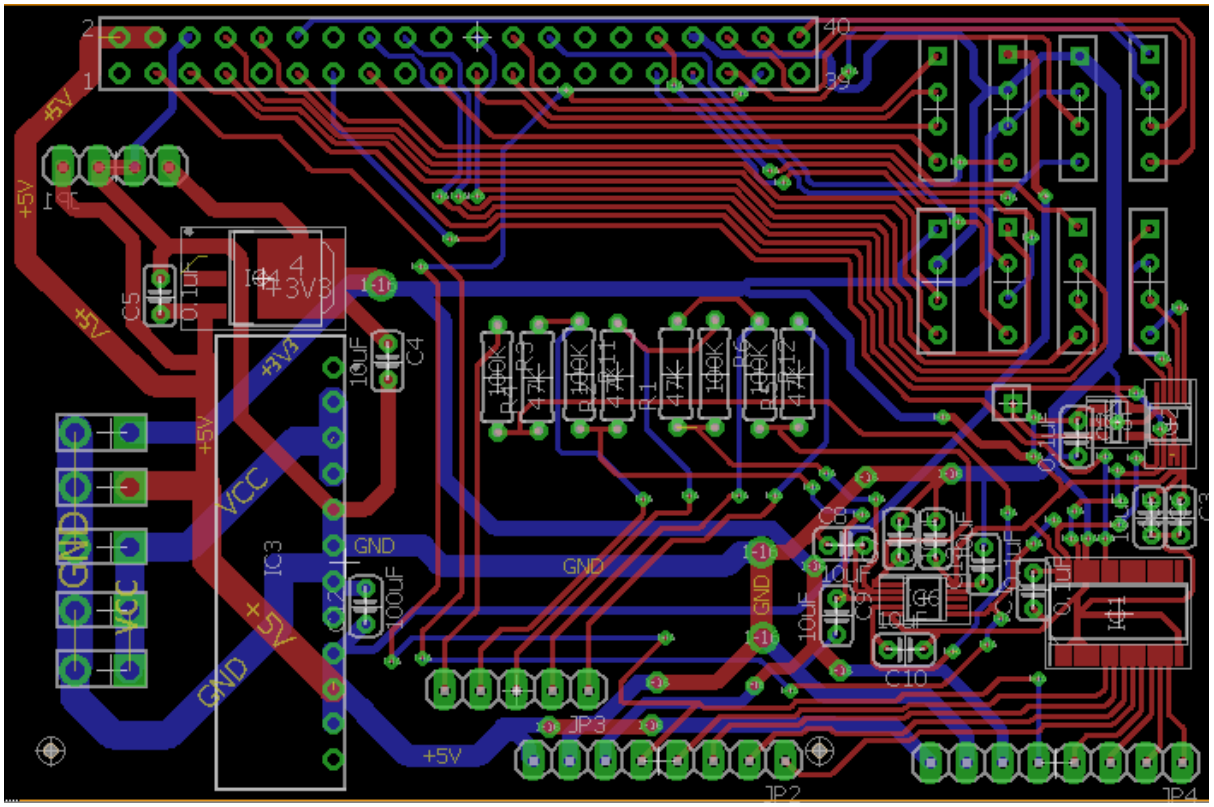


Figure 5.3.1, breakout board PCB layout

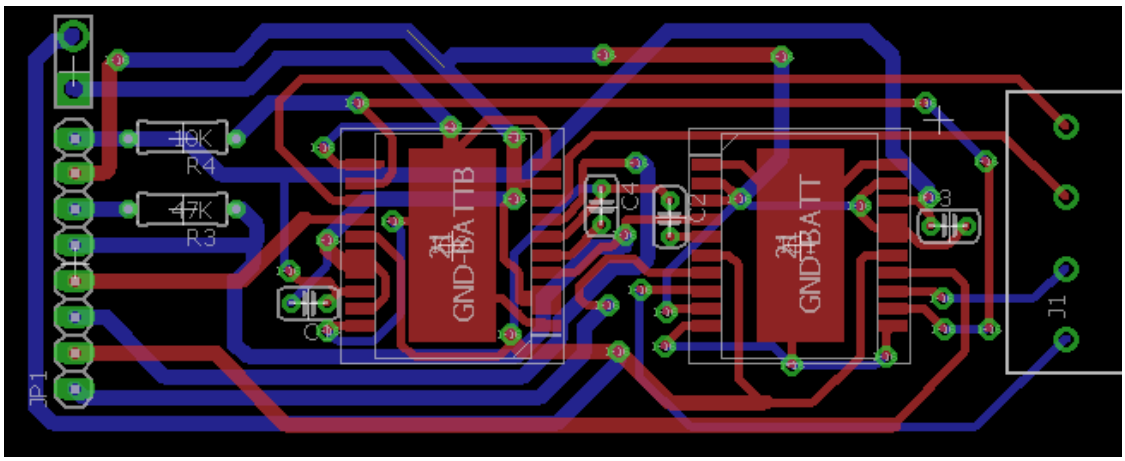


Figure 5.3.2, motor driver board PCB layout

These, Figure 5.3.1 and Figure 5.3.2, were the final designs selected for prototype PCBs to be manufactured – both of which were the result of four iterations of design. However, one issue which occurred during the manufactured caused the breakout board to require additional wire jumps between the tracks – as parts of the copper were caught on the PCB printer and as a result were wiped off the board, thus these wire jumps were required to compensate for this. The final boards (fully completed, after soldering and post-processing) are displayed overleaf in Figures 5.3.3 and 5.3.4.

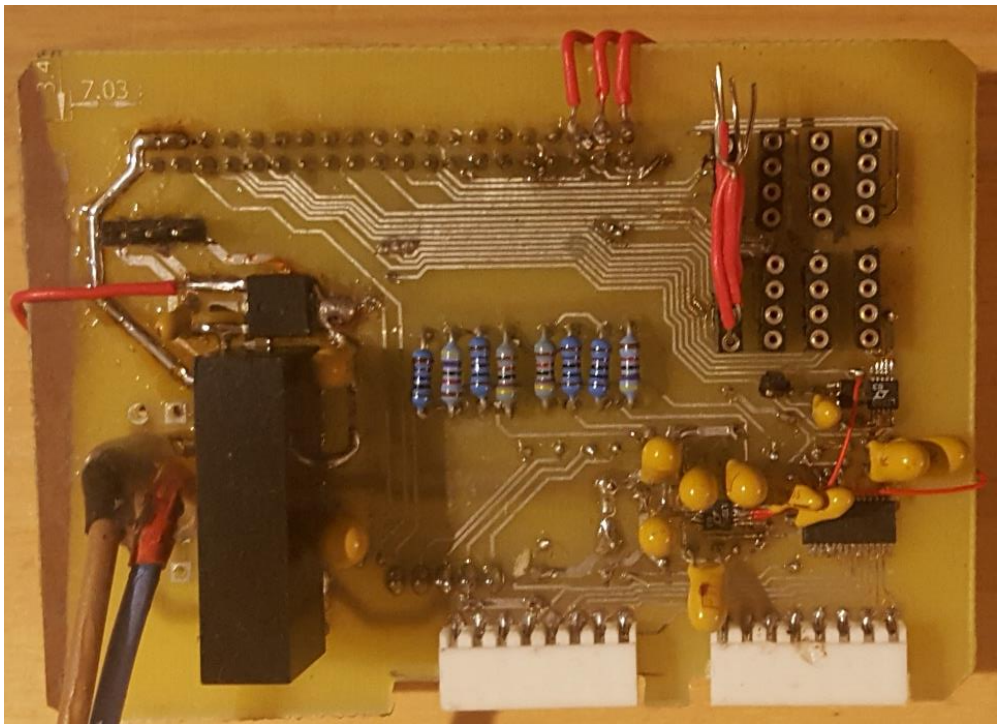


Figure 5.3.3, final breakout board

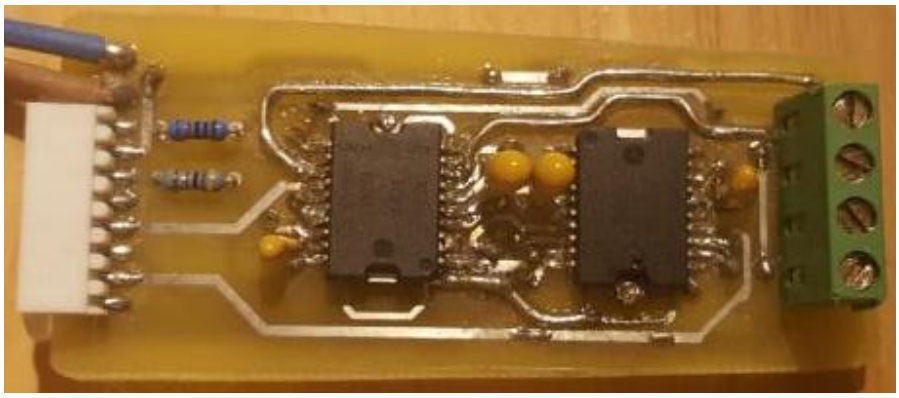


Figure 5.3.3, final motor driver board

Clearly, from the above graphics; most of the design (within 5.3.3) was accurate, however some errors were produced when the board was being printed – as mentioned previously. It should also be noted, that the traces required expanding in some sections of the boards (to increase current flow greatly, without expanding the width of the traces – due to restriction on size imposed by the frame of the AUV). This was done through simply layering a tinned copper wire (of diameter 0.6mm) atop the existing trace, then soldering along the entire trace to create a firm bond. This increases the current capacity of the trace from approximately 2A to 15A, without increasing the size of the copper trace (in the horizontal plane).

5.4 Review of hardware

Throughout the duration of the project it became apparent, that although the hardware worked as intended initially, the design could use some further thought to produce PCBs which not only work as intended; but also, are simple to debug while the system is in operation. Namely, this is desirable upon the breakout board (as displayed in Figure 5.3.3) which consists mainly of smaller components. From the authors design, it was noted that these smaller components (the PWM generator, four channel and single channel ADCs) should not be packed closely together, as when components such as bypass capacitors are mounted onto the board it become unpleasantly complex to measure voltages across certain traces.

From this, the author advises future iterations of the project take one of two options when progressing through the hardware design phase.

- Ensure adequate spacing is provided between all ICs, whilst ensuring they are distanced from the power circuit. Additionally, it should be noted that it is advisable to keep all ICs on the top side of the board, to ensure individual pins are easily accessed by oscilloscope probes. Additionally, only GPIO pins of the Raspberry Pi which are required should be mapped to the PCB, to reduce the number of small unused traces on the board.
- Move away from the Raspberry Pi based hardware and instead choose a controller which natively possess at least 8 unique PWM channels and as many ADC compatible pins as required. This would reduce a large bulk of the smaller, and difficult to debug, hardware which the project requires and would effectively reduce required hardware modules to: A power conversion circuit and a motor driver circuit. This would also prove to be useful in shrinking the codebase, the large class methods for the PWM generator and ADCs could simply be replaced with, in the case of an Arduino board (for example the Arduino Mega REV3), the simple functions *analogueWrite()* and *analogueRead()* for PWM and Analogue conversion respectively.

These are suggested by the author as the hardware provided a large variety of issues throughout the duration of the project, which shall be covered in detail in the following section.

5.5 Hardware Issues

During the project, the issues which arose may be separated into two separate categories: Hardware related, and Software related (which shall be documented in a later section). Of the hardware issues these may be further subdivided into issues relating to the breakout board and issues relating to the motor driver board.

Firstly, regarding the breakout board's issues, the following were experienced:

1. Clock stretching of the I2C bus, which resulted in the I2C communication between devices being both unable to carry data and execute significantly slower than its intended baud rate (speed).
2. Various PWM channels of the generator IC failing, and being unable to produce PWM signals above approximately 0.1 V.
3. Inability to communicate to two separate four channel ADC's as they possessed the same address on the I2C bus.

The reason for these issues and how they were resolved, will follow below:

1. Clock stretching of the I2C bus was caused by SCL line (the I2C clock line) being pulled low, on the board. This issue was solved by utilising a high magnification lens to individually check pins across the various I2C attached components and ensuring no solder bridges existed – in the end a small solder bridge between SCL and ground, which was amended by running a small knife between these pins to ensure separation.
2. The PWM generator on the board seems to be quite venerable to static electricity, with damage being possible during soldering, handling or even prior to purchasing the IC (Chip could be damaged before receival). This issue was noticed as during operation some motor configuration would not work, that is – as the 8 PWM channels each control one of four motors direction (forwards or reverse) – so it was observed that with all PWM signals set as a forward thrust configuration (utilising PWM channels 0, 2, 4 and 6), the motors operated; however in reverse (setting PWM channels 1, 3, 5 and 7) only two motors would spin. This issue was then diagnosed through the use of an oscilloscope to determine if the PWM generator was outputting the correct signals across all channels – from which it was observed that damaged channels would not produce the desired signal; thus the solution to this issue was to simply remove the damaged PWM generator and replace it with an undamaged one – which was then tested to ensure full operation.
3. After consulting the datasheet of the four channel ADC it was observed that the I2C address of the chips could be altered by applying a different voltage level to the address pins, thus this issue was solved by pulling all the address pins of one ADC high and pulling all address pins of the other low.

Regarding the issues related to the motor driver board, they were as follows:

1. H-bridge drivers were not receiving the correct voltages through the eight-pin connector to the breakout board.
2. H-bridge drivers were outputting a very small voltage despite receiving the maximum PWM signal from the generator on the breakout board.

These issues were resolved via the following:

1. Testing the individual pins of the eight-pin terminal mounted on the Breakout board to determine whether the correct voltages were being output. Further testing was then carried out to determine if the correct voltages were being carried through the eight-wire connector in both connected and unconnected (to the motor driver boards) to determine which piece of hardware the issue occurred on. If the eight-pin connector on the breakout board was producing incorrect voltages, then the breakout board required maintenance; if the voltages were correct on the breakout board and incorrect when applied across the eight-wire connection (which is unplugged to the motor driver board) then these wire's required reordering and reinforcing; if however the voltages were correct on the breakout board and the unconnected eight-wires, the issue was due to a fault on the motor driver boards (namely shorts to ground, or damaged H-bridge ICs – which would require replacement)
2. In this case, the checks as outlined above were performed and the issue was determined to be caused by the motor driver boards – specifically the H-bridge ICs on the boards – which had been internally damaged, in a way which shorted the outputs of the internal MOSFETs to ground rail. Thus, the chips required replacement, as it would be impossible to repair.

This ends the detailing of issues related to the hardware of the AUV, a variety of issue which resulted in the author expending far more hours than what could be considered acceptable to amend. Even more so when it is considered that these issues were not simply 'one off issues' but rather were recurrent and thus inspired the author's opinion on potential improvements to the hardware as outlined in '*4.4 Review of Hardware'* , as these would reduce the possibility of such issues occurring given the sensitive hardware would be contained as part of the controller and thus less likely to sustain damage. In fact, it is in the author's opinion that such changes would reduce the list of possible issues to the two documented as related to the motor driver board (as may be seen above).

6 Software Package

One of the greatest struggles which the author experienced in progressing forwards with the project, was the ability to understand software of the previous iteration of the project; the style of writing did not suit the idea of a modular project, through which pieces may be inherited from the previous as future iterations to build upon. The previous software package was both too great in length, complexity and possessed syntax which was difficult read. As combination of features which is deadly software which is intended to possess multiple editors over its lifetime.

Moving forwards, it was decided that the codebase should be redesigned – with Python 3 selected as the language to implement this software upon; as Python's nature leads itself to be ideal for a 'prototyping' based project, given it boasts of codebases which are generally much lesser in length when compared to other languages (C, C++, Java etc.) and possesses a 'style' of syntax which is much less complex. Both these features proved to be perfect for the idea of project continuality as software which is both short and simple to read, is much more suited to a project whose ownership is only retained for a short length of time – future owner's simply do not have the time to invest in fully understanding the existing codebase, rather then the codebase should be simple enough that it may be understood in one reading and short enough that it is not inefficient to re-examine the modules. Additionally, it should be noted that due to these features, software written in the Python language requires a much shorter development phase; which is excellent considering there is always the prosability of issues occurring – requiring debugging, refactoring or even re-developing to overcome.

The following sub-sections will outline the steps both setup the Raspberry Pi 3's operating system (OS), installation of various packages which were desirable and an overview of the developed software for the AUV 'Undyne'.

6.1 Raspberry Pi 3B Setup

Initially the Raspberry Pi was intended to utilise a lightweight, Ubuntu based, CLI operating system [40]; however, this proved to be unsuccessful as the bootloader of this OS (U-Boot [41], or Universal Boot) interfered with one of the software modules, which was unable to access the desired serial port due to it being assigned to the bootloader. This did not prove to be much of a setback, as ultimately Raspbian Stretch [42] was installed instead (an OS which did not experience this issue due to it's usage of the tradition Linux bootloader 'GRUB' [43].)

Firstly, with the Raspberry Pi connected to a monitor, keyboard and mouse, Raspbian Sketch was switched from desktop mode, to a CLI only mode; as for this project a GUI is unrequired. This was achieved through the following GUI options:

Menu > Preferences > Raspberry Pi Configuration

The Raspberry Pi may then boot to the command line, by default, by selecting “To CLI” in the boot option. Reboot the Raspberry Pi on prompt.

Once the Raspberry Pi has booted, next the network requires setup. This may be achieved through editing the ‘*wpa_supplicant.conf*’ file, in which wireless network data is stored, using the following command:

sudo nano /etc/wpa_supplicant/wpa_supplicant.conf

Enter the wireless network parameters using the following template (or use ‘sudo raspi-config’):

*network = { ssid=”**[network name]**”*

*psk=”**[network password]**”*

key=WPA-PSK

}

Now press ‘CTRL + X’ to exit and save the file.

Next, to enable various features of the Raspberry Pi which were required, type the following command:

sudo raspi-config

Navigate to option 5 – Interfacing options and enable the following: SSH, I2C and Serial (disable serial shell login, but enable serial port). Following this, it is now possible to remotely access the Raspberry Pi via SSH (Secure Shell).

First determine the Raspberry Pi’s IP address by typing: ‘*ifconfig’* into the command line, this should be noted.

Now from a device connected to the same network (preferably a Linux based OS) open the command line and type:

*ssh pi@**[IP Address]***

Confirm the authenticity of the shell when prompted, and then type the user password of the Raspberry Pi (which would have been assigned on first boot). The Raspberry PI, may now be remotely accessed utilising this method.

Next, it is desirable to move from the Raspberry Pi's native Wi-Fi to an alternative source; in the author's case a USB wireless dongle was selected and so the steps detailing the installation of its specific drivers will be outlined. In the case of the RTL8812AU [44] based wireless dongle – by default Linux does not provide drivers, thus the following steps were required:

git clone https://github.com/abperiasamy/rtl8812AU_8821AU_linux.git

sudo apt-get install linux-image-rpi-rpfv linux-headers-rpi-rpfv dkms build-essential bc

cd rtl8812AU_8821AU_linux

sudo nano Makefile

Next edit the following lines of text, to configure the make file to build drivers for the ARM architecture rather than the default x86 PC architecture.

CONFIG_PLATFORM_I386_PC = n

CONFIG_PLATFORM_ARM_RPI = y

Next build the Makefile by typing the following*: 'sudo ./Makefile'*

The drivers for the wireless should be successfully installed, finally edit the boot directories 'config.txt' to disable native Bluetooth and remap the full serial port 'tty/AMA0' to the GPIO pins, rather than the mini serial port 'tty/S0' which is tied to the GPIO by default.

This may be achieved through the following:

sudo nano /boot/config.txt

Then add the following two lines (to implement device tree overlays to achieve what was mentioned previously) to the bottom of the file:

dtoverlay=pi3-miniuart-bt

dtoverlay=pi3-disable-bt

Again press 'CTRL + X' to exit and save the file. Now reboot, and the Raspberry Pi should be successfully configured as required for the project.

6.2 Software Modules

The approach taken towards the development of the software took the form of an 'object-orientated' style, that is all the software modules of the AUV were split into separate classes dependant on their

functionality. With each class representing the physical objects (the sensor's, convertor's, motor driver's and the "flight controller".)

As such, the bulk of the software is contained within the following files:

bno055class.py, flightcontroller.py, ltc2990class.py, mcp3221class.py, motordriver.py and px3class.py

These classes contain the functions and methods which relate to the following:

bno055class.py – Defines functions relating to the BNO055 IMU (Inertial Measurement Unit), to output useful data such as euler co-ordinates, acceleration due to gravity, and linear acceleration

flightcontroller.py – Defines the functions which are used to calculate the control of the AUV, according to the sensor outputs from the BNO055 IMU and PX3 pressure sensor.

ltc2990class.py – Defines the functions which are used to operate the general purpose four channel ADC contained on the breakout board (Can display up to four voltage readings, and a temperature reading).

mcp3221class.py – defines the functions which are used to operate the single channel ADC used to calculate the battery voltage, and by extrapolation the remaining charge of the battery.

motordriver.py – defines the functions used to operate the four PWM controlled motors based on a motor demand of -100 to +100.

px3class.py – defines the functions used to read the data of the PX3 pressure sensor, from the four channel ADC, and convert this data to a depth reading (based on the gravity reading from the BNO055.

These modules are then utilised though the main programme – main.py – which calls each function on as required, whilst opening a communication link to the 'Command Center' GUI – GUI.py – which is operated from the user's laptop or similar device connected on the same network as the Raspberry Pi. This communication link is established via a TCP socket which sends (motor demand, PID parameters and desired Euler angles) and receives data (actual motor speed, after PID calculations, actual Euler angles and various other sensor data).

6.3 Software Module – bno055class.py

The first of the software defined classes, for the Adafruit BNO055 – a NDOF compass was created to access the following features of this sensor, via the Serial port 'tty/AMA0':

Absolute Orientation as Euler angles, the Magnetometer, the Gyroscope, the Accelerometer, Linear Acceleration, Gravity, Absolute Orientation as a Quaternion and Temperature.

The most vital piece of this process being the functions defined to enable communication via Serial (as determined from the datasheet of the BNO055 sensor), given simply writing and reading the Serial port is not possible on a device which possess an array of registers. Thus, the following three functions are essential to this class:

```
def writecommand(self, command, ack= True, maxattempts=3):
    attempt = 0
    while True:
        self.serial.flushInput()
        self.serial.write(command)
        if ack == False:
            return
        response = bytearray(self.serial.read(2))
        if not (response[0] == 0xEE and response[1] == 0x07):
            return response
        attempt += 1

        if attempt == maxattempts:
            raise RuntimeError("Too many re-send attempts")
```

Figure 6.3.1, the 'writecommand' function

The function as displayed above provided the means of taking a 'command' (which is determined by the two functions which will be outline below) and sending it via serial – then immediately confirming the status of this write to the BNO055, by reading the serial port and checking the first two bytes against the header '0xEE07'; the hex value which determines that the write was successful. If, however the write is not acknowledged, the function will resend the command until a maximum number of times is reached. It should be noted that is the variables 'ack' or 'maxattempts' are changed from their default values then the function will ignore the acknowledgement phase or set the maximum resends to this value of 'maxattempts'.

```
def serialwrite(self, address, data, ack = True):
    command = bytearray(4 + len(data))
    command[0] = 0xAA
    command[1] = 0x00
    command[2] = address & 0xFF
    command[3] = len(data) & 0xFF
    command[4:] = map(lambda x: x & 0xFF, data)

    response = self.writecommand(command, ack=ack)
    if response[0] != 0xEE and response[1] != 0x01:
        raise RuntimeError("Write unsuccessful")
```

Figure 5.3.2, the 'serialwrite' function

The previous figure provides the method for constructing a 'command' which will be written via serial, to interface with the correct address registers of the sensor. By default, error checking of the written message is turned off, however this may be turned off by passing assigning a False value to 'ack' when the function is called.

```
def serialread(self, address, count):
    command = bytearray(4)
    command[0] = 0xAA
    command[1] = 0x01
    command[2] = address & 0xFF
    command[3] = count & 0xFF
    response = self.writecommand(command)
    if not response[0] == 0xBB:
        raise RuntimeError("Read unsuccessful")
    count = response[1]
    response = bytearray(self.serial.read(count))
    if response is None or len(response) != count:
        raise RuntimeError("We have issues, my friend...")
    return response
```

Figure 6.3.3, the 'serialread' function

The above function provides the means of reading data from a register through the serial bus, by first assembling a command which is written to the sensor – which commands that data from the specified address is loaded onto the serial bus to be read.

In all the above figures the ability to easily debug the system was provided, using run time errors, which would be flagged if certain problems arose. This enables the user to determine the source of any potential ideas as the execution of the script would cease and the specified strings would be output to the terminal.

Through the use of these functions, it is possible to initialise the BNO055 chip (which is handled by a separate function, initialise() – which must be called before attempting to read data from the sensor). Following the initialisation of the chip, it is possible to read data from a specific address register; convert this data to a vector and then return these values. For example, the function geteuler() which returns a tuple of heading, roll and pitch.

6.4 Software Module – flightcontroller.py

This module provides the software implementation of PID control (inspired by YMFC-AL [45], an Arduino based quadcopter series) which provides decoupled PID control (as inspired by the NDRE-AUV autopilot system [46]) based on readings from the Euler angles (Heading, Roll and Pitch) as well as depth. With unique variable gain values across all controllers given all sensor outputs will not possess the same magnitude; as, for example, heading will produce values between -180 and 180 whereas the depth sensor will produce values between 0.5 and 4.5.

All PID calculations are handled by the functions headingPID(), rollPID(), pitchPID() and depthPID() which all operate similarly (as they are all PID controllers), with the main difference between each being the source of the error signal (actual sensor value subtracted from the desired) and the values of gain.

```
def headingPID(self, setpoint, K, error=0):
    self.Kp_Heading = K[0]
    self.Ki_Heading = K[1]
    self.Kd_Heading = K[2]

    heading = imu.geteuler()[0]
    self.Heading_Setpoint = setpoint
    self.headingerror = error
    # convert heading from 0:360 to -180:180
    if heading > 180:
        heading -= 360
    temperror = self.Heading_Setpoint - heading
    # Ensure the integral member is not in excess of the maximum demand
    self.imember_h += self.Ki_Heading * temperror
    if self.imember_h > self.maxdemand: self.imember_h = self.maxdemand
    elif self.imember_h < -1*self.maxdemand: self.imember_h = -1*self.maxdemand

    output = (self.Kp_Heading * temperror) + self.imember_h + (self.Kd_Heading * (temperror - self.headingerror))
    # Ensure output does not exceed maximum
    if output > self.maxdemand: output = self.maxdemand
    elif output < -1*self.maxdemand: output = -1*self.maxdemand
    # Add a slight deadband... The motors will thank you later.
    elif output > -1*deadband and output < deadband:
        output = 0
    self.headingerror = temperror
    return output, self.headingerror
```

Figure 6.4.1, the 'headingPID' function of the 'pidautopilot' class

The function outlined in the above figure, provides the PID control for the heading output of the BNO055 sensor which inherits the desired setpoint and gain values. With several global variables being utilised to enable storage of previous values as the controller iterates. In this case, as the sensor outputs heading as a value between 0 and 360, the sensor data need to be converted to enable the PID to return an output which reflects the shortest path to the desired setpoint. For example, instead of turning clockwise from a heading of 45 degrees to reach a setpoint of 0 degrees, the controller will signal the motors to turn anti-clockwise towards 0 degrees. Additionally, as the integral member of the PID calculation is the most likely to exponentially increase – the maximum value this may produce is limited to a maximum change in demand. The output is then calculated by combining the proportional, integral and derivative members of the PID calculation, which is also limited to a set value of maximum demand. Additionally, a small dead band region (and area in which the output is zero) is introduced due to the accuracy of the sensors – as they are subject to small variances when they should hypothetically be constant – this is required to ensure the life of the brushed DC motors of the AUV, especially considering a motor failing during operation could be disastrous for an

unmanned unit. This output is then either added or subtracted to the motors to produce differential thrust (where required – thrust should not be differential for pitch and depth control outputs).

As previously mentioned, the functions defining roll, pitch and depth PID control are similar in format and thus shall not be produced as figures – as the only change is in the variables which are utilised. All PID control functions are then called in separate function which returns the unique outputs and errors of each PID controller.

6.5 Software Module – ltc2990.py

This module defines the class used to operate the LTC2990 Analogue to Digital Converter, which support four channels of analogue input as well as temperature sensing capability. Communication with the between the IC and the Raspberry Pi, is handled via I2C – which is enabled through usage of the SMBus package.

Regarding the reading of the device, the most vital functions within the class are the functions: initadc(), temp() and voltage(). These provide the means of initialising and configuring the ADC, reading the temperature register and converting this to a float value and reading the voltage registers and converting to a float value.

```
def initadc(self):
    i2c.write_byte_data(address, creg, singlemode) # Write desired mode to the control register
    i2c.write_byte_data(address, treg, 0x00) # Trigger conversion
    time.sleep(1) # Give it a minute (well, a second) to think

    status = i2c.read_byte_data(address, sreg)
    control = i2c.read_byte_data(address, creg)
    #print("%s %s" %(status, control))
```

Figure 6.5.1, the 'initadc' function of the 'ltc2990' class

The figure above, displayed in 6.5.1, outlines the method utilised to initialise the ADC IC. This is handled by initially writing the constant variable 'singlemode' (which contains a value of '0x5F' – which corresponds to single acquisition of T, V1, V2, V3 and V4 registers being enabled) to the control register. Following this, a value of '0x00' is written to the trigger register – which starts the conversion process, 'triggering' it essentially. The execution is then paused briefly, to prevent reading of the registers before the IC is ready. The status and control registers may then be read back to determine if the write was successful (status register provides an output displaying which registers possess new readings, whereas the control register should contain the value written to it – that is '0x5F').

```
def temp(self, msb, lsb):
    # Determine flags which are set/unset
    DV = msb & 0b10000000 # Data Valid   - If greater than zero, data is new
    SS = msb & 0b01000000 # Sensor Short - If V1 voltage is too low during temp reading, will be greater than zero
    SO = msb & 0b00100000 # Sensor Open  - If V1 voltage is too high during temp reading, will be greater than zero

    msb = format(msb, '08b')[3:]
    lsb = format(lsb, '08b')
    temp = int(msb + lsb, 2)/16

    return temp #, DV, SS, SO
```

Figure 6.5.2, the 'temp' function of the 'ltc2990' class

This function, displayed in 6.5.2 provides the means of converting the values held in the MSB and LSB temperature registers (which are passed from the function 't1' – displayed in 5.5.3 – which simply reads these registers via I2C). Firstly, regarding the flags which can be raised within the temperature registers, bitwise operations are performed to check if the three most significant bits possess a binary value of one – which each correspond to the Data Valid, Sensor Short and Sensor Open flags. Following, the 'msb' and 'lsb' variables are formatted to binary form (with the first three bits of the 'msb' variable being sliced off).

The two binary values are then concatenated, converted to an integer value and divided by a value of sixteen (as per the method in the datasheet) to return a floating-point value of the temperature.

```
def t1(self):
    TEMPmsb = i2c.read_byte_data(address, tmpreg[0])
    TEMPlsb = i2c.read_byte_data(address, tmpreg[1])
    return self.temp(TEMPmsb, TEMPlsb)
```

Figure 6.5.3, the 't1' function of the 'ltc2990' class

```
def voltage(self, msb,lsb):
    # Determine flags which are set/unset
    DV = msb & 0b10000000   # Data Valid - If greater than zero, data is new
    SIGN = msb & 0b01000000 # Sign Bit - If greater than zero, implies negative voltage

    msb = format(msb, '08b')[2:]
    lsb = format(lsb, '08b')
    if (SIGN == 0):
        volt = int(msb + lsb, 2) * 0.00030518
    else:
        volt = int(msb + lsb, 2) * -0.00030518

    return volt
```

Figure 6.5.4, the 'voltage' function of the 'ltc2990' class

Finally, displayed in 6.5.4, the method for converting the values contained in the MSB and LSB voltage registers (which are passed from the functions 'v1', 'v2', 'v3', 'v4' and 'vcc' – a sample of format being displayed in 6.5.5). Firstly, the data flags are extracted from the variable 'msb' and a bitwise operation carried out to return a Boolean value of either '0' or '1' – to determine if a flag is set or unset. Next,

the ‘msb’ and ‘lsb’ variables are formatted to binary; with the first two bits of the ‘msb’ variable being sliced off. The sign bit is then utilised to determine whether the voltage read from the register is of a negative or positive value, then the ‘msb’ and ‘lsb’ values are concatenated and consequently cast to an integer value – then multiplied by positive or negative *0.00030516* to obtain the actual voltage (method according to the datasheet). A floating-point value of the actual voltage is then returned.

```
def v1(self):
    Vlmsb = i2c.read_byte_data(address, voltreg[0])
    Vllsb = i2c.read_byte_data(address, voltreg[1])
    return self.voltage(Vlmsb, Vllsb)
```

Figure 6.5.5, the ‘v1’ function of the ‘ltc2990’ class

6.6 Software Module – mcp3221.py

This module provides the class utilised to operate the MCP3221 – single channel – Analogue to Digital Converter, which similarly to the previous module for the LTC2990, communicates with the Raspberry Pi via the I2C bus – again utilising Python’s SMBus package to enable this.

The reading of this device is far simpler than that previous ADC, given it possesses only one channel of voltage conversion. The functions used to enable this are the ‘readadc’ and ‘voltage’, contained within the ‘mcp3221’ class.

```
def readadc(self):
    V = i2c.read_i2c_block_data(address, 0x00, 2)
    V = self.voltage(V[0], V[1])
    return self.vbatt(V), self.vbatt(V)/4
```

Figure 6.6.1, the ‘readadc’ function of the ‘mcp3221’ class

In the previous figure, 6.6.1, the method implemented in the ‘readadc’ function may be observed (in this case the ADC is implemented to read the terminal voltage of the battery, rather than it’s intended purpose of reading the water ingress sensor’s output). As for its operation, SMBus is utilised to read the MSB and LSB registers (‘0x00’ and ‘0x01’) – whose values are then passed to the ‘voltage’ function to determine the voltage across the ADC pin. This voltage was then scaled according to the voltage divider implemented to reduce the battery voltage to the ADC’s operating voltage range and returned as the total battery voltage and the estimated per cell voltage.

```
def voltage (self, msb, lsb):
    msb = msb << 8
    lsb = lsb
    v = msb + lsb

    return v*(3.3/4095)
```

Figure 6.6.2, the ‘voltage’ function of the ‘mcp3221’ class

Above, in 6.6.2, the function 'voltage' is displayed – which takes the arguments 'msb' and 'lsb' as inputs. The function then performs a bitwise operation to join the two binary values together, before utilising the scaling factoring (as determined from the datasheet) to output the voltage read by the device.

6.7 Software Module – motordriver.py

This module provides the methods required to operate the PWM generator IC, to produce an analogue output which may be interpreted the desired speed (by the H-Bridge ICs of the Electronic Speed Controllers).

The main functions utilised in this process are 'initialise', 'pwminit', 'setdemand', 'demandscale' and 'mapdemand' – contained within the class 'motordriver'. As with the previous modules, the PWM generator relies heavily on the I2C bus; so again, the SMBus package is used for access to this.

```
def initialise(self):
    # Enable output from the H-Bridge
    pi.write(27, pigpio.LOW)
    self.pwminit()
```

Figure 6.7.1, the 'initialise' function of the 'motordriver' class

The function displayed above, in 6.7.1, is simply used to activate the necessary components required to run the brushed DC motors of the AUV. This is done pulling the value of pin 27 – which corresponds to the OE pins of the H-Bridge IC – of the Raspberry Pi to a low logic value, thus enabling the ICs output pins. Secondly, the function pwminit() is called – which initialises the PWM generator IC for output.

```
def pwminit(self):
    # Mode settings
    mode1 = [0xA0, 0x04]
    sleep = [0xB0, 0x04]
    #print (mode1, sleep)
    # Commence the sleep period - Required to set prescaler bit (effects frequency)
    i2c.write_i2c_block_data(ADDR_PWM, 0x00, sleep)
    data = i2c.read_i2c_block_data(ADDR_PWM, 0x00, 2)
    #print(data)
    # Write PWM frequency
    freq = [0x03]
    i2c.write_i2c_block_data(ADDR_PWM, 0xFE, freq)
    freq = i2c.read_i2c_block_data(ADDR_PWM, 0xFE, 1)
    #print(freq)
    # Write to mode registers 1 and 2
    i2c.write_i2c_block_data(ADDR_PWM, 0x00, mode1)
    # Read back the mode registers
    mode = i2c.read_i2c_block_data(ADDR_PWM, 0x00, 2)
    #print(mode)
```

Figure 6.7.2, the 'pwminit' function of the 'motordriver' class

Above, in 6.7.2, the previously mentioned 'pwminit' function is displayed. The main purpose of this function is to set the PWM clock frequency as well as writing the desired modes to the PWM mode registers (with the sleep mode being utilised for setting the frequency, and mode1 being used to set the normal operation mode of the PWM generator). These operations are mainly carried out

utilising the I2C write functionality of the SMBus package, however the I2C read functionality may also be used to confirm the I2C communication was successful – which is essential in the case of debugging hardware issues.

```
def setdemand(self, demand):
    # Translate to 12-bit resolution
    for i in range(0, 4):
        #print("%s" % demand[i])
        if demand[i] > 100: demand[i] = 100
        elif demand[i] < -100: demand[i] = -100

        demand[i] = self.demandscale(demand[i], demandrange)
        #print("%s" % demand[i])

    block = [0] * 32
    demand, block = self.mapdemand(demand, block)
    #block = [0, 0, 255, 15, 0, 0, 0, 0, 0, 0, 255, 15, 0, 0, 0, 0, 0, 0, 255, 15, 0, 0, 0, 0, 0, 0, 255, 15, 0, 0, 0, 0]
    i2c.write_i2c_block_data(ADDR_PWM, MOTOR_REG, block)
    data = i2c.read_i2c_block_data(ADDR_PWM, MOTOR_REG, 32)
```

Figure 6.7.3, the 'setdemand' function of the 'motordriver' class

Displayed above in 6.7.3, is the function 'setdemand' which is utilised to set the desired demand of each motor – LT, LD, RT, RD – which is passed into the function, in the form of an array, and subsequently checked to ensure the values are in the range -100 to 100. The functions 'demandscale' and 'mapdemand' are then implemented to convert the demand from the previous range, to the range -4095 to 4095 – reflecting on the 12-bit resolution of the PWM generator – and to map these ranges across the eight PWM channels used. This 32-byte array returned, by the later function, is then written to the PWM generator via I2C – and subsequently may be read back, to ensure correct transmission when error checking.

```
def demandscale(self, demand, demandrange):
    demandscalar = 4095/demandrange
    return int(demand * demandscalar)
```

Figure 6.7.4, the 'demandscale' function of the 'motordriver' class

Above, in 6.7.4, the function 'demandscale' is outlined – which is implemented to, as previously mentioned, to convert from the input range (-100, to +100) to the output range (-4095, to 4095). This is achieved by, effectively, multiplying the demand by a value of 40.95 – then casting the output of this to an integer value, which is subsequently returned.

```
def mapdemand(self, demand, datablock):
    # LT, LD, RT, RD motor registers
    registers = [30, 20, 14, 4]
    # encode the demand accross 32 bytes
    for i in range(0,4):
        #Thrust Phase
        if (i%2 == 0):
            # Forward PWM - Thrust
            if (demand[i] >= 0):
                datablock[registers[i]] = demand[i]
                datablock[registers[i]+1] = demand[i] >> 8
            # Reverse PWM - Thrust
            else:
                datablock[registers[i] - 4] = abs(demand[i])
                datablock[registers[i] - 3] = int(abs(demand[i])) >> 8
        else:
            # Ensure max demand isn't 4095
            if (demand[i] == 4095):
                demand[i] = 4094
            elif (demand[i] == -4095):
                demand[i] = -4094

            # Forward PWM - Dive
            if (demand[i] >= 0):
                datablock[registers[i]] = 4095 - (demand[i])
                datablock[registers[i] + 1] = int(4095 - abs(demand[i])) >> 8
            # Reverse PWM - Dive
            else:
                datablock[registers[i] - 4] = 4095 -(abs(demand[i]))
                datablock[registers[i] - 3] = int(4095 - (abs(demand[i]))) >> 8

    return demand, datablock
```

Figure 6.7.5, the 'mapdemand' function of the 'motordriver' class

Displayed in 6.7.5, is the method used to create an array of data (possessing 32 indexes) to map the requested demand across the 32 registers of the PWM generator – consisting of four registers per PWM channel; two of which determine the MSB and LSB of the PWM off time, and two which determine the PWM on time. Motors mapped to each channel of the PWM generator is displayed below in 6.7.6. Where Thrust motors are set utilising the on time of the PWM signal, whereas Dive motors are set using the off time of the PWM signal – that is different phase PWM signals are used, which reduce the total current drawn per ESC, given each ESC provides the control of a single Thrust and Dive motor.

```
# Registers are mapped as such:

# LED0 - RIGHT THRUST (Forward)
# LED1 - RIGHT THRUST (Reverse)
# LED2 - RIGHT DIVE   (Forward)
# LED3 - RIGHT DIVE   (Reverse)
# LED4 - LEFT THRUST  (Forward)
# LED5 - LEFT THRUST  (Reverse)
# LED6 - LEFT DIVE    (Forward)
# LED7 - LEFT DIVE    (Reverse)
```

Figure 6.7.6, motor configuration and mapped PWM channel

6.8 Software Module – px3class.py

The purpose of this module is to receive the output of the PX3 pressure sensor, from the four channel ADC (the class defined in ltc2990.py shall be used to achieve this). Of course, since the operating voltage of the pressure sensor is 5V – a small piece of prototyping board and two 20K Ohm resistors were used to implement a voltage divider, to reduce the maximum output to roughly 2.5 Volts.

```
def pressure(self):
    voltage, temp = adc4.pressure()
    # Voltage divider will be used... ltc2990 cannot exceed 3.3V analogue input
    voltage = (2*voltage)
    pressure = voltage*pervoltage
    print("%s Bars, at %s Celsius" % (pressure, temp))
    return pressure, temp
```

Figure 6.8.1, the 'pressure' function of the 'px3' class

In 6.8.1, the function defined to perform the sensor reading, scaling and conversion to pressure is outlined. The voltage and temperature are obtained by reading the four channel ADC using the 'pressure' function – which simply returns a tuple of the voltage across pin V1 and the temperature, as recorded on the ADC. The voltage is then doubled – to counter the fact that the voltage divider halved the output voltage from the PX3 sensor. This corrected value is then scaled by the value of 'pervoltage' (which equates to 6.25), this converts the value from a voltage reading – to a reading of the pressure (in Bars).

```
def simpledepth(self):
    bars, temp = self.pressure()
    depth = bars * 1.019716
    print("Depth: %s m" % depth)
    return depth
```

Figure 6.8.2, the 'simpledepth' function of the 'px3' class

Above in 6.8.2, the function which provides the method to obtain a simple reading of the depth is displayed. Which simply uses the 'pressure' function to obtain the pressure, and then scaling this by a value of '1.019716' – a method described by Sea-Bird Scientific [47], for freshwater applications; where the depth is shallow.

```
def complexdepth(self):
    # More sophistcated method which accounts for compressity of water due to gravity
    # Proves more accurate at greater depth
    bars, temp = self.pressure()
    gravity = imu.getgravity()
    c1, c2, c3, c4 = -1.82*pow(10, -15), 2.279*pow(10,-10), -2.512*pow(10, -5), 9.72659
    # The following formula return depth in meters
    depth = ((c1*(bars + c2))*(bars - c3)*(bars + c4)*bars)/gravity
    print("Depth: %s m" % depth)
    return depth
```

Figure 6.8.3, the 'complexdepth' function of the 'px3' class

Additionally, a second method is employed to obtain a more accurate reading of the depth – as displayed in 6.8.3. In this case, both the pressure and gravity are used – as well as four constants (assuming a temperature of 0 °C and a sea surface salinity value 35 PSU). In this case, again the 'pressure' function is used to obtain the pressure in bars; additionally, the Adafruit BNO055 is used to obtain a value for the gravity. The method employed is used Professionally by Sea-Bird Scientific, and originally documented within a UNESCO (The United Nations Educational, Scientific and Cultural Organisation) technical marine science paper [48].

6.9 GUI

Regarding the GUI designed for the AUV, the "Undyne AUV Command Center", this module is launchable from the operators 'Command Center' – their laptop, or similar device with the ability to launch a python script.

This integration between the AUV and the user's hardware is performed by utilising a combination of the 'Socket' – to provide the methods required to setup a TCP socket, 'tkinter' – to provide the basic GUI functionality, and 'ttkinter' – an extension which allows for a more modern GUI, as well as features which the 'tkinter' package lacks.

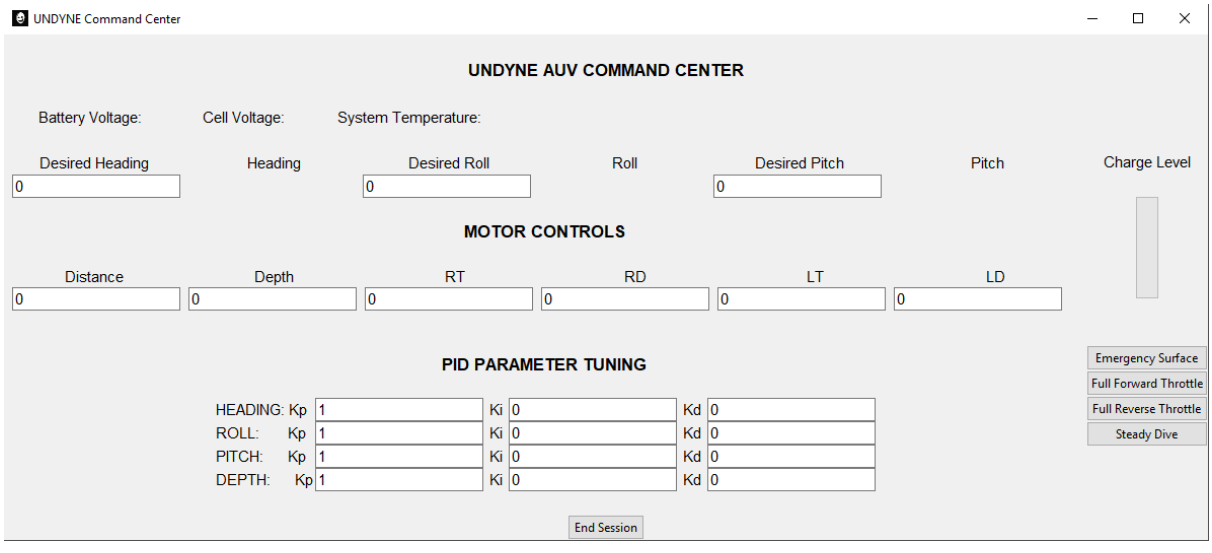


Figure 6.9.1, the GUI provided via Pythons 'tkinter' library

Regarding the features of the GUI, as observed within 6.9.1, from the top – there is system statistics: battery voltage, cell voltage and system temperature which are displayed via labels possessing a variable string value. Below, there is the means for setting: desired heading, roll and pitch – as well as labels displaying the current heading, roll and pitch. Below, there is an array of entries which are utilised to input distance to be travelled, desired depth, as well as the motor demands. And in the bottom most frame, there is a matrix of entries to set the PID control parameters for heading, roll, pitch and depth (it should be noted that distance could also be implemented as a PID – however the sensor method is too inaccurate to be deemed much more than a novel feature). In the right frame, there exist the means of displaying the current battery voltage in a bar, estimating the charge remaining in the battery pack – as well as several clickable buttons which provide useful pre-set options for the motor controls.

Any input into the GUI is then transmitted via a TCP socket – utilising 'codewords' or pre-transmission strings, to enable the Raspberry Pi to acknowledge that data is being transmitted and ensure the transmitted string of data is then parsed appropriately.

7 Control Design

As previously mentioned, the control theory applied within the software is that of PID (Proportional Integral Derivative) – for which the standard equation is displayed below in 7.0.1

$$u = \underbrace{K_p e}_{\text{Proportional Term}} + \underbrace{K_i \int_0^t e\,dt}_{\text{Integral Term}} + \underbrace{K_d \frac{d}{dt} e}_{\text{Differential Term}}$$

Figure 7.0.1, the typical PID equation

Using this type of control theory, decoupled control of the AUV was implemented – with a PID loop for each of heading, roll, pitch and depth, due to significant variance is the sensor output range; thus, it is desirable that each PID parameters may be adjusted independently and enabling each control loop to have control over the motors corresponding to the type of motion that occurs.

In case of the control loops which were applied differentially, this was applied to the control outputs of heading and roll. Heading being applied with opposite signage to the thrust motors and roll being applied similarly to the dive motors. On the other hand, the control outputs for pitch and depth were applied non-differentially – as both dive motors were required to actuate in the same direction to produce either a diving or pitch adjustment motion.

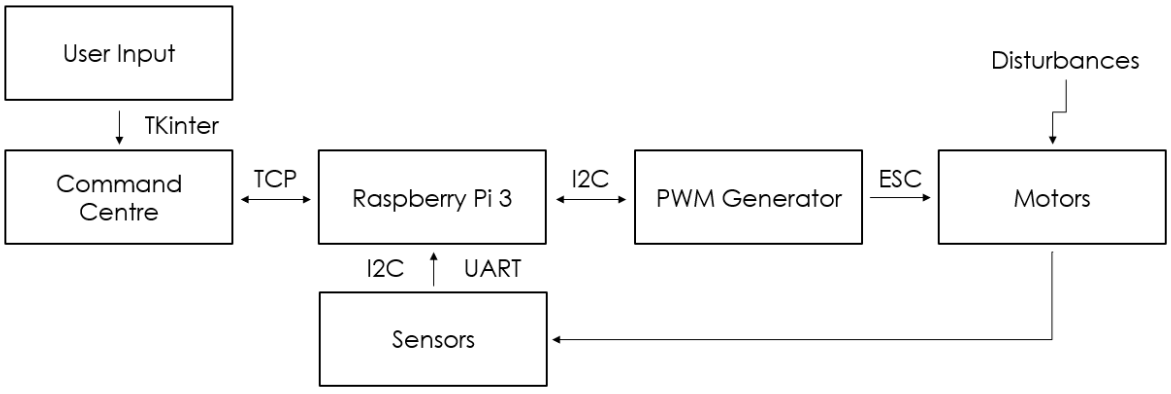


Figure 7.0.2, simplified control blocks for the submarine

In 7.0.2, it may be observed in a simplified form, how the control for the entire AUV system operates. With data being input from the GUI, to the main programme via 'tkinter' which then communicates with the main programme running on the Raspberry Pi (via TCP) which sends the required data to perform the various PID calculations, as well as the nominal operating speed desired for each motor. This data is then encoded via the PWM generator and applied across the motors (via the Electronic

Speed Controllers), which after experiencing the disturbance applied by water (as the result of heave, surge and sway) which results in the sensors reading a value – which is unlikely the desired values. A more complicated, and further broken-down version of this control system may be observed in 7.0.3 – with some assumptions being made as testing was not achieved, due to time constraints and sensor methods being unavailable.

The result of which lead to the ideal PID parameters of each control type being unobtained (the author had planned on implementing defaults on the GUI which loaded the submarine with ideal parameters, to simplify user experience). However, from some laboratory experiments it was possible to determine the system was functioning as desired (even without the ideal PID control parameters being known) – with each of heading, roll and pitch being tested via rotation in the appropriate direction, to determine motor output; which proved to be successful. The heading control loop, successfully applied a differential signal across the thrust motors of the AUV, providing confirmation of successful directional control. Roll provided a differential signal across the dive motors, as intended, which confirms the successful of the AUV's stability control. Additionally, the pitch control loop provided identical signals across the dive motors, as desired, confirming successful control of the AUV's pitch angle. Relating to the depth control, this was only tested at surface level – which proved successful, as the unit provided the correct atmospheric pressure as an output. Thus, under the assumption that the pressure sensor remains operational in submersion – the depth control of the AUV is a success.

Additionally, it was not possible to determine the correct transfer functions for the motors due to the lack of an onboard tacho meter, and the lack of testing to determine the velocity of the craft.

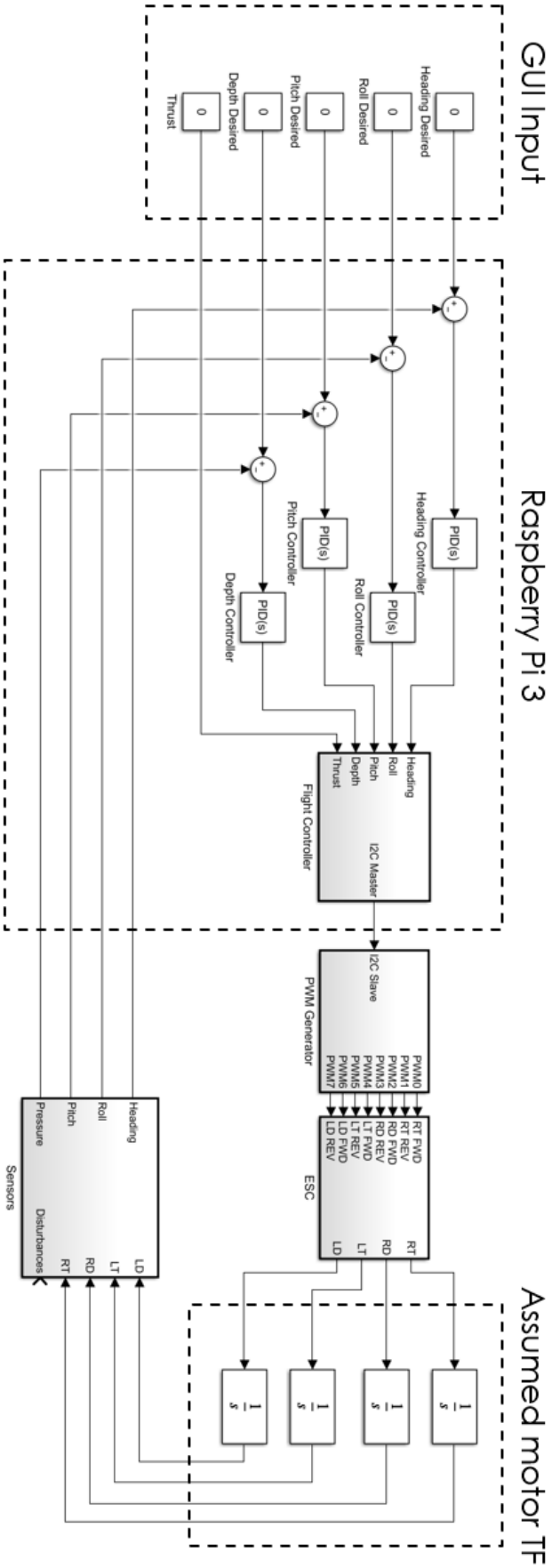


Figure 7.0.3, AUV mathematical model

8 Future Development

Regarding the issues which arose during the development of the AUV during this project, for progress in the future it would be advised that the Raspberry Pi 3 as a controller is a poor choice; given it possesses no native PWM or ADCs on the board, requiring additional hardware support to be installed – which can prove to be problematic as the PWM module chosen, through the author's experience, has proven to be quite susceptible to static electricity and the ADC modules included in the design are unnecessary when there are a multitude of existing microcontrollers which implement this, as well as onboard PWM.

One such example would be the Arduino Mega 2560 REV3 [49], which possesses 15 dedicated PWM pins (native to the board) and 16 analogue input pins (to support ADC). This alone greatly reduces the hardware which would be required on the breakout board, considering it mainly consists of ADCs, the PWM generator and remapping the GPIO. It should additionally be noted that the Arduino Mega supports an input of up to 20 Volts, which means it possesses an onboard power converter which reduces operating voltage to 5 Volts; which would prove ideal considering the battery voltage (of a four cell Li-Po) is 16.8 Volts at full charge, and the only remaining hardware would be the H-Bridge motor drivers (which may operate at a voltage of 5 volts). Thus, it would be possible to shrink the existing hardware down to a microcontroller, the motor driver boards and the selected sensor suite. The only apparent downside would be the communication difficulty that would arise; as the Arduino does not support Wireless natively, or USB for that matter (of course it is possible to breakout these); however communication with the AUV via a wireless-network proves to be an issue as it requires that submarine and command centre are connected to the same network – instead it would be far simpler to communicate via radio waves; perhaps by utilising a radio telemetry kit, where the transmitter/receiver floats upon the surface of the water (given radio waves do not propagate well through water as a medium). This would open an array of options for control, from wireless controllers with transmit/receive functionality (like controllers utilised within Quadcopter control) to wireless control via an android (or iOS) device.

Regarding future development, with respect to software, it would be possible to implement an underwater camera (utilising USB connection, would be advisable). This could open several further developments of the AUV, with the assistance of software packages such as OpenCV. Some of the following ideas could prove to be strong implementations of this:

- Using OpenCV [50] and an Underwater camera, to enable the AUV to follow, tape provided tracks along the bottom of the wave tank (with the possibility of using different coloured tape and a pressure sensor to determine the desire depth).

- A SLAM [51] (Simultaneous Localization and Mapping) control method could be implemented to provide the AUV with the means of detecting and avoidance of obstacles within the wave tank. It should be noted that this may be difficult to implement on a camera with poor submerged vision and may be simpler to devise a mapping method using an ultrasonic module.

9 Conclusion

With the project development cycle having reached its conclusion, it may be argued that management of the project was largely successful. Having fully developed a functional prototype, with a companion user interface for control – through which two points of the outlined specification were meet; 'To gain an understanding of unmanned systems' and 'To further develop and test a prototype robotic submarine. This includes both the physical system and related electronics'. However, due to time constraints leaving little margin for error in the development cycle – and the inevitable issues experienced, with the hardware and software design – testing of the AUV was not a possibility, leaving one member of the specification unchecked; 'To develop and test control algorithms (autopilots) on the vehicle.' In regards to which, the control algorithms were developed and their functionality tested – however without the ability to perform a submerged test, it was not possible to test these control algorithms in their target environment and thus, it was impossible to acquire the ideal control parameters to be implemented (which does not pose to much of an issue, outside of default PID values being less than ideal – as these parameters may be changed on demand, via the accompanying controller application). Additionally, as consequence of testing being out of the question, it was not possible to design an accurate mathematical model of the vehicle – as motor transfer functions, when surrounded by water, are unknown; as well as the disturbances experienced (which would be negligible, unless the water tank's ability to produce water waves was implemented). However, it was still possible to work around this by producing a mathematical model where the motor transfer functions were assumed, and disturbances are neglected (or assumed to be extremely small – in the case of still water, with no current – or external forces – acting on the vehicle). To this end it was possible to achieve, albeit not as precisely as desired by the author, the specification point outlined 'To develop a mathematical model of the provided robotic submarine.'

Additionally, as none of the possible risks outlined caused any issues throughout the development cycle; it can be argued that with the awareness of such risks, it required little trouble to ensure the project was developed safely. In fact, the closest the author came to experience one of these outlined hazards – was closer to the projects genesis; where when testing provided peripherals, it was observed that the original Li-Po battery was swiftly discharging and noticeably swollen; usage of this battery was immediately discontinued, as it posed a significant fire hazard.

Thus, concludes this report on the 'Development of an Unmanned Underwater Vehicle', which aimed to detail the development of this AUV – which has been titled 'The Undyne' [52], referencing the determination required to ensure completion of the vessel – which experienced no end of problems, due to the reliability of the selected hardware.

10 References

[1] QUB Online, 'ELE3001 Handbook' [online]. Available: https://learning.qol.qub.ac.uk/2171/ELE/3001-FYR-QUB/Resources/ELE3001%20Handbook%201718.pdf [Accessed: 12/01/2018].

[2] USGS, 'How much water is there on, in and above the earth' [online]. Available: https://water.usgs.gov/edu/earthhowmuch.html [Accessed: 12/01/2018].

[3] NOAA, 'How much of the ocean have we explored' [online]. Available: https://oceanservice.noaa.gov/facts/exploration.html [Accessed: 12/01/2018].

[4] Wikipedia, 'Deep Diving' [online]. Available: https://en.wikipedia.org/wiki/Deep_diving [Accessed: 12/01/2018].

[5] Wikipedia, 'Radio Propagation' [online]. Available: https://en.wikipedia.org/wiki/Radio_propagation [Accessed: 12/01/2018].

[6] Exploring Our Fluid Earth, 'Light in the Ocean' [online]. Available: https://manoa.hawaii.edu/exploringourfluidearth/physical/ocean-depths/light-ocean [Accessed: 12/01/2018].

[7] Wikipedia, 'Very Low Frequency' [online]. Available: https://en.wikipedia.org/wiki/Very_low_frequency [Accessed: 12/01/2018].

[8] Wikipedia, 'Dead Reckoning' [online]. Available: https://en.wikipedia.org/wiki/Dead_reckoning [Accessed: 12/01/2018].

[9] Python, 'Global Interpreter Lock' [online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock [Accessed: 12/01/2018].

[10] Diving Almanac, 'Aristotle describes a diving bell' [online]. Available: http://divingalmanac.com/aristotle-describes-diving-bell/ [Accessed: 12/04/2018].

[11] W. Bourne, 'Inventions or Devices'. 1578.

[12] P. Omodeo, 'Sixteenth Century Professors of Mathematics

at the German University of Helmstedt'. Max Planck Institute for the History of Science, 2011.

[13] G. Tierie, 'Cornelius Drebbel'. H.J.Paris, 1932.

[14] Paul E. Fontenoy, 'Submarines: An Illustrated History of Their Impact'. ABC-CLIO, 2007, pp. 1 – 2.

[15] Connecticuthistory.org, 'David Bushnell and his Revolutionary Submarine' [online]. Available: https://connecticuthistory.org/david-bushnell-and-his-revolutionary-submarine/ [Accessed: 12/04/2018].

[16] Encyclopedia Britannica, 'Submarine Naval Vessel' [online]. Available: https://www.britannica.com/technology/submarine-naval-vessel [Accessed: 12/04/2018].

[17] Marine Insight, 'Peral Submarine – The World's First Electric Battery Powered Submarine' [online]. Available: https://www.marineinsight.com/maritime-history/peral-submarine-the-worlds-first-electric-battery-powered-submarine/ [Accessed: 12/04/2018].

[18] Submarine Force Museum, 'History of USS NAUTILUS' [online]. Available: http://www.ussnautilus.org/nautilus/index.shtml [Accessed: 12/04/2018].

[19] H.R. Widditsch, 'SPURV – The first decade'. University of Washington – Applied Physics Lab, 1973.

[20] Kongsberg, 'Autonomous Underwater Vehicle, REMUS 100' [online]. Available: https://www.km.kongsberg.com/ks/web/nokbg0240.nsf/AllWeb/D241A2C835DF40B0C12574AB003EA6AB?OpenDocument [Accessed: 12/04/2018].

[21] Military and Aerospace Electronics, 'Navy asks Hydroid to upgrade MK 18 unmanned underwater vehicle (UUV) in $27.3 million contract' [online]. Available: http://www.militaryaerospace.com/articles/2017/08/uuv-unmanned-upgrade.html

[22] Kongsberg, 'Hydroid Littoral Battlespace Sensing AUVs Enter Full Rate Production' [online]. Available: https://www.km.kongsberg.com/ks/web/nokbg0238.nsf/AllWeb/CE7799BE8CF784D7C1257B170052DEC0?OpenDocument [Accessed: 12/04/2018].

[23] Woods Hole Oceanographic Institution, 'REMUS 600' [online]. Available: http://www.whoi.edu/osl/remus-600 [Accessed: 12/04/2018].

[24] European Nuclear Society, 'Fusion' [online]. Available: https://www.euronuclear.org/info/encyclopedia/f/fusion.htm [Accessed: 13/04/2018].

[25] PSFC MIT, 'Alcator C-Mod tokamak' [online]. Available: https://www.psfc.mit.edu/research/topics/alcator-c-mod-tokamak [Accessed: 13/04/2018].

[26] Science Direct, 'ARC: A compact, high-field, fusion nuclear science facility and demonstration power plant with demountable magnets' [online]. Available: https://www.sciencedirect.com/science/article/pii/S0920379615302337 [Accessed: 13/04/2018].

[27] Codecogs, 'Cavitation' [online]. Available: http://www.codecogs.com/library/engineering/fluid_mechanics/fundamentals/cavitation.php [Accessed: 13/04/2018].

[28] Telegraph, 'Will supercavitation make submarines the new Concorde?' [online]. Available: https://www.telegraph.co.uk/sponsored/technology/technology-trends/12121971/supercavitation-technology.html [Accessed: 13/04/2018].

[29] RChelisite, 'Li-Po battery charging and safety guide' [online]. Available: http://www.rchelisite.com/lipo_battery_charging_and_safety_guide.php [Accessed: 12/01/2018].

[30] RECOM, 'Innoline DC/DC-converter', R-7XXX datasheet, 2014 [online]. Available: https://docs-emea.rs-online.com/webdocs/1339/0900766b813390bd.pdf [Accessed: 13/04/2018].

[31] ON Semiconductor, '800 mA, Adjustable Output, Low Dropout Voltage Regulator', MC33269 datasheet, 2014 [online]. Available: https://docs-emea.rs-online.com/webdocs/13e4/0900766b813e44e3.pdf [Accessed: 13/04/2018].

[32] Infineon, 'SPI Programmable H-Bridge', TLE8209-2SA datasheet, 2012 [online]. Available: https://docs-emea.rs-online.com/webdocs/14d6/0900766b814d6cc8.pdf [Accessed: 13/04/2018].

[33] NXP, '16-channel, 12-bit PWM Fm+ I2C-bus LED controller', PCA9685 datasheet, 2010 [online]. Available: https://docs-emea.rs-online.com/webdocs/0f65/0900766b80f65101.pdf [Accessed: 13/04/2018].

[34] Linear Technology, 'Quad I2C Voltage, Current and Temperature Monitor', LTC2990 datasheet, 2010 [online]. Available: https://docs-emea.rs-online.com/webdocs/1329/0900766b81329f94.pdf [Accessed: 13/04/2018].

[35] Microchip, 'Low Power 12-Bit A/D Converter with I2C™ Interface', MCP3221 datasheet, 2006 [online]. Available: https://docs-emea.rs-online.com/webdocs/1380/0900766b81380b0c.pdf [Accessed: 13/04/2018].

[36] Bosch, 'Intelligent 9-axis absolute orientation sensor', BNO055 datasheet, 2014 [online]. Available: https://cdn-shop.adafruit.com/datasheets/BST_BNO055_DS000_12.pdf [Accessed: 13/04/2018].

[37] Adafruit, 'BNO055 Absolute Orientation Sensor with Raspberry Pi & BeagleBone Black' [online]. Available: https://learn.adafruit.com/bno055-absolute-orientation-sensor-with-raspberry-pi-and-beaglebone-black/hardware [Accessed: 13/04/2018].

[38] Honeywell, 'Heavy Duty Pressure Transducers', PX3 datasheet, 2016 [online]. Available: https://docs-emea.rs-online.com/webdocs/14d9/0900766b814d9e04.pdf [Accessed: 13/04/2018].

[39] Autodesk, 'Eagle' [online]. Available: https://www.autodesk.com/education/free-software/eagle [Accessed: 12/01/2018].

[40] Ubuntu Pi Flavour Maker, 'Download' [online]. Available: https://ubuntu-pi-flavour-maker.org/download/ [Accessed: 13/04/2018].

[41] GitHub, 'u-boot' [online]. Available: https://github.com/u-boot/u-boot [Accessed: 13/04/2018].

[42] Raspberry Pi, 'Raspbian' [online]. Available: https://www.raspberrypi.org/downloads/raspbian/ [Accessed: 13/04/2018].

[43] GNU, 'GNU GRUB' [online]. Available: https://www.gnu.org/software/grub/ [Accessed: 13/04/2018].

[44] Realtek, 'RTL8812AU' [online]. Available: http://www.realtek.com.tw/products/productsView.aspx?Langid=1&PNid=21&PFid=57&Level=5&Conn=4&ProdID=397 [Accessed: 14/04/2018].

[45] GitHub, 'ymfc' [online]. Available: https://github.com/eduardoscamargo/ymfc [Accessed: 14/04/2018].

[46] T. I. Fossen, ”Decoupled Control Design” in ‘Guidance and Control of Ocean Vehicles’. John Wiley & Sons, 1994, pp. 114 – 125.

[47] Sea Bird Scientific, ‘AN69: Conversion of Pressure to Depth’ [online]. Available: http://www.seabird.com/document/an69-conversion-pressure-depth [Accessed: 14/04/2018].

[48] UNESCO, ‘Algorithms for computation of fundamental properties of seawater’, UNESCO technical papers in marine science #44, 1983.

[49] Arduino, ‘ARDUINO MEGA 2560 REV3’ [online]. Available: https://store.arduino.cc/usa/arduino-mega-2560-rev3 [Accessed: 14/04/2018].

[50] OpenCV, ‘OpenCV’ [online]. Available: https://opencv.org/ [Accessed: 14/04/2018].

[51] Kudan, ‘An Introduction to Simultaneous Localisation and Mapping’ [online]. Available: https://www.kudan.eu/kudan-news/an-introduction-to-slam/ [Accessed: 14/04/2018].

[52] Undertale Wiki, ‘Undyne’ [online]. Available: http://undertale.wikia.com/wiki/Undyne [Accessed: 12/01/2018].

11 Appendix

11.1 Appendix 1 – Blank Risk Analysis Form

Health and Safety Regulations- Final Year Project

Project Title:	
Supervisor:	
Moderator:	

During your final year project you are expected to carry out your work independently of your supervisor and it is important that you are aware of the potential hazards involved in the work. You must discuss these hazards with your supervisor and determine the level of risk at each stage of the project. The level of risk is normally divided into three categories – HIGH, MEDIUM and LOW.

Risk Category				
HAZARDS	**LOW**	**MEDIUM A**	**MEDIUM B**	**HIGH**
Electrical				
Mechanical				
Chemical				
Radiation				

An example of LOW risk is the operation of a PC where access to the internals is not required. The MEDIUM risk category can be sub-divided into:

(a) After training the student is competent in the procedures and may operate without direct supervision, **but not in a room alone.** *Example* – PC-controlled low-voltage measurements.

(b) After training the student is competent in the procedures and may work without direct supervision, **but a staff member must be present in the laboratory.** ***Example* – operation of a rotating machine.**

The HIGH risk category requires continuous supervision by staff.

It is necessary to have proof that the risk categories of the various parts of the project have been assessed and agreed, so please complete the form and return it to your supervisor **before starting the practical work.**

I have discussed the risks involved in my Project and have agreed that the following areas fall into the risk categories.

CATEGORY	**DETAILS**
LOW	
MEDIUM A	
MEDIUM B	
HIGH	

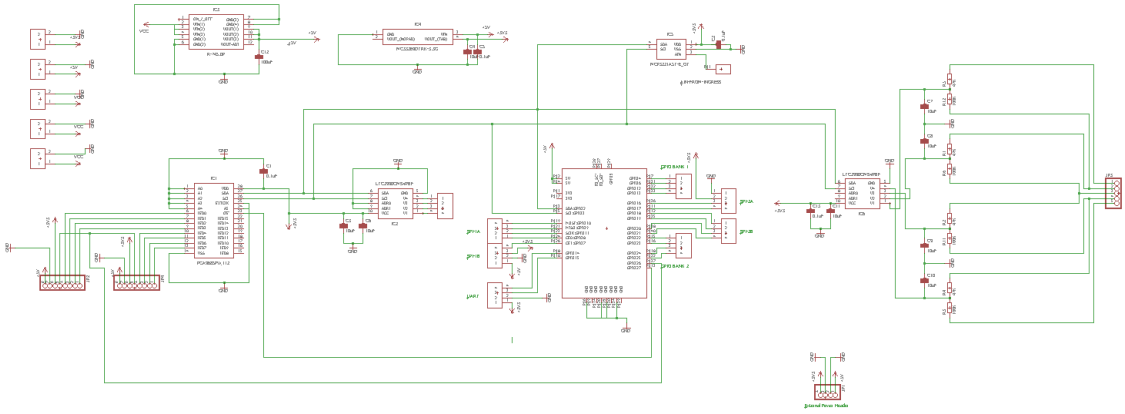
I have received training from my supervisor on the correct operation of the equipment necessary to complete my project. I have been made aware of other projects within the area and any dangers that may be associated with them. I have been advised that undergraduates are not permitted to work in isolation

Student's name:

Student's signature: Date:

Supervisor's signature: Date:

11.2 Appendix 2 – Full Breakout Board



11.3 Appendix 3 – Complete Software Repository

Available: https://github.com/wparr02/Undyne-V1.0