MONASH MOTORSPORT

FINAL YEAR THESIS COLLECTION

Development and Integration of a Computing Platform for an Autonomous Vehicle

James Wyatt - 2018

The Final Year Thesis is a technical engineering assignment undertaken by students of Monash University. Monash Motorsport team members often choose to conduct this assignment in conjunction with the team.

The theses shared in the Monash Motorsport Final Year Thesis Collection are just some examples of those completed.

These theses have been the cornerstone for much of the team's success. We would like to thank those students that were not only part of the team while at university but also contributed to the team through their Final Year Thesis.

The purpose of the team releasing the Monash Motorsport Final Year Thesis Collection is to share knowledge and foster progress in the Formula Student and Formula-SAE community.

We ask that you please do not contact the authors or supervisors directly, instead for any related questions please email info@monashmotorsport.com

MONASH University
Engineering

Development and Integration of a Computing Platform for an Autonomous Vehicle

James Wyatt

October 2018

Supervised by Prof. Tom Drummond

Department of Electrical and Computer Systems Engineering

Significant Contributions

In pursuit of this project the author:

- Chose the computing hardware that will be used for the autonomous system. This hardware was purchased, assembled and configured, ready for other subsystems to utilize.
- Investigated various autonomous software architectures, and eventually chose Robot Operating System (ROS). From this, in consultation with the other subsystem designers, the author defined the autonomous system architecture. This includes the processing split (what algorithms run in what node, on what machine), as well as the data flow and message structures between the nodes.
- Created a 'Dashboard' utilizing technologies such as MQTT and NodeJS, which enables easy monitoring of the autonomous computers from a web-based application.
- Designed a system for monitoring the health of each processing node using heartbeats, which ensures the system quickly fails into a safe state every time a fatal error occurs.
- Developed the ROS 'master' library to facilitate global state control, and easy integration of processing nodes into the autonomous system.
- Developed a robust custom message protocol for use over UART, which facilitates dynamic payload sizes and message integrity checks.
- Programmed a low-level micro-controller (PSoC 5LP) to implement UART communication using the aforementioned message protocol. Also implemented digital and analogue input and output logic, as well as laid the foundation for the autonomous state-machine (as defined by competition rules).
- Designed a PCB circuit for the PSoC 5LP, which protects the input/output pins, enables CAN bus communication, and interfaces with the EBS interlock circuitry(which was designed by another subsystem).
- Collaborated in discussions with the 'LV' subsection on how to cool the computing units once on the vehicle, and from this, performed thermal finite-element-analysis to validate the proposed cooling solution.
- Provided assistance to the other subsystem designers, and created ROS utility packages to facilitate processing and saving of data.

MONASH University Engineering

Department of Electrical and Computer Systems Engineering

ECE4095 Final Year Project 2018

James Wyatt

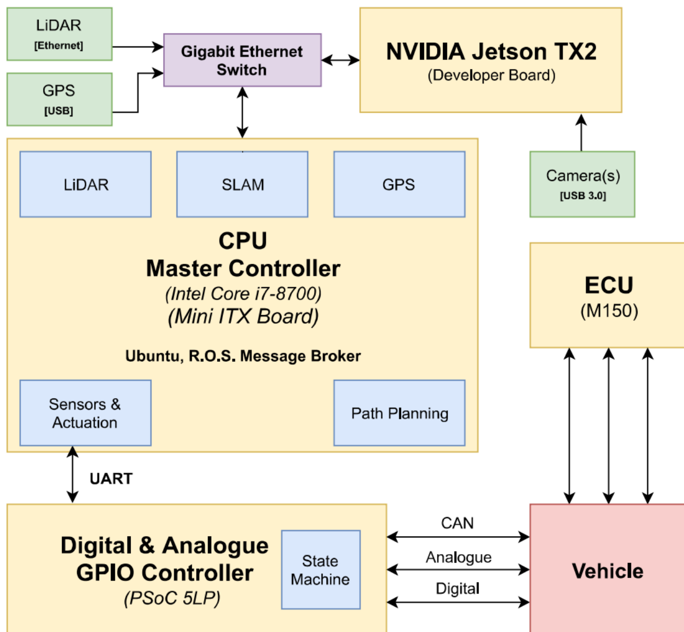
Development and integration of a computing platform for an autonomous vehicle

Supervisor: Professor Tom Drummond

The future is autonomous, and **Monash Motorsport** is taking great strides towards realizing this future. Development of an autonomous racing vehicle is underway and scheduled to be completed mid 2019. It must be capable of perceiving various cone-marked tracks and completing laps as quickly as possible.

This project focuses on the **computational platform** which the other subsystems use, as well as how the system **integrates** with the existing electric vehicle.

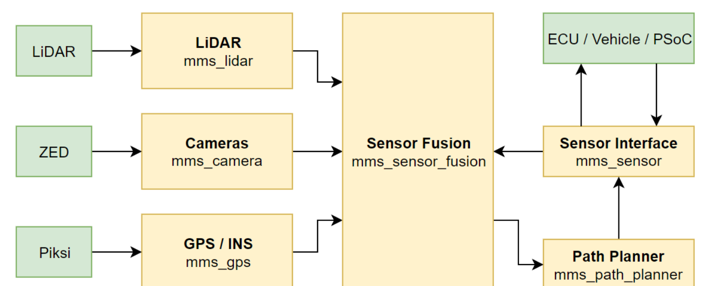
The 2017/2018 Electric Vehicle, M17-E



Hardware Overview

By leveraging the power of parallel processing using **CUDA**, **OpenCL** and **multi-threading**, the entire autonomous system is theoretically capable of processing over **1.5 trillion 32-bit floating-point** numbers every second.

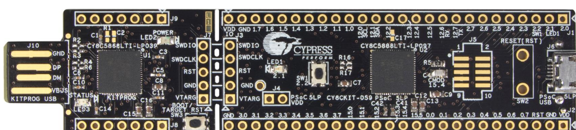
The platform uses **Robot Operating System**, which is a flexible, open-source framework for writing robotic software. It helps integrate computing clusters, and enables easy communication between multiple processing 'nodes' in the system.

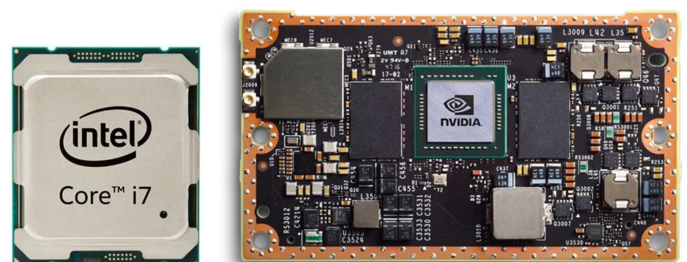


ROS Node Processing Overview

The autonomous system currently runs on 3 computing units: An **Intel i7 8700 CPU**, an **NVIDIA Jetson TX2**, and a **PSoC 5LP**. The 'i7' is best for **sequential** processing, whilst the 'Jetson' is best for **parallel** processing, thanks to its 256 CUDA-enabled cores.

The **PSoC** houses the autonomous state-machine, and is used for low-level digital and analogue logic.





Processing Hardware

ROS NVIDIA CUDA OpenCL NVIDIA intel

Executive Summary

What follows in this report is the design process of an autonomous racing vehicle's computational system and structure. This covers the original research performed, the concept generation phase, and the final design decisions. It concludes with detailed documentation of the system architecture, and various tasks that were performed in tandem.

Three processors were ultimately chosen for their speed, price and power/performance characteristics. Each of these computing devices play an important role in satisfying the requirements of an autonomous vehicle; which is to perceive and react to a fast-changing external environment.

1. An Intel i7, 8700 Processor
2. An NVIDIA Jetson TX2
3. A PSoC 5LP - Mounted on a custom PCB board

Robot Operating System is used as the backbone of the autonomous environment to integrate all the computing processes together into a single unified system. All R.O.S. processing nodes are required to implement a state-machine, and are controlled by one 'master' node which is the main controlling unit of the autonomous computing system.

A custom message protocol is defined, which allows for robust communication over a serial hardware layer using UART. This system allows for multi-byte payloads, and facilitates verification of received message structures through the use of two cyclic-redundancy-checks.

The system is designed with safety in mind, and utilizes a chain of heartbeats to ensure that all processing nodes are alive and functioning correctly. The last line of defence is built using non-programmable physical logic gates which are unable to fail due to poor programming.

This computing system shall be used in 2019 as the foundation for Monash Motorsport's first autonomous vehicle, M19-D.

Acknowledgements

Thanks to Monash Motorsport and the wonderful team of people there who supported me and my peers every step of the way, and gave us the opportunity and resources to begin building an autonomous racing vehicle.

My gratitude also extends to Xenon and NVIDIA, for sponsoring the team and this project, by providing two NVIDIA Jetson TX2s for us to test and develop with.

Furthermore, I extend my thanks to my supervisor, Professor Tom Drummond, for his weekly assistance and his dedication and enthusiasm for the project and the team.

I'd also like to thank my wonderful family for their continual support, food, and a warm home.

And finally, thanks to Georgia for late night company and a constant supply of free food.

Contents

Significant Contributions i

Poster ii

Abstract iii

Acknowledgements iv

1 Introduction 1
1.1 Motivation 1
1.2 The Competition 1
1.3 The Team 2
1.4 Project Objectives 2

2 Background 4
2.1 Existing Work 4
2.2 Literature Review 4
2.3 Previous Teams 5

3 Concept Generation 6
3.1 Hardware 6
3.1.1 Jetson TX2 7
3.1.2 Intel i7 8
3.1.3 Computing Summary 8
3.1.4 Low-Level Processor 9
3.2 Software 9
3.2.1 Nodelets 11
3.2.2 Multi-language Support 11
3.2.3 Services 11
3.2.4 Transform Frames 11

4 Physical System Overview 12
4.1 Computational Hardware 13
4.2 Data Collection Hardware 14
4.2.1 LiDAR 14
4.2.2 Cameras 14
4.2.3 GPS 14
4.3 Connectivity 15
4.4 PSoC PCB 15

5 Software 18

5.1 ROS Node Architecture . . . 18
5.1.1 LiDAR Node . . . 18
5.1.2 Camera Node . . . 18
5.1.3 GPS Node . . . 19
5.1.4 Sensor Fusion Node . . . 19
5.1.5 Path Planning Node . . . 19
5.1.6 Sensor Interface Node . . . 19
5.2 Master Node . . . 19
5.3 Child Nodes . . . 20
5.3.1 Child Node State Machine . . . 20
5.4 Heartbeats and Safety . . . 20
5.5 PSoC Processing . . . 22
5.5.1 Custom UART Message Protocol . . . 24

6 Computing Constraints **26**
6.1 Jetson TX2 CUDA . . . 26
6.2 Intel Integrated Graphics . . . 26
6.3 Eigen MKL . . . 28

7 Testing **29**
7.1 Testing Box and Trolley . . . 29
7.2 ROS Message Delay . . . 30

8 Miscellaneous Work **32**
8.1 Camera Testing . . . 32
8.1.1 Basler Camera Testing . . . 32
8.1.2 ZED Camera Stereo Image Recording . . . 33
8.2 Remote Control Actuation . . . 33
8.3 Kalman Filter and Time Message Reordering . . . 34
8.4 M19D Dashboard . . . 35
8.5 Thermal FEA . . . 36

9 Conclusion **39**
9.1 Requirement Satisfaction . . . 39
9.2 Future Work . . . 40
9.3 External Links . . . 40
9.3.1 Code Repository . . . 40
9.3.2 Presentation Video . . . 40

Bibliography **41**

Glossary of Terms **42**

A - PSoC Pinout **43**

B - PSoC Command List **44**

B - PSoC Command List **45**

C - ROS Message Structures **45**

D - PSoC Circuitry **46**

List of Figures

1.1 Skidpad Track Layout . . . 2

2.1 AMZ Driverless, 2017, At FSG . . . 5

3.1 The Neousys NUVO-5095GC . . . 7
3.2 The Jetson TX2 Module . . . 7
3.3 The two mini-ITX boards. Left: i7, Right: Jetson TX2 . . . 9

4.1 High-Level Hardware Diagram . . . 12
4.2 The PSoC 5LP . . . 13
4.3 The MoTeC M150 Engine Control Unit . . . 13
4.4 The VLP16 LiDAR . . . 14
4.5 The Stereolabs ZED Camera . . . 14
4.6 The Swiftnav Piksi-Multi GNSS Module . . . 15
4.7 The PSoC PCB . . . 17

5.1 ROS Node Overview Diagram . . . 18
5.2 Child Node State Diagram . . . 20
5.3 Example Heartbeat Flow Diagram . . . 22
5.4 Pictorial Representation of PSoC Processing Flow . . . 23
5.5 Autonomous Statemachine . . . 23

6.1 ArrayFire Monte Carlo algorithm timing . . . 27

7.1 Testing trolley on track . . . 30
7.2 Round trip message delay between two ROS nodes on networked computers . . . 31

8.1 Stereo Image Pair, Captured with two Basler Cameras . . . 32
8.2 Screenshot of RVIZ Vehicle Visualization . . . 34
8.3 Screenshot of the M19D Web Dashboard . . . 36
8.4 Autonomous Computing Box Thermal FEA Results . . . 37
8.5 Intel Stock Heatsink Thermal FEA Results . . . 38

List of Tables

3.1 Potential Hardware Options 6
3.2 Common Message Transmission Methods 10

5.1 UART Message Structure 24

Chapter 1

Introduction

1.1 Motivation

In the year 2000, Scott Wordley and a group of friends built Monash's first Formula SAE vehicle. This laid the foundation for Monash Motorsport, a student-led team based at Monash University Clayton, which is still operating to this day. Every year Monash Motorsport partakes in global competitions, based both in Australia and Europe, and strive to achieve a place on the podium. To accommodate the new changes occurring in the automotive industry, new classes of vehicles have been introduced over the years. In 2016, electric vehicles were allowed to compete at competition, side by side with the classic high-performance combustion cars that had sustained the competition for over 20 years. Exactly one year later in 2017, another class of vehicles were introduced into the F-SAE family: driverless. Due to significant and fundamental differences compared to the previous classes, these vehicles required a large set of new rules and regulations, as well as a modified set of events to compete in. In 2017 Monash Motorsport investigated the feasibility of developing their own autonomous vehicle. After a year of research, the decision was made to pursue the development and manufacture of a driverless vehicle, to potentially and hopefully compete in the 2020 Formula Student Germany competition, amongst others.

1.2 The Competition

The original competition involves 4 dynamic events:

- Skidpad – A figure-eight track. The vehicle must complete 2 laps of each side.
- Acceleration – A straight track with a length of 75 meters.
- Autocross – A short (less than 1.5km) but difficult track, to test the handling of the vehicle
- Endurance – A long track, approximately 22km in total length

For the driverless competition, the 'endurance' event is swapped for the 'track-drive' event. This event uses the same track as the autocross event, however the vehicle only needs to drive up to 10 laps, of a 200-500m track. This means that the maximum distance the vehicle needs to drive is 5km, compared to the 22km for endurance. Competitions are held all over the world, and include countries such as the USA, Germany, the UK, Austria and Australia. Different competitions have slightly different sets of rules, however ultimately the same 4 dynamic events still occur.

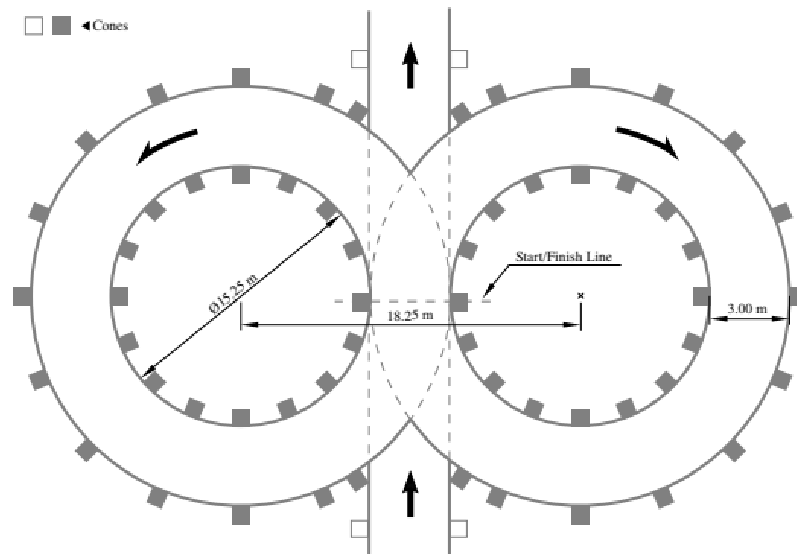


Figure 1.1: Skidpad Track Layout

The competition in Europe is slightly more competitive compared to the Australian competition. With high-performing teams such as AMZ and KA-Racing consistently out-performing almost all other teams. In Australia, the driverless vehicle category has still not been integrated into the competition, however there are high hopes that this will occur soon. For all driverless events, the track is laid out with 4 types of cones. Blue and yellow cones define the left and right sides of the track respectively, whilst orange cones define the start and finish boundaries. These cones are guaranteed to be a specific shape, size and colour, which means that the algorithms being developed require significantly less processing power compared to a full autonomous vehicle.

1.3 The Team

The team in 2018 consisted of over 100 students across multiple engineering and non-engineering disciplines. The team fully handles its own finances, management and manufacturing of the vehicle. Oversight from academic supervisors help guide the direction of the team and the engineering choices made, however ultimately the design and direction of Monash Motorsport is at the discretion of the leadership team. The team is split into multiple sections, these include aerodynamics, powertrain, suspension, chassis, business and autonomous systems. With the development of the electric vehicle in 2017, powertrain was split into two sections, one for combustion and one for the electric powertrain. At the beginning of 2018 the autonomous section was created to handle the development of the new driverless vehicle. This section consisted of 6 students undertaking their Final-Year-Projects, and another 12 students with various roles in the subsection.

1.4 Project Objectives

At the beginning of 2018, the autonomous management team decided upon a single goal:

To build an autonomous vehicle capable of completing all competition events in 2019.

This goal underpins everything that the autonomous subsection has done and shall continue to do into the future. Whilst some teams choose to optimise their vehicles for specific events,

Monash Motorsport aims to be the most respected FSAE team in the world, and performing well in just one event is not enough to achieve this goal. At the beginning of 2018, the autonomous team was split into subsections, these include

- LV Systems
- Cameras
- LiDAR
- GPS / INS
- Computing
- Path Planning
- Vehicle Actuation
- Software Infrastructure

Specifically, in the case of computing, the following requirements were generated at the beginning of the year:

- Provide a unified platform for other subsystems to use for processing.
- Provide a method to share information between multiple subsystems
- Interface with the existing ECU, or, implement the functionality of the existing ECU
- Be able to process images and feed-forward pre-trained neural networks
- Implement a state-machine to keep track of the vehicle status

From these requirements, and the requirements determined by the other subsystem designers, the following simple performance targets were chosen:

Metric	**Target**	**Notes**
Image Processing Rate	At least 12 FPS	Of 1080p images
Computational Power	More than 1 TFLOP	32-bit floating point
Message Latency	Less than 100ms	Between any two points of the system

The image processing rate was chosen based on the maximum speed of the vehicle and the spacing of cones as specified in the rules. The computational power requirement was rather arbitrary; however, it was based on empirical evidence from FSAE teams who competed and were successful in previous competitions. To satisfy these requirements, a combination of hardware and software was chosen, purchased, assembled and tested. These components are explored in-depth throughout the rest of this report.

Chapter 2

Background

2.1 Existing Work

To ensure the entire autonomous system is developed, built and tested within a reasonable time-frame, it was decided to re-use the 2018 electric vehicle, rather than build a new one from scratch. This will require a minimal amount of modifications to the chassis and low-voltage loom in order to accommodate the new sensors and computing units, and avoids the hassle of building an entire electric-vehicle from the 'ground up'. In 2017, a member of Monash Motorsport conducted research into the feasibility of building an autonomous vehicle to compete at competition in 2019. Off the back of this research the new autonomous division was recruited, and work began immediately in January of 2018.

2.2 Literature Review

A large body of work has been undertaken in the area of autonomous vehicles. Whilst the low-level details of each implementation can differ quite significantly, there still exists a set of core components which constitute an autonomous vehicle. These can be broadly defined as perception, decision and control, and vehicle platform manipulation [1]. Perception relates to the sensing of the world, and creating understanding from the data measured from various sensors. Decision and control involves determining the actions that should be taken, based on this understanding of the world. Finally, vehicle platform manipulation involves using these decisions, usually in the form of a desired motion, and actuating the various actuators to achieve this result.

Whilst this broad overview is helpful in segmenting the different components of an autonomous system, it avoids the question of "How do we link these components together?", and "Where do these components exist and operate?".

Hardware implementations of this high-level overview take many forms, and depend heavily on the resources available to the engineering team and the purpose behind the autonomous vehicle. For small F-SAE racing teams like Monash Motorsport, success has been seen with low-powered computing units such as a Raspberry Pi or a laptop[2]. This appears to work well due to the reduced complexity of the problem at hand, which only requires localization within a small track, with clear and well-defined boundaries. More complex problems, such as level 5 autonomous driving, require significantly more processing power as the system must perceive a large number of object types, and is required to perform complex path planning to ensure the maximum amount of reliability and safety in a busy and messy environment. For

these problems, high-compute installations are used, which can use in excess of 3000 Watts of electrical energy, and provide over 60 tera-FLOPS (60×10^{12}) of computational processing power[3].

Software implementations also vary wildly from project to project. The architecture is usually the same as described above, however the broad components are split into more specific sub-components, e.g. Sensor Processing, Localization and Mapping, Path Planning and Vehicle Actuation. These can be focused on in isolation, and helps to simplify the overwhelming task of creating an autonomous vehicle. However, eventually these components must be combined into a single entity, and there is a need for a system to enable these components to communicate easily and effectively. A highly common approach is to utilize Robot Operating System (ROS), which is a flexible, open-source framework for writing robotic software[4][5].

2.3 Previous Teams

Whilst the driverless competition is still relatively new, there's already some outstanding work being performed by teams all over the world. One team stands out specifically: Academic Motorsports Zurich (AMZ). This team has competed at Formula Student Germany (FSG) twice with their driverless vehicle, and won both competitions. In order to aide other teams developing their own vehicle, they've kindly provided various sets of algorithms, white-papers and datasets which have helped shape and direct the Monash Motorsport autonomous section. A ROS bag containing various sensors was vital in the development of algorithms for other subsystems such as LiDAR and GPS, and a recent whitepaper also provided some useful insight into how to structure the computing processing pipeline and cooling system.[6]

In 2017, 15 teams competed in the driverless category at the Formula Student Germany (FSG) competition. The hardware used by these teams was quite diverse, ranging from a BeagleBone computer, to an NVIDIA Drive PX2. Most teams failed to pass scrutineering and weren't allowed to drive their vehicles at competition, however, the four teams that did all saw moderate success with their vehicles. Once again, AMZ performed the best out of all teams. They used two computing units, one the 'robust master', the other the 'high-performance slave'. It was stated to only have a combined processing power of 368 giga-FLOPS. In 2017 AMZ didn't use any GPU computing, as they weren't familiar with the techniques involved in parallelizing the processing pipeline, and hence decided to only use conventional processing units.

Figure 2.1: AMZ Driverless, 2017, At FSG

Chapter 3

Concept Generation

What follows is the thought process behind the concept that was implemented. This includes both the hardware and software aspects of the system.

3.1 Hardware

Various pieces of hardware were originally considered.

Name	**Processing Power**	**Power Consumption**	**Price**
Raspberry Pi 3 B+	∼3.5 GFLOPS[7]	12 Watts	$60
BeagleBone Black	∼4 GFLOPS	2.3 Watts	$75
Jetson TX2	1.3 FP16 TFLOP	7.5 Watts	$600
NVIDIA Drive PX2	20 FP16 TFLOPS	250 Watts	$15,000
NVIDIA Drive Pegasus	320 INT8 TOPS	500 Watts	>$15,000
NVIDIA Xavier	30 INT8 TOPS	30 Watts	$2,500

Table 3.1: Potential Hardware Options

The NVIDIA Drive PX2 and NVIDIA Drive Pegasus were discarded as potential candidates due to their high price point. The NVIDIA Drive Xavier seemed like a good option, however it was unavailable for purchase at the time. The Raspberry Pi and BeagleBone Black were also discarded quite early on in the concept generation process, due to their low computational power.

Another potential option that's not included in the above table, is to build a custom computer from off-the-shelf components. The exact specifications of such a machine are undefined, as it depends entirely on the amount of money available to spend. With enough funds, a custom machine could well outperform all of the above options. On a similar note, all-in-one units were also considered, such as the Neousys Nuvo-5095GC. These units are also customizable. The main difference between an all-in-one and a custom-built computer is the packaging, cooling and pricing. For example, the Nuvo-5095GC comes in a sturdy (but not waterproof) aluminium case with heatsinks to dissipate the 150+ watts of energy it consumes. Whilst this is ideal for a standard vehicle which has a waterproof interior and plenty of space, it's not ideal for a space-constrained vehicle such as M19-D, which is also exposed to the elements. This would require extra packaging to ensure the system is waterproof, and a new cooling solution too.

Determining exact computational requirements is a difficult task, especially when the algorithms

Figure 3.1: The Neousys NUVO-5095GC

haven't been written yet. The camera subsystem designers stated that they wished to achieve over 12 frames per second when processing camera data. Assuming a 640×480 image, at 12 FPS, with three 8-bit colour channels, this would result in 11 MOPS if each 8-bit colour byte was processed just once. However, difficult questions arise after this point. How many operations are required on each byte of image data? If just averaging the 3 colour channels, this would required 1 operation. A 3×3 kernel, for detecting edges or blurring the image, would require at least 18 operations per byte. A convolutional neural network could require hundreds! This also fails to account for inefficiencies such as a cache-miss or a 'bubble' in the processor pipeline, which can delay processing significantly. It also depends on compiler optimizations and the instruction set available (e.g. is the fused-multiply-add instruction being used?).

3.1.1 Jetson TX2

After some very rough calculations, it was originally decided that the Jetson TX2 was an affordable, low-powered option that should in theory be able to provide the computational power required. However, there were concerns from other team members that this would still not be able to provide the amount of processing power required. The TX2 is theoretically capable of processing 1.3 trillion 16-bit floating-point numbers every second. However to extract this performance, CUDA code must be written to specifically take advantage of the special instruction set, which multiplies 2 16-bit 4×4 matrices together, and creates one 32-bit 4×4 matrix[8]. If the problem at hand can't be made to utilize this instruction, then the performance of the TX2 decreases to 0.6 TFLOPS. Furthermore this assumes the problem can be parallelized over the 256 CUDA cores available. If the computation is purely sequential, then the performance decreases even further and it's no longer beneficial to use the CUDA cores. Consequently, using the ARM64 processor instead leads to a theoretical performance of approximately 0.06 TFLOPS.

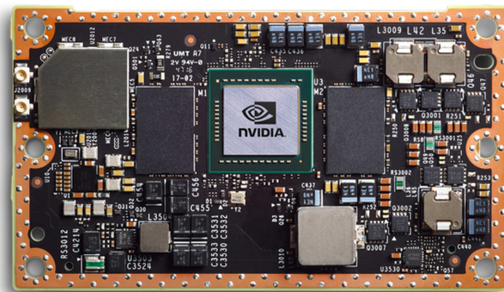


Figure 3.2: The Jetson TX2 Module

3.1.2 Intel i7

Due to the performance concerns with the Jetson TX2, it was decided to include another processing unit in the autonomous system, which utilizes an Intel i7 processor. This is better suited for workloads which can't be parallelized on the GPU. Due to the nature of the vehicle, it was decided to build this computer using off-the-shelf components, which allows for a smaller form factor compared to units such as the Neousys, and enables the development of a custom cooling solution which works best for an F-SAE vehicle.

To determine which processor was best, benchmark results of over 500 high-end CPUs were gathered and analysed. By using benchmark scores instead of theoretical performance (FLOPS), a more accurate comparison can be made as these numbers are not theoretical 'best-case' scenarios, but actual standardized measurements of processing power.

First, the results were ordered in descending order by benchmark score. They were then filtered to remove any processor which consumed more than 80 watts of power. Immediately one processor stood out as the best option: an Intel i7 8700. This CPU has a 65W Thermal-Design-Power (TDP), a passmark benchmark score of 15,240, a 3.2 GHz base frequency and a 4.6 GHz turbo boost frequency. Looking at current benchmark scores, the Intel i7 8700 is still one of the highest performing consumer-grade processors available.

3.1.3 Computing Summary

Various other physical components are included in the autonomous computing platform. These include RAM, SSDs and Mother/Carrier boards. These aren't covered in detail as they're simple necessities which are required for the computing units to function. In summary, the computing hardware for the autonomous system consists of the following:

- An Intel i7, 8700 CPU. Mounted on a Mini-ITX motherboard, with 8GB of RAM and a 250GB M.2 SSD.
- An NVIDIA Jetson TX2, mounted on the developer carrier board (also Mini-ITX form-factor), with 8GB of LPDDR4 RAM, 32GB internal eMMC memory and an external 250GB SSD. Contains 256 NVIDIA Pascal CUDA cores and a 4-core ARM Cortex-A57 64-bit processor.

Note that both processing units are situated on a $170mm \times 170mm$ mini-ITX board. This was purposeful, as it facilitates easier mounting on the vehicle, and consumes less space than the larger ATX-style motherboards.

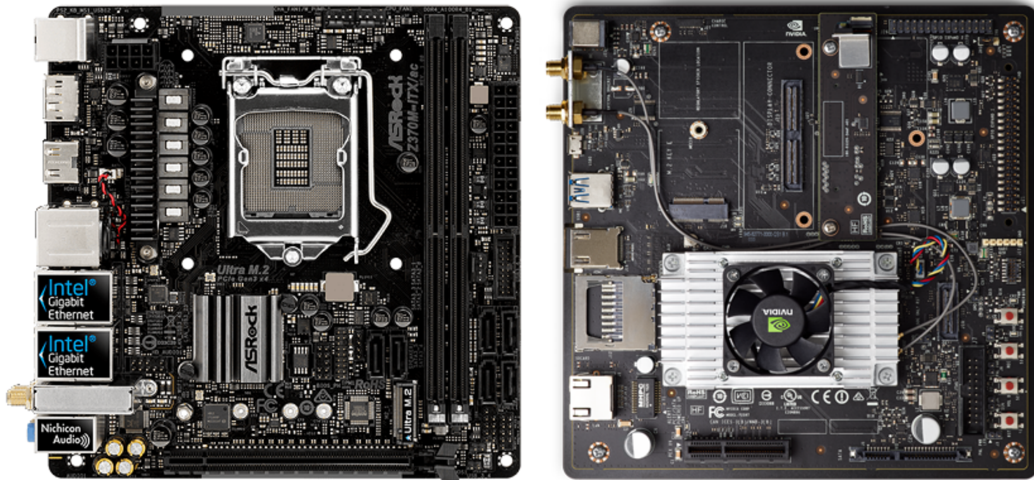


Figure 3.3: The two mini-ITX boards. Left: i7, Right: Jetson TX2

3.1.4 Low-Level Processor

On top of these two computing units, another processing device is required. The autonomous system must interact with the vehicle. This requires CAN communication, as well as digital and analogue inputs and outputs. Whilst the Jetson TX2 can provide some of this IO, the 3.3v voltage levels used by the Jetson make this difficult, and it is beneficial to not have to rely on the Jetson TX2 in case the team wishes to take another direction with the computing hardware in the future. Thus, another processing unit is required which is capable of providing this low-level physical communication. This piece of hardware has been chosen to be a PSoC 5LP. While there are many options possible, the PSoC was chosen due to its versatility, and the fact that the team already has a wealth of experience with the PSoC from previous projects.

The PSoC has features such as

- 4 Digital to Analogue Converters
- 4 Analogue to Digital Converters
- 48 Configurable Input / Output Pins
- A CAN controller
- UART Communication
- I2C Communication
- SPI Communication
- Reprogrammable Universal Digital Blocks. Allows the creation of modules such as counters from reconfigurable logic blocks.

These make it a very versatile machine which is capable of performing all the functions we require.

3.2 Software

There exists a wide array of possibilities when it comes to software. As with any complex system, it is necessary to split the components into smaller units of work, which can be delegated to team-members and worked on in isolation. However, this introduces a new problem:

"How are these components integrated back together to create the final product?" Ultimately a computing system is simply a processor of data. Data enters the system in various forms, work is performed on this data, and new data is output. This data must flow between the aforementioned components in an efficient manner. In a distributed computing system, there exists many different mechanisms to facilitate this, outlined in the table below:

Mechanism	**Description**
Request-Based Server / Client	A central process, which a set of clients connect to. All communication occurs through the 'server', which routes messages between the various clients. Each process can 'request' data from another client through the 'server'.
Publish / Subscribe	A central process, which maintains an open connection to a set of clients. Each client 'publishes' data to 'topics'. Upon publishing the data, it is transferred to the server, which routes the message to any clients which are 'subscribed' to the topic.
Single Process, Multiple Threads	The entire system is contained within a single process. This process starts multiple threads, which are all capable of sharing the same memory space, allowing for fast transfer of data.
Shared Memory	The system is comprised of multiple processes, which use an external mechanism (such as memory mapped files) to share data.

Table 3.2: Common Message Transmission Methods

These all have various advantages and disadvantages. Special care must be taken when it comes to sharing memory between processes, and if not handled correctly data corruption can easily occur. Purposefully copying and transferring the data between separate threads using a networking protocol solves this problem, however it now introduces latency into the system. Also, large heaps of information such as raw images can't be copied and transferred quickly, and the overhead involved will consume unnecessary and unwanted amounts of processing power.

Thankfully, these problems have already been discovered and solved, and the solution is open-source and freely available: Robot Operating System. This is currently the industry standard when it comes to integrating robotic components together. It's used by research institutions and commercial businesses such as Bosch. There's a wide variety of support for ROS, with yearly updates, documentation and tutorials, as well as a slew of libraries and utilities which aid in the development and testing of robotic systems.

Robot Operating System, or ROS for short, is a meta-operating system which provides a frame work for communication between robotic components that is independent of the hardware and runtime context. On a basic level, ROS consists of a 'master' process, and a set of 'nodes' which connect to this master process to become part of the robotic system. Each node is a separate process, and can communicate with other nodes using a publish/subscribe model. ROS topics are a many-to-many relationship. Any node can publish data to a topic, and any node can 'subscribe' and receive published messages from a topic.

This architecture enables the individual components to be created in isolation by the various subsystem designers, and easily combined in the end. It also ensures redundancy; if a single process fails, the rest of the nodes are 'protected' from the failure, and won't crash too. An external mechanism can relaunch these nodes upon failure, or take other actions if required, such as shutting down the entire system and bringing the vehicle to a safe stop. Roslaunch,

a command-line-interface which comes with ROS, already provides some of this functionality.

ROS also has many other features which make it beneficial to use:

3.2.1 Nodelets

The standard ROS component is a ROS 'node' This is a single process which can access the ROS system using the provided ROS API. Each ROS node interacts with other nodes using a network protocol such as TCP/IP or UDP. However this is not congruent with transferring large amounts of data. It's quite wasteful to copy large chunks of information, and transfer them over TCP/IP, especially when this data already exists in another memory location. This is unavoidable when transferring between two computers which don't share memory, however it can be solved on a single machine. ROS 'nodelets' provide a solution to this problem. A nodelet runs within a ROS node, and is able to communicate with other ROS 'nodelets' by sharing memory between the two ROS nodelets. ROS handles the intricacies of shared memory objects which allows the developer to focus more on the higher level design of the system, rather than worrying about shared memory-management techniques such as mutexes and semaphores.

3.2.2 Multi-language Support

ROS is able to be used across various languages. The default language is C++, which has high performance however can be painful to continually compile and test with. ROS also supports python out-of-the-box, and enables faster prototyping as the code does not need to be compiled. There is also support for other languages, such as Ruby and NodeJS, however these are no longer actively maintained. Testing that was performed toward the end of the year showed that the NodeJS library appears to still function, however if used in the future it should be rigorously tested before being utilized on the vehicle.

3.2.3 Services

The 'roscore' process maintains a list of the current ROS nodes which are running. Each ROS node is capable of advertising 'services', which are exposed through the roscore interface. These are basically Remote Procedure Calls (RPC), where a specific function can be called remotely from another ROS node. Examples of service functions include setting the rate of an IMU or setting camera parameters on-the-fly. Whilst it is possible to implement similar functionality using the default ROS topic publish/subscribe model, it's much easier to implement a service. A 'service' will return a result to the calling process, however if using standard topics, then there must be another topic created and subscribed to in order to receive the result of the function call.

3.2.4 Transform Frames

A common problem in robotic applications is determining where the robot is in the world, and where all of the 'limbs' of the robot are too. ROS comes with a module called 'tf', which provides a set of utilities to solve these problems. One can define three-dimensional 'frames' and the transformations between then at given points in time. The 'tf' module maintains a history of these transformations, and can be queried to determine the transformation matrix between the frames at a previous point in time.

Chapter 4

Physical System Overview

In summary, the physical architecture of the autonomous system is as shown:

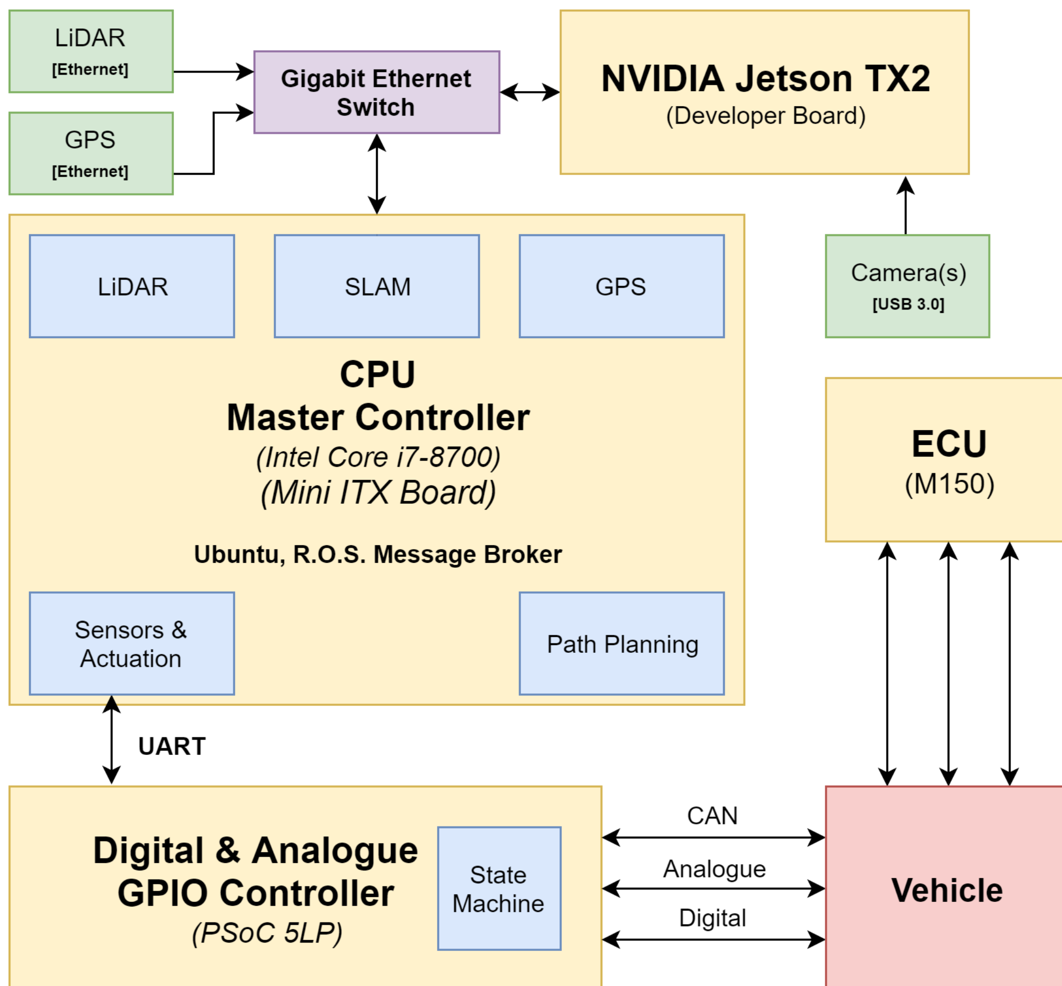


Figure 4.1: High-Level Hardware Diagram

The following sections outline the above components which constitute the autonomous vehicle, as well as the various interactions between these components.

4.1 Computational Hardware

As outlined earlier, the autonomous system adds three processing units to the vehicle:

- Intel i7 8700, 8GB RAM, mounted on a Mini-ITX motherboard
- NVIDIA Jetson TX2, 8GB RAM, mounted on a Mini-ITX carrier board
- PSoC 5LP[9], mounted on a custom-built PCB

The **Intel i7** is the main processing unit, and contains processing nodes which control the state of the entire autonomous vehicle. The **Jetson TX2** is solely used for processing camera image data. Due to its 256 CUDA cores it's best suited to this task compared to the other computing units. These two units are connected together through an Ethernet switch. This allows the two computers to communicate easily using ROS over a TCP/IP link.

The **PSoC 5LP** is a different kind of processor all together. The i7 and Jetson both run Ubuntu 16.04, and unlike these POSIX based systems which have an underlying scheduler which shares processing time between multiple processes, the PSoC 5LP only runs a single program/process. This fundamental difference in computing architecture makes the PSoC a more stable processing machine, which is ideal for safety critical sections of code which must be executed at regular intervals and must not crash. The PSoC communicates with the Intel i7 processor using UART.

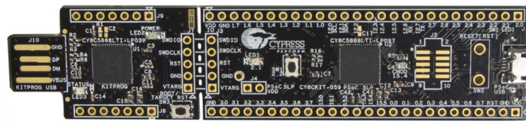


Figure 4.2: The PSoC 5LP

The existing electric vehicle already contains its own processing unit: the **MoTeC M150**. This is aptly named the 'Engine Control Unit' (ECU) and is the current brains of the electric vehicle. It's nowhere near powerful enough to drive the vehicle autonomously, however it already handles various engine and vehicle related actions, such as data-logging and throttle control. Without this device the vehicle would cease to function, and thus the autonomous system must integrate with the ECU to implement the new functionality that the autonomous system requires. The ECU shall communicate with the PSoC over CAN.

Figure 4.3: The MoTeC M150 Engine Control Unit

4.2 Data Collection Hardware

There are various new sensors which are going to be integrated into the vehicle.

4.2.1 LiDAR

The LiDAR sensor is a sweeping infra-red laser which detects straight-line distances from the sensor to points in the world. It's low-latency high-frequency data makes it incredibly accurate. Many commercially developed autonomous vehicles use one or more LiDARs to sense and perceive the world around the vehicle with great success. The LiDAR communicates with the Intel i7, through the Ethernet switch.

Figure 4.4: The VLP16 LiDAR

4.2.2 Cameras

The solution developed by the camera subsystem designers uses a stereo vision camera. Work was done at the beginning of the year to test a dual-camera setup, however it proved difficult synchronizing the camera shutters and aligning the cameras correctly. Eventually a Stereolabs ZED camera was purchased, which is an all-in-one IP66 rated plug-and-play unit. This is much easier to use and solves the difficulties associated with synchronizing image captures and aligning two separate cameras on the vehicle. The Stereolabs ZED camera communicates with the Jetson TX2 over USB 3.0.

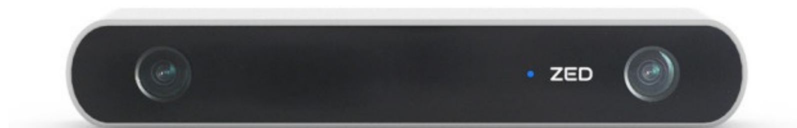


Figure 4.5: The Stereolabs ZED Camera

4.2.3 GPS

GPS signals are inherently noisy and inaccurate. When driving on the road, the GPS signal can be complemented with a map of the road to produce a more accurate estimation of the vehicle location. However, out on track at competition with an F-SAE vehicle, this is no

longer an option as the track isn't well defined like a road system. Thus it was decided to invest in a Swiftnav Piksi-Multi GNSS Module. This GPS module is capable of communicating with a base-station, which provides centimetre-grade accuracy to the GPS measurements. The GPS communicates with the Intel i7 over Ethernet. The Piksi-Multi does provide UART communication methods too, however these aren't being utilized by the autonomous system as Ethernet is more convenient.

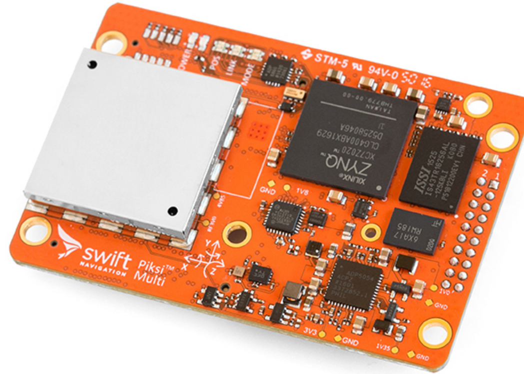


Figure 4.6: The Swiftnav Piksi-Multi GNSS Module

4.3 Connectivity

The two main computing units, the Intel i7 and the Jetson TX2, are connected through an Ethernet switch. On each machine a static IP address is assigned, within the *192.168.0.X* subnet. For the i7, it's been assigned *192.168.0.1*, and the Jetson has been assigned *192.168.0.10*. These IP addresses have also been mapped in the */etc/hosts* file, which allows the computing units to easily reference each other as 'mms-tx2-1' and 'm19d' for the Jetson and the i7 respectively. External computers can connect to the Ethernet switch, assign themselves a static IP address such as *192.168.0.20* and then communicate with the computing system.

The PSoC communicates with the i7 using UART. A custom message structure is being used to transmit the payloads, and is discussed later in chapter 5. The PSoC then communicates with the vehicle using digital and analogue input/output pins, and also interfaces with the vehicle's CAN bus so it can communicate with the inverter for motor control, and also monitor signals from other sensors already on the car.

4.4 PSoC PCB

The PSoC 5LP requires various pieces of supporting hardware to enable it to interact with the vehicle and its systems. A Printed Circuit Board was created for this purpose:

- **Voltage Regulation** - The PSoC 5LP requires a 5v power supply. The vehicle however runs using 12v, and thus the voltage needs to be stepped down.
- **Controller Area Network** - A CAN transceiver converts CAN HI and CAN LO signals to RX and TX signals which can be used by the PSoC's CAN module.
- **Digital Input Filtering** - A Schottky diode pair, two resistors and a capacitor filter the incoming digital signals; clamping the voltage between zero and five volts, as well as filtering out high frequency noise.

- **Digital Output Filtering** - Similar to input filtering, a Schottky diode pair clamps the voltage, and a resistor limits the current draw.
- **Analogue Input and Output Filtering** - Analogue inputs and outputs both use a Schottky diode pair to clamp the voltage between zero and five volts.
- **ASSI MOSFET** - The vehicle is required to have 'Autonomous System Status Indicators' which indicate what state the PSoC state-machine is currently in. The PSoC is the source of truth for the state-machine's state, and hence has control over the ASSI's. Two digital outputs from the PSoC switch two MOSFETs, one for the yellow LEDs and one for the blue LEDs. When switched, the MOSFETs connect drain to source, providing a path for the LEDs to sink current and turn on.
- **EBS Interlock** - Multiple inputs and outputs from the PSoC are required by the EBS Interlock circuit, which was developed by the Low-Voltage subsystem designers. These connections are routed on the PCB. The EBS Interlock Circuit interfaces with his PCB through a set of header pins.

The PCB board was designed in Altium 16 and aims to serve all of the above requirements. All inputs and outputs to the PCB are routed through Molex Microfit connectors, which the team has past experience with. The latest revision of the PCB board contains

- 12 Digital Outputs, 7 of which are routed to the EBS interlock PCB
- 8 Digital Inputs, 3 of which are routed from the EBS interlock PCB
- 8 Analogue Inputs
- 4 Analogue Outputs
- CAN TX and RX
- UART TX and RX (routed to i7 processor)
- 12v Input and Ground
- Shutdown Circuit In and Out
- Brake power In and Out

The board has been designed with more digital and analogue IO than required to allow for future expansion of the system if required. See Appendix A for a full list of inputs and outputs.

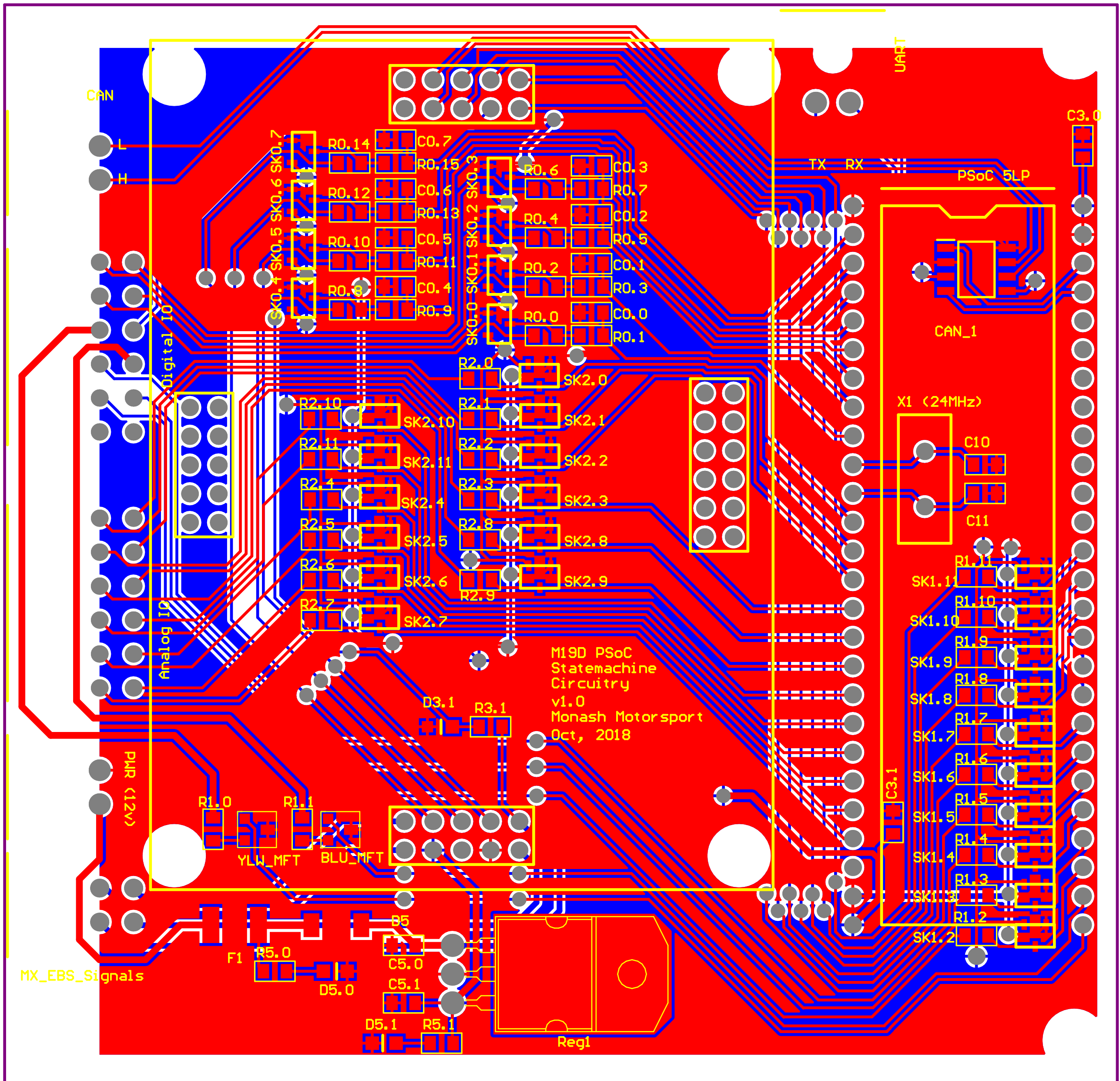


Figure 4.7: The PSoC PCB

Chapter 5

Software

5.1 ROS Node Architecture

The autonomous system is comprised of numerous processing nodes. A pictoral representation of the system is described below

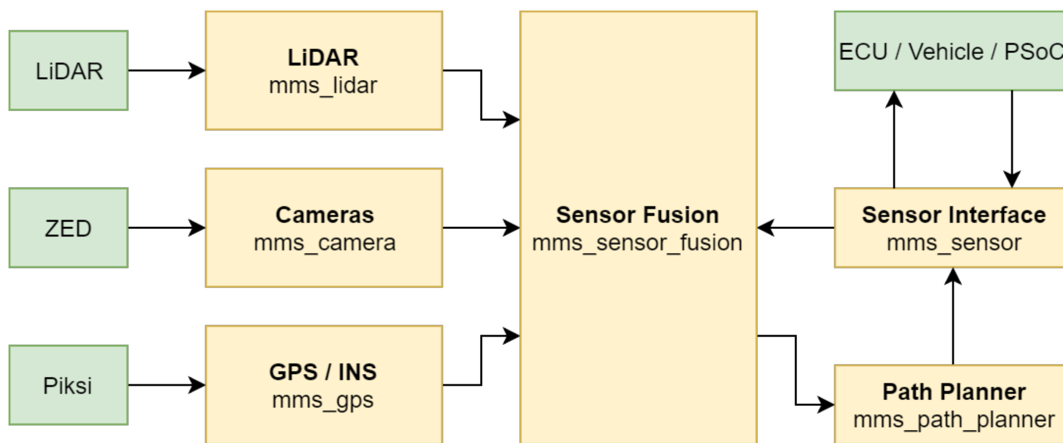


Figure 5.1: ROS Node Overview Diagram

A description of each node and it's inputs and outputs are listed below.

5.1.1 LiDAR Node

This node connects directly to the Velodyne VLP16 LiDAR via Ethernet using the UDP protocol. It consumes incoming point clouds from the sensor, and processes the sweeps to determine where cones are in the world. These cone positions are then relayed to the sensor fusion node for further processing.

5.1.2 Camera Node

The camera node connects directly to the Stereolabs ZED camera. It then processes the incoming images using YOLOv3 to determine cone positions in one of the images. Stereo-matching is then performed to determine cone depth, which results in a 3-dimensional cone measurement. Then is also relayed to the sensor fusion node for further processing.

5.1.3 GPS Node

The GPS node connects directly to the Swiftnav Piksi-Multi GNSS Module, and receives incoming GPS, IMU and Magnetometer data. This data is then also forwarded on to the sensor fusion node.

5.1.4 Sensor Fusion Node

This node contains an EKF which performs state estimation and Simultaneous Localisation and Mapping (SLAM). It takes two types of cones measurements as inputs, one from LiDAR and one from Cameras. Within the filter it performs data-association, which allows it to associate new incoming measurements with previously seen measurements. It keeps track of the car's position, velocity, acceleration, yaw and yaw rate, as well as all of the cones and their position in the world. Measurements from the inertial measurement unit enables this node to update it's state rapidly. If just the IMU was used, then drift would occur as time progressed. However by incorporating cone position measurements and GPS measurements, this issue is solved.
Ultimately this processing node outputs two things.

- The Vehicle State. This include position, velocity, acceleration, as well as yaw and yaw rate.
- A list of cones. These are simply Cartesian X-Y coordinates of the cones in the origin (world) frame.

These two outputs are passed to the path planning node.

5.1.5 Path Planning Node

This node receives the list of cone positions and the estimated vehicle state. Using this data, and depending on the event type and lap number, a path trajectory is calculated. From this path, and the current vehicle state, desired vehicle torque and steering angles are generated, and sent to the sensor interface node to be relayed to the vehicle through the PSoC.

5.1.6 Sensor Interface Node

This node provides an interface for other nodes to transmit and receive data from the vehicle. It opens a serial connection to the PSoC micro-controller, and relays commands and data to and from the device. This abstracts away the complications that arise using UART, and provides a ROS-compatible interface for the other nodes to utilize.

5.2 Master Node

This node doesn't appear in the above diagram, however it does exist and is arguably one of the more important nodes in the system. The master node is responsible for controlling the state of the ROS system, as well as performing sanity checks to ensure the computing system is in a safe and reliable state. When the autonomous system is started, the master node as well as all of the above nodes are spawned. Each 'child' node 'connects' to the master node through ROS, by publishing and subscribing to specific topics.

5.3 Child Nodes

Each of the above nodes is classified as a 'child' node in the system. These nodes all subscribe to the */mms/master/command* topic, and take appropriate actions when commands such as 'GO' and 'STOP' are received from the master node. They also publish to a heartbeat topic, classified by the schema */mms/<NodeName>/heartbeat*. The master node listens to these heartbeats, and sends its own heartbeats to the PSoC.

5.3.1 Child Node State Machine

Each child node maintains a small internal state-machine.

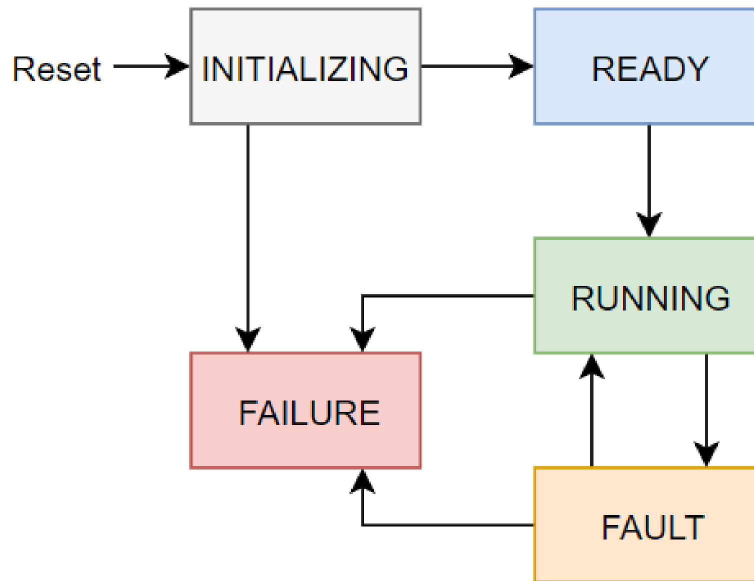


Figure 5.2: Child Node State Diagram

Initializing	The node is currently setting up, connecting to other nodes and sensors.
Ready	The node is ready to begin performing its required function instantly upon request.
Running	The node is processing data, doing what it's supposed to. Everything is fine.
Fault	A non-fatal error has occurred. The node may be able to recover. e.g. The GPS has lost the base-station signal.
Failure	A fatal error has occurred. The node can't recover without human intervention. The vehicle should stop.

The transition from 'Ready' to 'Running' is triggered by the Master node. All others occur due to internal mechanisms within each node, which are specific to the node.

5.4 Heartbeats and Safety

Each child node must publish regular heartbeat messages, at a rate of 5Hz to the master node. The master node receives these heartbeat messages, and ensures the system is running correctly.

It knows what node state configurations are valid, and will attempt to stop the car if the ROS system enters an invalid state. For example, the camera node may enter the ’fault’ state, this is potentially valid as the system can still use LiDAR for cone detection. However if the camera and LiDAR nodes both enter the ’fault’ state then the vehicle must come to a stop as the system is potentially no longer stable.

If an invalid configuration of child nodes occurs, or a child node ’falls’ off the network and is no longer publishing heartbeat messages, then the master node will attempt to stop the vehicle by sending the appropriate command to the PSoC. The master node publishes it’s own heartbeat messages too. These are routed to the PSoC through the sensor interface node. The PSoC main loop monitors these heartbeats, and shuts down the vehicle if requested, or if the heartbeat messages go missing.

The electric car that the autonomous system is being implemented on contains what is known as the ’shutdown circuit’. This is an electrical loop that runs through the vehicle’s loom, which is normally open. When closed, the vehicle is able to operate, however, when opened power from the vehicle’s accumulator is disengaged and the vehicle’s tractive system is disabled. The shutdown circuit will latch open in the event of failure, and a power cycle is required to get the vehicle started again.

The PSoC and its supporting circuitry utilizes this shutdown circuit to stop the vehicle. The PSoC has two digital outputs which are routed to the ’EBS Interlock Circuitry’, which was developed by the Low-Voltage subsystem designers. This circuitry handles the ’Emergency Braking System’, and it’s main purpose is to stop the vehicle in the case of failure. It’s implemented solely using digital logic gates, and thus common computer related issues don’t occur. Housed on the EBS Interlock circuit is an opto-MOSFET, which is wired in series with the shutdown circuit. A watchdog timer is connected to the PSoC. Every cycle of the main loop, this digital output is toggled by the PSoC, resetting the watchdog timer. The PSoC also outputs a constant HIGH (5v) signal to the EBS circuitry, which is fed into an AND gate. If this signal goes LOW (0v) the shutdown circuit is opened. If the watchdog timer doesn’t receive a pulse in a timely fashion from the PSoC (due to it being stuck in an infinite loop), then the shutdown circuit is also opened.

If the PSoC wishes to stop the vehicle, either because it’s detected a fault (missing heartbeats from ROS) or it’s been told to do so by the ROS system, then it will simply drive the aforementioned digital output pin to 0v, which activates the EBS interlock circuitry and opens the shutdown circuit.

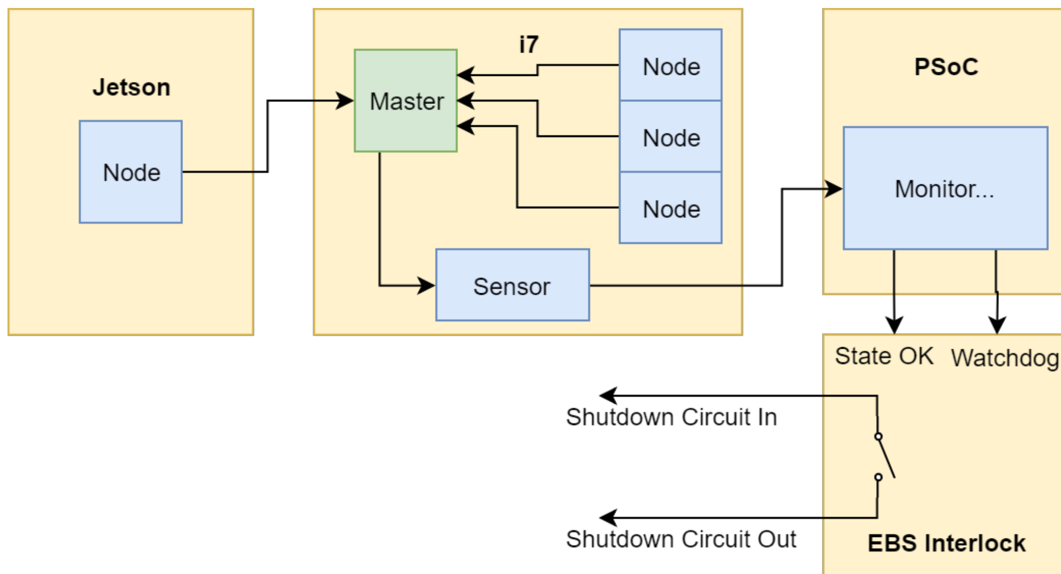


Figure 5.3: Example Heartbeat Flow Diagram

The entire system, by design, will fail into the off state and bring the vehicle to a stop. Every component within the autonomous system must actively work hard to keep the vehicle running. If any piece of the processing pipeline fails, the EBS will trigger and bring the vehicle to a stop.

5.5 PSoC Processing

As with all microcontrollers, the PSoC contains an infinite main loop. Within the main loop, the following actions are performed:

Update UART / ROS Communication	Processes any bytes in the UART RX buffer, and when ready, constructs a message object from this data. The message's integrity is checked using checksums, and then sent to the message processor. A switch-case statement selects the appropriate handling function, and the function is executed, generating and queuing any response messages generated.
Update EBS State	Parses the current digital / analogue IO values to determine the state of the Emergency Braking System.
Update Service Brake	Measures service brake sensors, and monitors the state of the service brake.
Update Steering and Throttle Control	Updates any control signals being sent to the inverter and service brake actuator. If the PSoC statemachine is not in the AS_DRIVING state, then this section must ensure that no actuation requests from ROS are serviced here.
Update State machine	Updates the PSoC / Autonomous state-machine. This is defined and required by rules.

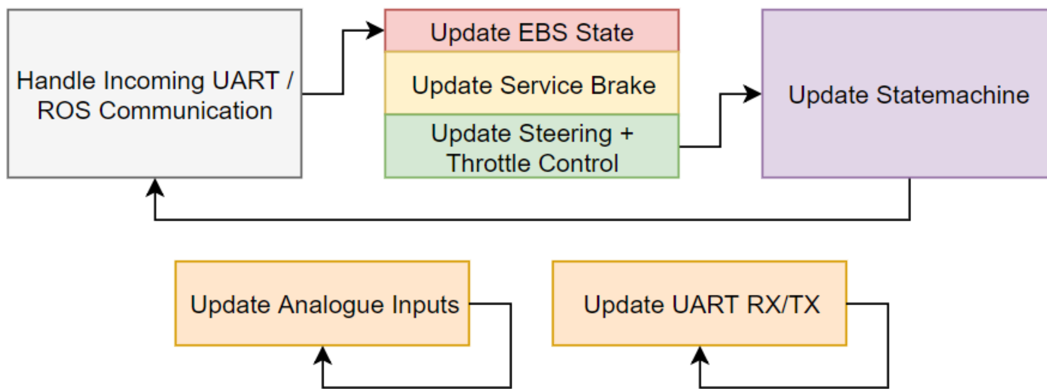


Figure 5.4: Pictorial Representation of PSoC Processing Flow

It is required by rules to implement a state-machine to control the autonomous system. This is implemented on the PSoC due to its robustness and 'closeness' to the vehicle and its sensors and actuators.

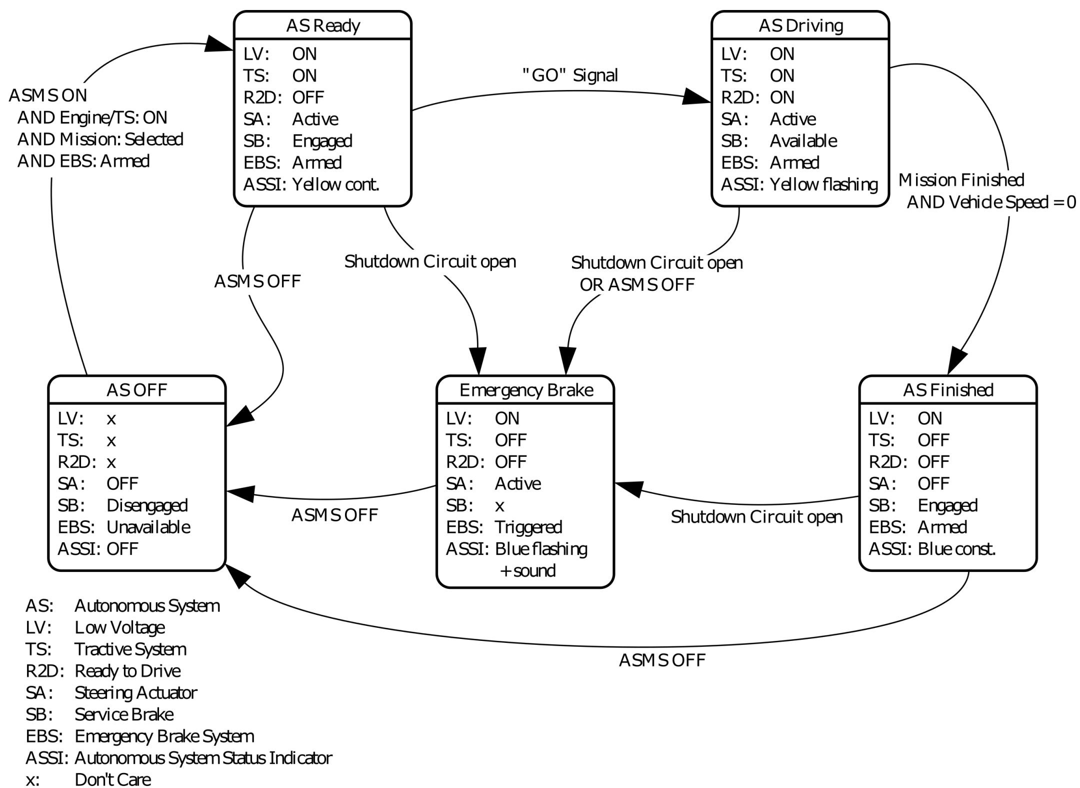


Figure 5.5: Autonomous Statemachine[10]

Two other processes are occurring in the background. These are interrupt-driven and don't require intervention from the main processing loop:

Update Analogue Inputs	A single Analogue to Digital Converter reads analogue signals from 8 analogue inputs. After each read, an analogue multiplexer is switched so the next analogue input can be read.
Update UART RX / TX	New bytes from the i7 are queued up in the RX buffer. Any bytes to be sent are removed from the TX buffer and sent to the i7.

5.5.1 Custom UART Message Protocol

The Intel i7 processor and the PSoC 5LP communicate using UART. When using UART, it is guaranteed with relatively high accuracy that a received byte is identical to the byte that was sent. This is done by appending a CRC to the bit-stream, and verifying that the received CRC bit matches the computed CRC. Whilst this helps verify each individual byte, it doesn't verify that a sequence of bytes are correct. If bit errors occur, bytes can be discarded and not received. When a message spans over multiple bytes, it requires extra mechanisms to ensure all of the bytes have been received.

To ensure corrupt messages sent between the i7 and PSoC are caught and handled appropriately, a custom message protocol has been designed. By appending and prepending the payload with extra bytes, a more robust messaging system is created which ensures message integrity. The structure of this message structure is outlined below:

Start Byte	MID	Command	Size	Payload	CRC ADD	CRC XOR
0x9d	0-255	0-255	0-250	Dynamic Data	0-255	0-255

Table 5.1: UART Message Structure

A description of each component of the message structure is outlined below:

- **Start Byte** - This byte signifies that what follows is a message structure. This is particularly useful for when an error does occur and the expected 'end' of a message is actually mid-way through the next message. Bytes must be skipped so that the receiving end knows where the new message begins.
- **Message ID** - This byte is used for response matching. This should be incremented for each message sent that requires a response. Any response message from the PSoC will use the same message ID as the request.
- **Command** - This byte represents what action should be taken. The meaning of this is also context-dependent. For example, if the PSoC receives a 'get digital pin' command it will expect the payload to contain a list of digital pins to read. The response message from the PSoC to the i7 will use the same command and message ID, however the payload will be slightly different; containing both pin numbers, and the read values.
- **Size** - This is simply the size of the following payload. To enable easier development on the PSoC, this should be limited to 250 bytes, so 8-bit integers can be used to reference bytes within the message structure.
- **Payload** - This is a dynamic sized payload, the contents of which depends of the command type

- **CRC ADD** - A summation of all the previous bytes, from 'Start Byte' to the end of the payload
- **CRC XOR** - An XOR summation of all the previous bytes, from 'Start Byte' to the end of the payload. Does not include the 'CRC ADD' byte.

This message structure has been implemented both on the PSoC in C, and in a ROS node in C++.

Chapter 6

Computing Constraints

The autonomous computing system must be able to run on the vehicle without being tethered to a power supply. This constrains the amount of power the system can consume. In conventional desktop computers, it's not uncommon to have a single computer which consumes over 500 Watts or more. Running a system such as this on a 20 Amp-hour 12V battery, not including all of the other components such as actuators and sensors, results in a maximum of 30 minutes of runtime. Whilst this is potentially long enough to last an entire track drive, it's still quite short for testing purposes, and would make testing sessions difficult.

What follows is some techniques that have been or can be employed in the future to reduce the required processing power, and hence electrical energy consumption.

6.1 Jetson TX2 CUDA

Conventional GPU's can consume over 100 Watts of power, however the Jetson TX2 module only consumes 7.5 Watts of power when processing is at a maximum. This reduced power consumption does come at the cost of computational power. The TX2 is only capable of approximately 0.6 single-precision TFLOPS, however if the precision is halved then the performance doubles to 1.3 TFLOPS (due to the TX2 having a CUDA Compute Capability of 6.2 [8]). This allows more performance to be extracted, however only in specific circumstances.

The algorithms running on the Jetson TX2 are yet to take advantage of any reduced precision computing. The camera subsystem designers are still developing stereo matching, and are currently following the philosophy of 'design now, optimize later'. Once the algorithms are complete and bottle necks are determined, CUDA code can potentially be written to increase performance in the required areas.

6.2 Intel Integrated Graphics

On the Intel i7 processor, there is still performance to be extracted. The i7 contains an integrated graphics processing unit: Intel® UHD Graphics 630. This contains 184 cores, has a 1150 MHz boost clock speed, consumes 15 Watts and in theory, can produce 400 GFLOPS of processing power. This is usually used to render the desktop, however on the autonomous vehicle there is no screen to be displayed, and this processing potential is being unused.

OpenCL can be used to tap into this processing power, however writing OpenCL code is quite difficult, and there are many optimizations required to make it worth while. This issue has

already been solved by the 'ArrayFire' library and others like it, which abstracts the parallel computing language from the C++ code, and allows code to be written like follows:

```
// sample 40 million points on the GPU
array x = randu(20e6), y = randu(20e6);
array dist = sqrt(x * x + y * y);
// pi is ratio of how many fell in the unit circle
float num_inside = sum<float>(dist < 1);
float pi = 4.0 * num_inside / 20e6;
af_print(pi);
```

This is much simpler than OpenCL, and already performs many optimizations 'under-the-hood' to speed up the processing time significantly. To test the potential processing power of the integrated Intel graphics processor using ArrayFire, the Monte Carlo pi-approximating algorithm was run on a laptop, utilizing:

- The CPU with standard C++
- The CPU with ArrayFire, using all 4 processing cores
- The Intel GPU with ArrayFire
- The NVIDIA GPU with ArrayFire

The results of this test were quite impressive.

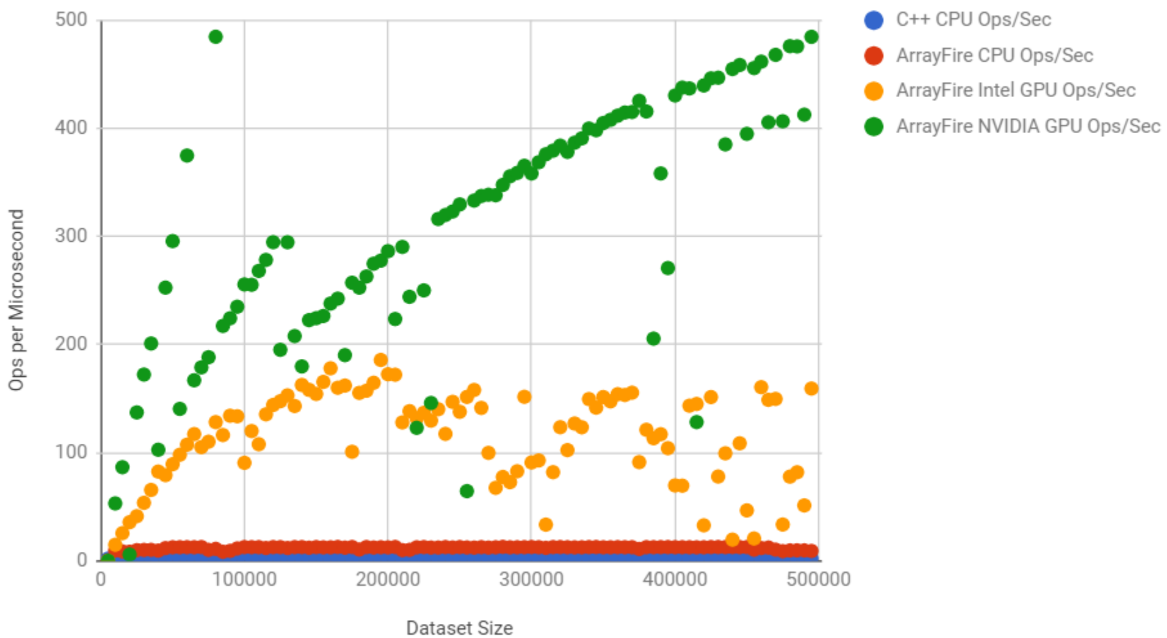


Figure 6.1: ArrayFire Monte Carlo algorithm timing

Using just the CPU resulted in rather poor performance. Using the Intel Integrated Graphics processor provided a significant improvement, almost 50 times faster than a naive C++ implementation. Finally, the NVIDIA GPU can be seen to give the best performance gains, with a processing speed potentially 140 times faster, however only when operating on large datasets.

Whilst this isn't exactly representative of what performance gains will be seen when optimizing algorithms for the autonomous vehicle, it is indicative of the potential improvements that can be expected.

$$\text{Jetson TX2 } = 1300 \times 10^6 \times 256 \times 4 = 1331 \times 10^9 \text{ FLOPS}$$
$$\text{Intel UHD Graphics } = 1150 \times 10^6 \times 184 \times 2 = 423 \times 10^9 \text{ FLOPS}$$

6.3 Eigen MKL

Many of the algorithms being run on the vehicle require the processing of large matrices. For example, the SLAM EKF maintains XY coordinates of potentially up to 300 cones. This results in a 600×600 covariance matrix, which must be multiplied by various other vectors and matrices every time cone estimates are updated. If multiplied by another 600×600 matrix, this requires approximately 216 million multiplications and 215.65 million additions.

Intel provides the *Intel Math Kernel Library* which aims to improve processing times by fully utilizing all of the features provided by an Intel processor. This includes parallelizing processing over multiple CPU cores, and utilizing special instructions within the CPU to optimize calculation times. This library has been integrated with Eigen, a library for linear algebra, matrix and vector operations, amongst various other mathematic related functions.

'Eigen MKL' is being used by the SLAM EKF processing node, and in its current state, enables it to run in real time on the Intel i7 processor. Future additions are required to be added to the EKF to integrate camera and GPS measurements, however these additions are being carefully constructed to not require exorbitant amounts of unnecessary processing, so the current performance being seen should not be degraded significantly.

Chapter 7

Testing

The system has been periodically tested over the entire year, with numerous testing sessions every month collecting sensor data through ROS. Various other tests have been performed to validate design decisions and verify the system will operate correctly once the other processing components are ready.

7.1 Testing Box and Trolley

The Low-Voltage subsystem engineers designed the 'Autonomous Testing Box'. This is simply constructed out of wood, and houses the two computing units, as well the GPS boards, the LiDAR interface box, and the low-voltage 12V battery. It also includes the various switches required on the vehicle, such as the GLVMS, ASMS and TSMS. This box was mounted on a 4-wheeled trolley with the ZED and LiDAR sensors mounted, and was manually pushed around various tracks to collect representative data.

Figure 7.1: The testing trolley, recording a straight-line track marked by yellow and blue cones. Credit: Jack Coleman

7.2 ROS Message Delay

The designed autonomous computing system consists of two networked computers. These computers must share information quickly and efficiently to ensure the vehicle can operate correctly. Tests were conducted to measure the latency between two ROS nodes over a network connection. A laptop was connected directly to the Jetson TX2. A ROS package was created, which contained two nodes: an 'echo' node and a 'transmit' node.

- The **echo** node received payloads on one topic, and piped them to another topic immediately. This was run on the Jetson TX2.
- The **transmit** node published payloads to one topic, and subscribed to another to receive the response messages from the echo node. This was run on the laptop.

The payload size was gradually increased, starting at 1 byte, and finishing at 1500 kilobytes. Each payload size was sampled multiple times to get an accurate measurement of the round-trip time.

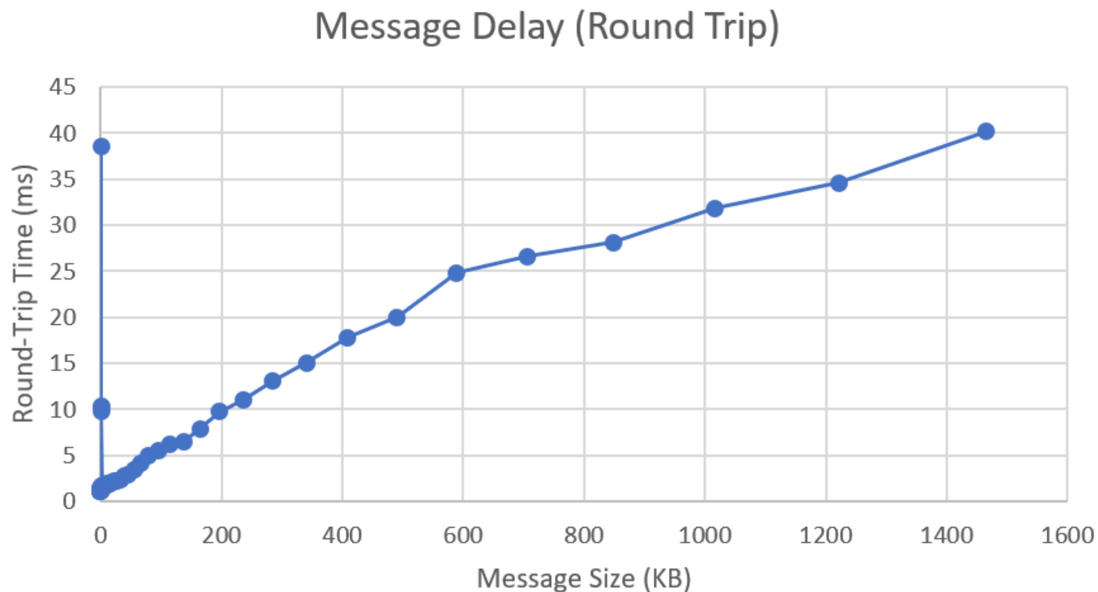


Figure 7.2: Round trip message delay between two ROS nodes on networked computers

There is a significant spike at approximately 1 kilobyte. This is due to the TCP protocol trying to respect the Maximum Transmission Unit (MTU) of Ethernet, 1500 bytes. If the payload contained within a TCP packet is greater than approximately 1000 bytes, then the packet will be split into multiple chunks. Each chunk must be sent, received, and the sender must then receive a response acknowledgement from the receiving end, before the next chunk is sent. This introduces significant delay, and seems to cause issues at the transition point. Once the payload size increases significantly beyond the MTU size, the TCP protocol begins to operate efficiently once again.

For small packet sizes, the transmission delay is sub-millisecond. The round-trip time is approximately one millisecond or less for packets smaller than 1000 bytes. The majority of messages being transferred between the Jetson TX2 and the i7 are quite small. These include cone position estimates, and heartbeats / commands to and from the master node. These messages are expected to be below 1000 bytes, and thus the time it takes to transmit data one-way between the two computing units is expected to be approximately half a millisecond.

If transmission times do become a problem, it is possible to change the transmission protocol from TCP/IP to UDP. TCP is safer and has more guarantees, however if something does go wrong or missing, it can take precious time for the protocol to realize, and correct the issue. With UDP this doesn't occur. The packet is simply transmitted and then forgotten about. If it doesn't make it to the destination, then it is lost forever, unless another external mechanism retains the message and resends it. This sounds terrible, and it is for certain situations, such as when downloading a file; a missing packet would corrupt the file. However for this particular use-case, UDP may be quite beneficial. If a packet goes missing, by the time it is realized by TCP, new data may be ready to transmit. In this case it's better to just use the new piece of information, as this is more up-to-date and accurate, especially in the case of cone position measurements from a high-speed vehicle.

UDP is supported by ROS, and can be requested as the transmission protocol when advertising or subscribing to a ROS topic.

Chapter 8

Miscellaneous Work

In pursuit of realizing Monash Motorsport's goal of an autonomous vehicle, miscellaneous pieces of work have been performed. Whilst they don't necessarily relate directly to the computing system, they are explained below for completeness.

8.1 Camera Testing

8.1.1 Basler Camera Testing

Extensive camera testing was performed in the first semester of 2018. The purpose of these tests were to collect stereo image data using different baseline widths to determine the accuracy of a stereo camera. In pursuit of this a python script was written, which interfaces with ROS and OpenCV to record images from two Basler Cameras.

Figure 8.1: Stereo Image Pair, Captured with two Basler Cameras

This python script consumed images from two other ROS nodes, one node for each Basler camera. These images were displayed in an OpenCV window. If the user pressed the **s** key, the stereo images were saved to disk. An Arduino was connected to the computer using UART, and provided a physical 'trigger' signal to the two cameras to force them to capture an image at the same time. The python script connected to this Arduino and would trigger a new stereo image capture when required.

8.1.2 ZED Camera Stereo Image Recording

Originally the Stereolabs ZED ROS Wrapper was being used to retrieve left and right images from the ZED camera. This had some issues, particularly when running at a high resolution ($1920 \times 1080 \times 2$). The image pair couldn't be transmitted between ROS nodes fast enough, and this resulted in a frame-rate of approximately seven FPS, instead of 30. A new ROS package was created, which interfaces directly with the ZED SDK and wrote the images to a ROS bag, bypassing any message transmission. This improved the frame-rate slightly, however it was still only performing at 15 frames-per-second. After disabling the saving of images to disk, it appeared that the ZED SDK was the bottle neck, and so it was removed from the system entirely. OpenCV was used instead to capture images from the ZED. These images were then written to a ROS bag. Whilst it worked correctly on the testing laptop, it failed to achieve 30 frames per second on the Jetson. At first it was believed that the bottleneck was the disk writing speed, and so an attempt was made to compress the images using the L4Z algorithm. This was a rather terrible idea and only made the situation worse, reducing the recording rate to 7 FPS again.

Finally, it was decided to bypass all things ROS. The stereo image pair was simply captured from the ZED using OpenCV, and written straight to disk in raw format with a custom header structure. The processing was split over multiple threads, one thread for capturing images, one thread for transforming image data into a writeable format, and another for writing the images to disk. Pre-allocated *std::vector*s were used to prevent reallocation of memory, and a pool of these vectors was kept and drawn from when required.

This finally resulted in a 30 FPS recording of 2 1920×1080 images on the Jetson TX2. Another script was written later to extract the images from the raw image file, and add them to a ROS bag for easy playback later. The only downside to this approach is the data size. Two 1080p images, with 3 colour channels at 30 FPS requires:

$$1920 \times 1080 \times 2 \times 3 \times 30 = 356 \text{ Mega-Bytes Per Second}$$

The Jetson TX2 has a 250 GB Solid-State Drive attached, this allows for only 12 minutes of data recording before the drive is full. This is too short for a track-drive, and a larger SSD may be required in the future if we wish to record stereo image data when running at competition.

8.2 Remote Control Actuation

The first milestone for the vehicle is early 2019, where the vehicle shall be remotely controlled. This will allow the team to test the actuation system without worrying about the complexities involved with the sensors and path planning components.

To facilitate this, a simple ROS node was created, which interfaces with a Linux joystick input device, and publishes ROS messages which contain information about the actuation states of the triggers and buttons. To test this system, another ROS node was created which receives these inputs, and simulates a simple vehicle using the kinematic bicycle model[11]. Outputs from this simulation were published as ROS visualization messages, and displayed in RVIZ; a utility for visualizing data and topics in ROS.

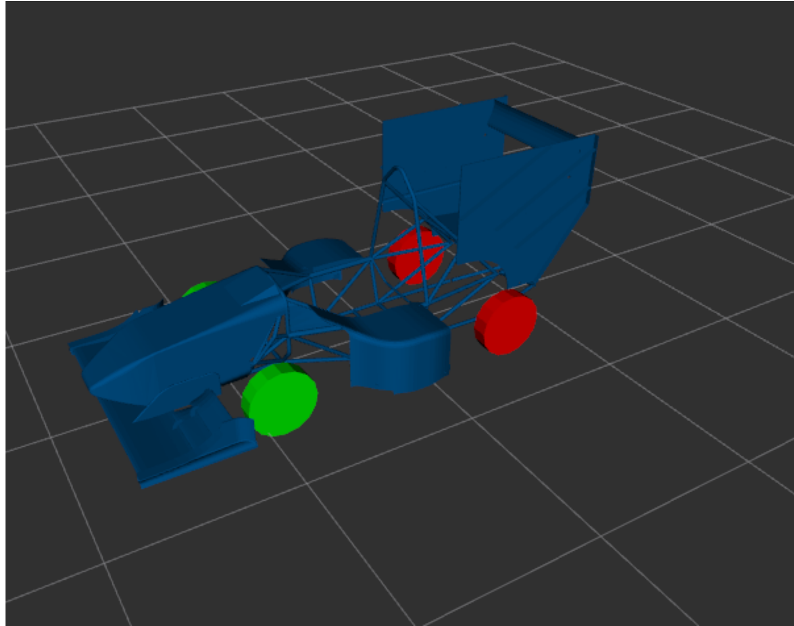


Figure 8.2: Screenshot of RVIZ Vehicle Visualization

Whilst there is still a lot of work to be done with integrating this component into the autonomous system, the foundation has been laid and future work can build upon the system that has already been designed.

8.3 Kalman Filter and Time Message Reordering

One issue that needs to be solved is the latency of the various sensors. The SLAM EKF requires that all measurements are applied in the order they were measured in. The LiDAR is low latency, and so a measurement may only take 10 milliseconds to reach the Kalman filter. However, the camera system exhibits high latency, and when a measurement arrives at the Kalman filter node, it may be from over 100 milliseconds ago. If the LiDAR measurement has already been applied, then the camera measurement cannot be applied without rolling the Kalman filter back, applying the camera measurement first, and then the LiDAR measurement.

There are a number of ways to solve this issue

- Using the aforementioned 'rollback' method, where the previous states are recorded and stored. When an old measurement arrives, the filter is 'rolled back' to the previous state, the new measurement is applied, and the proceeding measurements are re-applied. This is potentially very slow and will waste computation time.
- Propagate the measurement forward in time. Using the change in estimated state, the measurement can be transformed to behave as if it was measured now, instead of 100 milliseconds ago. This has the downside of not being numerically stable in certain cases, as the filter is basically using a previous estimate to update a new estimate.
- Delay incoming measurements by the highest latency sensor. e.g. delay LiDAR measurements by 100 milliseconds, so camera measurements have a chance to be applied first. This has the downside of increase estimation latency.

The current plan is to utilize the last option, and combine it with the first option. The first option is ideally the best solution, however the increased computation time is detrimental to

system performance. If the Kalman filter is constructed correctly however, then this performance loss can be reduced significantly. Measurements from the GPS and IMU sensors only affects a small portion of the Kalman filter state. Thus these measurements will be applied as they are received. The LiDAR shall be delayed to be synchronized with the camera latency. When it's time to apply a cone measurement from the LiDAR or cameras, the Kalman filter will be rolled back. However only IMU and GPS measurements are being rolled back in this case, which is significantly faster than rolling back cone measurements. The cone measurement is then applied, and the IMU and GPS measurements are re-propagated through the Kalman filter to determine the current state.

Simply put, measurements from the LiDAR and Camera help localize the vehicle within the track. These are slow and cumbersome to apply to the Kalman filter. IMU and GPS help determine where the vehicle has moved in a short period of time. These are small and fast to apply to the Kalman filter. The IMU does introduce drift, however the periodic measurements from GPS, LiDAR and Cameras remove this drift.

To delay these measurements, a simple library has been written. When instantiated, the *TimeReorder* object creates a vector which is sorted by the time the message should fall out of the buffer. New messages are inserted into the vector in the correct ordered position. A ROS Timer waits for the shortest period and calls a callback function with the delayed message when ready. To use, it's quite simple. Simply create the object, specify a callback function and add messages as required. This will be used by the SLAM system to reorder LiDAR and Camera measurements as required.

```
mms::TimeReorder<geometry_msgs::PointStamped> reorder(ros::Duration(1.0),
    [](const geometry_msgs::PointStamped& msg){
    // Delayed messages exit the buffer here
});
geometry_msgs::PointStamped msg = getMessage();
// Messages enter the buffer here
reorder.addMsg(msg.header.stamp, msg);
```

8.4 M19D Dashboard

Whilst both computing units are capable of outputting video data to a display over HDMI, this is unnecessary and thus is not being used on the vehicle. However this makes interacting with the computing units rather difficult. One must SSH into the machine and execute commands from the command-line. To do this, the IP address of the machines must be known. When connecting to a WiFi network such as Monash eduroam, IP addresses change almost daily. To solve this issue, the M19D Dashboard was created.

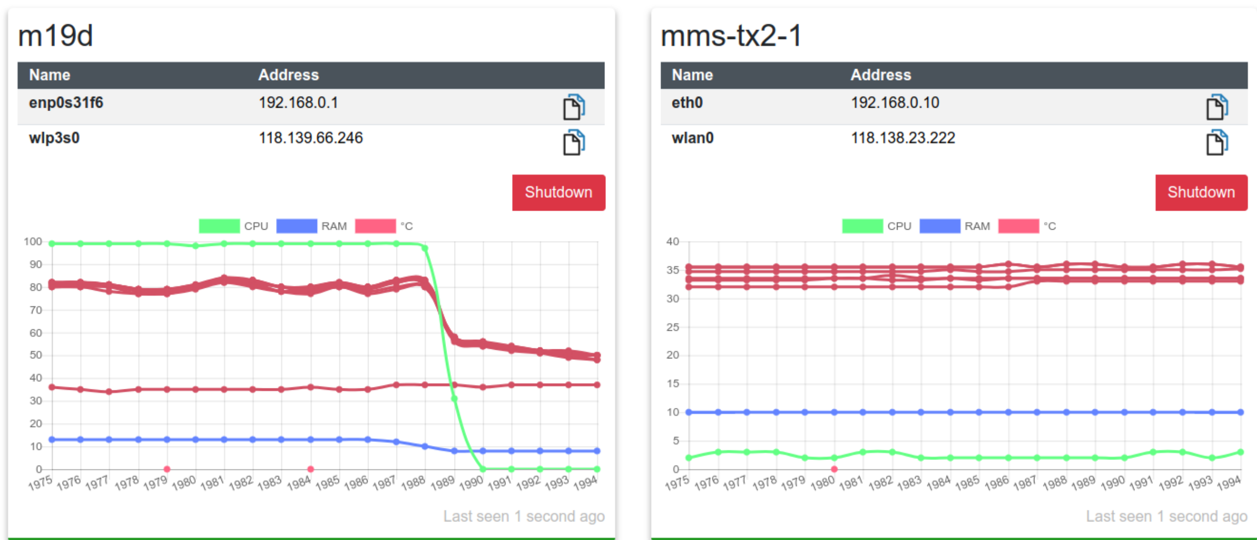


Figure 8.3: Screenshot of the M19D Web Dashboard

Upon startup, the two computing units instantiate a NodeJS process. This process connects to an MQTT broker that is situated online, and periodically publishes telemetry data to the broker. This includes RAM usage, CPU usage and temperature sensors. Another NodeJS process running side-by-side with the MQTT broker monitors these MQTT messages, and retains the latest telemetry information for each connected machine. It also maintains an HTTP server, which serves static resources such as scripts, CSS and the main html page. It also provides a REST API for querying the IP addresses of connected machines.

Users of the dashboard are first presented with a bare screen. Upon clicking a button, they are also connected to the MQTT broker using WebSockets. All messages published by the computing units are now forwarded to the web page. A script on the page extracts the relevant data, and displays live updates of performance metrics and IP addresses of the various network interfaces. The dashboard also contains a button to shutdown the computing units when they're no longer required.

The NodeJS server component exposes an endpoint, which returns the IP address of the requested computing unit. The URI is as follows:

/ip/<HostName>/<NetworkInterface>

This allows the IP address of the machines to be fetched dynamically from within a bash script, and allows the creation of scripts which automatically SSH into the desired machine, without the end user worrying about what the actual IP address is.

The server components are currently running on a personal EC2 instance using Amazon Web Services. In the future it is planned to decouple the server components from AWS and implement them on the computing units themselves, so no external internet connection is required.

8.5 Thermal FEA

One test that is performed at competition is the 'rain test', where water is lightly sprayed over the vehicle to ensure that nothing is damaged. Thus the computing units must be waterproof when mounted on the vehicle, otherwise this test will fail. However making computers waterproof, while still allowing heat to leave the system is a difficult task. The devised solution

utilizes blocks of aluminium to extract heat from the processing units, and transfer that heat to the externals of the computing box to be cooled by a waterproof radiator and fan. A previous test of the computing system showed that under maximum load, the processor isn't cooled fast enough, and performance is throttled to ensure the CPU temperature remains at 80 degrees. Thus a more efficient cooling design will enable higher performance of the processor.

To ensure this system shall work, thermal Finite-Element-Analysis was performed. The following assumptions were made:

- Only the externals of the box and heatsink radiate heat.
- Only the externals of the box and heatsink convect heat.
- The heatsink has a convection coefficient of 60 $W/(m^2K)$
- The box externals have a convection coefficient of 10 $W/(m^2K)$
- Only the processors generate heat; 65 Watts at the i7 and 15 Watts at the Jetson TX2.
- All interfaces are perfectly conductive, i.e. no loss in heat transfer
- The ambient air temperature is 40 degrees
- The box is in complete darkness, i.e. it's not in sunlight

Whilst there is at least another 40 Watts of heat being generated within the box due to passive PCB components, this is difficult to model accurately. The FEA simulation performed doesn't take into account heat transferred between components, and the air inside the box. These missing components will skew the results slightly, and should be kept in mind. The extra heat generated will raise the ambient temperature, however the heat transfer through the air will also lower the ambient temperature. How much each factor contributes is hard to determine.

The stock Intel heatsink was also simulated using similar assumptions. From previous testing it was expected that the heatsink would reach a temperature higher than 80 degrees. However, it did not.

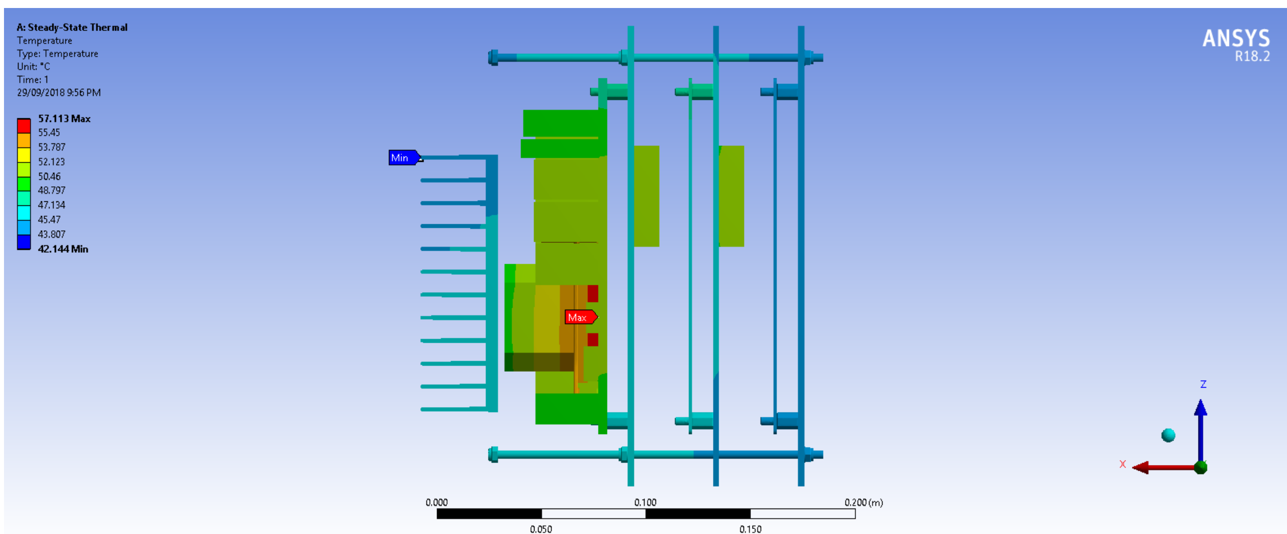


Figure 8.4: Autonomous Computing Box Thermal FEA Results

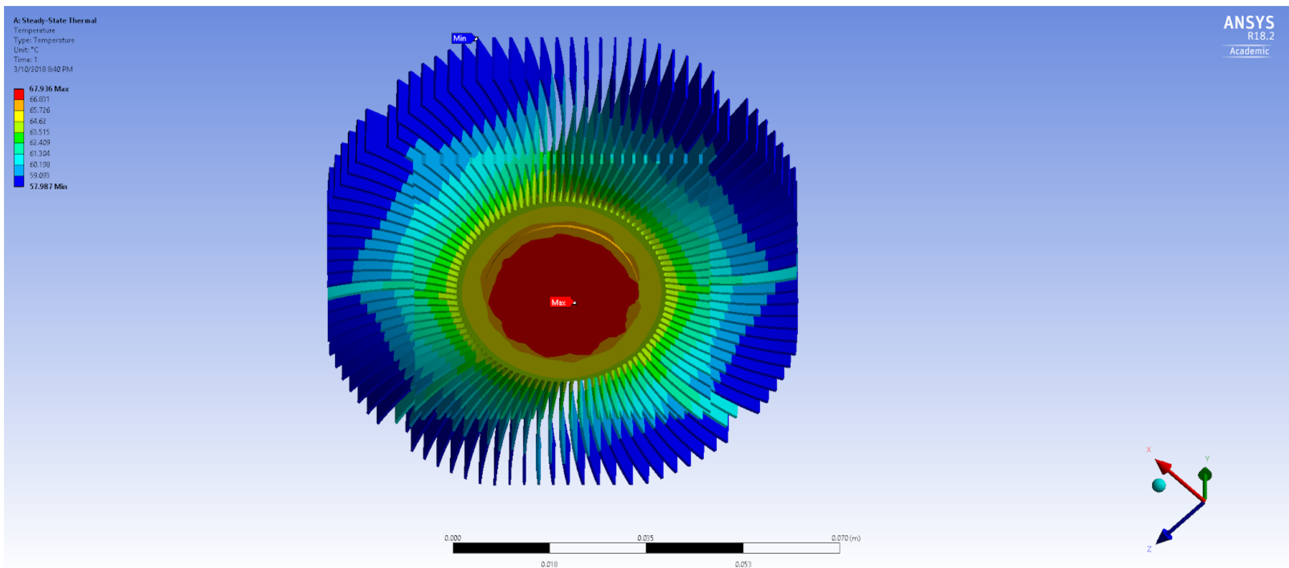


Figure 8.5: Intel Stock Heatsink Thermal FEA Results

From simulation, the aluminium heat-block solution is expected to reach a maximum temperature of 57 degrees, whilst the Intel heatsink is expected to reach a maximum temperature of 68 degrees. This is both good and bad:

- **The Bad:** The thermal FEA performed was not entirely accurate, and isn't indicative of actual real-world thermal performance.
- **The Good:** The aluminium heat-block solution is thermally more efficient at extracting heat from the system than the Intel cooler.

So while it is difficult to say exactly what kind of performance will be seen with the aluminium heatblock design, it can be assumed that the performance will be better than the stock Intel heatsink that is currently being used. Although, this isn't exactly a very promising statement, as the Intel heatsink is known for being sub-par.

Chapter 9

Conclusion

9.1 Requirement Satisfaction

To reiterate, the high level requirements of the computing system were as follows:

1. Provide a unified platform for other subsystems to use for processing.
2. Provide a method to share information between multiple subsystems
3. Interface with the existing ECU, or, implement the functionality of the existing ECU
4. Be able to process images and feed-forward pre-trained neural networks
5. Implement a state-machine to keep track of the vehicle status

These requirements have been satisfied:

1. The hardware and utilities to interface with the hardware have been created. All of the computing components are linked, either through Ethernet or UART, and there exists a mechanism to interface with the vehicle.
2. Robot Operating System provides a robust method to share information between subsystems.
3. Through the PSoC 5LP the system can interact with the ECU over CAN.
4. The Jetson TX2 is doing both of these tasks, processing images and feed-forward neural networks.
5. The PSoC implements the state machine.

On top of these initial requirements, new issues have come to light that needed to be solved. Safety of the autonomous system was not originally considered, however through the course of development it was realized that safety is actually the number one concern. The method implemented, using a combination of heartbeats and physical hardware solves this issue elegantly and robustly.

Due to the ongoing nature of Monash Motorsport, this project is not complete. The autonomous vehicle is not scheduled to be driving until mid 2019, and there is still much work to be done by all subsystems. Off the back of the work done in this project the autonomous system shall be built, and hopefully it will help Monash Motorsport win respect and glory in the European campaign of 2020.

9.2 Future Work

As mentioned, the project is far from over. Actuation components must be purchased and fit to the vehicle. The autonomous computing box must be cut and assembled. The SLAM EKF must be upgraded to work with Camera cone measurements, and the Path Planning algorithms still require some effort before they're ready to control the vehicle on-track.

Specific to computing, the PCB boards must be sent for manufacture, and then assembled and tested. The statemachine on the PSoC must be completed, with all states and state transitions. It must subsequently be tested to ensure it behaves as expected.

The camera subsystem designers have struggled with the Jetson TX2 over the past few months. The ARM processor is far from powerful, and it's difficult to optimize code for CUDA, using reduced precision computing. As a result, cone detection is only just running at 12 frames per second, and ideally should be running faster. In the future it may be beneficial to either upgrade the TX2 to a more powerful machine, such as the NVIDIA Xavier which came onto the market quite recently. Alternatively a consumer-grade graphics card such as a GTX 1050 could be utilized, connecting it to the i7 processor through the spare PCI-Express slot. This has a theoretical performance of 1,862 GFLOPS, just under 3 times the performance of the Jetson when working in single precision mode. This would require an additional 70 Watts of power and would require a larger battery to compensate for the extra energy consumption. However this is far from impossible. The Jetson TX2 was designed for highly constrained applications, such as a drone; an autonomous racing vehicle has far fewer constraints and it's more than possible to add a second battery when more power is required.

Bibliography

[1] B. Sagar and T. Martin, "A functional architecture for autonomous driving," 2015. [Online]. Available: http://sagar.se/files/wasa2015.pdf

[2] T. H. Drage, "Development of a navigation control system for an autonomous formula sae-electric race car," 2013. [Online]. Available: http://robotics.ee.uwa.edu.au/theses/2013-REV-Navigation-Drage.pdf

[3] L. Shaoshan, T. Jie, Z. Zhe, and G. Jean-Luc, "Caad: Computer architecture for autonomous driving," 2017. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1702/1702.01894.pdf

[4] G. Granosik, K. Andrzejczak, M. Kujawinski, R. Bonecki, L. Chlebowicz, B. Krysztofiak, K. Mirecki, and M. Gawryszewski, "Using robot operating system for autonomous control of robots in eurobot, erc and robotour competitions," 2016. [Online]. Available: https://doi.org/10.14311/APP.2016.6.0011

[5] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," 2009. [Online]. Available: http://www.willowgarage.com/papers/ros-open-source-robot-operating-system

[6] M. I. Valls, H. F. Hendrikx, V. J. Reijgwart, and F. V. Meier, "Design of an autonomous racecar: Perception, state estimation and system integration," 2018. [Online]. Available: https://arxiv.org/abs/1804.03252

[7] M. F. Cloutier, C. Paradis, and V. M. Weaver, "A raspberry pi cluster instrumented for fine-grained power measurement," 2016. [Online]. Available: http://dx.doi.org/10.3390/electronics5040061

[8] *CUDA Toolkit Documentation*, NVIDIA. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions

[9] *PSoC 5LP Architecture TRM*, Cypress Semiconductor, 2017. [Online]. Available: http://www.cypress.com/file/123561/download

[10] *Formula Student Rules, 2019*, Formula Student Germany, 2018. [Online]. Available: https://www.formulastudent.de/fileadmin/user_upload/all/2019/rules/FS-Rules_2019_V1.1.pdf

[11] P. Polack, F. Altché, B. d'Andréa Novel, and A. de La Fortelle, "The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles?" in *2017 IEEE Intelligent Vehicles Symposium (IV)*, June 2017, pp. 812–818.

Glossary of Terms

F-SAE	Formula - Society of Automotive Engineers. A sub-division of the SAE which oversees formula student competitions around the world.
ROS	Robot Operating System. A meta operating system for robotic applications.
PSoC	Programmable System on Chip. A microcontroller developed by Cypress Systems. Very versatile.
MQTT	A publish/subscribe messaging service, originally developed for connecting oil pipelines over unreliable satellite networks.
NodeJS	Javascript for a server. Javascript usually runs in a web browser, however it's also possible to run javascript on a server. Due to the structure of Javascript (single threaded, with an underlying event-queue) it's very good for processing tasks which are required to wait on other tasks. e.g. persisting data in a database, or querying a web-service and waiting for the response. NodeJS is built on Google's V8 Javascript engine.
UART	Universal Asynchronous Receive Transmit - A computer hardware device for asynchronous serial communication. 'Universal' meaning configurable, you can change the baud rate, start and stop bits, flow control etc.
SLAM	Simultaneous Localization And Mapping - An algorithm which uses an extended Kalman filter to determine not only where the observer is, but where landmarks in the world are too. This is exactly what the autonomous vehicle needs to do.
Kalman Filter	An algorithm which combines a series of statistically noisy measurements over time, to produce estimates of unknown variables, which are more accurate than any single measurement alone.
EKF	Extended Kalman Filter - A version of the Kalman filter, which enables you to use non-linear measurements, i.e. measurements involving sine or cosine.
GLVMS	Grounded Low Voltage Master Switch - Enables low-voltage power to the vehicle.
ASMS	Autonomous System Master Switch - Enables the autonomous system. When this is off the vehicle is incapable of performing any autonomous action.
TSMS	Tractive System Master Swtich - Enables / Disables the vehicle's tractive system
FLOPS	Floating Point Operations Per Second - A measure of how many numbers can be processed every second. An operation is usually classified as an addition, subtraction, multiplication or division.
Floating Point Number	A number, which is represented by a base and exponent. The exponent causes the decimal place to 'float', allowing a wide range of possible values.

Appendix A - PSoC Pinout

Name	PSoC Pin	Description
Digital Output 1	P2.0	ASSI Yellow GND Sink (Sink LED current to ground)
Digital Output 2	P2.1	ASSI Blue GND Sink (Sink LED current to ground)
Digital Output 3	P2.2	TBD
Digital Output 4	P2.3	TBD
Digital Output 5	P2.4	TBD
Digital Output 6	P2.5	Initial Brake Check
Digital Output 7	P2.6	Watchdog PWM Out (Trigger EBS when missing)
Digital Output 8	P2.7	Shutdown Close (manually turn off tractive system)
Digital Output 9	P12.5	Set Driving State (EBS Ref Guide)
Digital Output 10	P12.4	ARM EBS
Digital Output 11	P12.1	Statemachine OK
Digital Output 12	P12.0	Set Finish State
Digital Input 1	P3.0	ASMS
Digital Input 2	P3.1	TSAL
Digital Input 3	P3.2	TBD
Digital Input 4	P3.3	TBD
Digital Input 5	P3.4	Shutdown Circuit Is Closed
Digital Input 6	P3.5	Driving State Active
Digital Input 7	P3.6	EBS Is Armed
Digital Input 8	P3.7	EBS Is Activated
Analog Input 1	P15.2	Steering Angle Sensor Feedback
Analog Input 2	P15.3	Brake Actuator Feedback
Analog Input 3	P15.4	Brake Line Pressure 1
Analog Input 4	P15.5	Brake Line Pressure 2
Analog Input 5	P0.0	EBS Pneumatic Energy Storage Pressure
Analog Input 6	P0.1	TBD
Analog Input 7	P0.2	TBD
Analog Input 8	P0.3	TBD
Analog Output 1	P0.4	Steering Angle Output
Analog Output 2	P0.5	Brake Pedal Actuation Output
Analog Output 3	P0.6	TBD
Analog Output 4	P0.7	TBD
CAN RX	P1.6	CAN Receive
CAN TX	P1.7	CAN Transmit
UART RX	P12.6	UART (to i7) Receive
UART TX	P12.7	UART (to i7) Transmit
XTAL 1	P15.0	External Crystal Oscillator, Pin 1 (24 MHz)
XTAL 2	P15.1	External Crystal Oscillator, Pin 2 (24 MHz)

Appendix B - PSoC Command List

ID	Command Name	Payload In	Payload Out	Description
1	Get State	None	1 Byte - The state	Gets the state of the internal PSoC statemachine. This is the state machine as defined in FSG rules.
2	Select Mission	1 Byte - The mission	None	Selects the mission
10	Set Digital Out	PinPairList	None	Writes a digital HIGH or LOW to a set of output pins.
11	Get Digital Out	PinList	PinPairList	Gets the written digital HIGH or LOW output status from the output pins.
12	Get Digital In	PinList	PinPairList	Returns a list of digital HIGH or LOW values
13	Set Analog Out	PinPairList	None	Sets the desired analog pin voltage. 16mV per bit.
14	Get Analog Out	PinList	PinPairList	Returns the set desired analog pin voltages.
15	Get Analog In	PinList	PinPairList	Returns the last value read from a set of analog input pins
255	Error	None.	String Message	Errors from the PSoC are sent back to the i7 using this command.

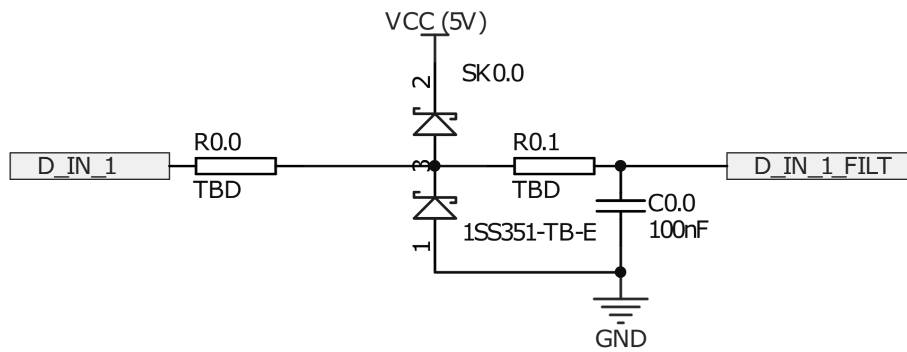
PSoC Payload Structures

Name	Description
PinList	A payload of size **n**, where **n** is the number of pins being queried. Every byte simply represents a pin ID, starting at 0. For example. **0x00010203** Refers to pins 0, 1, 2 and 3. This is the same for both digital and analogue.
PinPairList	A payload of size **2n**, where **n** is the number of pins being sent. Every 2 bytes represents a pair, first value is the pin-ID, the second is the pin-value. For analogue pins, this is 0-255. For digital pins, 0 = LOW > 0 = HIGH. For example, a digital command **0x01010200** Would represent Pin1 = HIGH, Pin2 = LOW. For an analogue command **0x03FF040A** Would represent Pin3 = 255/255, Pin4 = 10/255

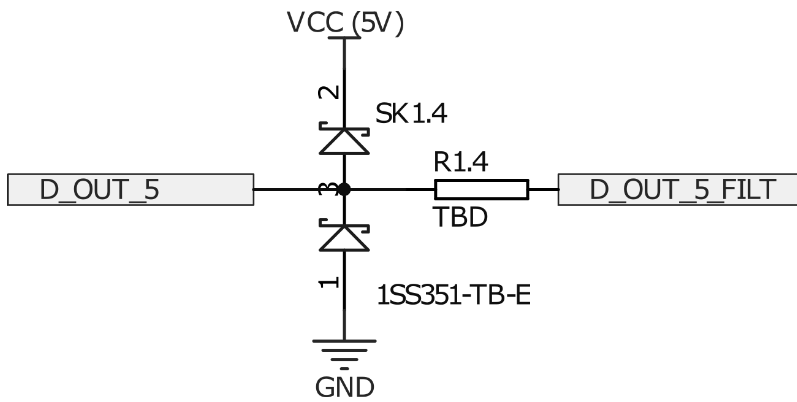
Appendix C - ROS Message Structures

Message	Structure	Description
Cone	Header header uint16 id float32 x float32 y Time time_seen uint8 confidence uint8 cone_type uint8 coordinate_frame	A single cone. header: The time the message was generated id: The local identifier of the cone. This is used to update measurements in the Kalman filter. x: The x position of the cone y: The y position of the cone time_seen: When we believe the cone was seen in the world cone_type: An enum which specifies the type of cone: Blue, Yellow, Orange Big, Orange Small. The enum values are stored in the message definition. coordinate_frame: Our own custom coordinate frame designation. 0 for world, 1 for lidar, and 2 for camera.
ConeArray	mms_master_msgs / ConeLIST	Simply an array of cones. Allows us to bundle up multiple cones into a single message to avoid the overhead that comes with TCP transmission of many small packets.
Heartbeat	uint8 state string message	state: The state of the node sending the heartbeat 0 = Init, 1 = Ready, 2 = Running, 3 = Fault, 4 = Failure message: A descriptive message of why the node is in the same. This can be blank, but if it's an error, then it's good to specify why the error occurred, for logging purposes.

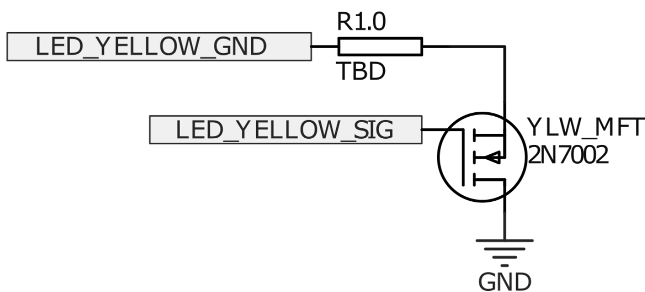
Appendix D - PSoC Circuitry



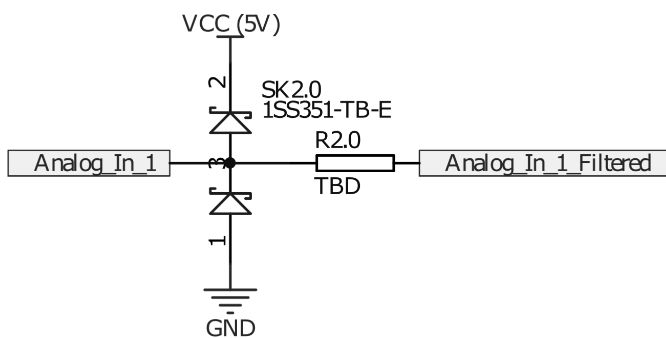
Digital Input Schematic



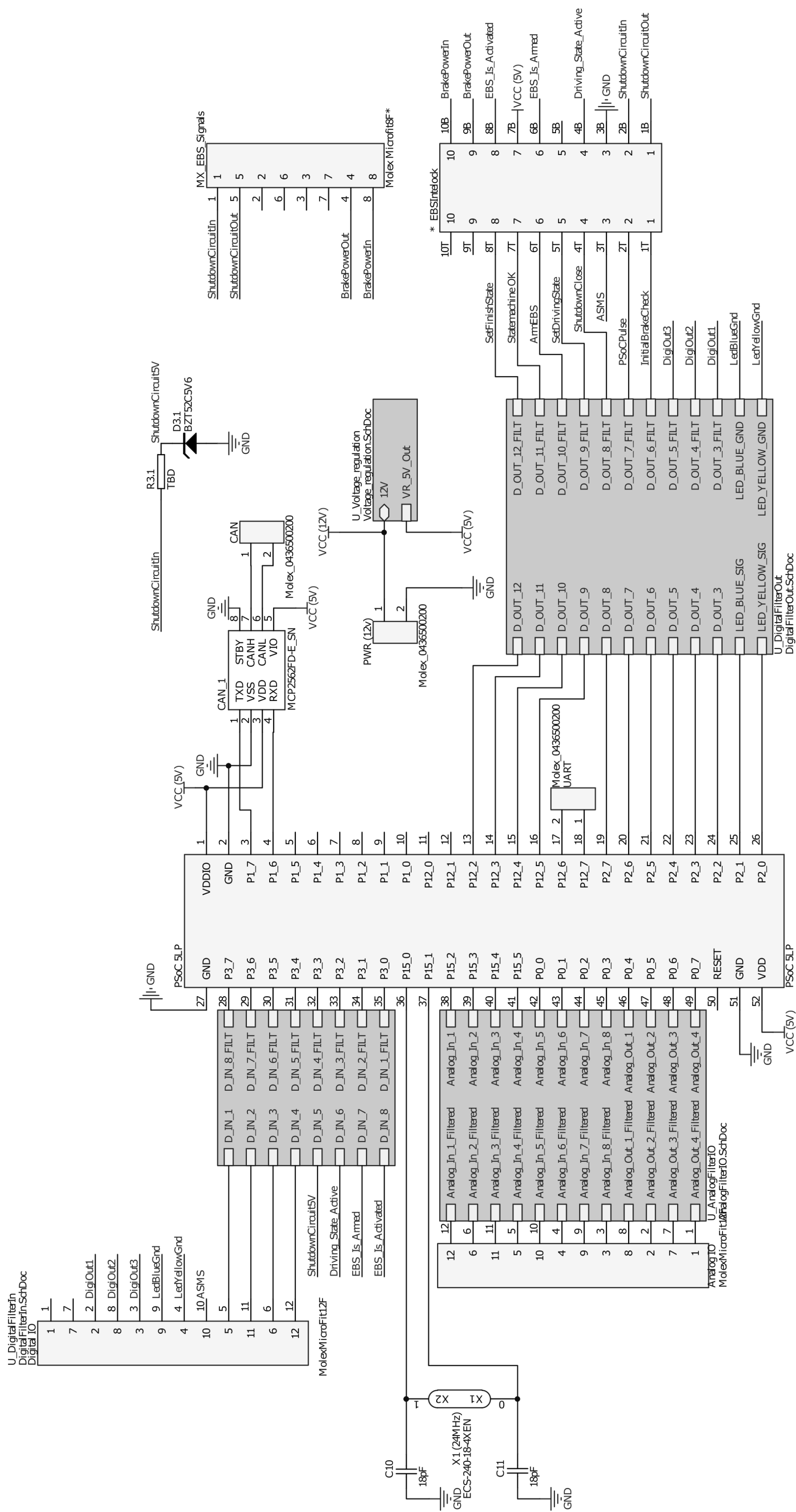
Digital Output Schematic



ASSI MOSFET Schematic



Analogue IO Schematic



Main Circuit Schematic