Engineering Design Project Report

Software and System Integration

By: Liam Gritters
Student Number: 100976928

Reviewed By: Trever Middleton

1125 Colonel By Dr
Ottawa, ON

liamgritters@cmail.carleton.ca

Table of Contents

Executive Summary

The Integrated Autonomous Vehicle (IAV) project is a new 4th year capstone project created this year and is part of the Mechanical and Aerospace Department at Carleton University. The overall goal of the project is to create an autonomous shuttle to transport students around the Carleton campus. This goal must be broken down into smaller, more achievable goals for teams to meet each year. This year's objective it to create a prototype vehicle by re-engineering an old buggy frame and equipping it with the necessary sensors and software for it to be able to drive autonomously in ideal conditions. An important aspect of robotics is the software implemented on it. Through the developed software, the controls and logic of the vehicle are realized. The focus of this report is the software integration completed this year on the IAV prototype, more specifically the framework, system architecture, and control system that will allow the vehicle to smoothly operate.

Highlights

This report discusses five aspects of the project:

1. The virtual simulation of the vehicle
2. The network and messages between the various software components of the system
3. The software system architecture
4. The control system
5. The device drivers developed for the hardware components

This is followed by a conclusion and a section discussing future work and improvements of the system. Additional work completed is also included at the end of the report.

Introduction

Software integration is the process of aggregating diverse sub-systems to combine and form one coherent multi-functional system [1]. Within this project, software integration has played a key role in the overall design and formation of the system components and has largely influenced the outcome of the final product. In this manner, software and the software integrator act as the glue between the other groups within the project. Moreover, within such a complex system such as an autonomous vehicle, software integration also incorporates the hardware and sensors used, and thus evolves into a role of a system integrator with a large focus on software integration. Therefore, in the later half of this report, the software developed will also be discussed in relation to the hardware components used.

IAV Simulator

Testing is critical to software development especially in a high-risk environment such as that associated with an autonomous vehicle. A simulator or simulation program offers a method to test changes and new features in a virtual space, lowering the risk of an accident occurring.

The IAV simulator was developed to do just that, to test algorithms developed on the vehicle in a virtual space. This also drastically speeds up the time required to develop software for the vehicle, as it can be tested without the need for a physical vehicle. However, the simulation operates in ideal conditions and testing in a physical environment is still required to properly examine the algorithm due to the noise related to sensors.

The simulation was built on Webots Simulator, which is a complete development environment to model and program robots [2]. Within the simulation the vehicle is split into three parts; the buggy model, the sensors added to it, and the software controller to function the robot. Additionally, real world maps with associated GPS data can be imported into the simulator to test the robot in a familiar space. This was done with a portion of the Carleton campus since the vehicle will inevitably be tested there, as shown in Figure 1.

Figure 1 Simulated map of the Carleton University Campus

Buggy Model

A model of the buggy frame was uploaded into the simulator using a previously created CAD model as show in Figure 2. It was determined that the buggy with all the sensors, motors, batteries and a single person, would have a design weight of 700 lbs. The vehicle in the simulator was thus also modelled to weigh 700 lbs. Hinge-joints were incorporated on either side of both axes to model the rotation of the wheels. These joints can be actuated through the robot's controller

software and are comparable to motors. Unfortunately, in the simulator it is difficult to model Ackermann steering and currently the simulator incorporates skid steering. This should be fixed in the future years, as the real buggy has been modified to allow for Ackermann steering. Currently, the rear wheels in the simulator act independently replicating a split axle design, which is currently included in the real buggy.

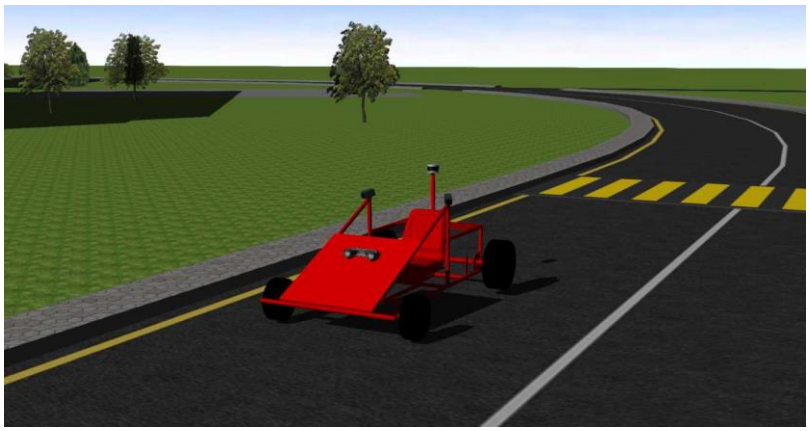

Figure 2 Buggy Simulator Model

The simulator includes a physics module which more accurately models the dynamics of the vehicle. This in turn creates noise in the environment, due to vibration from rough terrain and slip in the vehicle's wheels. The noise produced is important when developing an algorithm which relies on encoder data from the wheels or images from a camera, as both these sensors will be affected. Friction is also introduced into the environment with the physics node. The coefficient of friction has currently not been adjusted but it can be tuned to improve the overall model of the system. To do this, the friction of the rubber wheels and asphalt road would have to be determined.

Sensors

The simulator includes all 6 different sensors used on the real vehicle; LiDAR, stereo cameras, two monocular cameras, GPS, IMU and rotary encoders. The sensors are included as nodes in Webots, making it quite easy to model. Each sensor can be accessed through the robot's software controller.

The LiDAR publishes an array of range values with each row representing a planar scanline and each column representing the range at that angular location. For instance, if a LiDAR published a range every 5 degrees with a horizontal field of view of 40 degrees and vertical field of view of 10 degrees, the data outputted could be represented by Table 1. The columns of the table represent the horizontal angular values, while the rows represent the vertical elevation of the range.

Table 1 LiDAR Data Example

	-20°	**-15°**	**-10°**	**-5°**	**0°**	**+5°**	**+10°**	**+15°**	**+20°**
+5°	1.1	1.1	1.2	4.0	4.0	0.5	3.0	3.1	2.0
0°	1.0	1.0	1.1	3.8	3.8	0.4	2.9	3.0	1.9
-5°	1.1	1.1	1.0	2.0	0.4	0.5	3.1	3.0	0.5

The LiDAR data is stored in a dynamic array represented as a multidimensional vector in C++. This is to allow for the array dimensions to be changed instantly if a different lidar, which publishes a different amount of data, is used.

The camera publishes image data as an array of pixels. The resolution of the camera can be specified in the camera node embedded in the robot node of the simulator. The pixel values are represented by 3 colour values; red, green, and blue, which are adjacent to each other in the array. Therefore, if an image has a resolution of 1280 x 720, there would be 3840 x 720 values in the array. Similar to a camera, the stereo camera also publishes an array of pixels but with an additional channel representing depth. So instead of 3 colour values representing a pixel there would be 4 values; 3 for colour and 1 for depth. Both sets of data are stored in dynamic arrays, like the ones used for the LiDAR data.

The simulated GPS publishes 4 values; longitude, latitude, altitude and GPS speed. The coordinates are published as a vector with 3 indices, with each index representing one of the coordinates. The GPS speed is published as a single precision (float in C++) value. The IMU publishes data similar to the GPS, but with each index representing either the roll, pitch or yaw of the vehicle.

Finally, the rotary encoders publish the rotation of the wheels represented as a number of ticks, with one full rotation represented as 1024 ticks. This data can be used to estimate the amount of displacement of the vehicle.

Software Controller

The controller is the software interface to the robot developed in the simulator. It has access to all the sensors incorporated on the robot and provides control commands to the robot's actuators, in this case the wheels. The controller is written in C++, which is a commonly used programming language for robotic systems. The controller is broken up into different modules, known as classes in C++, which focus on one aspect of the robot. From a high-level perspective there are 3 main classes; System Manager, Sensor Manager and Controls Manager. The System Manager is the highest level in the controller. It manages the other two classes and sets up all the processes and messages related to the program. The Sensor Manager receives and stores all the data from the simulated vehicle. The Controls Manager deals with everything related to the motors and encoders. Commands are sent to the Controls manager to move and steer the vehicle. All the code for these classes is included in the appendix section of this report.

Software Utilities

Three utilities were created to display the data produced in the simulator from three different sensors. These utilities were made to act as an example of how to receive data from the simulator. The first utility displays the middle scanline of the Lidar and shows it as a 2D planar view. The second, prints out the GPS data to a terminal. The third prints out IMU data to a terminal. The utilities can be seen in Figure 3.

Figure 3 Simulator Utility View

System Network

As it is the first year of the project, it is crucial to lay a strong foundation that can be built upon in further years. The designed system should have the following key features:

1. Expandable
2. Adaptable
3. Resistant to Failure
4. Real Time Operation
5. Concurrent Operation

Expandability focuses on the ability of the software system to grow and incorporate new features and more advanced levels of autonomy. The system must be designed in such a way that future students can add to the software while also taking advantage of the current capabilities.

Adaptability addresses the flip side of this, taking away or changing a portion of the current capabilities without remaking the entire system. This allows for the system to be adapted in future years to allow for new designs and iterations.

The system must also be resistant to failure and in the occurrence of failure, it must act in an appropriate manner. The software system must not completely shutdown if a sensor or software component fails.

Real time operation and concurrent operation address a similar aspect of the system. Essentially, all the software components must run fast enough to handle issues occurring in real time in the environment it is acting in. In order for such a complex system to accomplish this, multiple software components must run in parallel or at the same time, thus the system must allow for concurrent operation.

All the above features are answered with the use of a communication network. A communication network allows for multiple programs running on a computer to communicate with each other. Additionally, the network can be expanded to communicate with outside devices and computers, allowing for communication between multiple vehicles in future years.

By sharing data between multiple components running on the same computer, real-time and concurrent operation can be achieved. Additionally, this adds a level of security to the overall system, increasing its resistance to failure. If a component were to fail then the overall system would not fail with it, instead it would detect the fault and handle the error in a safe manner. Furthermore, components can be changed or added within the system without affecting the operation of other components allowing for expandability and adaptiveness of the overall software system.

The communication network used within this system is called a Lightweight Communications and Marshalling (LCM) network.

LCM Network

LCM is a software network used to publish and receive messages [3]. More specifically it is a set of libraries and tools for message passing and data marshalling. The reason it was selected was because it is targeted at real-time systems where high bandwidth and low latency are critical. These are two very important factors within an autonomous vehicle. There

is a need for high bandwidth due to the amount of data published from all the sensors onboard the vehicle. Likewise, the latency related to messages must be as low as possible due to the critical environment the vehicle will operate in.

LCM provides functions to publish and listen to messages on an Internet Protocol (IP) Network. The messages are in the form of User Datagram Protocol (UDP) messages. UDP is a connectionless communication model and does not require a port to receive the messages on the other end. Furthermore, it does not perform any handshaking protocol to confirm that the messages were received. Although this can lead to unreliability in the network, it avoids overhead in processing and decreases latency, making it more suitable for real-time applications. The unreliability in the messages can be resolved by error-checking within the program. LCM also provides functionality to encode and decode all the data.

Figure 4 describes the process of publishing and receiving a message from one software program to another. Within the IAV system, an example of this would be the Sensor Manager, represented as Program 1, publishing LiDAR data to the Obstacle Detection component, represented as Program 2. The data is encoded as a UDP message which is then published to an IP Network. Concurrently, the Obstacle Detection component is listening into the IP network for LiDAR data. Once it hears that data has been published it then receives that UDP message and decodes it back into LiDAR data.

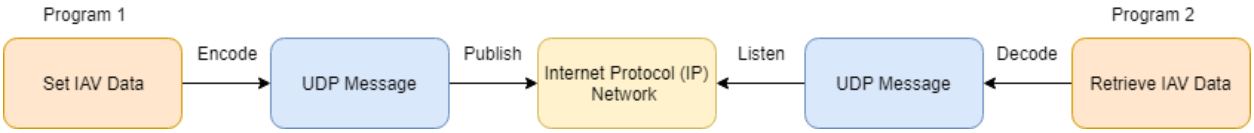

Figure 4 LCM Message passing

The LCM network lays the foundation for the system architecture by allowing for communication between its associated software components.

Software System Architecture

One of the most important parts of a system is how it comes together. Typically, when developing a complex system like an autonomous vehicle the system architecture is planned and laid out. Like all designs, this goes through an iterative process before being finalized.

Initial Design

The initial design followed a flow pattern, which provides hierarchy to each component. This system architecture is shown in Figure 5. The System Controller is the highest level of the system, only being exceeded by inputs from the user via an interface such as a gamepad/joystick or graphical user interface (GUI). The System Controller would then manage the 3 main autonomous components of the system; the Path Planner, Path Follower and Pose Estimator.

1. The Path Planner is responsible for planning the vehicles route. It receives a destination and determines the best possible path to take to get there. This module would also deal with rerouting the path if an object were to get in the vehicle's way.
2. The Path Follower component would be responsible for following the path set out by the Path Planner. It would constantly monitor the data published by the sensors to determine if the path generated is safe to take and that no objects move in front of it.
3. The Pose Estimator works together with the Path Planner and Path Follower to give the best possible estimate of the vehicles position and heading in real space. This component fuses data from the GPS, IMU and

encoders to reduce positional error in the system. Additionally, it corrects the vehicles position to stay within a determined lane.

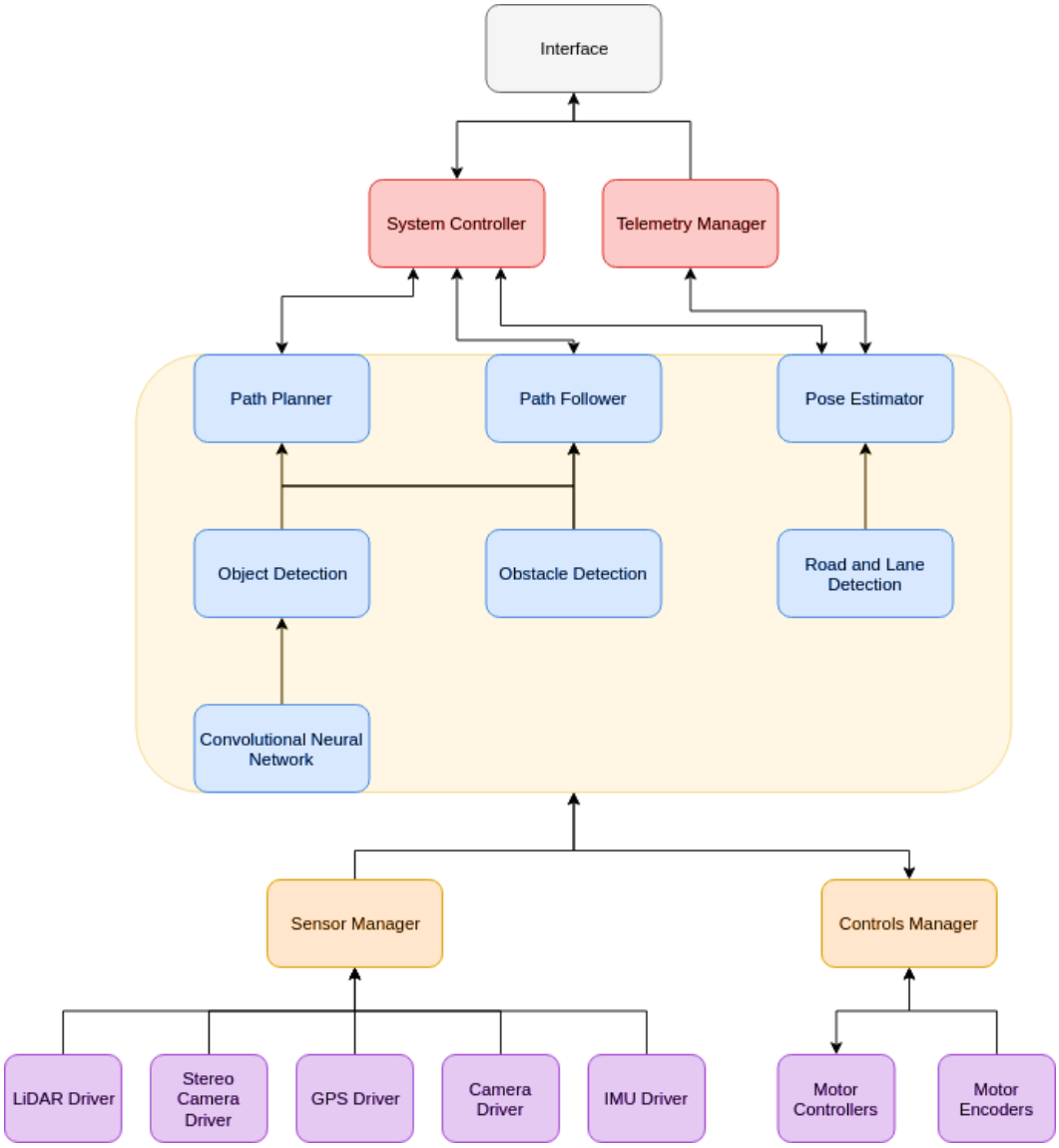

Figure 5 Initial System Architecture

The components below these previously described ones, each interpret the data gathered by sensors of the surroundings of the vehicle.

1. The Object Detection module uses a convolutional neural network and a stereo camera to determine what kind of objects appear in front of the vehicle.
2. The Obstacle Detection component uses LiDAR data to find any kind of obstacle around it. The type of obstacle is unknown but is very reliable at detecting some kind of obstacle compared to the Object Detection component. Moreover, these components provide some redundancy within the system, if one were to fail.
3. The Road and Lane Detection component detects the lane the vehicle must stay within. This is very important when driving on a public road. This data is used to correct for the vehicles position.

Finally, the last two modules are the Sensor Manager and Controls Manager components. As the name implies, they manage all the drivers and data associated with either the sensors or the motors. These components were mentioned earlier when discussing the simulator.

Autonomous Process Logic Diagram

Once an initial design had been created, a logic diagram was required to evaluate the flow between each component. Figure 6 shows the autonomous process with regards to each software component.

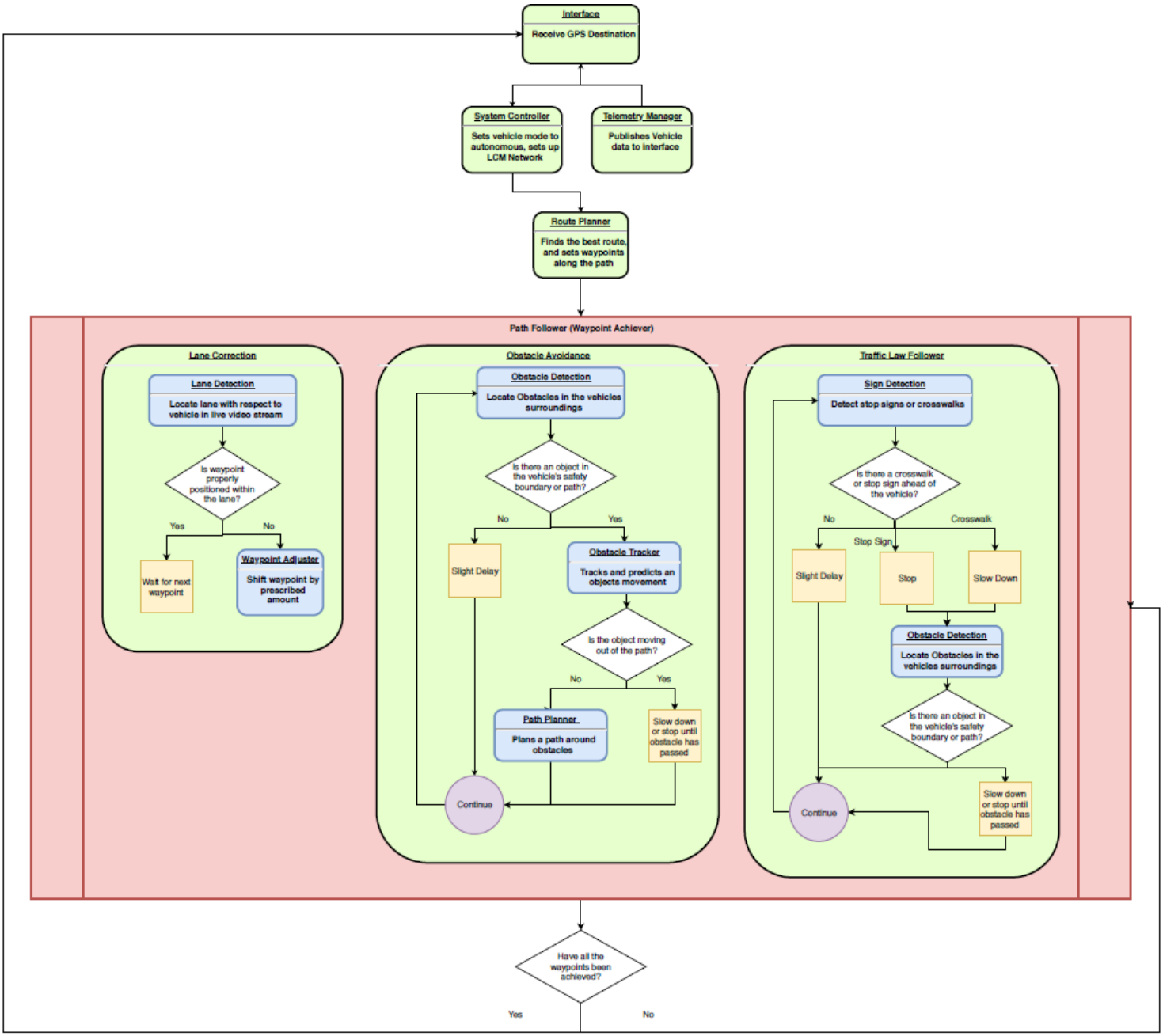

Figure 6 Autonomous Process Logic Diagram

A destination is sent to the route planner to plan the initial route the vehicle will take, similar to google maps providing directions. Waypoints are incrementally set within the planned route. This data is then sent to the Path Follower which tries to achieve each of these waypoints. The Path Follower runs three components in parallel; lane correction, obstacle avoidance and traffic law following. Once the way point has been achieved safely, the vehicle then moves to the next

waypoint in the route. If there are no more waypoints, the route has ended, and the vehicle stops. Going through this process has caused the initial design of the system to change to incorporate the new modules shown in the diagram.

Current Design

The current system architecture design abandons the traditionally hierarchal approach to programming. Instead it packages each component into a subsystem, to allow for more parallelism in a complex system. The system is comprised of three main subsystems; vision, positioning and controls. Within each of these subsystems are the components and algorithms the other students in the project are developing. The whole system is called the Integrated Autonomous Vehicle System (IAVS) and is shown in Figure 7.

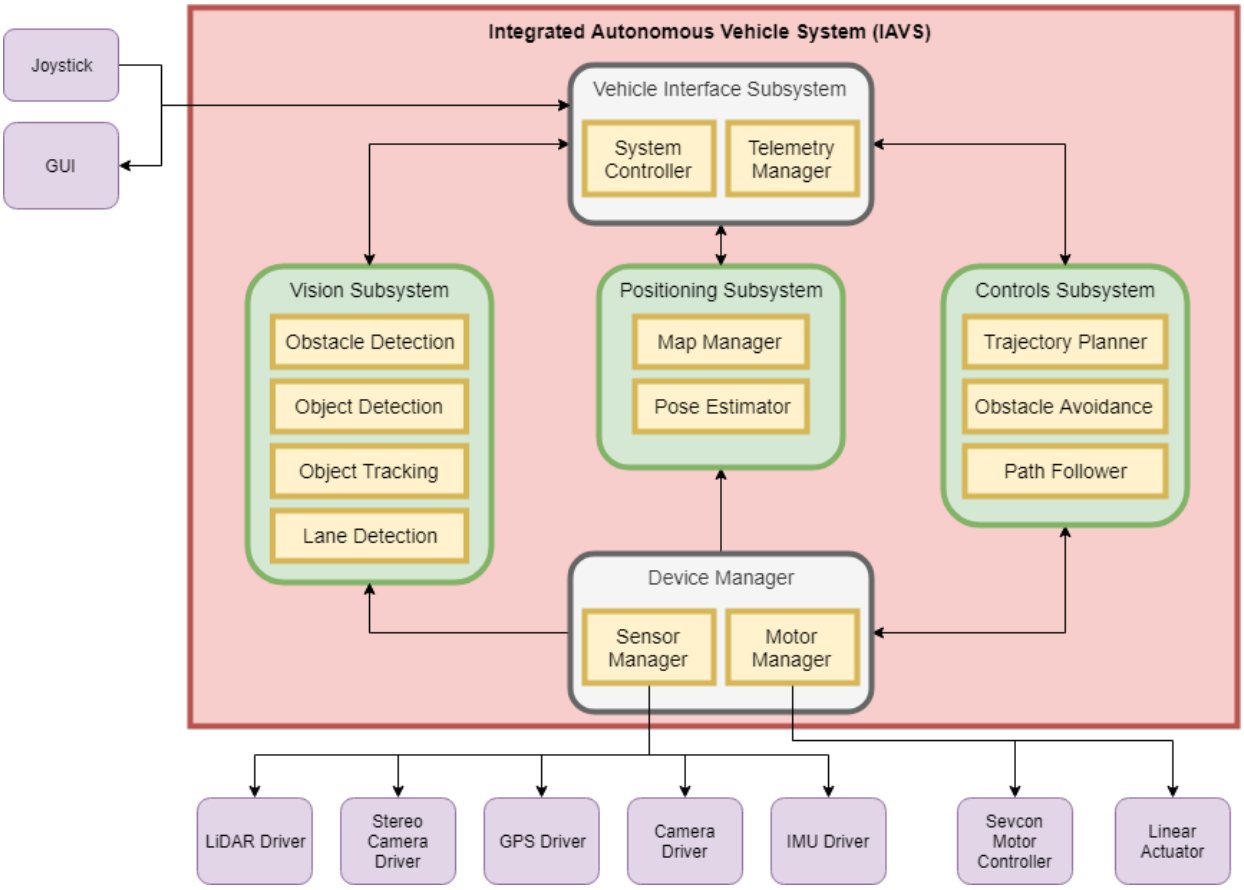

Figure 7 Integrated Autonomous Vehicle System

Everything within the red box has access to the LCM network. Therefore, messages can be quickly passed to and from each individual component. Everything outside of the red box is considered an external software component, which is an interface or driver for the system. The purpose of this is to simplify the system but also make it dynamic if more sensors or interfaces are added to it in the future. The manager components would just have to connect to a new driver in order to incorporate the new device.

For clarity purposes the Controls Manager component was renamed the Motor Manager and the Path Planner component was renamed the Trajectory Planner.

Control System

The control system highlights the logical flow between each software component associated with it. The flow chart is shown in Figure 8.

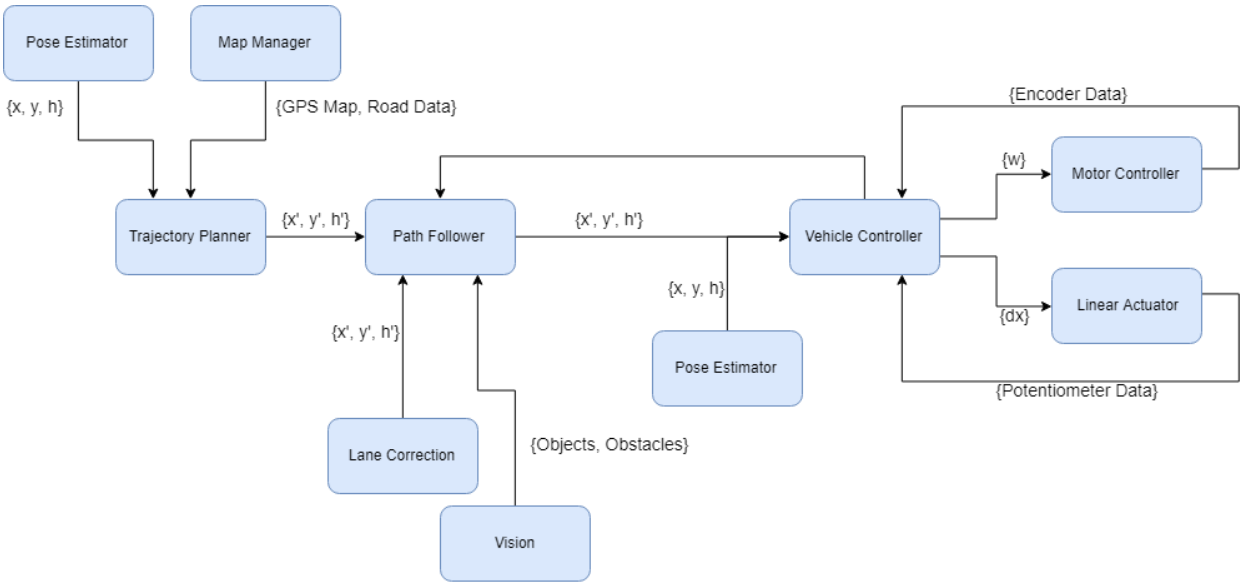

Figure 8 Control System Flow Chart

The Trajectory Planner component receives GPS map and road data from the Map Manager component which stores this data on file, so it is available off-line, similar to GPS navigators found in cars today. The Trajectory Planner uses this data along with its initial position, received from the Pose Estimator, to determine a path, represented as a list of GPS waypoints the vehicle must follow to get to its destination. This list is shown as {x', y', h'} which relate to the target easting and northing position and the target heading. This data is sent to the Path Follower component which fuses the data with a corrected position from the Lane Correction component and obstacles and objects, detected from the vision subsystem, that may be in the vehicles path.

From the Path Follower component, a new adjusted target position is outputted to the Vehicle Controller component which also receives an estimate of the current position represented as {x, y, h} for easting, northing and heading. The Vehicle Controller component implements a unique feedback controller developed by Ryan Kidney and is further discussed in his report. The component outputs an angular velocity demand and position change demand to the motor and linear actuator respectively. These devices respond with their current readings, in the form of encoder data and potentiometer data.

The Vehicle Controller than loops back to the path follower which repeats the process again for the following waypoint in the path.

Device Drivers

A device driver is a software program that controls a hardware device that is attached to a computer [4]. Within the IAV system this includes all the sensors, the motor controller, and the linear actuator.

Motor Controller

The motor controller used to actuate the ME1114 Motenergy Motor is a Sevcon Gen 4 Motor Controller [5] [6]. It is connected to the IAV computer, which is a Nvidia Drive AGX Computer, through a CAN-Bus line [7].

Controller Area Network (CAN)

Controller Area Network or CAN is a communication protocol which allows signals containing messages to be passed to multiple devices on the network [8].

The line is represented electrically as nodes connected by two 20-gauge AWG wires; CAN-high and CAN-low. The connections are terminated at either end by 120-ohm resistors, as shown in Figure 9.

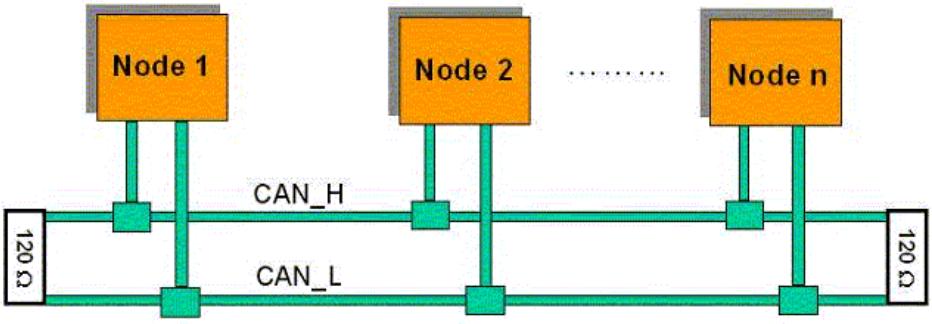

Figure 9 CAN-Bus Electrical Diagram [9]

The protocol allows for multiple nodes to be connected to the same bus, however in the IAV system there are only two; the computer and the motor controller.

In order to properly communicate and read messages over the CAN-Bus line the Nvidia computer required a transceiver to interpret the changes in voltage. The transceiver is connected to CAN0_IN, CAN0_OUT, 3.3V, and GND on the computer as shown in Figure 10 and Figure 11.

Figure 10 Nvidia Xavier Pin Out [10]

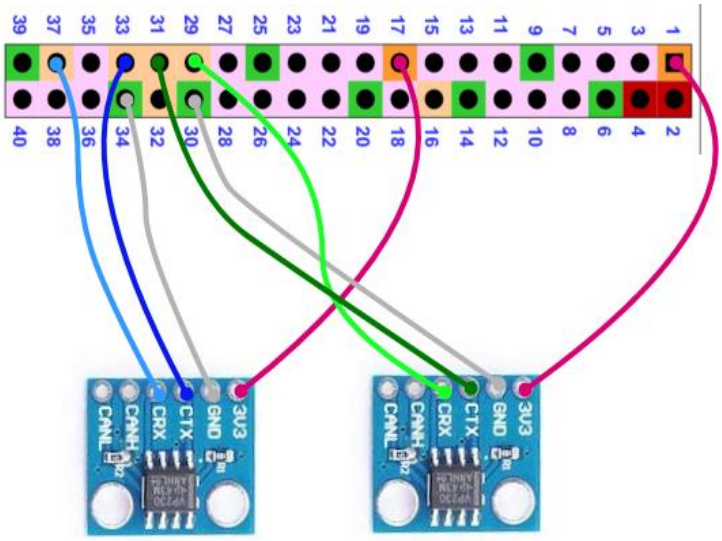

Figure 11 Transceiver Connection Diagram [10]

The above figure shows the connection for two transceivers, but currently for this project only one is required. In future years the second can be utilized for new CAN enabled devices.

CANopen

The Sevcon Gen 4 Motor Controller used in the IAV system utilizes a high-layer protocol called CANopen [6]. CANopen is a higher layer protocol based on CAN-Bus which allows for motion control and parameter adaptation of a device on the network [11]. It is a common software protocol used in robotics, medical devices and automotive devices. It was original designed for motion control of machines and motors. CANopen incorporates a range of new concepts to CAN. These include communication models and protocols, states, and an object dictionary.

The communication model used for the IAV system was designed to follow the Master/Slave approach (Figure 12). One node, the main computer, acts as the master and is able to send and request data from slaves, in this case the motor controller. There can be 0 to 127 slaves in the standard application.

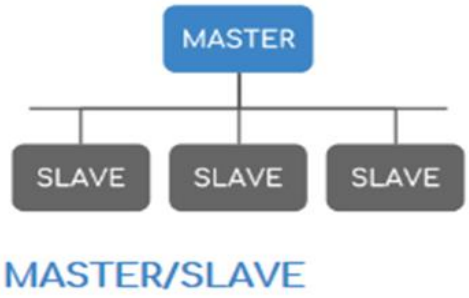

Figure 12 Master - Slave Tree Model [11]

Unfortunately, the motor controller purchased does not utilize the speed control loop integrated within it while in slave mode. This caused the motor to oscillate in an unstable manner when a command was sent to it. To remedy this, it was decided to leave the motor controller in master mode, as the motor functioned in a stable manner in this setting. This is discussed further later in this report.

The protocol also uses an object dictionary to store a nodes parameters and variables [11]. An object dictionary (OD) is represented as a hexadecimal index, which can be accessed through two message types; a Service Data Object (SDO) and a Process Data Object (PDO) [12] [13]. The object dictionary is a series of indexes with corresponding sub-indexes that have an associated value, as shown in Figure 13.

Index	SubIndex	Example Value	Description
1801h	00h	100	Highest Subindex
	01h	00000181h	PDO COB ID
	02h	0	
	03h	1000	Inhibit Time
	04h	Unused	Unused
	05h	0 (disabled)	Event timer
1A01h	00h	3	Number of Entries
	01h	Index 2001h, Subindex 00h	Mapped OD item 1
	02h	Index 2003h, Subindex 00h	Mapped OD item 2
	03h	Index 2005h, Subindex 00h	Mapped OD item 3

Figure 13 Example of a CANopen Object Dictionary [14]

The standard CANopen CAN message, also referred to as a frame, is composed of a COB-ID, data length indicator, and the data [11]. The structure of a CAN Frame is shown in Figure 14.

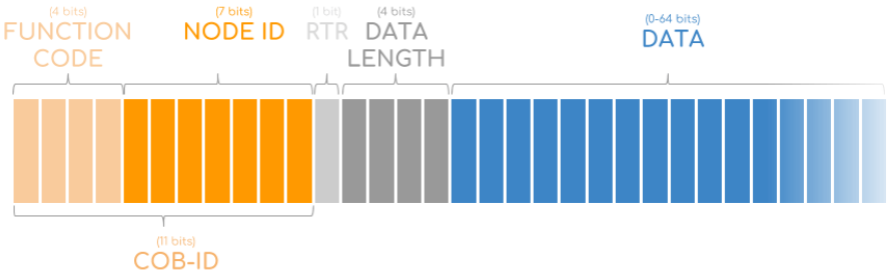

Figure 14 CANopen Frame Structure [11]

A COB-ID or Communication Object Identifier is split into two parts; a 4-bit function code and a 7-bit node ID. The function code relates to the function the master node would like to commit, while the node ID relates to the slave node the function is acting on. Figure 15 summarizes the different types of messages, also known as communication objects, CANopen supports. The figure also shows the corresponding COB-ID for that communication object.

	COMMUNICATION OBJECT	FUNCTION CODE (4 bit, bin)	NODE IDs (7 bit, bin)	COB-IDs (hex)	COB-IDs (dec)	#
1	NMT	0000	0000000	0	0	1
2	SYNC	0001	0000000	80	128	1
3	EMCY	0001	0000001-1111111	81 - FF	129 - 255	127
4	TIME	0010	0000000	100	256	1
5	Transmit PDO 1	0011	0000001-1111111	181 - 1FF	385 - 511	127
	Receive PDO 1	0100	0000001-1111111	201 - 27F	513 - 639	127
	Transmit PDO 2	0101	0000001-1111111	281 - 2FF	641 - 767	127
	Receive PDO 2	0110	0000001-1111111	301 - 37F	769 - 895	127
	Transmit PDO 3	0110	0000001-1111111	381 - 3FF	897 - 1023	127
	Receive PDO 3	1000	0000001-1111111	401 - 47F	1025 - 1151	127
	Transmit PDO 4	1001	0000001-1111111	481 - 4FF	1153 - 1279	127
	Receive PDO 4	1110	0000001-1111111	501 - 57F	1281 - 1407	127
6	Transmit SDO	1011	0000001-1111111	581 - 5FF	1409 - 1535	127
	Receive SDO	1100	0000001-1111111	601 - 67F	1537 - 1693	127
7	HEARTBEAT	1110	0000001-1111111	701 - 77F	1793 - 1919	127

Figure 15 CANopen Communication Objects [11]

An SDO message can access all the parameters in a node's object dictionary. There are two types of SDOs; a Transmit SDO (TSDO) and Receive SDO (RSDO). The associated COB-IDs for a TSDO and RSDO are 0x580 + Node ID and 0x600 + Node ID.

The Transmit SDO message is represented in Figure 16 and the Receive SDO message is represented in Figure 17.

Command byte	OD main-index	OD sub-index	Data (max. 4 bytes)

Figure 16 Transmit SDO Message Frame [12]

Command byte 0x60	OD main-index	OD sub-index	Empty (4 byte)

Figure 17 Receive SDO Message Frame [12]

The command byte shown in the above figures denotes the type of message being sent, Transmit or Receive, and the number of bytes of data contained within that message. This is summarized in Figure 18.

Command Code	Meaning	Command Code	Meaning
0x43	Read Dictionary Object reply, expedited, 4 bytes sent	0x23	Write Dictionary Object reply, expedited, 4 bytes sent
0x47	Read Dictionary Object reply, expedited, 3 bytes sent	0x27	Write Dictionary Object reply, expedited, 3 bytes sent
0x4B	Read Dictionary Object reply, expedited, 2 bytes sent	0x2B	Write Dictionary Object reply, expedited, 2 bytes sent
0x4F	Read Dictionary Object reply, expedited, 1 bytes sent	0x2F	Write Dictionary Object reply, expedited, 1 bytes sent

Figure 18 SDO Command Byte Summary [15]

Two additional command bytes can be received from a node, these are the acknowledge byte (0x60) and the error byte (0x80 + node ID) [12]. OD main-index and OD sub-index describe the object dictionary index and sub-index of the parameter the message relates to. Finally, the last section of an SDO is the data being sent or received. It has a maximum size of 4 bytes and is in a hexadecimal format.

A PDO message is a faster alternative to an SDO message and can send a maximum data length of 8 bytes [13]. However, a PDO message only has access to select variables in the object dictionary and must be chosen prior to operation. These select variables are mapped to an associated index and sub-index and must be the same on all the nodes the PDO message applies to. Similar to an SDO message, there are two types of PDO messages; Transmit PDO (TPDO) and Receive PDO (RPDO). 4 of each message (TPDO or RPDO) can be mapped for a single node and is summarized in Figure 15.

Multiple variables can be mapped to one PDO if the total size does not exceed 8 bytes. This is shown in Figure 19 along with the associated mapping of the receiving and transmitting nodes.

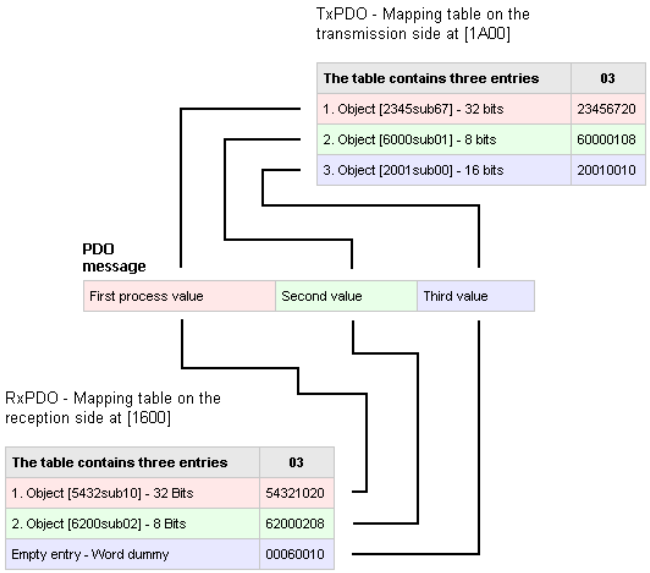

Figure 19 PDO Message and Associated Mapping [13]

The mapped values shown under 2nd column in the above figure relate to the index, sub-index, and number of bytes. For instance, in the RPDO mapping table, row 1, the mapped value is 54321020. 5432 pertains to the index, 10 pertains to the sub-index, and 20 is the hexadecimal representation of 32, for the number of bits that denote the size of the variable (First Process Value) being sent.

SocketCAN

All the different message types and functions mentioned were written in C++ using simple functions from a library called SocketCAN. SocketCAN is the official CAN API of the Linux kernel. SocketCAN offers the user a hardware independent socket-based API for CAN based communication and configuration [16]. The library offers functions to connect to the CAN-Bus port and send and receive messages from it. Using these functions, the capabilities of CANopen were incorporated to allow for communication from the IAV computer to the motor controller.

Motor Controller Issues

As mentioned earlier, it was decided that the motor controller will operate in master-mode, with the computer acting as the slave. This was due to the motor not operating in a stable manner while the motor controller was in slave mode as the control loop built inside of it was not used. This also affected the use of PDO messages being sent, as the motor controller was unable to map new PDO messages while in master mode. Fortunately, the motor does not need to receive speed demands at a high frequency and SDO messages were used in place. SDO messages were also used to receive the current angular velocity of the motor.

Although, control of the motor worked in this scenario, there was still an issue of initiating the motor controller and putting it into operational state. The motor controller programming software known as Sevcon DVT was able to place the motor controller into operational state, and then allow the motor to be controlled from the computer [6]. However, once the controller was turned off, it would reset to preoperational state on boot-up. To avoid using the DVT software every time, the computer must be able to recreate this initiating process.

In order to understand how the controller was forced into operational state, all 3 devices; the IAV computer, the motor controller, and the laptop containing the DVT software, were connected to the same CAN-Bus line. The IAV computer then listened in to the commands being sent from DVT to the motor controller. These commands were recorded and recreated using CANopen messages to place the controller into operational state. By doing this, the computer then had full control over the motor controller without the need for the DVT software and additional laptop.

Linear Actuator

After many design iterations, it was decided that a linear actuator would be the simplest method to control the steering of the IAV prototype. Linear actuators have some setbacks with regards to speed and control. Fortunately, a high speed, high force linear actuator was chosen. However, the issue of controlling the device still remained, as it is only able to extend, retract, or remain stationary. A control loop had to be designed to address this issue.

Linear Actuator Control Loop

To solve the issue of controlling the device, the actuator selected must have a sensor, such as a potentiometer, that can read the current position of it. Using this reading a feedback control loop was designed to allow for stable operation and control. The loop is described in Figure 20.

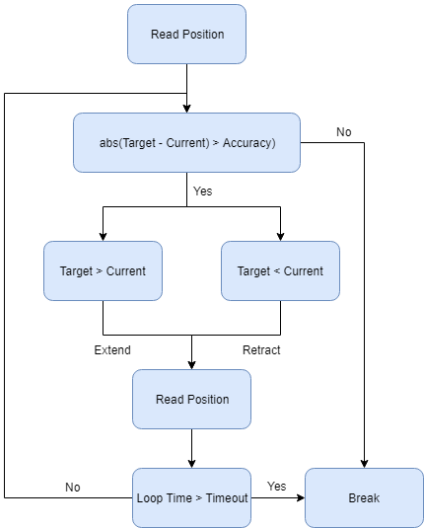

Figure 20 Linear Actuator Control Loop

Beginning from the top of the control loop, the current position of the actuator is read from the potentiometer. The potentiometer outputs the position in millivolts with 30 being fully retracted and 930 being fully extended. These values can then be used to determine the position of the actuator by linearly interpolating them with the true extension of the actuator in meters. Once converted the current position is compared to the target position of the actuator, as shown in the second block. An accuracy value had to be added to stabilize the device as the potentiometer reading varies by + or – 10 millivolts. Depending on the outcome of this block, the actuator either extends or retracts. The position is read again and checked against the target position and continues operation until it is at its desired position. A timeout condition was added, in case the actuator gets stuck and is unable to meet its goal. In this scenario, it will exit the loop and wait for a new target position.

Linear Actuator Communication

The linear actuator is driven by a motor driver. The motor driver receives commands from an Arduino Uno, which receives the potentiometer reading from the linear actuator. The control loop mentioned above, operates on the Arduino Uno. Although this system can control the actuator and drive it to a desired position, it still needs to be told when to do so. Thus, there must be communication from the IAV computer to the Arduino Uno.

The Arduino connects to the computer through USB and uses serial communication to send and receive messages. A device driver was written in C++ to allow the IAV computer to send a target position and receive a current position reading. On the Arduino end, a driver was written to receive a target position and send out the current actuator position. By creating these device drivers, the steering of the IAV prototype was then incorporated into the overall IAV system. The hierarchy of the steering system is shown in Figure 21.

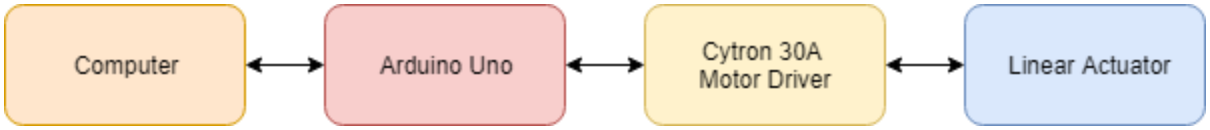

Figure 21 Linear Actuator System

Future Work

The next step, for future students is to begin developing the autonomy of the system. This should be done by integrating GPS waypoint following into the system. Once this has been done, object and obstacle detection should be integrated. Slowly these features will build upon each other to achieve higher autonomy. With regards to the topics highlighted in this report, more work also needs to be done.

Simulator

As mentioned earlier the simulator needs to be updated to better represent the real buggy. Ackermann steering needs to be added to properly represent the controls and motion of the vehicle. The friction coefficient needs to be adjusted to model a real environment. Additionally, the buggy CAD model needs to be updated to properly represent the redesigned vehicle.

System Network

The LCM Network is mostly complete. New messages can be made when needed in the future years. The software related to the network and the overall system is available to future years, for them to incorporate into their programs. Commenting of each function is still required and some classes need to be improved.

Software System Architecture

Parts of the software system still needs to be developed; this relies heavily on future years developing all the mentioned components. The Sensor Manager and Motor Manager components, as well as most of the device drivers have been completed. Following the designed system will ease development for future years.

Control System

As mentioned above, software components such as the Map Manager, Trajectory Planner, Path Follower, Lane Correction, Vision Subsystem, and Pose Estimator must be developed properly in order to function the vehicle in higher autonomy. The Vehicle controller has been partly completed but will need to be changed as the other components are integrated.

Device Drivers

The motor controller times out if left too long without a speed input other than 0. This should be addressed in future years. The linear actuator control is slightly delayed and can be approved upon further by adding the control loop to the IAV computer and removing the Arduino Uno from the system. To do this the IAV pins need to be adjusted and an analog to digital converter (ADC) is required to read the actuator position from the potentiometer.

IAV Computer

The IAV computer must be shutdown in a safe manner, by typing "sudo shutdown now" in a terminal or clicking shutdown through the graphical user interface. A proper shutdown circuit should be designed in the scenario where power is lost to the computer. This is further discussed in the Nvidia Driver Xavier Manual [7].

Conclusion

Although the vehicle didn't achieve autonomy this year, the system built will act as a very good foundation for future years to build upon. Currently, the system can read all of the data produced from the sensors as well as control and actuate the motor and linear actuator. The vehicle can also be controlled through a joystick.

This page was intentionally left blank

References

[1] "What is Software Integration," 2018. [Online]. Available: https://whatis.ciowhitepapersreview.com/definition/software-integration/. [Accessed 09 04 2019].

[2] "Webots," Cyberbotics, 2019. [Online]. Available: https://cyberbotics.com/. [Accessed 10 04 2019].

[3] "Lightweight Communications and Marshalling (LCM)," [Online]. Available: https://lcm-proj.github.io/. [Accessed 10 04 2019].

[4] M. Rouse, "device driver," Tech Target, 03 2019. [Online]. Available: https://searchenterprisedesktop.techtarget.com/definition/device-driver. [Accessed 09 04 2019].

[5] "ME1114," Motenergy, 2019. [Online]. Available: http://motenergy.com/me1114.html. [Accessed 10 04 2018].

[6] "Sevcon Gen 4," Sevcon, 2019. [Online]. Available: http://www.sevcon.com/products/low-voltage-controllers/gen4-dc/. [Accessed 10 04 2019].

[7] "Drive AGX," Nvidia, 2019. [Online]. Available: https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/. [Accessed 10 04 2019].

[8] "CAN Bus Explained," CSS Electronics , 2019. [Online]. Available: https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en. [Accessed 09 04 2019].

[9] "Automotive bus technologies," 24 10 2016. [Online]. Available: https://www.slideshare.net/RadwaTarek7/automotive-bus-technologies. [Accessed 10 04 2019].

[10] R. Nabati, "Enabling CAN on Nvidia Jetson Xavier Developer Kit," Medium, 08 11 2018. [Online]. Available: https://medium.com/@ramin.nabati/enabling-can-on-nvidia-jetson-xavier-developer-kit-aaaa3c4d99c9. [Accessed 10 04 2019].

[11] "CANOPEN EXPLAINED - A SIMPLE INTRO (2019)," CSS Electronics, 2019. [Online]. Available: https://www.csselectronics.com/screen/page/canopen-tutorial-simple-intro/language/en. [Accessed 10 04 2019].

[12] "Device Configuration via SDOs," CAN open Solutions , [Online]. Available: https://www.canopensolutions.com/english/about_canopen/device_configuration_canopen.shtml. [Accessed 10 04 2019].

[13] "Process data exchange with PDOs ("Process Data Objects")," CAN open Solutions, [Online]. Available: https://www.canopensolutions.com/english/about_canopen/pdo.shtml. [Accessed 10 04 2019].

[14] "The Basics of CANopen," National Instruments , 19 03 2019. [Online]. Available: http://www.ni.com/en-ca/innovations/white-papers/13/the-basics-of-canopen.html. [Accessed 10 04 2019].

[15] RobinC, "SDO – Service Data Objects – CanOpen," Byte Me, 07 11 2015. [Online]. Available: http://www.byteme.org.uk/canopenparent/canopen/sdo-service-data-objects-canopen/. [Accessed 10 04 2019].

[16] M. Kleine-Budde, "SocketCAN - The official CAN API of the Linux kernel," CAN CIA, 2012. [Online]. Available: https://www.can-cia.org/fileadmin/resources/documents/proceedings/2012_kleine-budde.pdf. [Accessed 10 04 2019].

Additional Work

A lot of additional work was put into the project this year to determine the progression and structure of the project. Additionally, tasks had to be assigned to each group member to continue development and keep everyone busy. In this sense, I took on a project management role within the project group.

Some of the additional tasks done the semester were:

- Wrote the GPS Driver
- Implemented the Joystick Driver into the system
- Picked and assisted in the selection of sensors and other components for the vehicle
    - Lidar, Cameras, GPS, Computer
    - Assisted Hadi with motor and motor controller selection
    - Assisted Tarek with linear actuator and motor driver selection
- Assisted a lot of other project members with their responsibilities when it came to software
    - Assisted Gareth with implementing Lidar cluster algorithm
    - Assisted Adonis with implementing lane detection algorithm
    - Assisted Ryan with EKF implementation and control system implementation
    - Assisted Jia with machine learning
    - Assisted Chris with ZED driver (just a little)
    - Assisted Shanshui with position fusion and GPS driver use
    - Assisted Jiacun with route planning
    - Assisted Tarek with the linear actuator Arduino code
    - Assisted Aslam with the joystick driver
- Assisted with the design of the overall system (hardware and software)
- Assigned tasks to each member of the project
- Organized meetings outside of class to plan the project
- Organized a meeting with the project professors and TA (fall semester)
- Wrote software tutorials for all the group members
- Software Lead for the IAV club
    - Organized tasks for the software volunteers
    - Met up every week to work on tasks
    - Organized a guest speaker
- Created and designed project website (iav.mae.carleton.ca)
- Assisted Aslam with sponsorship and fundraising
- Designed project logo (as seen in title page)
- Created Github for project
- Create Linux Virutal Machine (VM) for group members
- Assisted Matt with wiring the sensors and components to the vehicle, and the electrical diagram
- Worked with Hadi to wire up the motor controller and batteries
- Created Control loop for the motor controller
- Created utilities to test the motor controller and linear actuator

Appendix

All the software pertaining to the project can be found on my Github: https://github.com/LiamGritters/IAV

C++ was chosen as the programming language to develop all the IAV software as it is commonly used in robotics and has multiple libraries to support it. Additionally, the software was all developed on a Linux device running Ubuntu 16.04. The IAV computer also runs Linux, with Ubuntu 18.04.

If any questions arise from the software developed from this year, please do not hesitate to contact me at liam.gritters@gmail.com or through my github repository by creating an Issue.

Best of luck to future years!