Notre Dame Electrical Engineering
Senior Design: EE-41440-01

Variable **L**ocation **E**lectronic **T**ransportation
VaLET Group

William Chestnut, Alden Kane, Jonathan Peterson,
Perfect Mfashijwenimana, Sean Scannell

May 7th, 2020

Table of Contents

1 Introduction

Amazon made a huge splash when it announced it was investigating the use of drones to deliver packages directly to customers doorsteps a couple of years ago. Start-up Kiwibot also has a ground-based delivery solution, targeted primarily at delivering food on college campuses. Drones usurping the UPS man once seemed like something out of *The Jetsons*, but the autonomous vehicle industry has demonstrated through several partnerships (e.g Nuro and Dominos, Waymo and Walmart) that the prospect and viability of unmanned food, grocery, and package delivery is becoming more real every day.

Autonomous travel has several unique technical challenges, including inaccurate GPS navigation, sensing obstacles, negotiating these obstacles, and choosing the appropriate delivery point for a person or parcel.

Once inexpensive, effective, and accurate methods of doing so are implemented, a tremendous proliferation in automated deliveries will take place. The financial upside of automated deliveries is tremendous — there is no labor overhead, these vehicles will navigate with more efficiency and fewer accidents than their human counterparts, and can run all day without tiring (within their battery constraints).

Additionally, the rise of the COVID-19 pandemic has illustrated the need for automated delivery even further. The risk of infection from this virus is reliant on human to human interaction. With the attempt to reduce all unnecessary human interaction, the VaLET system could reduce the number of workers required in package and food delivery supply chains. This is an especially important group to reduce headcount in, given that these workers have heightened risks of becoming infected and transmitting the disease to others due to their heightened level of travel and human interactions in picking up and delivering their packages.

Proposed Solution

We aimed to develop and design a vehicle capable of implementing the challenge laid out in our problem description, ultimately safely delivering a package to a GPS-determined location. With redundant sensing, including GPS, LiDAR, optical, and Wi-Fi, VaLET would have the ability to navigate to an ultimate GPS location, sense objects that are obstructing its path there, pick up predetermined targets and navigate to those as well, and communicate with smart devices while doing so.

More specifically, VaLET will tackle the challenges laid out in the problem statement with the features of:

1. Object Detection/Avoidance

2. GPS Path Following
3. Wi-Fi Communication with Smart Devices
4. Delivery Verification

Ultimately, a sensor suite and basic controls schema will navigate VaLET to the desired drop off location.

Expectations and Results

Due to restrictions from the COVID-19 virus, we were not able to complete our project as initially planned. The University of Notre Dame's decision to suspend classes on March 18th was the correct one given the circumstances, but it unfortunately meant that we were unable to regroup as a team post Spring Break to finish the construction and begin testing our system. It is our belief that we would have completed the system on time and met the project deadlines. Therefore it is a great disappointment that we will not get the chance to do so.

As we approached spring break, we had demonstrated the functionality of almost all of our subsystems individually, but we were not able to integrate the subsystems together to demonstrate a working system. Our microcontroller unit and printed circuit board have been designed, in addition to our power control system. The other subsystems, including the LiDAR, vision, Raspberry Pi, and GPS, were pre-constructed with communication ports. All had been proven to be functional but the communications between these systems were not yet attempted because we had not yet reached that point in our timeline.

---

2 Detailed System Requirements

End Goals for the Project

At the conclusion of our project, our goal was to be able to set the VaLET system down at a random location in Debartolo Quad at the University of Notre Dame and have it travel to the door of Stinson Remick Hall on the west side of the quad. We would set the location of Stinson Remick on our website, and the Raspberry Pi would pull that information before departure. Then the system would read in its location from the GPS and compare that to the destination. With the respective distances to be traveled in longitude and latitude, the system would determine the orientation it needed to travel in. This information would be compared to the e-compass to determine how much the system needed to change orientation. From this, the VaLET system would travel in a straight line to the destination while avoiding any obstructions that were placed in its way. Upon arrival, VaLET would start scanning for a QR code. When presented a matching QR code, the system would confirm that the delivery had taken place, thus ending our demo.

Features to be Demonstrated

The VaLET delivery system will be used for variable-location delivery. It will be designed to address the accuracy of shortcomings of solely GPS-based delivery systems. With this system, we intend to demonstrate:

1. Dispatch Feature: Directs car to predetermined GPS location. In order to implement this feature, VaLET needs to be capable of receiving a Bluetooth or Wi-Fi signal from an external transmitter. This would simulate a delivery company inputting a unique address or objective for each delivery
2. Communication with a Smart Device: Relevant for dispatch feature, Bluetooth pinging
3. GPS Path Following: VaLET will follow GPS paths to its final objective
4. Object Detection/Avoidance: VaLET will detect and avoid obstructions to its path, while ultimately navigating closer to its objective
5. Target Identification: With the ultimate goal of a seamless delivery process, VaLET with use GPS data and computer-vision based target recognition to finish the delivery process. This could include the user showing a QR code to VaLET, to release whatever parcel is being carried and finish the delivery.

With all of these tools, the VaLET system will have the capability to travel from a changing initial location to a destination using GPS to determine the path while avoiding crashes with any object that could reasonably take up space on that path, such as a human, bike or other miscellaneous object.

---

3 Detailed Project Description

Operating Market: Food Delivery Industry

There are a variety of systems on which this project depends. The general business transaction is that of food delivery. In this industry, the consumer is able to place an order to a restaurant and have it delivered from that restaurant to the customer's location of choice. This incurs a fee on the customer for the added convenience. The majority of this fee goes to the delivery service. Therefore, the removal of that individual would add value to the restaurant as they could reduce their labor costs.

System Tools

MPLAB IDE

Our main programming tool that we used was the Microchip MPLAB IDE. This application was created by Microchip Technology for the purpose of programming the PIC32 microchips that they sell. It is common practice in our Senior Design Class to use these chips and we are taught in the first half of the academic year to do so. Therefore, it seemed a natural progression to use this as our microchip of choice for this project. Our specific Microchip model was the PIC32MX795F512H.

C

C is a coding language that was developed by Bell labs in the 1970's. It is a widely used programming language and is the language of choice for MPLAB IDE. Therefore, to use the MPLAB IDE and the PIC32, all code designed to run on this platform is written in C.

PIC32

PIC32 is a family of microchips created by Microchip Technologies. It is a widely used microchip and one that our professor is familiar with. It is the chip that we were trained with in the fall of 2019 when we were learning how to code microchips.

The PIC32MX has a series of configuration registers that set the clock, I/O, and other settings. These can be set in in the MPLAB environment using the #pragma config command followed by the register bits keyword and the desired value (reference Appendix 2 for PIC32MX Data Sheet). The configuration bit values VaLET uses are contained in the C header file configbits.h found in Appendix 13.

Raspberry Pi Model 3B+

The Raspberry Pi model 3B+ is a full-service computer that is approximately the size of a credit card. It has a diverse set of input/output ports and pins that allow for image processing and interfacing with external sensors. Our model is configured with Raspbian Buster OS, a Linux OS for desktop and embedded computing on the Raspberry Pi. It can complete almost any function that a desktop computer can, but has the form factor needed for an embedded project.

Python

Python is an interpreted programming language developed by Guido van Rossum and first released in 1991. It is high-level and has several useful libraries for image processing, computer vision, and interfacing with external sensors on the Raspberry Pi that makes it an ideal choice for vision functions and sensor fusion.

Description of Block Diagram

Figure 1 below shows the block diagram for our system. There needs to be power to start the VaLET system, so that is where the block diagram starts. The first part of the block diagram is

connecting the battery to the power management board. This is done by connecting each cell of the battery to the phoenix connector on the board, as well as connecting the power cables of the battery to the board. Also, the power wires for the battery have to connect to the drive system. After the battery is connected to the drive system as well as the power management board, the TI integrated circuit can then perform cell balancing, voltage, and current regulation on the battery. This ensures there is no damage to the battery during operation. Also, on the power management board there is a regulator that outputs 5 volts and 3 amps. This regulator's output is connected to two USB hubs so that the 5 volts can be applied to power the MCU board as well as the Raspberry Pi. The Raspberry Pi holds the LiDAR and the camera for the VaLET system. This Pi cam allows for object avoidance and can also detect QR codes for object delivery. The LiDAR is also used for object avoidance, but can detect objects at a further distance compared to the camera. The MCU board contains the logic for the motor and the drive system, and also uses the data from the e-compass to communicate with the drive system. Figure 2 shows the motor controller block diagram, which expands out the drive system in Figure 1. This system has an input from power to run the motors and UART connection to the MCU. The ecompass uses a magnetometer as well as an accelerometer, so it can provide data about the vehicle's acceleration and direction to the MCU. This allows the MCU to accurately control the VaLET system's direction. The MCU board also implements a regulator that lowers the voltage from the 5 volt USB connector output to 3.3 volts, which is what the microcontroller needs for proper functionality. Information from the Pi is communicated to the MCU by using a SPI interface. The MCU can then use the information from the Pi to form logic for the drive system. This allows the vehicle to actively detect objects and avoid collisions. A GPS for the system is also connected to the MCU. The GPS uses UART communication to send data to the MCU. Using the GPS, the VaLET system can receive pick-up and drop-off locations from the user. The location for drop-off is given through Wi-Fi by inputting the desired location through our website and then transmitting that data to the Raspberry Pi. Once the VaLET reaches the desired destination, the Pi cam can scan a QR code and notify the user that it has successfully reached the drop-off location.

System Block diagram

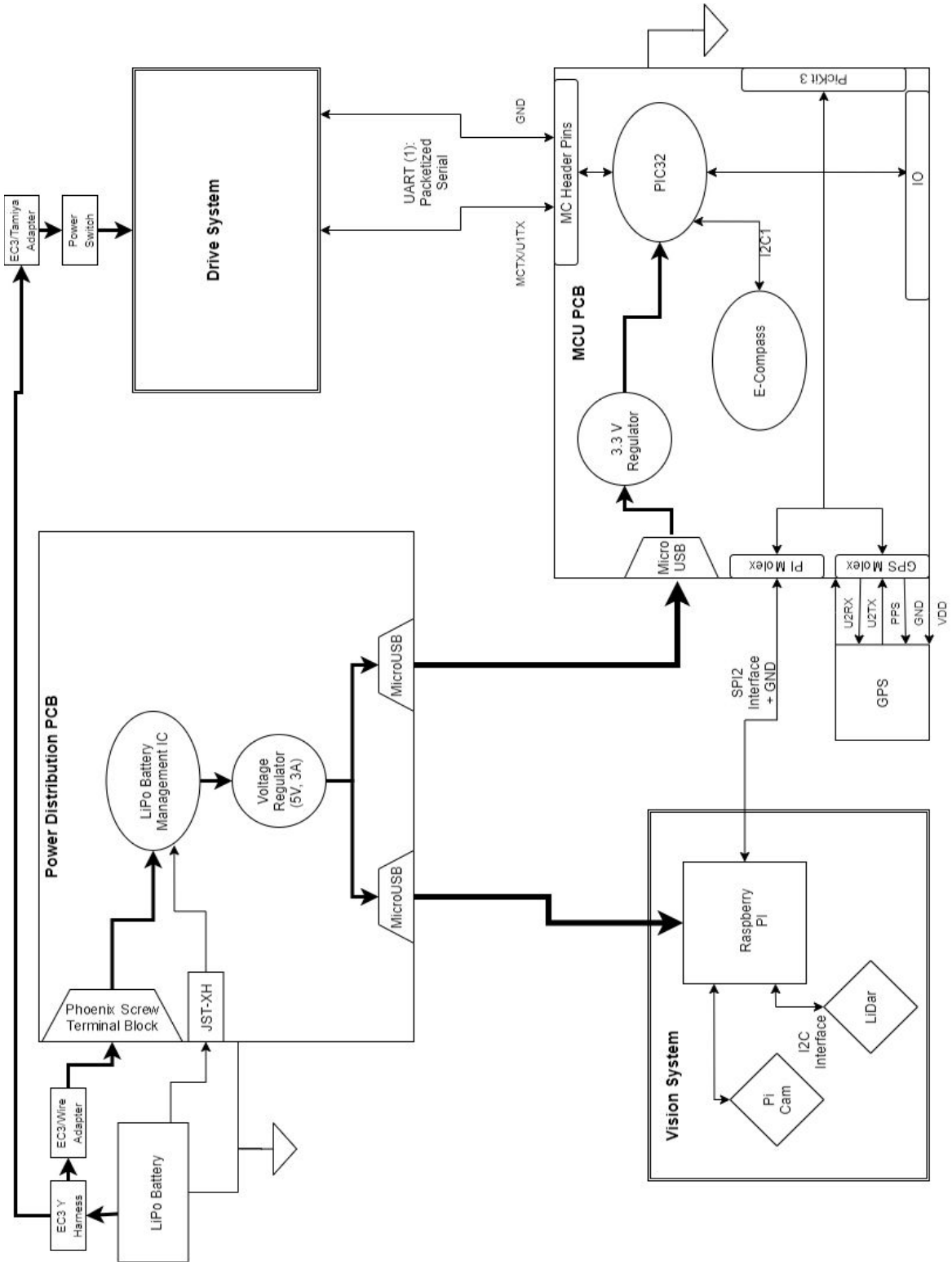


Figure 1. System block diagram

Drive System

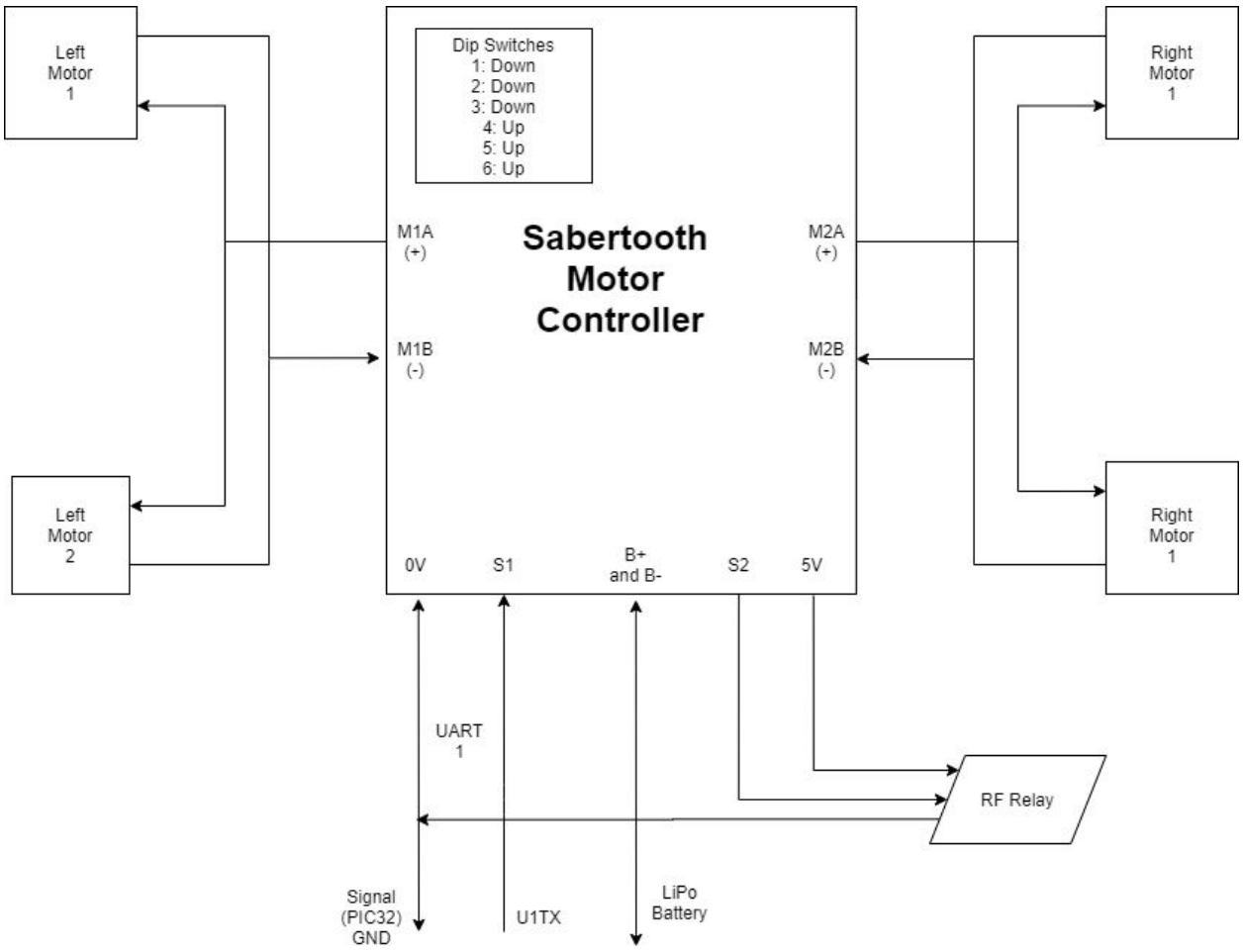


Figure 2. Drive system block diagram

3.1 E-Compass (Accelerometer/Magnetometer Combo)

Software Description

The Accelerometer/Magnetometer (NXP FXOS8700CQ) system is part of the e-compass system designed to provide orientation and motion data through error corrected magnetic and accelerometer readings. It uses the I2C interface to communicate with the PIC32MX where the PIC32MX is a master and the e-compass a slave. The software to interface with the e-compass is set up by creating crucial I2C functions. These functions are then utilized to construct a software driver for the e-compass.

I2C Interface

The e-compass I2C write function is contained in the EC_I2C. h header file. It must be included in any MPLAB project wishing to communicate with the e-compass. It is comprised of the following basic functions which utilize the PIC32MX's I2C1 interface and were custom written for VaLET (a full software listing can be found in Appendix 10):

Generic I2C Functions

- I2Cl_init:
  - Description: Initializes the I2C1 interface of the PIC32 for I2C communication
  - Inputs:
    - rate = desired interface clock speed (we used 400 khz)
- I2C_start:
  - Description: Initiates an I2C start condition on the bus
- I2C_restart:
  - Description: Initiates an I2C restart condition on the bus
- I2C_stop:
  - Description: Initiates I2C stop condition on the bus
- I2C_ack
  - Description: Initiates a master acknowledge
- I2C_write
  - Description: Transmits a byte of data on the I2C bus
  - Inputs:
    - data = byte of data to transmit
  - Output:
    - result = 0 if transmission acknowledge by slave, -1 if not acknowledged by slave (indicates error)
  - Theory of Operation: The master sends a start condition to the bus containing the slave address. The last bit of the R/W register is set to 0. The master sends the register address to write to upon receiving acknowledgement from the slave.
- I2C_read
  - Description: Reads a byte of data from the bus
  - Inputs:

        - ack = indicates if master should or should not acknowledge the read
    - Outputs:
        - data = byte of data received
    - Theory of Operation: Master sends a start condition to slave and the register to read from. The slave sends data to the master and the master terminates the operation by sending a stop condition.

NXP FXOS8700CQ Specific I2C Functions

- I2C_ec_read_byte
    - Description: Reads the contents of a specific e-compass register (one byte of data)
    - Inputs:
        - dev = device address (e-compass hard set to 0x1E, see MCU PCB section)
        - reg = address of register to be read (consult data sheet in Appendix 3)
    - Outputs:
        - data = value of register, -1 indicates error
    - Theory of Operation:The I2C read function transmits a start condition from the master and indicates the slave address followed by an acknowledgement. The R/W byte is set to 0 for a write and the master transmits the register to read from. This is followed by repeated start conditions from the master. This is followed by reading the data from the designated register. Consult the Appendix 10 and the E-compass data sheet for a complete description.
- I2C_ec_write_byte:
    - Description: writes a single byte to a specific e-compass register
    - Inputs:
        - dev = device address (e-compass hard set to 0x1E, see MCU PCB section)
        - reg = address of register to be written to (consult data sheet in Appendix 3)
    - Outputs: -1 if error, 0 if success
    - Theory of operation: Master transmits start condition to e-compass followed by the slave address with the last bit of R/W set to 0. The e-compass sends an ACK. The master transmits the address of the register to write to, and the e-compass sends ACK. The master transmits 8-bit data followed by an ACK from e-compass. The master transmits a stop to end the transmission.
- I2C_ec_read_multi:
    - Description: Reads the data contained in multiple consecutive (e-compass auto increments address if specific protocol is followed) e-compass registers (burst read)
    - Inputs:
        - dev = device address (e-compass hard set to 0x1E, see MCU PCB section)
        - reg = address of first register to be read from(consult data sheet in Appendix 3)
        - read_len = number of registers to read from
        - *data = pointer to array where read data will be stored
    - Outputs: -1 if error, 0 if success

- Theory of Operation: In the multi byte read function, the PIC32 requests a multi register read beginning at the e-compass register reg. The PIC32 stores the read data in the array pointed to by *data. After a successful read, the PIC32 acknowledges and increments the *data pointer. Meanwhile, the e-compass increments its own register pointer and transmits the data of the next register. Once again the PIC32 acknowledges, saves the data to the array, and increments the array address pointer. This occurs until the read_len is reached at which point, the PIC32 does not acknowledge the read (NACK) and terminates the operation. Refer to Appendix 3 and E-compass data sheet (Fig 6) for a bus diagram describing the operation.
- I2C_ec_write_multi
- Description: Writes data to multiple consecutive e-compass registers
- Inputs:
  - dev = device address (e-compass hard set to 0x1E, see MCU PCB section)
  - reg = address of first register to write to(consult data sheet in Appendix 3)
  - write_len = number of registers to write to
  - *data = pointer to array where write data is stored
  - Theory of Operation: In the multi write function, the e-compass memory array address pointer increments immediately after receiving a write command. The PIC32 then increments its own data array pointer and writes a new data byte to the e-compass. This continues until the specified number of bytes to be written (write_len) is reached. Refer to Appendix 3 and E-compass data sheet (Fig 6) for a bus diagram describing the operation.

E-Compass Software Driver

The e-compass software driver is used to read/write data to/from appropriate registers to accomplish magnetic measurements, and auto calibration.

All driver code is contained in the software file ec_driver.c. The driver is structured as followed.

Custom Data Types

A struct called SRAWDATA is constructed to hold the x, y, and z components of the magnetometer and accelerometer data.

Functions:

- accel_mag_config
  - Description: Configure's e-compass for operation in VaLET platform
  - Inputs: N/A
  - Outputs:
    - -1 = error
    - 0 = configuration complete

    - Theory of Operation: Configuration of FXOS8700CQ for 200Hz hybrid mode. The function first checks the WHOAMI register. Then assigns appropriate values to different registers of the accelerometer and magnetometer.
- ReadAccelMagnData
    - Description: Reads raw accelerometer and magnetometer readings from the e-compass, stores in structs
    - Inputs:
        - SRAWDAT *pAccelData = pointer to struct containing x,y,z components of accelerometer data
        - SRAWDATA *pMagnData = pointer to struct containing x,y,z components of magnetometer data
    - Outputs:
        - -1 = error
        - 0 = success
    - Theory of Operation: In a single operation, the software driver does a block read of the status byte, three 16 bit accelerometer channels, three 16-bit magnetometer channels-13 bytes in a single operation. To read the data pAccelData and pMagnData are defined as pointers for accelerometer and magnetometer data and are passed to the read data function. The multi read function inputs the slave address(0x1E), status(0x00), enable, and buffer. The obtained data from accelerometer and magnetometer is copied in 16 bit words. There is data for the x,y,z coordinates. They are all copied to 16 bit words. An error (I2C_ERROR) is returned if the operation can't be carried out. Otherwise, the return in (I2C_OK).

Driver

The driver is contained in the main function. It:

- initializes the I2C interface
- configures the e-compass using the accel_mag_config function
- instantiates structs for both the accelerometer and magnetometer data
- Creates pointers to the structs
- Continuously reads magnetometer and accelerometer data using the ReadAccelMagnData function

Further Work

The collected accelerometer and magnetometer data is used to compute magnetometer headings or acceleration vectors. These can be calculated from the x, y, and z axis readings using simple trigonometry. We looked into implementing a tilt-compensation algorithm which would adjust magnetometer readings (and headings) based on the angle of VaLET. However, VaLET's motion is rather jerky and it accelerates and changes direction frequently. The tilt compensation algorithm depends on accelerometer readings influenced only by gravity (no extra linear motion). This would have potentially resulted in even greater error in VaLET orientation data. Furthermore, the operating theory behind VaLET is that it continuously checks

its current heading against the desired one, and directs course accordingly. Its operating environment won't include a large amount of hills, or tilting for extended periods that would disrupt the orientation readings by a large amount for an extended period, such that VaLET's control logic couldnt compensate for slightly erent readings.

While the FXOS8700CQ includes an auto calibrate function for eliminating hard iron magnetic interference, it does not incorporate one for soft iron interference. Future work could be dedicated to implementing this feature.

VaLET also includes several pre-programmed interrupts. These include alerts for orientation deviation, flipping, and acceleration thresholds. They could potentially be useful to alert the control software if VaLET deviates from its course, crashes, or turns over.

3.2 GPS Subsystem

Description

The GPS system uses a MediaTek 3339, an all in one GPS system on a chip. This system was given to the team already built into an integrated circuit board with input and output using UART communication protocol. The pins were built in, but needed to be connected by a wire harness to the MCU PCB.

MediaTek 3339

This device was chosen because of its availability to the team. Professor Schafer was already in possession of a MediaTek 3339 in an integrated circuit board, so that was the majority of the selection process for the team. The MediaTek 3339 can provide data via UART, SPI, or I2C communication protocols. The integrated circuit board that was provided was designed for UART communication, which determined our communication protocol of choice for this subsystem. The data transmitted from the MediaTek 3339 is composed of NMEA formatted sentences which are transmitted as ASCII letters.

The schematic diagram of the integrated circuit is shown below:

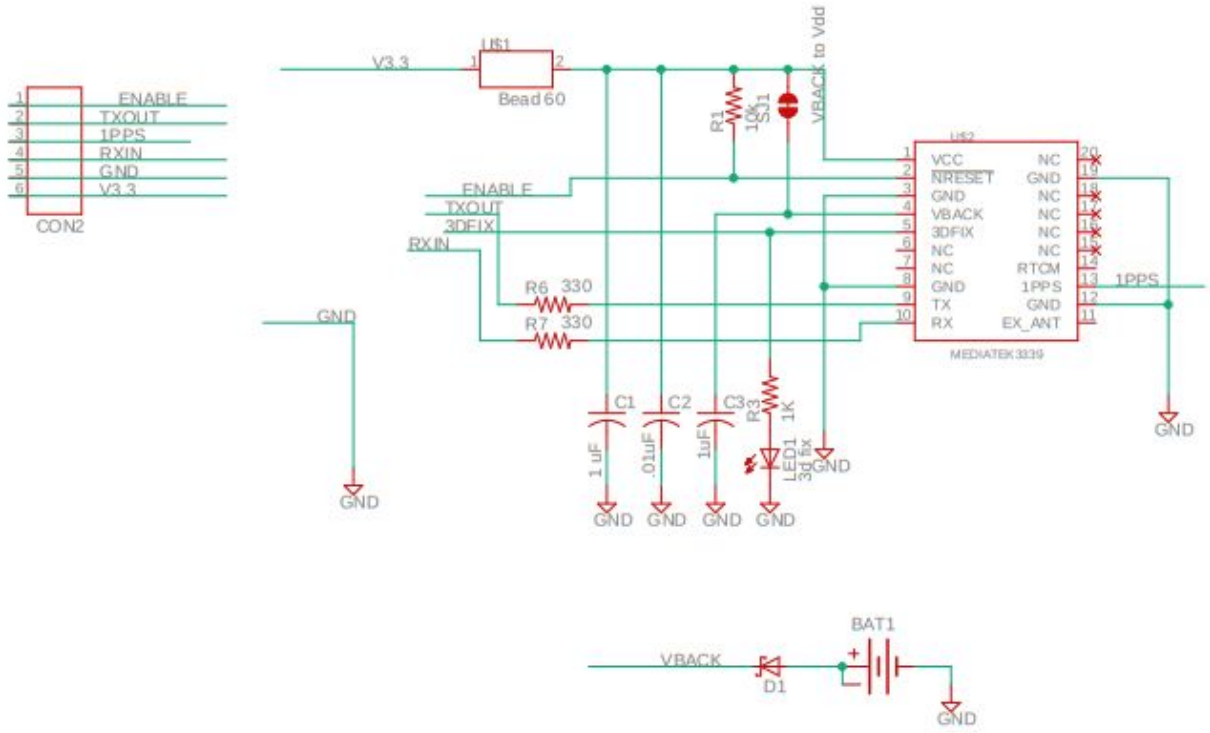


Figure 3. Schematic of the GPS integrated circuit with Mediatek 3339

Communication (Is this necessary if it's described in the PCB section)

The communication of the MediaTek 339 is transmitted using a UART connection. The schematic for this system is shown below.

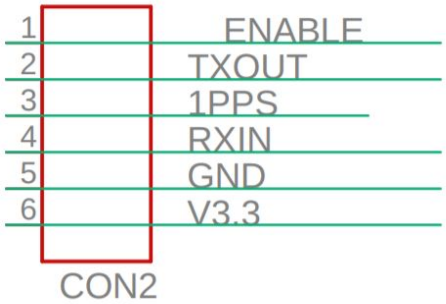


Figure 4. Schematic of UART Port

These different ports are connected as shown in the table below

Communication Pin	MediaTek 3339 Pin
Enable	2
TXOUT	9
1 PPS	13
RXIN	10
GND	GND
V 3.3	1

Table 1. GPS communication pins

NMEA Data

The National Marine Electronics Association is the most common form of transmitting GPS receiver communications. NMEA data sentences start with a "$" and then followed by the data type, which defines how the remainder of the sentence is interpreted. There are several different sentences in NMEA style data, but some of the basic and most used include GGA, which pinpoints the 3d location and accuracy, and GSA, which provides accuracy level and the number of satellites the antenna is receiving from. More information can be found in the Appendix 7.

Requirements

This subsystem needs to be able to determine the location of the VaLET system and share that data with the MCU to determine the system's next action. This includes determining the current location and determining relative distance and orientation of the destination.

Theory of Operation

This subsystem relies on the greater theory of global positioning systems. This is a satellite based radio navigation system owned and operated by the United States Government. This system includes a receiver and a fleet of man made satellites that maintain an orbit around planet earth. When four or more satellites have a direct line of site to the receiver, that receiver is able to determine its position. The satellites consistently broadcast a signal containing its time as determined by an onboard atomic clock. The receiver picks this up, and from the differences in time between the four signals due to the differences in distance traveled, is able to determine the position of the receiver.

For communication with the MCU, the system would use UART, which is laid out in detail in the MCU subsystem later in this paper. That section also includes pin assignments for the corresponding UART connector on the MCU PCB.

Software component

The software component of this subsystem would be coded in C using the MPLAB IDE. This is the program we would use to program the MCU PIC32, which is where the majority of the processing would take place in this system. We would have these functions stored in a function library focussed on overall functions, for all necessary subsystems.

In the final system, a powered mediatek 3339 continuously sends data in the form of NMEA data in ascii sentences. Once it reaches the end of its message, it begins again at the beginning. When the location was needed, the PIC32 would start reading the messages being broadcasted by the Meidatek, specifically looking for "$", which precedes the MediaTek 3339 message definition. After that, the Mediatek would check that the following five letters were GPGGA, which indicates that this message was location and time. The location is stored as numbers separated by commas. After waiting for the correct number of commas, the software would read in the longitude, wait for another comma, and then read latitude.

To calculate the distance to be traveled, the software should transfer the destination from strings to floating numbers. Then, subtraction of the longitudes gives the east/west distance to be traveled and subtraction of the latitudes gives the north/south distance to be traveled. Positive values for longitude and latitude would represent west and north respectively while negative values would correspond to east and south. Then, the calculation of distance to travel and orientation are two geometric calculations, one a pythagorean theorem and the other a tan calculation.

Command/Function Description

- Fetch Location
    - Description: Fetches the location of the program from the GPS module
        - Valid Data: no input required for this program
    - Example Command: float Fetch_Location()
- Determine Direction
    - Description: Determines the total direction to be traveled by the system
        - Valid Data: Final Destination (Floating Number), Current Destination (Floating Number)
    - Example Command:float Fetch_Distance(Final_Destination, Current_Destination)
- Determine Orientation
    - Description: Determines the required orientation to be traveled by the system
        - Valid Data: Final Destination (Floating Number), Current Destination (Floating Number)
    - Example Command: int Fetch_Orientation(Final_Destination, Current_Destination)

3.3 Vision System

Requirements

VaLET's vision system ought to detect obstacles immediately in front of the vehicle, and send a signal to the MCU with motor controls when needed. It also ought to have the function of detecting users via the presentation of QR codes at the end of a delivery, through the camera. The vision system should give the vehicle an understanding of its local surroundings, as well as be able to identify users through the presentation of a known QR code. The vision system uses a blend of sensors to do so: (1) a camera, and (2) a LiDAR.

Theory of Operation

This subsystem employs both computer vision and LiDAR (light detection and ranging) for operation.

Computer vision allows for understanding of the local environment around VaLET's delivery vehicle using a camera with RGB pixel information. It uses a camera for imaging, and acquires an RGB image via a CMOS sensor. Photons in the visible light spectrum (380-740 nm) are acquired. A series of operations can be performed on an image or video stream to derive understanding about the local scene — an image can be segmented based on color, shape of objects, motion of objects from one frame to the next, and so on. Using OpenCV, a countless number of computer vision algorithms can easily be implemented.

LiDAR, or light detection and ranging, allows for the measurement of distance from the sensor to a physical point based on the reflected signal of a pulsed laser. Based on the time it takes for a laser pulse to return to the sensor, and using the known speed of light, accurate distance measurements can be performed with a LiDAR sensor. The Garmin LiDAR Lite v4 uses a pulsed IR laser at wavelength = 940 nm.

Hardware Description

The following components are needed to construct the vision system:

- Raspberry Pi Model 3B+
    - Description: Runs scripts for processing of camera and LiDAR information and houses interfaces for camera and LiDAR. Contains full Linux kernel and Raspbian OS for processing.
    - Connections: Connects to USB webcam via one of the USB ports onboard the Raspberry Pi. Connects to Garmin LiDAR Lite V4 via I2C interface on Raspberry Pi GPIO pins
- Logitech c615 Webcam

- Description: Acquires RGB image when commanded to by the Raspberry Pi
- Connections: Connects to USB webcam via one of the USB ports onboard the Raspberry Pi

Figure 5. Logitech c615 USB Webcam

- Garmin LiDAR Lite V4
  - Description: Acquires distance to object in front of VaLET vehicle when commanded to do so by Raspberry Pi
  - Connections: Connects to Raspberry Pi via I2C interface. A detailed description of pin connections is given below, in Figure 6 and Table 2.

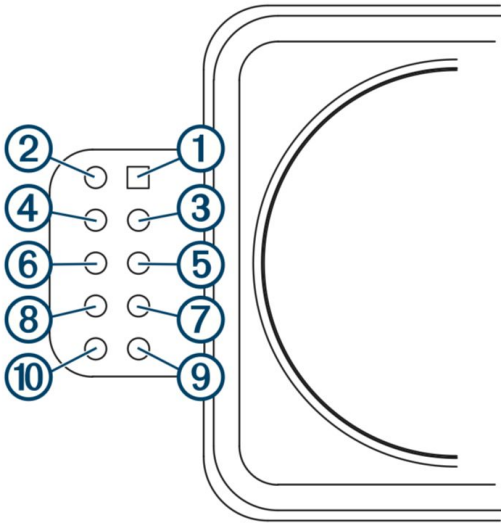


Figure 6. Garmin LiDAR Lite V4 Pinout

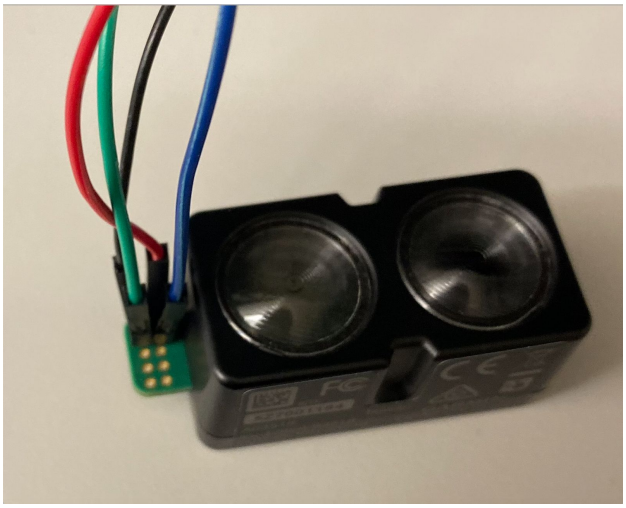


Figure 7. Garmin LiDAR Lite V4 w/ Power, I2C Connections to Raspberry Pi

LiDAR Lite V4 Pin #	Raspberry Pi GPIO Pin #	Pin Description	Notes
1	2	5V Power sourced from Raspberry Pi to LiDAR Lite V4	Red Wire
2	6	Ground LiDAR Lite V4 to Raspberry Pi	Black Wire
3	3	I2C Data	Blue Wire
4	5	I2C Clock	Green Wire

Table 2. Garmin LiDAR Lite to Raspberry Pi Pin Connections

Once the LiDAR Lite V4 is configured with the pin connections described above, the software functions described forthwith can be used to fetch a distance reading from the device.

Software Description

The vision system on the Raspberry Pi runs two primary functions: get_Camera_Headless() and get_LiDAR(). These two functions are used for obstruction detection and QR code recognition in the vision system. Each of these functions is initiated in their own file, then imported and called into system.py for more compact management of the system. The full code for each is provided in Appendix 15, but a description of each function is provided forthwith:

Functions

- get_Camera_Headless()
  - Description: Obstruction detection and QR code recognition from USB webcam attached to the Raspberry Pi
  - Arguments: None
  - Returns: obstruction, barcodeData
  - Theory of Operation: Python3 function that uses OpenCV to first initiate a webcam capture, then measure whether or not there is an obstruction in front of the camera. To do so, it uses the intuition that obstructions will present a uniform surface in front of the camera, i.e. there will be a low variance of all pixels on the screen if a tree is about 1 meter in front of the camera. It takes the standard deviation of the pixels in the three color channels of an image, and if the standard deviation drops below a set threshold, it sets an obstruction detection variable. get_Camera_Headless() also provides QR code detection using the pyzbar library. It scans for all barcodes in the frame using pyzbar.decode(img), then searches for QR code data within all decoded barcodes. It then returns the data encoded in a QR code
- get_LiDAR()
  - Description: Fetches the distance reading from the Garmin LiDAR Lite v4 sensor over the Raspberry Pi's I2C interface
  - Arguments: None
  - Returns: distance_LiDAR_1
  - Theory of Operation: Python3 function that uses smbus library to get data over I2C interface with LiDAR Lite v4. The function first writes to the ACQ register on the LiDAR Lite v4, then reads the status register. If the status indicates that the LiDAR Lite v4 has a readable distance, then it will read FULL_DELAY_LOW at address 0x10 on the Garmin LiDAR Lite v4. The read distance is then returned as distance_LiDAR 1. All of this is per the datasheet, found in Appendix 16.

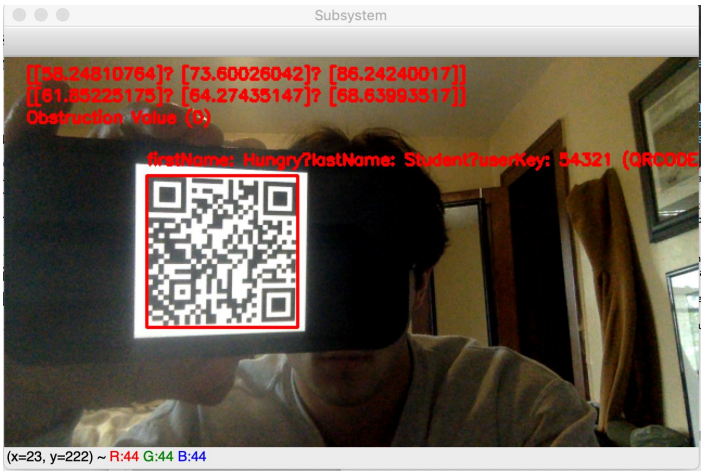


Figure 8. QR Code Detection, Obstruction Detection on a Video Steam

3.4 Wi-Fi

Requirements

VaLET's system ought to connect to Wi-Fi to get information about who and where to deliver packages to. Wi-Fi serves as a mock-LTE in this case, as Wi-Fi is easy to implement on the Raspberry Pi. Once the module connects to Wi-Fi, VaLET's vehicle can check a database for dispatches and get the GPS coordinates of its final destination.

Theory of Operation

The Wi-Fi subsystem uses the 2.4 GHz Wi-Fi connection on the Raspberry Pi Model 3b+. Using this connection, it downloads pertinent information about a user to deliver a parcel to.

Hardware Description

A Wi-Fi connection atop an OS that can scrape the web and parse .json files is needed for the operation of the W-iFi system. All of this is provided by the Raspberry Pi model 3b+.

Software Description

Information about a mock user is housed on VaLET's website at http://seniordesign.ee.nd.edu/2020/Design%20Teams/valet/users.json. This .json file, which contains information about only one delivery, is used in lieu of a database that deploying VaLET at scale would entail. This .json file has the following contents:


```
{
 "firstName": "Hungry",
 "lastName": "Student",
 "age": 21,
 "userKey": 54321,
 "streetAddress": "205 Stinson Remick Hall",
 "city": "Notre Dame",
 "state": "IN",
 "postalCode": "46556",
 "latitude": "4141.512 N",
 "longitude": "8614.141 W"
}
```

This .json file is downloaded and parsed by get_User_Info(), which takes a target url and parses the file for the first name, last name, latitude, longitude, and userKey of the target user. The first name and last name of this user is checked against a QR code that is presented at the end of operation, while the latitude and longitude of the user is transmitted to the MCU for GPS location.

A full listing of the code is given in Appendix 20.

Functions:

- get_User_Info()
    - Description: Fetches user data from a known web location, in lieu of a database of users for VaLET
    - Arguments: target
    - Returns: firstName, lastName, userKey, lat, long
    - Theory of Operation: Python3 script function that parses json information from a web location, taken as the function argument target. It returns the user's name, key, latitude, and longitude for use in other subsystems. It uses the urllib and requests Python libraries.

3.5 Power Management

Reason to use a Power Management System

Our small autonomous vehicle was going to need to deliver power to the drive system, the PIC32 MCU board, as well as the Raspberry Pi. With all of these different components requiring power, we decided it would be in our best interest to create a power management system so the right amount of power for each subsystem could be allocated properly.

Design Requirements for Power System

When designing the power management system, we needed to take in account what components we were going to use, what voltage those components need, and what current those components would need. After coming to the conclusion that we would need 5 volts for the Microchip PIC32MX795F512H as well as the Raspberry Pi, as well as 12 volts for the motor, we decided that a 15 volt battery would suffice if we put all of the parts drawing voltage in parallel. Since we wanted to put the devices that were drawing voltage in parallel with the battery, we knew that we needed a battery that could safely output the desired current that we would need to power the motor, the PIC, as well as the PI.

What Battery to Use and Proper Connections

We chose to purchase the Gens ace 3300mAh 14.8V 45C 4S1P Lipo Battery Pack with EC3. LiPo batteries tend to be small and light, which was great for our system because we did not want any additional weight. Also, the discharge rate for Lithium batteries is high, which allows us to obtain the current that we need. EC3 bullet-type connector allows the battery to connect easily to the power system on the vehicle as well as the charging dock for the lithium

battery. We knew that this would give us the voltage we needed, as well as the current we needed for sustained lengths of time. After deciding on the battery to use, we knew that we needed some type of system that would allow cell balancing of the battery, as well as over discharge protection. This way the cells of the battery would not damage themselves. We also needed to regulate the current and voltage of the battery.

Battery Protector Integrated Circuit

The regulation mentioned in the previous section is why we decided to implement the Texas Instruments BQ7791500PWR. This device would allow us to regulate the current and voltage, as well as provide cell balancing. The Sabertooth Dual 12A 6V-24V Regenerative Motor Driver that we were using for a motor controller has a built in safety mechanism for lithium batteries. The Sabertooth motor controller shuts down once the battery cells fall below 3 volts. Once we selected the Texas Instruments BQ7791500PWR IC that would allow us to provide cell balancing, as well as voltage and current regulation, we then needed to choose a product that would regulate the voltage that was going into the PIC as well as the Pi.

Regulator for the PIC32 and the Pi

We knew that we needed to deliver 5 volts to the PIC and the Pi, so we decided upon a regulator that would regulate the voltage to 5 volts. The part that we chose to regulate the Pi and the PIC was the MIC29300-5.0WU-TR. This device allowed for a maximum input of 30V and an output of 5V and an output current of 3A. The MIC29300-5.0WU-TR delivers the right amount of current and voltage needed to power the Pi and the MCU board. The schematic drawing of the power management layout as well as the board design is given below.

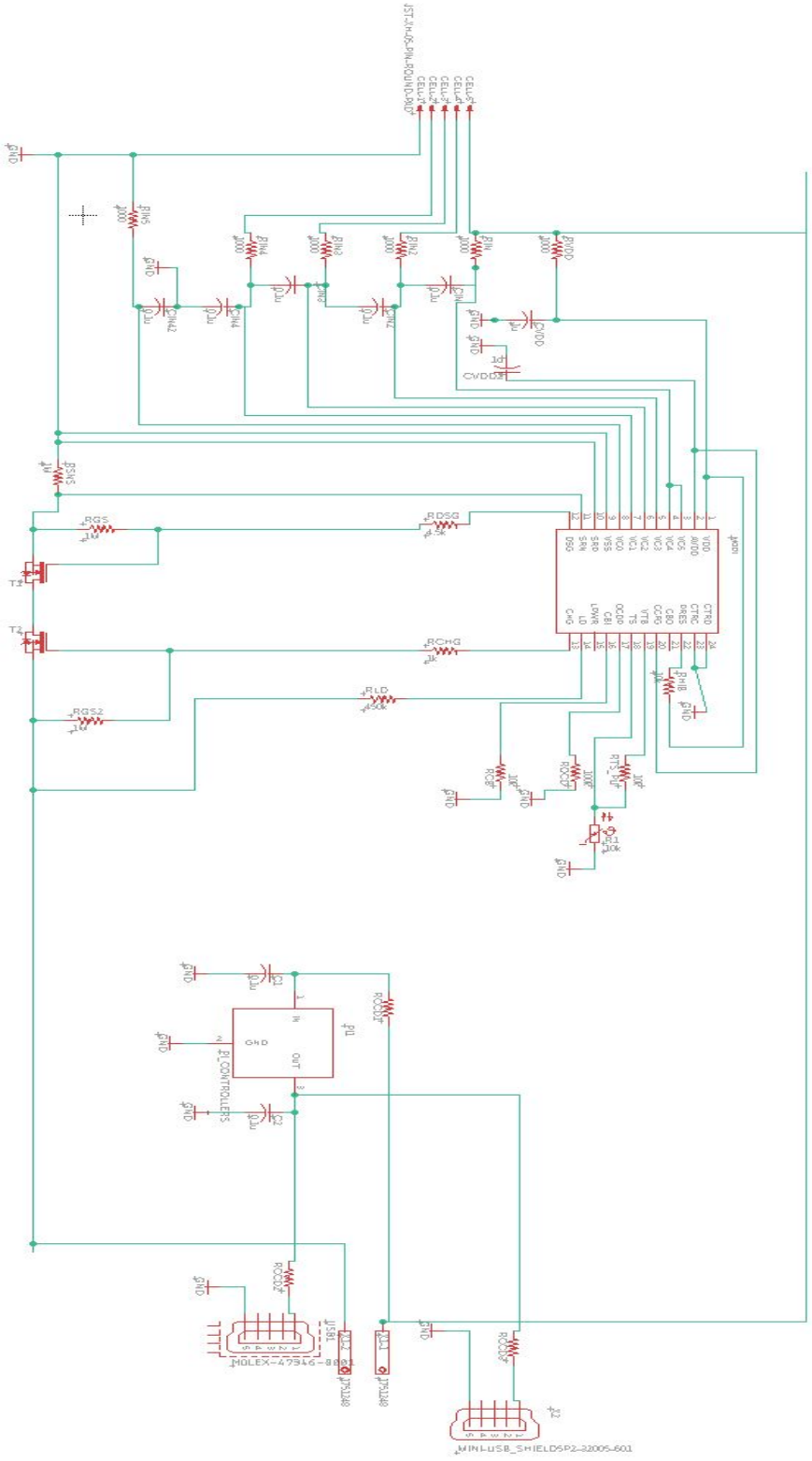


Figure 9. Schematic of power management subsystem

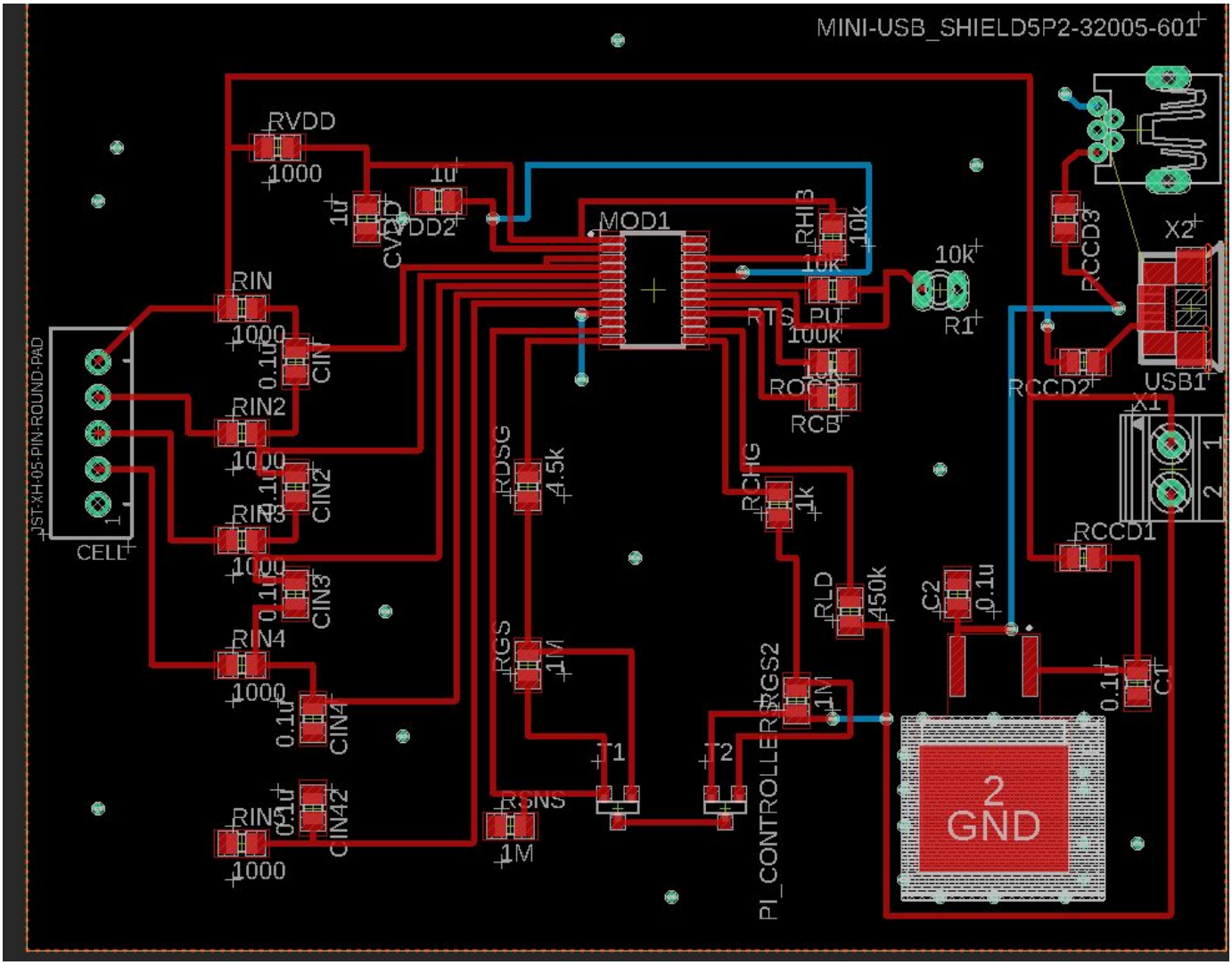


Figure 12. Board design of power management subsystem

name	type	value
RVDD	resistor	1000
RIN	resistor	1000
RIN2	resistor	1000
RIN3	resistor	1000
RIN4	resistor	1000

RIN5	resistor	1000
CVDD	capacitor	1 uF
CVDD2	capacitor	1 uF
CIN	capacitor	0.1 uF
CIN2	capacitor	0.1 uF
CIN3	capacitor	0.1 uF
CIN4	capacitor	0.1 uF
CIN42	capacitor	0.1 uF
RSNS	resistor	1 M
RGS	resistor	1 M
RDGS	resistor	4.5 K
RGS2	resistor	1 M
RCHG	resistor	1 K
RLD	resistor	450 K
RTS_PU	resistor	10 K
ROCD	resistor	100 K
RCB	resistor	10 K
R1	resistor	10 k
RHIB	resistor	10 K
C1	capacitor	0.1 uF
C2	capacitor	0.1 Uf
RCCD3	resistor	0.1
RCCD4	resistor	0.1

Table 3. Power management board details

Also included within the schematic is the as well as the board is two SOT-23 footprint MOSFET. These MOSFETs were needed to make the TI BQ7791500PWR IC function properly. The values of the resistors and capacitors were obtained from the data sheets for the TI BQ7791500PWR IC and the MIC29300-5.0WU-TR. When designing the board, we decided to use a large width for the wires because we needed to allow the larger current from the battery to pass through the wires. Also, we decided to make the copper pour over the MIC29300-5.0WU-TR regulator open so that it could allow for better heat dissipation. Each cell of the battery needs to be connected to the board, so we implemented a phoenix connector to connect each cell of the battery to the board. The wires that extend from the battery that connect to each cell of the battery easily connect to a phoenix connector. We also implemented a phoenix connector on the right side of the board to allow for the power from the battery to be connected to the board. We decided that a phoenix connect would be the best way to connect the power wires from the battery to the board due to the phoenix connectors simple design. All you have to do is put the wires into each of the allotted holes and it connects to the board.

3.6 Motor/Drive

Hardware

Rover Kit

Early on in the design and project discernment process, we decided to rely on a pre-built robot rover chassis as the basis for our design. This decision was made to ensure we had a sound mechanical platform and could devote the majority of our engineering efforts to the complex electrical and software systems VaLET requires to operate.

Several kits were considered including tracked, 6 wheel, and wheel designs. The greatest priorities were cost, reliability, and flexibility to accommodate the unknowns of our stated problem. Ultimately, the options were narrowed down to the Lynxmotion A4WD1 v2 Robot and the ServoCity Scout™. Both kits included a 4 wheel chassis and 4 12VDC motors and offered a straightford design that was customizable with different mounting points and screw on panels. Supplied motors were of the reliable and easy to control brushed DC type and also roughly comparable. Ultimately, the Lynxmotion A4WD1 v2 Robot was chosen for larger internal cargo bay, greater online product support, and advertised support for a recommended motor controller and 4 wheel drive from a two motor controller.

Chassis and Wheels

Figure 13. Image of Lynxmotion chassis

The chassis came in pieces and required assembly. We followed the detailed instructions found in Appendix 4. Looking at Figure 13, the central enclosed portion of the chassis contains space for the battery, motor controller, and processing electronics. Several holes prefabricated in the lexicon panels provide locations for additional and external devices such as the camera and GPS (to facilitate signal integrity) to be mounted and for wires to be routed through. A summary of the chassis specifications can be found in Table 4 below.

Platform Specifications

Parameter	Measurement
Overall Length	12.00”
Overall Width	13.50”
Tire Height	4.75”

Chassis Length	9.75”
Chassis Width	8.00”
Chassis Height	4.00”
Ground Clearance	1.63”
Weight	4lbs 6 oz.
Max Speed	36” per second

Table 4. Lynxmotion chassis details

Motors

The Lynxmotion A4WD1 v2 Robot comes with 4 12VDC 200RPM motors. They are designed and built by Lynxmotion and custom sized and specced for this particular combination of wheels and chassis. The motors include mounting points for encoders. However, we decided against equipping VaLET with them as the increased cost and complexity outweighed the potential benefit (precision control is not required). The numerous other sensing systems VaLET employed also made them slightly redundant as other systems can be utilized for feedback. A summary of their specifications is outlined in Table 5. below, while a full description and component diagram can be found in the Data Sheet (LINK).

Specifications:

Rated Voltage	12 VDC
Voltage Operating Range	6-12 VDC
Rated Load @ 12VDC	0.78 kg-cm
No Load Speed @ 12VDC	200 RPM +/- 10%
Speed @ Rate Load	163 RPM +/- 10%
No Load Current @ 12 VDC	< 115 mA
Current at Rated Load	< 285 mA
Stall Current	1.5 A / 83 oz-in

Table 5. Lynxmotion motor details

Motor Controller

Figure 14. Image of Sabertooth motor controller

The Sabertooth 2x12 Regenerative Motor Driver by Dimension Engineering is the heart of the drive system. A brushed DC motor includes commutators to mechanically switch the polarity of the input voltage, thus motor control is reduced to managing the speed, torque, and direction of the motor. The motor controller is responsible for scaling the input voltage (which is proportional to the motor speed) usually through PWM modulation, while sourcing the input current required for sufficient torque. It also must monitor the current input in order to not overdrive and damage the motor. Furthermore, it must reverse the polarity of the input voltage (without damaging the motors) when reversed movement is required. We investigated the idea of using a PIC32MK to create an integrated and custom motor control/microcontroller solution. These devices come equipped with PWM and QEI hardware blocks and built in support for motor control applications. However, they still require significant software and hardware overhead, including an H-Bridge motor driver and current sensing input, for bidirectional brushed DC operations. Which, when coupled with the fact that designing a motor controller was outside the purview of our stated project goals, meant we turned towards market solutions. The Sabertooth 2x12 was recommended by Lynxmotion for use with our robot chassis/motors and it came highly recommended by the Notre Dame Robotic Football Team. They have successfully worked with sabertooth motor drivers for many years now.

The Sabertooth 2x12 is a two channel motor controller with support for 12V, 12A continuous, and 25A peak per channel (well above our motors 1.5A stall current). To enable four motor operation, the left robots left side motors are wired in parallel to channel 1, and the right side are wired in parallel to channel two. The complete controller/motor configuration is diagrammed in Figure 2. We initially had concerns this configuration may result in unequal motor speeds on each side, however, Lynxmotion support for the configuration and eventual testing eased our concerns.

Communication to the motor controller is provided by a single asynchronous (UART) transmit channel (and Ground connection) in our application. The device offers support for 4 different communication standards including Analog, R/C, Simplified Serial, and Packetized Serial. We opted to utilize the Packetized Serial mode. It uses TTL level RS-232 serial data to set speed and

direction control. While the controller is a 5V TTL level device, it employs the same voltage cutoffs as the 3.3V level logic of the microcontroller ensuring transmission compatibility without a level shifter. The communication modes are selected via the dip switches located on the microcontroller. Dip switches 1 and 2 select the communication interface and placing them in the down position enables the packetized serial mode.

When operating in serial mode, dip switches 4, 5, and 6 set the address of the device. All three switches in the up position correspond to an address of decimal 128 which is the address used by VaLET.

Dip switch 3 enables or disables lithium cutoff protection. The Sabertooth 2x12 has built in hardware support for lithium batteries to ensure they are not over drawn and damaged during operation. The device automatically detects the number of cells in use and cuts off the power if a minimum voltage level is reached. As VaLET uses a LiPo battery, this ensures the battery is protected from over discharge by the drive system and the high currents of this system don't have to be drawn through the Power Supply PCB.

The final important component of the Sabertooth 2x12 is its support for emergency shutoff when operated in packetized serial mode. The S2 (Serial transmit) input can be configured as an active-low emergency stop. Considering VaLET is an autonomous prototype, we felt very strongly that a remote emergency shutoff was a must have feature to ensure safe operation. Though we never got the opportunity to implement it, a simple RF relay (Proposed Model: eMylo DC 12V Relay Switch) can be wired to bring the S2 input to ground upon activation, remotely shutting off the VaLET drive system.

A full description of the Sabertooth 2x12 Regenerative Motor Controller and its unutilized features can be found within its data sheet.

Software

Theory of Operation

VaLET utilizes the Sabertooth 2x12's Packetized Serial Communication mode. In this mode, commands are sent to the motor controller via UART transmit in 4 byte bursts. The packets consist of an address byte, a command byte, a data byte, and a seven bit checksum.

- Address: Motor Controller Address as set by dip switches
- Command: Unique number (decimal 0-17) corresponding to the drive or setting operation desired
- Data: Speed of motor if issuing a drive command or value of setting if issuing setting command.
- Checksum: Packets are terminated with a checksum to ensure data integrity. It must be correct or the command will not be acted upon. The checksum is the summation of all 3 previous 8 bit unsigned integers ANDed with the mask 0b01111111.

As far as drive commands, the Sabertooth 2x12 includes support for both independent (one side of motors at a time) and mixed mode commands (both motors/sides simultaneously). Commands cannot be mixed as switching between commands will result in vehicle stoppage until new data is received for both motors. VaLET was designed to use the mixed mode commands (however the function library includes support for independent commands as well). When using the mixed mode commands, the Sabertooth 2x12 requires both turn and drive data

before it will begin to move for the first time. Once data has been sent for both, turn and drive commands can be updated independently. For example, the function DriveForwardMixed must be transmitted together with TurnLeftMixed (even if in one the speed is set to 0 indicating it is not actually acted upon) when starting from stop.

Not all of the Sabertooth 2x12's supported commands are utilized by VaLET and supported by it's software. For example, there is a command for setting the baud rate of the motor controller (decimal 15). However, VaLET just uses the default baud rate of 9600. A full list of commands, used and unused, can be found in the Sabertooth 2x12 data sheet in Appendix 1.

UART Interface

The Motor Controller utilizes the PIC32 UART 1 hardware block. Transmit occurs via pin 51 labeled MCTX/U1TX in Figure 19/20, and is received via the Sabertooth input S1. Support for the UART interface is contained in the header file UART.h found in Appendix 8 It is written in C code specifically for the PIC32MX795F12H. This header file must be included in any software utilizing motor control functions. UART.h contains three basic functions:

- serial_init
  - Description: Initializes PIC32 UART 1 interface
  - Inputs:
    - rate = Desired baud rate
    - pb_clk = peripheral bus clock frequency

```
void serial_init(unsigned long rate, unsigned long pb_clk) {
  U1MODEbits.ON = 1;
  U1MODEbits.BRGH = 1;
  U1STAbits.URXEN = 1;
  U1STAbits.UTXEN = 1;

  int brg = ((pb_clk/(4*rate))-1);
  U1BRG = brg;
}
```

- getu (not necessary for motor control operation as no recieve required)
  - Description: retrieves data byte from receive buffer

```
#define u1_data_avail U1STAbits.URXDA
unsigned char getu(void) {
  while (!u1_data_avail); //wait till data ready

  char dat;
  dat = U1RXREG;
  return (dat);
}
```

- putu
  - Description: Places data byte in transmit buffer, waits till it is empty before doing so

- Inputs:
  - dat = data byte to be transmitted

```
#define u1_buff_full U1STAbits.UTXBF
void putu(char dat) {
        while (u1_buff_full); // wait if buffer full
        U1TXREG = dat;
}
```

This file also contains functions and supports for using the UART 2 interface utilized by the GPS.

Motor Control Function Library

The motor control function library is written in C code and contained in the MPLAB function library MC_packserial.a found in Appendix 9. Functions are declared in the header file MC_packserial.h also found in Appendix 9 MC_packserial.a must be added to the library files folder of any project that uses the motor control functions. Meanwhile, MC_packserial.h must be added to the Header files folder and included in any files that utilize the motor control functions. Calling a function results in the corresponding command being sent to the motor controller and movement being initiated.

All motor control functions follow the function outline below.

```
void DriveFunction(char address, char speed)
{
        Putc(address);
        Putc(command);
        Putc(speed);
        Putc((address + command + speed) & 127);
}
```

The supported functions include the following individual drive commands:

- DriveBackward1 (Left Motors) - Command 1
- DriveBackward2 (Right Motors) - Command 5
- DriveForward1 (Left Motors) - Command 0
- DriveForward2 (Right Motors) - Command 4

The following mixed mode commands are supported:

- DriveBackwardsMixed - Command 9
- DriveForwardsMixed - Command 8
- TurnLeftMixed - Command 11
- TurnRightMixed - Command 10

Summary

I/O Pins

- MCU PCB
    - 2 Position Vertical Header
        - GND
        - PIC32 UART Interface 1
            - MCTX1/U1TX (Pin 51)Motor Controller
- Motor Controller
    - S0 (GND)
    - S1 (MCTX1/U1TX)

Software Hierarchy

- MC_packserial.a (MPLAB Function Library)
    - Description: Contains all functions for communication with motor controller via packetized serial. Add to any project requiring motor control
- MC_packserial.h
    - Description: Contains function definitions for library. Must be included with any project using the library
- UART.h
    - Description: Contains basic UART configuration and transmission functions for UART blocks 1 and 2. Must be included in pack_serial.h

Function Outline

4 data bytes sent via UART

- Motor Controller Address (Set to 128 via dip switches 4, 5, and 6), input to function
- Command Value: Unique number corresponding to the desired drive command, hard coded into each function
- Speed: Motor speed, input to function
- Checksum: (Address + Command Value + Speed) & 0b01111111, calculated by function

Command/Function Description

- Drive Forward
    - Description: Commands both right and left sets of motors to drive forwards
    - Valid Data (Speed): 0-127
    - Example Command: DriveForwardMixed(char address, char speed);
- Drive Backwards
    - Description: Commands both right and left sets of motors to drive backwards
    - Valid Data (Speed):0-127
    - Example Command: DriveBackwardsMixed(char address, char speed);
- Turn Left
    - Description: Commands vehicle to turn left
    - Valid Data (Speed): 0-127
    - Example Command: TurnLeftMixed(char address, char speed);
- Turn Right
    - Description: Commands vehicle to turn right
    - Valid Data (Speed): 0-127
    - Example Command: TurnRightMixed(char address, char speed);

3.7 MCU PCB

Description

The MCU PCB integrates the Microchip PIC32MX795F512H microcontroller, NXP FXOS8700CQ 3D Accelerometer + Magnetometer, power supply, and all relevant I/O connection points onto a single PCB. Duplicate I/O connections along with several extra microcontroller pins and communication interfaces are brought out to a series of header pins to simplify debugging and provide developmental flexibility.

MCU

The Microchip PIC32MX795F512H was chosen as the centerpiece of the board and the VaLET system design in general primarily due to familiarity with its design and firmware requirements. The access to numerous Kit Boards to develop and troubleshoot along with its combination of processing power and I/O support made it an easy choice.

Power Supply

Power is supplied by a Mini USB (5V) connection. This configuration was chosen due to the availability of Mini USB connectors and cords in the senior design lab stock, as well as the flexibility it affords. The board can be easily removed from the robot chassis and powered up by a computer or wall adapter for programming and debug purposes.

In order to supply the 3.3V required by the PIC32 (2.3-3.6V) and E-Compass (VDD = 1.95-3.6V, VDDIO = 1.62-3.6V), a voltage regulator is required. The STMicroelectronics LD1117D is utilized on all the Kit Boards in the Senior Design Course and we had all implemented previously on our demo boards during fall semester. Its proven functionality and reliability combined with its familiarity and availability made it an easy choice for the task. It is connected directly to the Vin, the input supply, and supporting circuitry was designed according to the spec sheet.

Figures 15 and 16 below showcase the power supply circuit schematic and layout on the board respectively. A green LED is connected to Vin and lights up when it and the board are receiving power.

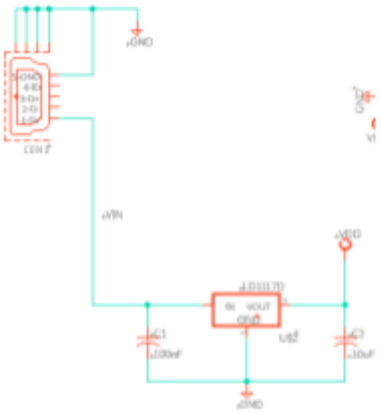


Figure 15. Power supply circuit schematic

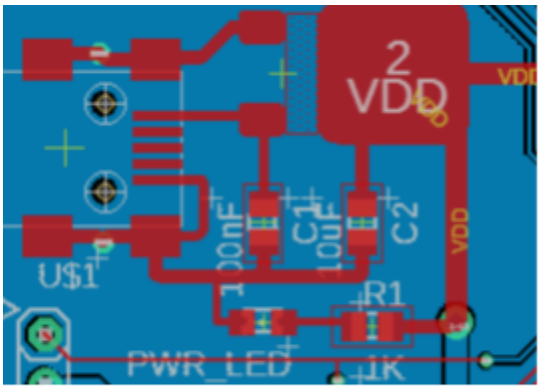


Figure 16. Power supply circuit layout

Programming

The PIC32 is programmed via the standard Microchip PICKIT3 device. The connection is located on the bottom left side of the board using a right angle through hole connector.

LEDs

The board is equipped with 3 Red LEDs in a cluster near the top left, as seen in Figure 17, to assist in debugging. Their implementation is modeled after the LED bank present on the Kit Board. They are labeled ER_LED1, ER_LED2, ER_LED3 and are designed to light up in different sequences to indicate different error states during device operation enabling rapid system feedback and debugging. However, they could be used for any purpose. The connections to the PIC32 are outlined below in Table 6.

	ER_LED1	ER_LED2	ER_LED3
PIC32 Port	E1	E2	E3
PIC32 PIN	61	62	63

Table 6. LED connections to PIC32

Ports are configured for operation with LEDs by configuring them to output (TRISE bits to 0) and driving them high (corresponding PORTE bits = 1) or low (PORTE bits = 0) to turn the LEDs off or on respectively.

All LED signals are also brought out to the header pin labeled PORTE so the signals can be viewed or utilized elsewhere. If the user would like to use the ports for another purpose, power to the LEDs can be disconnected with the jumper labeled LED PWR in Figure 18. Thus, maximum flexibility is provided allowing the device to evolve over time.

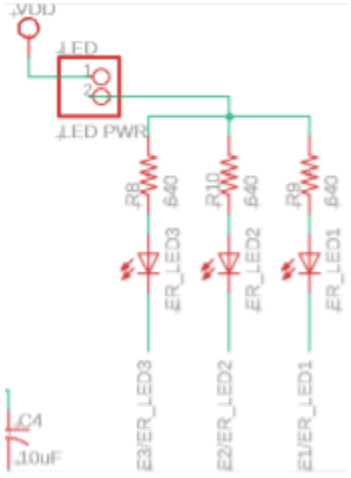


Figure 17. LED schematic

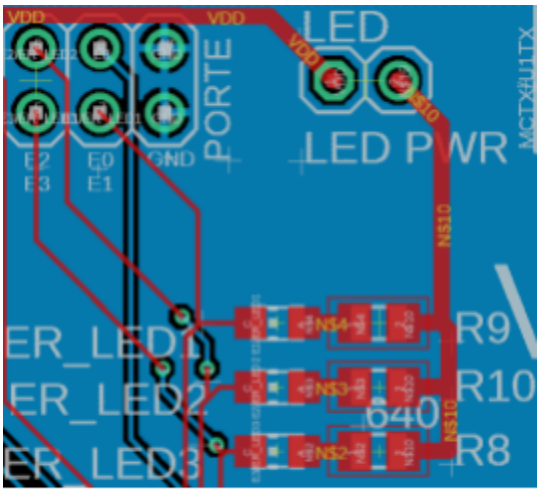


Figure 18. LED board layout

Motor Controller

The sabertooth motor controller is sent commands via a singular UART transmit pin. Looking at Figure 19, transmission is accomplished through a 2 pin Molex connector providing both data and ground. This connection is located near the top right of the board and the signals are duplicated, along with extra I/O and corresponding UART receive port for flexibility, on the header labeled MC_HEADER in Figure 20.

The UART transmit signal originates in the PIC32 UART 1 interface/block (U1TX) corresponding to pin 51 on the PIC32.

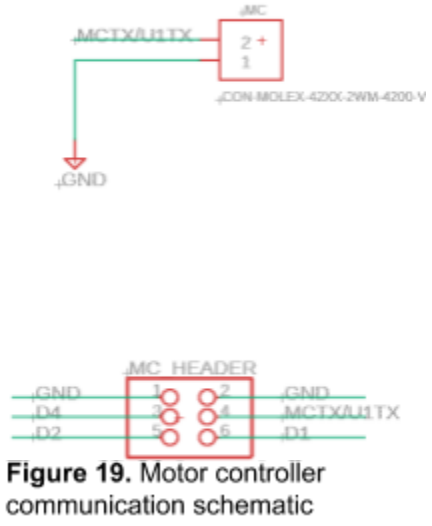


Figure 19. Motor controller communication schematic

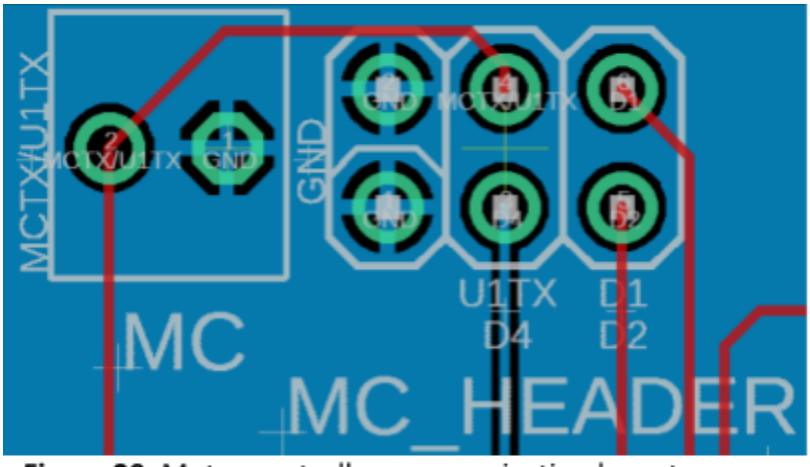


Figure 20. Motor controller communication layout

Raspberry Pi

The Raspberry Pi and vision subsystem are proposed to communicate via a custom SPI protocol. SPI was chosen for its speed, the flexibility of its 4 channel configuration affords, and

hardware support by both the Raspberry Pi and PIC32. An exact communication standard was never developed due to the onset of COVID-19.

The SPI bus is connected to the SPI 2 interface of the PIC32. As seen in Figure 21 and 22, signals are routed to both a Molex connector for communication to the Raspberry PI and a 6 pin header labeled PI_HEADER for inspection and error checking. Signal names and PIC32 connections are summarized below in Table 7.

	PI/SDI2	Pi/SDO2	PI/SCK2	PI/SS2
PIC32 Port	G7	G8	G6	G9
PIC32 PIN	5	6	4	8

Table 7. Raspberry Pi communication details

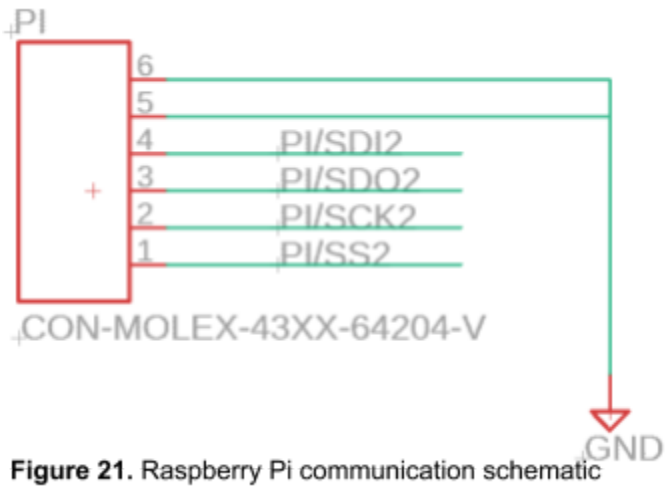


Figure 21. Raspberry Pi communication schematic

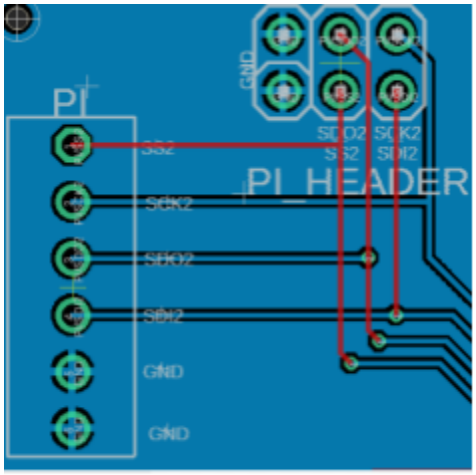


Figure 22. Raspberry Pi communication layout

GPS

The GPS sensor is self-contained on a seperate small PCB designed to be connected via a wire harness with molex connectors. The relevant signals are routed from the PIC32 and MCU PCB to a right angle, 6-pin molex connector on the bottom right of the board. The connections are summarized in Table 8. GPS and some extra I/O are routed to a 8-pin header leabled PORTBH.

	VDD	GND	GPSTX/U2 TX	B15/PPS	GPSRX/U2 RX	!MCLR!
PIC32 Port	N/A	N/A	F5/U2TX	B15	F4/U2RX	!MCLR!
PIC32 PIN	N/A	N/A	32	30	31	7

Table 8. GPS communication details

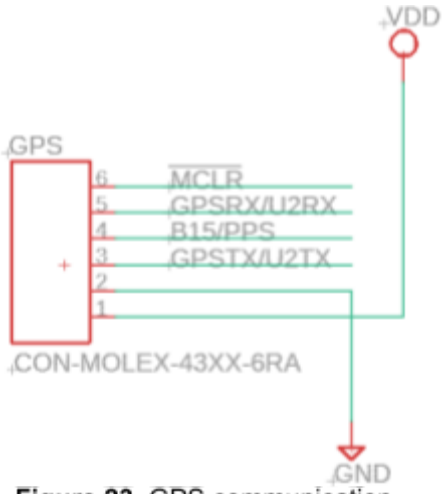


Figure 23. GPS communication layout

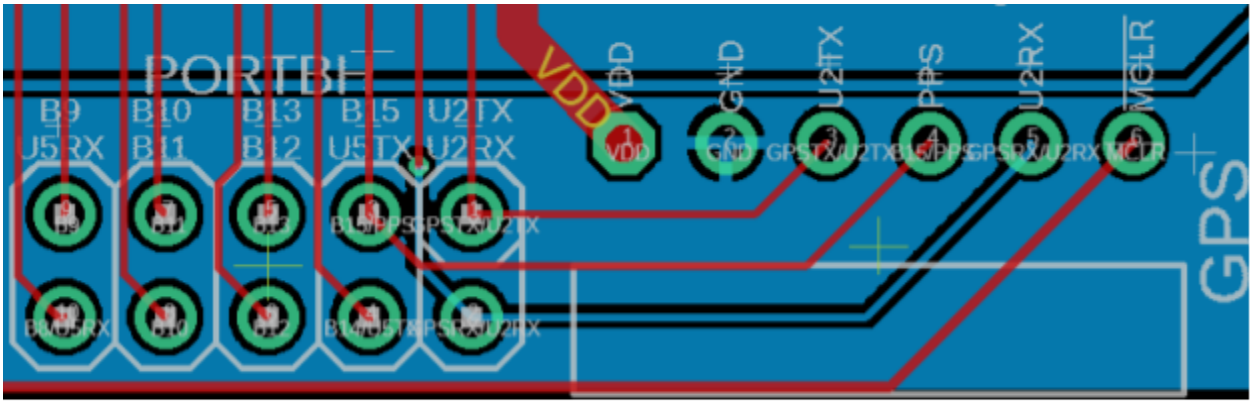


Figure 24. GPS communication layout

E-Compass

Hardware Justification

The NXP FXOS8700CQ was chosen to provide orientation and motion data for VaLET. It is a digital motion sensor with an integrated 3D linear accelerometer and magnetometer providing a complete 6-axis electronic compass solution. Knowing orientation data would be required for VaLET to navigate to GPS waypoints, this particular part was chosen for several reasons.

1. 3.3V Supply: The majority of the magnetometer and e-compass solutions on the market operate using a 1.8V supply. Meanwhile, the FXOS8700CQ accepts any supply voltage between 1.95 and 3.6V and an IO supply voltage between 1.62 and 3.6V. 1.8V is outside the operating range of the PIC32 and thus would have necessitated the implementation of a second voltage regulator to power the device and a level shifter to facilitate communication with the PIC32.

2. Development Board Availability: Enabled software and integration development and testing without a fabricated MCU PCB.

3. Integrated DSP and Interrupt Function Blocks: The device has hardware support for DSP functions including a high pass filter for accelerometer data and a built in compensation engine for eliminating hard iron effects from magnetometer data. These are critical to achieving accurate and noise/error free readings. It also has two dedicated interrupt ports which can be configured to trigger based a series of several events/operations. This potentially enables VaLET to automatically detect if it crashes, flips-over, or isn't moving when it should be automatically via a simple interrupt.

4. Accelerometer Support: Although not explicitly required for the navigation of VaLET, the opportunity to essentially get accelerometer functionality for free was a great asset. Accelerometer functionality enables VaLET to potentially detect accidents and/or unforseen motion or lack thereof. It also opens the door for the potential to implement a tilt compensated electronic algorithm. Such an algorithm adjusts magnetometer readings based on the angle of the device ensuring accurate heading information even if VaLET is operating on an incline. Such an algorithm was ultimately not integrated as is outlined in the E-compass subsystem section.

Hardware Implementation

The FXOS8700CQ is located on the right side of MCU PCB. Table 9. provides a summary of all the e-compass connections. Supporting circuitry can be seen in Figure 28 (Schematic page 3) and 25 and primarily consists of decoupling capacitors as mandated by the data sheet. The device supports communication via both SPI and I2C, however the same pins are used for both. The chip auto-detects the communication interface being used and the SPI MOSI (SA0) and SPI Chip Select (SA1) are used to set the I2C address when I2C communication is desired. Setting these pins to high (VDDIO) or low (GND) allows for the selection of 4 I2C addresses. In the implementation seen here, both are grounded making the I2C address 0x1E. Pull up resistors are implemented on both bus channels for correct operation.

The reset also requires some extra circuitry. Contrary to the GPS/PIC32, the reset of the FXOS8700CQ is active high. A 2 channel reset switch enables this functionality (See sheet 1 of the schematic). One channel pulls the active low components to ground, the other pulls the e-compass reset high by connecting it to VDD. When not active, the reset pin is connected through a resistor to ground as shown in Figure 25. This resistor ensures that when VDD is applied, voltage is dropped across the resistor to ground and the signal remains high at the reset input. A capacitor is also connected between VDD and the reset pin. Normally this does nothing, however upon power up, it acts like a short and instigates a power on reset of the e-compass ensuring the communication interface auto detection function operates properly (VDDIO must be applied before are at the same time as VDD, or a reset initiated). This is most likely redundant, but it pays to be safe.

All FXOS8700CQ communication connections (along with some extra I/O) are also brought out to the 8-pin header above it labled E_COMPASS. They are there to assist in debugging the device.

Symbol	FXOS8700CQ Pin	PIC32 Pin	PCB Signal	Description
VDDIO	1	VDD	VDD	Interface supply voltage
BYP	2	-	GND Via C11 Capacitor	Internal regulator output bypass capacitor connection
Reserved	3	-	GND	Test reserved
SCL/SCLK	4	44	ECSCL/SCL1	I2C serial clock/SPI clock
GND	5	GND	GND	GND
SDA/MOSI	6	43	ECSDA/SDA1	I2C serial data/SPI master out, slave in
SA0/MISO	7	GND	GND	I2C address selection bit 0/SPI master in, slave out
CRST	8	-	GND Via C10 Capacitor	Magnetic reset capacitor
INT2	9	45	ECINT2/INT4	Interrupt 2
SA1/CS_B	10	GND	GND	I2C address selection bit 1/SPI chip select (active low)[
INT1	11	42	ECINT1/INT1	Interrupt 1
GND	12	GND	GND	GND
Reserved	13	GND	GND	Test Reserved
VDD	14	VDD	VDD	Supply Voltage
N/C	15	-	-	No Connection
RST	16	-	EC_RST	Reset Input, active high

Table 9. E-compass connections

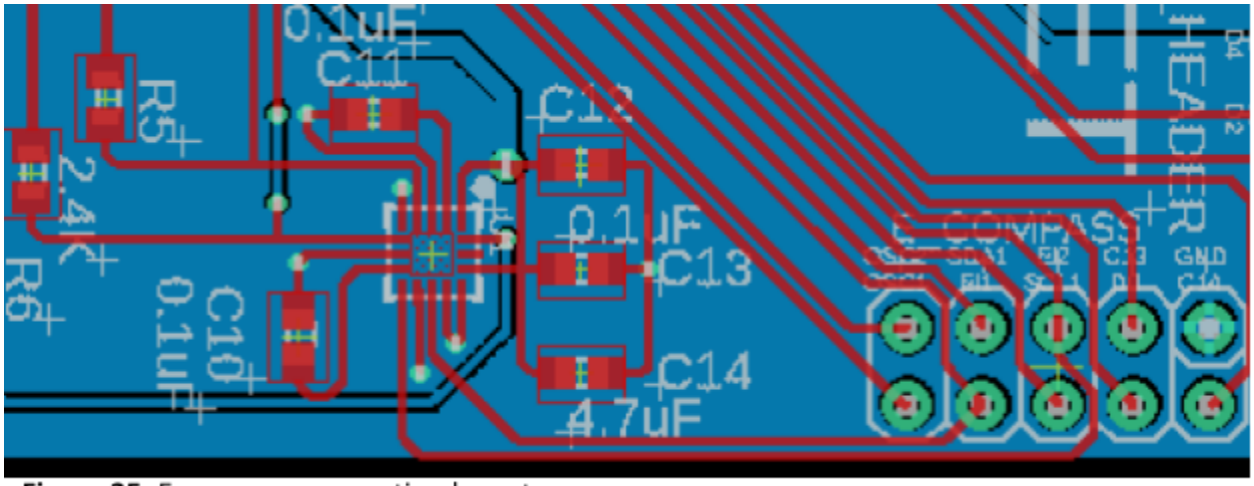


Figure 25. E-compass connection layout

Extra I/O

Extra PIC32 designated for debugging or expanded board capability enabling future upgrades to VaLET is supplied to several header pins throughout the board design. The signals and their connections are outlined in Table 10 below.

Signal Name	PIC32 PIN	Header Label
E0	60	PORTE
E4	64	PORTE
E4	1	PORTE
E6	2	PORTE
E7	3	PORTE
B8/U5RX	21	PORTBH
B9	22	PORTBH
B10	23	PORTBH
B11	24	PORTBH
B12	27	PORTBH
B13	28	PORTBH
B14/U5TX	29	PORTBH
C13	47	E_COMPASS

C14	48	E_COMPASS
D0/INT0	46	E_COMPASS
OSC1	39	E_COMPASS
OSC2	40	E_COMPASS

Table 10. Extra I/O pin connections

BOM

Qty	Value	Device	Manufacturer	M/F Part Number	Package	Reference Designator	Description	Supplier	Supplier P/N
3		LED	Stock	Stock	CHIP-LED0805	ER_LED1, ER_LED2, ER_LED3	Red Error LEDs	Stock	
1		LED	Stock	Stock	CHIPLED_0805	PWR_LED	Green Power Indicator LED	Stock	
1		ACCEL/MAG	NXP Semiconductor	FX0S8700CQ	QFN-16	U1	E-Compass	Mouser	841-FXOS8700CQR1
1		MCU	Microchip	PIC32MX795F512H	TQFP-64	IC1	32 Bit MCU	Microchip	PIC32MX795F512H
3		PINHD-2X3	Stock	Stock	2X03	MC_HEADER, PI_HEADER, PWR_RST	PIN HEADER	Stock	
3		PINHD-2X5	Stock	Stock	2X05	E_COMPASS, PORTBH, PORTE	PIN HEADER	Stock	
1		USB-MINI-B	Stock	Stock	USB-MINI-FCI10033527	CON1	Mini USB Power Connector	Stock	
9	0.1uF	C-US	Stock	Stock	C0603	C5, C6, C7, C8, C9, C10, C11, C12, C13	CAPACITOR	Stock	
1	0.1uF	C-US	Stock	Stock	C0805	C3	CAPACITOR	Stock	
1	10	R-US	Stock	Stock	R0603	R4	RESISTOR	Stock	
1	100	R-US	Stock	Stock	R0603	R3	RESISTOR	Stock	
1	100nF	C-US	Stock	Stock	C0805	C1	CAPACITOR	Stock	
1	10K	R-US	Stock	Stock	R0603	R2	RESISTOR	Stock	
1	10uF	C-US	Stock	Stock	C0603	C15	CAPACITOR	Stock	
2	10uF	C-US	Stock	Stock	C0805	C2, C4	CAPACITOR	Stock	
1	1K	R-US	Stock	Stock	R0805	R1	RESISTOR	Stock	
2	2.4K	R-US	Stock	Stock	R0603	R5, R6	RESISTOR	Stock	
1	4.7uF	C-US	Stock	Stock	C0603	C14	CAPACITOR	Stock	
3	640	R-US	Stock	Stock	R0805	R8, R9, R10	RESISTOR	Stock	
1	8.2K	R-US	Stock	Stock	R0603	R7	RESISTOR	Stock	
1		CON-MOLEX-2-Vert	Stock	Stock	WM-4200	MC	Molex 2-pin Vertical	Stock	
1		CON-MOLEX-6-Vert	Stock	Stock	WM-4204	PI	Molex 6-pin Vertical	Stock	
1		CON-MOLEX-6-RA	Stock	Stock	WM-4304	GPS	Molex 6-pin Right Angle	Stock	
1	3.3V	Voltage Regulator	STMicroelectronics	LD1117D	DPAK	U$2	SMT voltage regulator	Stock	
1		PINHD-1X2S	Stock	Stock	1X02N	LED	Jumper LED power	Stock	
1		PICKIT3	Stock	Stock	PICKIT3	U$1	Programming Connection	Stock	
1		SWITCH-PB	Stock	Stock	PBSWITCH	SW1	Reset Switch	Stock	

Schematic

Figure 26. Schematic page 1

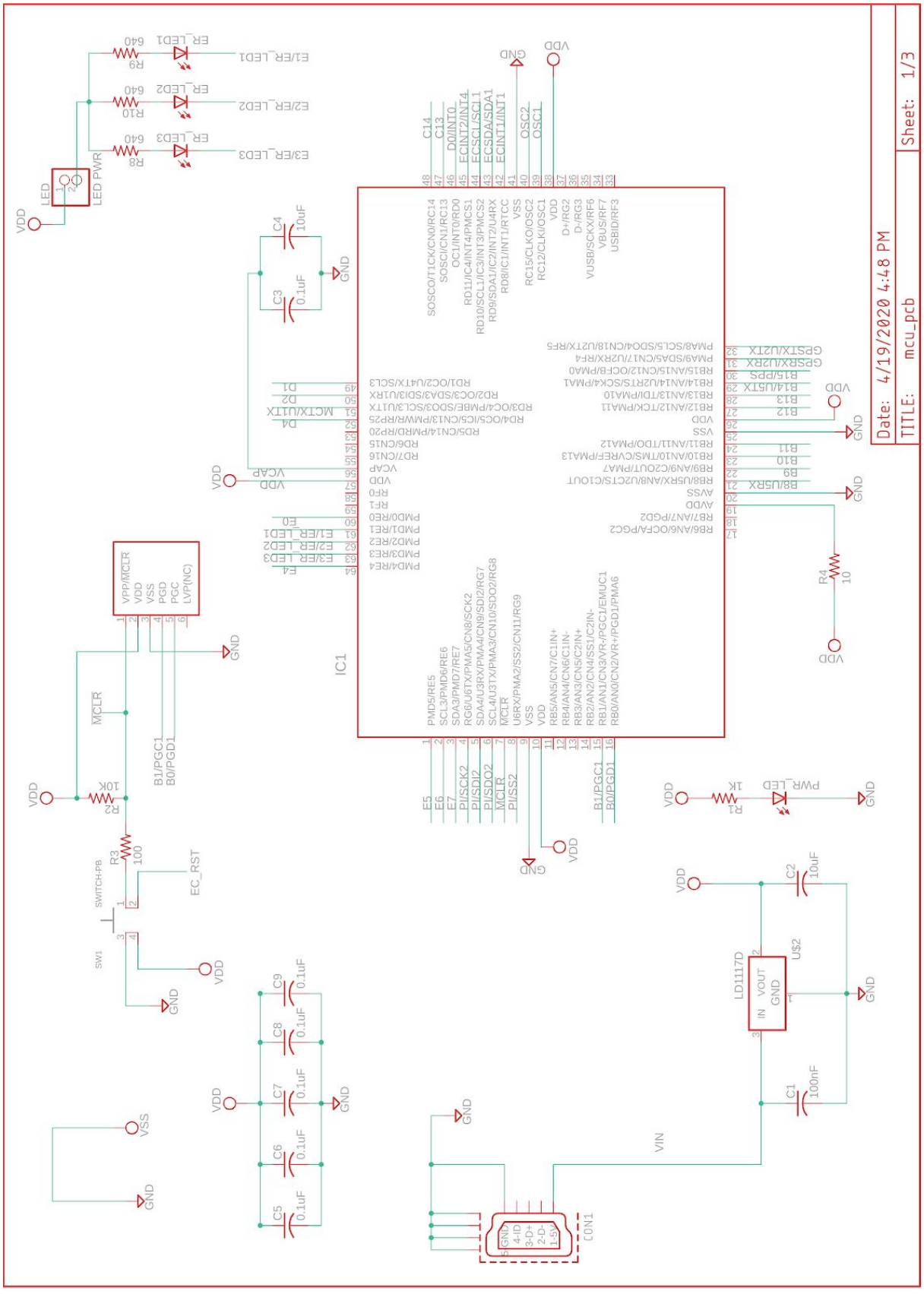


Figure 27. Schematic page 2

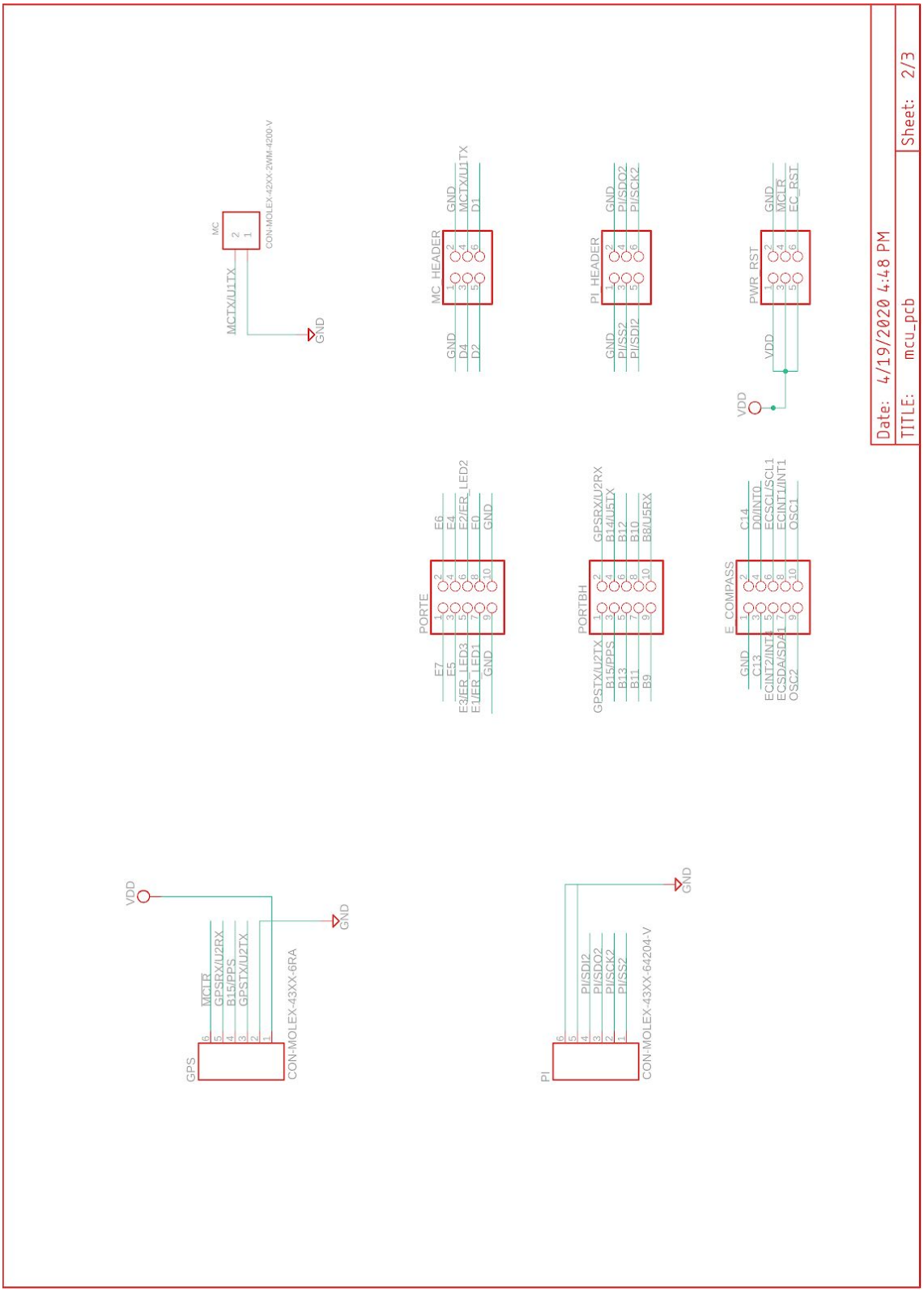
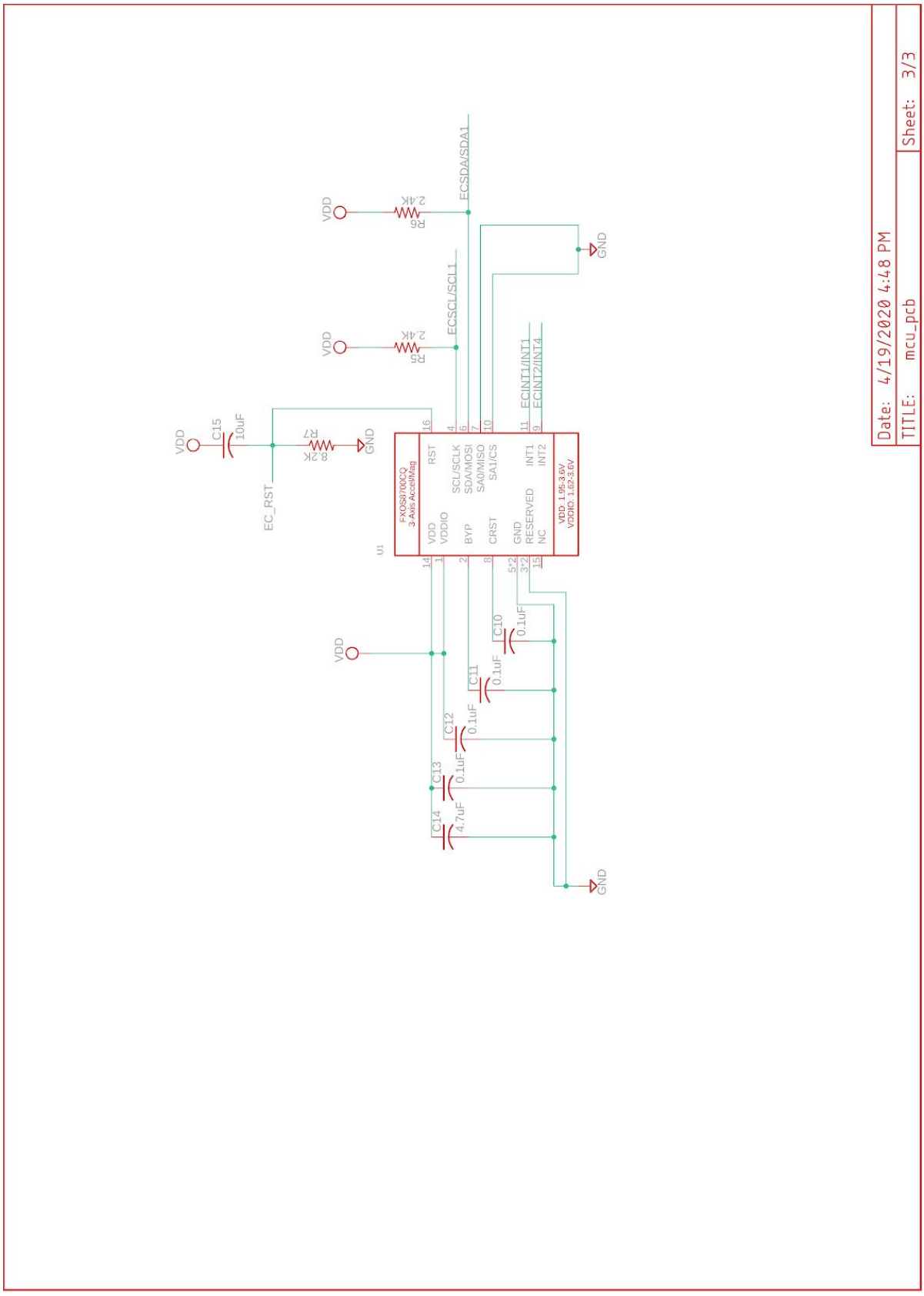
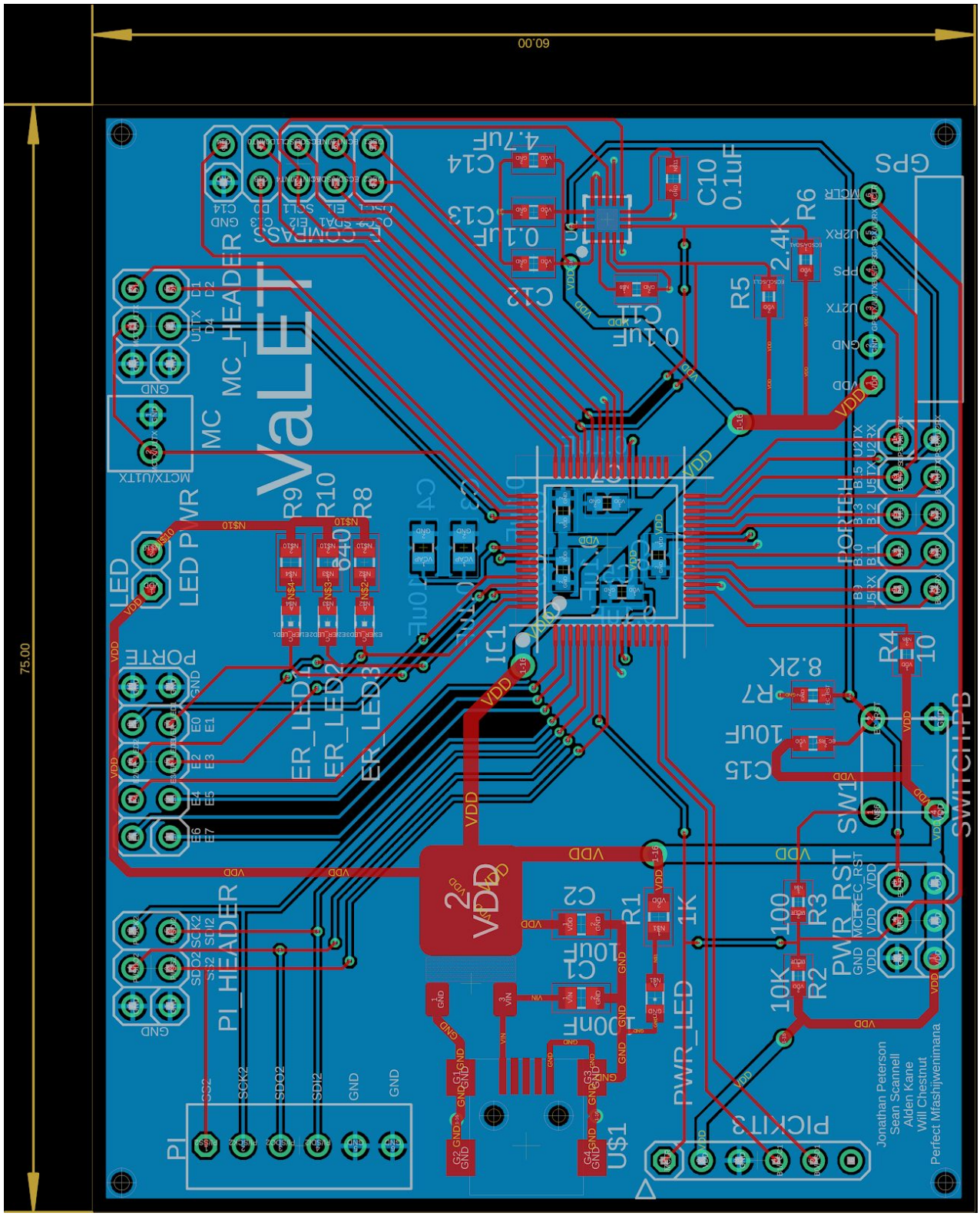


Figure 28. Schematic page 3



Board Layout

Figure 29. Board Layout



3.8 Interfaces

Overview

Before the advent of Covid-19, we had just completed our second design review in which we demonstrated the independent functioning of VaLET's various subsystems. We therefore never got the opportunity to integrate the systems into a cohesive whole and engineer all the interfaces this would require. The interfaces that do exist revolve around a system communicating with their host processor (e-compass with PIC32, motor controller with PIC32, GPS with PIC32, camera and Garmin LIDAR with Raspberry PI). For all the systems to communicate, the two processors must be linked together. What follows is our proposal for communication between the PIC32MX and Raspberry Pi.

Raspberry PI and PIC32 Interface

The main interface that exists in this subsystem is that between the Raspberry Pi and the PIC32 using SPI protocol. The main reason for this integration was the convenience of the Raspberry Pi's with regards to the camera systems and wi-fi access. Raspberry Pis have preexisting camera modules that are relatively easy to use. Raspberry Pi makes it trivial to use openCV on its Linux OS. One of our team members was well acquainted with these systems which lowered a substantial barrier. This software would have been extremely difficult to create from scratch and Professor Schafer gave us approval to do so. Additionally, there was a benefit due to Wi-Fi accessibility of the Raspberry Pi.

---

4 System Integration Testing: High Level State Machine of Integrated System and Proposed Testing Method

Introduction to the State Machine

To best visual how this system would operate, we decided to create a High Level State Machine. This exercise provides a comprehensive, yet simple to understand, view of our vision for our final project. We would have modeled our logic and code around the different states in this state machine to achieve our final goal. We have included descriptions of all of the states below to assist the reader in understanding the state machine. With this tool, we attempted to describe every aspect of normal operations for our system. As is the case with all state machines, this figure does not describe every situation and set of circumstances that the VaLET system could be placed in, but we attempted to cover the issues that are most likely to arise. We also outline a proposed testing protocol for our now hypothetical system.

State Machine

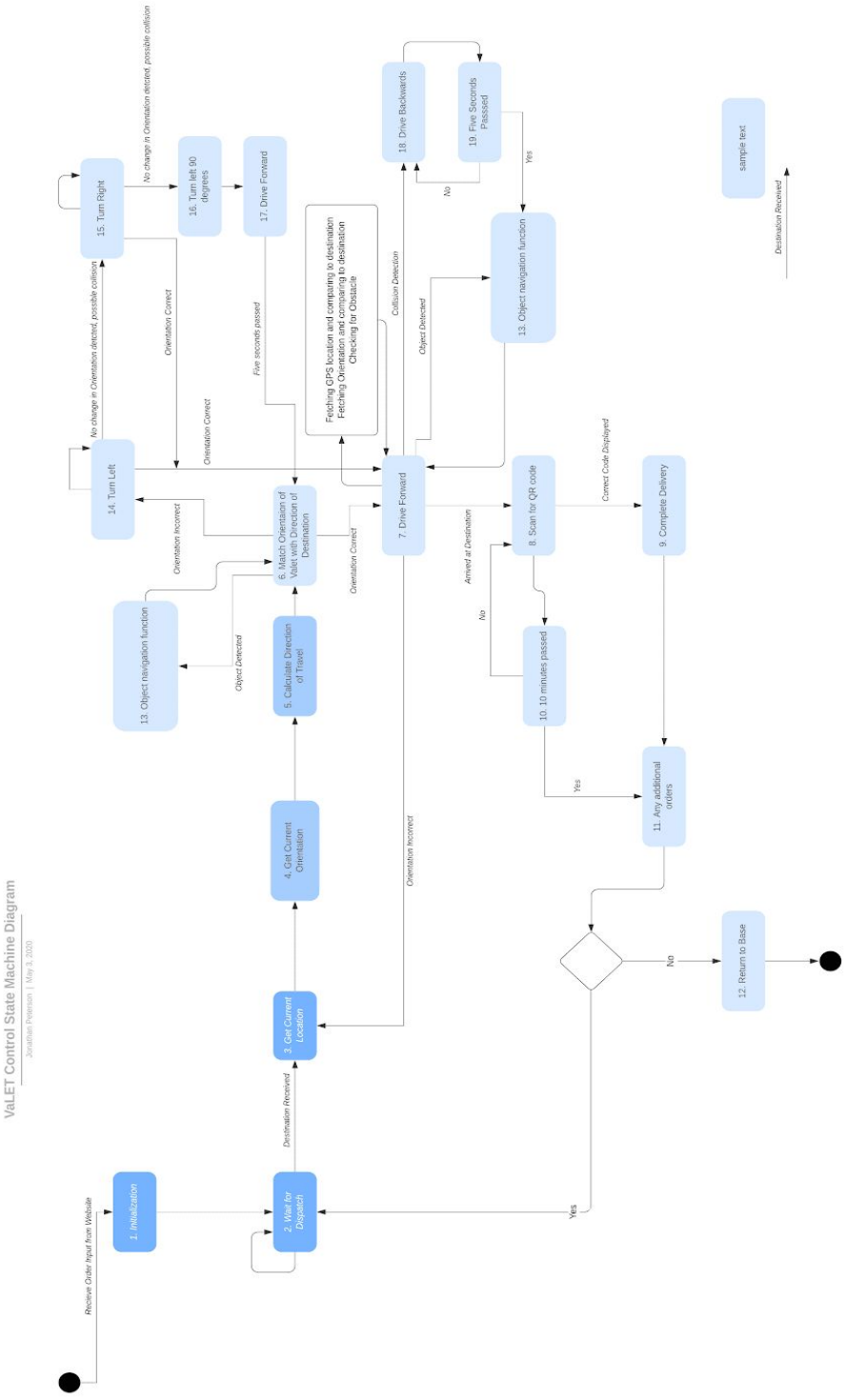


Figure 30. State Machine

State Descriptions

1. **Initialization**: Occurs when the system is turned on. Includes powering up and checking for communication between the subsystems. PIC32 pings the Raspberry Pi and checks for response. Run initialization code for e-compass. Check power and communication to GPS. If Raspberry Pi is communicating, check internet connection.
2. **Wait for Dispatch**: Wait for GPS coordinates and dispatch signal to be fetched from the project website via Wi-Fi (Allows for dispatch of potential fleet of VaLET vehicles).
3. **Get Current Location**: Read in NMEA data from GPS and confirm that the data is readable.
4. **Get Current Orientation**: Read in orientation data from the e-compass. Compute heading from raw x,y,z magnetometer data
5. **Calculate Direction of Travel**: Compare GPS coordinates of destination to GPS coordinates of current location. From this, determine the direction of travel necessary.
6. **Match Orientation of VaLET with Direction of Destination**: Compare current orientation with the necessary direction of travel and determine how VaLET needs to turn to match the two.
7. **Drive Forward**: call drive forward function
8. **Scan for QR code**: start scanning using the raspberry pi camera, once received, compare that qr code to check if correct
9. **Complete Delivery**: Allow users to retrieve packages from the system. Press button on system to complete.
10. **10 minutes passed**: Timer to wait 10 minutes
11. **Any additional orders:** Check if there are any additional orders to be completed.
12. **Return to Base**: Return to the location that is designated as the base location. Potential for function to remap steps taken to get to the current location.
13. **Object navigation function**: check the lidar subsystem for any obstructions in front of the system. If an object is detected, call the function turn left until no obstruction is detected. Call function drive forward for a specified amount of time. Transition to check orientation state again to course correct.
14. **Turn Left**: Call Turn Left function. Continuously compare orientation with required direction of travel until they match.
15. **Turn Right**: Call Turn Right function. Continuously compare orientation with required direction of travel until they match.
16. **Turn Left 90 degrees**: Call Turn Left function until orientation has been changed by 90 degrees
17. **Drive Forward**: Call Drive Forward function.
18. **Drive Backwards**: Call Drive Backwards function.
19. **Five Seconds Passed:** Timer to maintain current operations for 5 seconds.

Proposed Testing Method

To start testing, begin by finding a wide open space with few tall buildings directly overhead. Any quad at Notre Dame was envisioned as an ideal place to conduct testing of the system. VaLET could be incredibly useful in this environment, yet it is also contained and relatively obstacle free.

GPS and motor drivers

First, set the destination to a location that is due east of your current location and observe the systems reaction. If the system drives in a straight line to the destination, it is operating as expected. Repeat the process for a location that is due north of your current location. If both of these tests are passed, several of the subsystems are operating as expected. For the final test of this system place the destination at a location with different longitude and latitude from your current location. If the system travels in a straight line to the destination, it is operating as expected.

LiDAR and obstruction

To test that the LiDAR is working correctly, once again set the destination to a location that is different from current location. When the VaLET system is en route, place an obstruction in its path that is visible to the system. If the VaLET system stops, turns until the path forward is clear, and then drives past the obstruction, it is operating as expected.

QR Scanner

To test the QR scanner, set the correct QR code to be one unique code and set the delivery destination to be the current location. From this, the system will believe that it is at the correct destination and attempt to deliver the package. Next, attempt to confirm delivery with an incorrect code. If the VaLET system does not accept this code, but does accept the correct code when displayed, it is operating as expected.

If all of the tests described in this section are completed without failure, then the system has demonstrated its ability to travel from one destination to another while avoiding obstructions, and on arrival delivering an item. This satisfies the system requirements as described in our problem statement.

---

5 Users Manual/Installation manual

For this section, we will assume that the VaLET system has reached a point in its product life cycle that it is available to be used by a company for food delivery. This is not the point we are at now, but in order to visualize a customer's initialization and testing, it is a necessary step. With that in mind, we need to assume that the system has completed the to-market design challenges that are listed in this paper. Additionally, there would need to be a customer facing software system that could be used as a tool for the customer. This would include a website or application that consumers could order on and that customers would have access to. These

software offerings would additionally need security systems to protect both the customer and consumer's information.

If all of these aspects were accounted for, then the VaLET system would be sold in a package that included the chassis with all of the subsystems stored on the interior and the developed item storage container on top. The wheels would be unattached.

5.1 How to install your product

Upon receiving the system, unpackage the wheels, system and charging cable. Screw the left wheels on the left side of the chasis and the right wheels on the right side of the chassis. Once completed, plug in the charging port and let sit until the system indicates that it has been fully charged. While the system is charging, visit the VaLET website on your web browser of choice. If this is your first VaLET system, create an account with the corresponding information. If not, log in to your existing account. Select the add new device and input the serial number of the device being set up. This will prompt you to download the application to your smartphone or smart device. Log on to the application and follow the instructions.

5.2 How to setup your product

Construction, initialization, setting up the home base, getting the system built
Once the device has completed charging, the application on your smart device will walk you through the process of setting up the system. This will include placing the item in an open area with clear access to the sky so that a clear GPS signal can be determined. When the system has established a clear GPS signal, it will prompt you asking the user if the current location should be set as the home base. Confirm this if the current location is satisfactory, or move the system to the desired location.

5.3 How the user can tell if the product is working

To test the systems after the product has been set-up, simulate a customer order by entering a destination for the system to travel. If the system travels from its origin to your destination, releases the item when the correct QR code is displayed and then returns to its home base, then the user knows that the product is operating correctly.

5.4 How the user can troubleshoot the product

There are several likely issues that can arise during normal operations. If the device will not turn on, try plugging the device in and charging it. If the device does not turn on, then the issue is likely beyond the user's ability to fix the device and they should contact a VaLET support member.

If the device turns on, but can not establish a GPS signal, check that there is a clear skyline and that the device's signal is not clogged by trees or a busy skyline. If this is the case, try and move

the device to a clear open area to improve the connect. If not, then the issue is likely beyond the user's ability to fix the device.

If the device turns on and begins to spin indefinitely, there is likely an issue with the e-compass. Try turning the device off and on again and check if that resolves the issue. If not, then the issue is likely beyond the user's ability to fix the device.

If the device turns on and travels in the correct direction, but does not avoid objects, there is likely an issue with the camera or pi subsystem. Attempt to clear the camera lense and reboot the device to check if the issue is solved. If not, then the issue is likely beyond the user's ability to fix the device.

---

6 To-Market Design Changes

Due to the Covid-19 crisis, VaLET is not even a functioning prototype. However, this section is constructed as if we had built a prototype that satisfies the rudimentary design goals. The following are a list of changes and updates that would need to occur to transform the VaLET prototype into a product that is ready for market.

1. **Safety:** VaLET is not currently equipped with the quantity and quality of sensors and processing power to ensure safe operation in all conditions. Required safety features include:
    a. Low Light/Night Vision: VaLET only has one sensor, the LiDAR, which works in low light conditions. It is practically blind in the rain or low light conditions and could potentially be dangerous. VaLET must be equipped with multiple higher quality camera sensors to facilitate visibility in all conditions. A larger suite of sensors that can compensate if one or more is compromised is also necessary.
    b. Field of View: VaLET's vision system is designed to view and detect whatever is directly ahead of it. Furthermore, the field of view of the LIDAR sensor is less than 10 degrees. Multiple sensors enabling a 360 degree field of view are necessary for safe operation
    c. Vehicle Brakes: VaLET is not currently equipped with brakes. Unpredictable real world scenarios will make them necessary to stop at a moment's notice to protect people and the environment.
    d. Increased Processing Speed: The PIC32MX is a capable microcontroller, however, its CPU performance is biased towards low power and signal aggregation. It does not possess the power to operate the advanced algorithms nor reaction time necessary for perfectly safe operation. It also necessitates a separate processor for the vision system (Raspberry Pi). A single piece of custom hardware operating on an ASIC/FPGA or a high performance microprocessor is required.

e. More Advanced Navigation Algorithm: Our current navigation algorithm is rather rudimentary. Adaptability (potential for machine learning) and baked in safety hardstops are necessary for robust and safe operation.

2. **Security:** VaLET must be outfitted with features to enable secure delivery of the package as well as the security of the VaLET vehicle itself. As it stands, packages are designed to merely ride on top of the vehicle. An enclosed and locked box that opens upon delivery recipient confirmation would ensure package security. To protect the VaLET vehicle, distress call protocols would need to be implemented in the event someone tried to steal or damage the vehicle. If it is being stolen, the GPS would begin to read a location not consistent with where it is attempting to navigate to. In the scenario, VaLET must have a way to send out a signal alerting its operator of what is happening while also emitting a loud noise to attract attention and dissuade the thief. A similar protocol could exist if it detects damage to one of its systems.

3. **Scale:**
    a. Size: VaLET must be appropriately sized for its missions. As it currently stands, its carrying capacity is quite small. Sizing up VaLET or introducing variants of different sizes would enable VaLET to capture more of the marketplace.
    b. Power: Greater motor power is required to operate a larger vehicle and potentially operate on city streets at high speeds. This may also necessitate the switch to an AC motor (and controller) and integration of a higher voltage system.
    c. Quantity: VaLET must be able to be mass produced in order to be a viable market solution. Integrating all electronic components on a single PCB would facilitate mass production as well as eliminate wires/costs. Modular parts would also make it easy to add and subtract systems and build VaLET vehicles on a production line.

4. **Performance:**
    a. Adaptability:
        i. VaLET's current navigation algorithm is designed for the very controlled environment of a Notre Dame quad and is reliant on VaLET not encountering any situations that we did not envision when designing it. Endowing VaLET with a more advanced navigation algorithm potentially incorporating deep learning would enable it to adapt and tackle the greater variety of situations in the real world. A great set of tracking and navigation features would help too such as the ability to follow roads/sidewalks, detect and identify people and other objects, and scan and map out environments using 3D tools such as radar.
        ii. In the event the GPS signal is lost or compromised, VaLET must still be able to navigate to a certain extent. The implementation of an inertial navigation system would help it stay on track, or return to base in the case of spotty coverage.

iii. Over the air updates would be critical to ensure that VaLET is easily serviced and is compatible with the newest technologies.
iv. Single Processor: Sourcing of a chip that can handle all processing duties rather than separate CPU (PI+PIC) for vision on processing functions.
v. Battery Life: The current battery is plenty large and can handle hours of continuous operation. However, more features, more processing power, and a greater size will significantly reduce battery life. A larger battery (perhaps Lithium Ion for greater lifetime) may be required.

b. Durability:
i. Waterproofing: VaLET needs to be sealed to allow it to operate in the rain.
ii. Dust: Current exposed components and PCBs are susceptible to damage from dust. These must be protected.
iii. Strength: A for market VaLET must have a long usage life and be able to perform many missions day after day. The current chassis constructed of aluminum brackets and Lexan (polycarbonate) panels is very lightweight, however, it does not possess the durability for the hard life of a market VaLET vehicle.

5. **Adherence to Regulatory Requirements:** There is not currently a standard set of regulations concerning the design and deployment of autonomous delivery vehicles in the United States. However, any market solution would have to adhere and be designed with these regulations (which may vary by state/country) in mind. VaLET may need to be adapted to meet regulations across the United States and the world.

---

7 Conclusions from the Team

Jonathan Peterson

VaLET has been an immensely challenging yet satisfying project to work on. I am incredibly saddened by the fact that due to the COVID-19 crisis, we were unable to complete the project (to see our baby walk if you will). However, the journey is just as important as the conclusion, and I developed significantly as an engineer along the way.

My primary areas of contribution to the product include, speccing and ordering all of the components, designing the MCU PCB, assembling the chassis, and writing and testing all of the PIC32MX C code (Interfaces, motor control, and e-compass). Having the opportunity to work with such a large variety of systems sharpened my problem solving skills and forced me to think outside the box as we faced a multitude of new challenges. No decision could be made without thinking about how it may impact every other system and attention had to be paid to every detail.

I am proud of what this team has accomplished in VaLET. It fills a legitimate market need and fits into many current areas of research and development in electrical engineering. I

hope that in the coming years, other student design students may pick off where we left things, and finish what we started. I would love to come back to campus at some point and see a bunch of little VaLET's zooming around.

Perfect

In any situation, hitting milestones is just as important as the final product. It is a process that teaches project coordination and exposes to design complexities. Working on VaLET was a complete process since it started with theoretical design thinking. Through this process, I learned to make design decisions for battery systems and all the power systems that glue the project together (decisions on power management, regulation, and testing).

After the thinking process, I learned through working on the e-compass module deciding on different interfaces. Once decided on I2C, I got to work on the software side of the E-Compass as well as the testing. It was a great experience and I hope that what we worked on might get to see an end.

Alden

I'm very fortunate to have worked on this project with phenomenal people. We had a good collaborative team —– characteristic of ND engineering. When we were on campus this semester, I had a tremendous amount of fun building subsystems and modules to go on our drone. I learned a fair amount about practical computer vision, serial communication, and LiDAR through this project.

My primary responsibilities for this project were to manage vision sensors (camera, LiDAR), manage all subsystems housed on the Raspberry Pi (vision, Wi-Fi), and maintain our web presence. More importantly, I got to work on a team of talented engineers and legitimately interface systems to get one hand talking to the other.

It's a shame that quarantine severely curbed our progress on this project, as we were progressing towards a functioning prototype. It surely would've had bugs and kinks characteristic of any proof-of-concept module. But, we would've learned a tremendous amount in the process. Losing the ability to work on this project in the senior design lab makes me value my education even more —– cliche, but you don't know what you have until it's gone. Starting from an idea and seeing a working module is one of the joys of engineering. Nevertheless, I'm extremely thankful for my ND education and the joys of NDEE.

Will

The Power Management System was a great way to further improve the group's Autodesk Eagle skills. Creating a board that was going to be used for the Power Management System allowed our group to design a board outside of the typical MCU designs. Also, having to create

new parts and libraries for the subsystem greatly improved the skill of creating custom parts in Eagle.

Sean

I find it terribly unfortunate that the semester ended the way it did. I fully believe that we would have completed this project and not having the opportunity to do so is disheartening. It is an extremely unceremonious time to end this team's time at Notre Dame and in Senior Design. I greatly enjoyed my time working on this project and getting to know my team members.

My main area of contribution was the GPS subsystem, and I found this an extremely interesting area to learn about. I researched the different elements of this subsystem and built out the logic for the communication between the GPS and the MCU, as well as the direction and orientation determination softwares. GPS strikes me as a very high level engineering tool, but this project taught me how accessible it is and showed me that we have advanced much further as engineering students than I had previously thought.

I hope that no more teams experience the disruption that our year has experienced. Even with that, I am thankful for the opportunity to work on this project and the entirety of my time at Notre Dame.

8 Appendices

Appendix 1: Motor Controller Data Sheet

Motor Controller: Dimension Engineering Sabertooth 2x12 Data Sheet
https://www.dimensionengineering.com/datasheets/Sabertooth2x12.pdf

Appendix 2: PIC32 Data Sheet

Microcontroller: PIC32MX Family Data Sheet
http://ww1.microchip.com/downloads/en/DeviceDoc/PIC32MX5XX6XX7XX_Family)Datasheet_DS60001156K.pdf

Appendix 3: E-Compass Data Sheet

E-Compass: FXOS8700CQ Data Sheet
https://www.nxp.com/docs/en/data-sheet/FXOS8700CQ.pdf

Appendix 4: Robot Kit Information

Robot Chassis: Lynxmotion A4WD1
http://www.lynxmotion.com/c-111-a4wd1-no-electronics.aspx
Assembly Instructions: **http://www.lynxmotion.com/images/html/build122.htm**

Appendix 5: LiPo Battery

LiPo Battery: Gens ace 3300 mAh 45C 4S EC3
https://www.genstattu.com/ga-b-45c-3300-4s1p-ec3.html

Appendix 6: GPS Data Sheet

GPS Antenna: MediaTek 3339
https://labs.mediatek.com/en/chipset/MT3339

Appendix 7: NMEA Data

NMEA Data: GPSinformation.org
https://www.gpsinformation.org/dale/nmea.htm

Appendix 8: Battery Monitoring IC Data Sheet - TI BQ77915

BQ77915 Data: ti.com
https://www.ti.com/lit/ds/symlink/bq77915.pdf?ts=1588535759132

Appendix 9: MIC29300-5.0WU-TR

MIC29300 Data: digikey.com
http://ww1.microchip.com/downloads/en/DeviceDoc/MIC2915x-30x-50x-75x-High-Current-Low-Dropout-Regulators-DS20005685B.pdf

Appendix 8: UART Software

UART.h

```
#ifndef _UART_H_
#define _UART_H_
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>

/************************************************************
 *  UART Serial Functions
    Created by Jonathan Peterson
*/


#define u1_data_avail U1STAbits.URXDA
unsigned char getu(void) {
   while (!u1_data_avail); //wait till data ready

   char dat;
   dat = U1RXREG;
   return (dat);
}

#define u2_data_avail U2STAbits.URXDA
unsigned char getu2(void) {
   while (!u2_data_avail); //wait till data ready

   char dat;
   dat = U2RXREG;
   return (dat);
}

#define u1_buff_full U1STAbits.UTXBF
void putu(char dat) {
```

```
  while (u1_buff_full); // wait if buffer full
  U1TXREG = dat;
}

#define u2_buff_full U2STAbits.UTXBF
void putu2(char dat) {

  while (u2_buff_full); // wait if buffer full
  U2TXREG = dat;
}

void serial_init(unsigned long rate, unsigned long pb_clk) {
   U1MODEbits.ON = 1;
   U1MODEbits.BRGH = 1;
   U1STAbits.URXEN = 1;
   U1STAbits.UTXEN = 1;

   int brg = ((pb_clk/(4*rate))-1);
   U1BRG = brg;
}

void serial_init2(unsigned long rate, unsigned long pb_clk) {
   U2MODEbits.ON = 1;
   U2MODEbits.BRGH = 1;
   U2STAbits.URXEN = 1;
   U2STAbits.UTXEN = 1;

   int brg = ((pb_clk/(4*rate))-1);
   U2BRG = brg;

}

void __attribute__((weak)) _mon_putc(char c) {
  putu(c);
}

#endif //UART
```

Appendix 9: Motor Control Software

MC_packserial.h

```
//Jonathan Peterson
ifndef _MC_PACKSERIAL_H_
```

```
#define _MC_PACKSERIAL_H_
#include <xc.h>
#include "UART.h"

/*************************************************************
 *  Sabertooth 2x12 Motor Controller Packetized Serial Functions
 * Mixed and Normal Drive Commands
*************************************************************/

void DriveBackward1(char address, char speed);
void DriveBackward2(char address, char speed);
void DriveBackwardMixed(char address, char speed);
void DriveForward1(char address, char speed);
void DriveForward2(char address, char speed);
void DriveForwardMixed(char address, char speed);
void TurnLeftMixed(char address, char speed);
void TurnRightMixed(char address, char speed);

#endif //Motor Control Packetized Serial
```

MC_packserial.a

Includes source files:

DriveBackward1.c

```
//Jonathan Peterson
#include <xc.h>
#include <stdio.h>

void DriveBackward1(char address, char speed)
{
        putu(address);
        putu(1); //drive motor 1 backwards command
        putu(speed);
        putu((address + 1 + speed) & 0b01111111);
}
```

DriveBackward2.c

```
//Jonathan Peterson
#include <xc.h>
#include <stdio.h>

void DriveBackward2(char address, char speed)
{
        putu(address);
```

```
        putu(5); //drive motor 1 backwards command
         putu(speed);
        putu((address + 5 + speed) & 0b01111111);
}
```

DriveForward1.c

```
//Jonathan Peterson
#include <xc.h>
#include <stdio.h>

void DriveForward1(char address, char speed)
{
        putu(address);
        putu(0); //drive motor 1 backwards command
         putu(speed);
        putu((address + 0 + speed) & 0b01111111);
}
```

DriveForward2.c

```
//Jonathan Peterson
#include <xc.h>
#include <stdio.h>

void DriveForward2(char address, char speed)
{
        putu(address);
        putu(4); //drive motor 1 backwards command
         putu(speed);
        putu((address + 4 + speed) & 0b01111111);
}
```

DriveBackwardMixed.c

```
//Jonathan Peterson
#include <xc.h>
#include <stdio.h>

void DriveBackwardMixed(char address, char speed)
{
        putu(address);
        putu(9); //drive motor 1 backwards command
         putu(speed);
        putu((address + 9 + speed) & 0b01111111);
}
```

DriveForwardMixed.c

```
//Jonathan Peterson
#include <xc.h>
#include <stdio.h>

void DriveForwardMixed(char address, char speed)
{
        putu(address);
        putu(8); //drive motor 1 backwards command
         putu(speed);
        putu((address + 8 + speed) & 0b01111111);
}
```

TurnLeftMixed.c

```
//Jonathan Peterson
#include <xc.h>
#include <stdio.h>

void TurnLeftMixed(char address, char speed)
{
        putu(address);
        putu(11); //drive motor 1 backwards command
         putu(speed);
        putu((address + 11 + speed) & 0b01111111);
}
```

TurnRightMixed.c

```
//Jonathan Peterson
#include <xc.h>
#include <stdio.h>

void TurnRightMixed(char address, char speed)
{
        putu(address);
        putu(10); //drive motor 1 backwards command
         putu(speed);
        putu((address + 10 + speed) & 0b01111111);
}
```

Appendix 10: E-Compass I2C Functions

```
*
 * File:   EC_I2C.h
 * Author: Jonathan Peterson
 *
 * PIC32MX795 I2C1 Output
```

```
 */

#ifndef EC_I2C_H
#define          EC_I2C_H
#include <xc.h>
#include "sourcecode/configbits.h"

/*
 using internal FRC (80 MHz)
 peripheral clock = at 40 MHz (80 MHz/8)
 */



void I2C1_init(unsigned long rate, unsigned long pb_clk) {

        AD1PCFG = 0x1111;
        DDPCONbits.JTAGEN = 0;
        I2C1CONbits.ON = 0;
        I2C1BRG = (pb_clk()/(2*rate)) -
        (pb_clk()/10000000UL) - 2;
        I2C1CONbits.DISSLW = 1;
        if(rate == 400000ul)I2C1CONbits.DISSLW = 0;
        I2C1CONbits.ON = 1;
        while(!I2C1CONbits.ON)
        return;
}

void I2C_start(void) {
        IFS0bits.I2C1MIF = 0; // SSP intflag
        I2C1CONbits.SEN = 1; // send "start bit"
        while(!IFS0bits.I2C1MIF);  // wait for flag set
        return; //success
}

void I2C_restart (void) {
        IFS0bits.I2C1MIF = 0; // clear master inter
        I2C1CONbits.RSEN = 1; // send "start bit"
        while(!IFS0bits.I2C1MIF);  // wait for flag set
        return;
}

void I2C_stop (void) {
        IFS0bits.I2C1MIF = 0; // master intflag
        I2C1CONbits.PEN = 1;
```

```
	while(!IFS0bits.I2C1MIF);  // wait for flag set
	return;
}

#define i2c_ack_not_recv I2C1STATbits.ACKSTAT  // 1 = ack not recieved
unsigned char I2C_write (unsigned char data) {
	unsigned char result;
	IFS0bits.I2C1MIF = 0; // SSP intflag
	I2C1TRN = data; // send the data
	while(!IFS0bits.I2C1MIF);
	if (i2c_ack_not_recv)
		result = -1; // no ack
	else
		result = 0; // ack
	return result;

}

#define i2c_ack  I2C1CONbits.ACKEN  // master send of ackor not
#define i2c_nack I2C1CONbits.ACKDT  // bit that is sent by ack(1 = NAK!)
void I2C_ack (void) {
	IFS0bits.I2C1MIF = 0;
	i2c_ack = 1; // enable acksequence to go
	while(!IFS0bits.I2C1MIF);  // wait for flag set
	return;
}

unsigned char I2C_read (unsigned char ack) {
	unsigned char data;
	IFS0bits.I2C1MIF = 0; // SSP intflag
	I2C1CONbits.RCEN = 1; // set receive enable
	while(!IFS0bits.I2C1MIF);  // wait for flag set
	data = I2C1RCV; // get data
	if (ack)
		i2c_nack = 0; // 0= here means send ack
	else
		i2c_nack = 1; // 1 means don't  ack
	 /* do the ack*/
	I2C_ack();
	return data;

 }

unsigned char I2C_ec_read_byte (unsigned char dev, unsigned char reg) {
	unsigned char ack;
```

```
	unsigned char data;
	I2C_start();
	I2C_write(dev<<1);
	I2C_write(reg);
	I2C_restart();
	ack = I2C_write(dev<<1 | 1);
	data = I2C_read(0);
	if (ack == -1 ){
		return -1; //error
	}
	I2C_stop();
	return data;
}

unsigned char I2C_ec_write_byte (unsigned char dev, unsigned char reg, unsigned char data) {
	unsigned char ack;
	I2C_start();
	ack = I2C_write(dev<<1);
	I2C_write(reg);
	ack = I2C_write(data);
	if((ack == -1) | (I2C1STATbits.IWCOL))/* If write collision occurs,return -1 */
	{
		I2C_stop();
		return -1; //error
	}
	else
	{
		I2C_stop();
		return 0;
	}
}

unsigned char I2C_ec_read_multi (unsigned char dev, unsigned char reg, int read_len, unsigned
char *data)
{
	unsigned char ack;
	//unsigned char * ptr = data;
	unsigned char result;
	I2C_start();
	ack = I2C_write(dev<<1);
	ack = I2C_write(reg);
	I2C_restart();
	ack = I2C_write(dev<<1 | 1);
	if (ack == -1 ){
		return -1; //error
```

```
	}
	int i;
	for (i=1; i < read_len; i++) {
		result = I2C_read(1);
		*data = result;
		data++;
	}
	*data = I2C_read(0);
	I2C_stop();
	return 0;
}

unsigned char I2C_ec_write_multi (unsigned char dev, unsigned char reg, int write_len,
unsigned char *data)
{
	unsigned char ack;
	I2C_start();
	I2C_write(dev<<1);
	I2C_write(reg);
	int i;
	for (i=1; i < write_len; i++) {
		ack = I2C_write(*data);
		if(ack == -1)
		{
		return -1;
		}
		else
		{
		while(I2C1STATbits.TRSTAT); //Wait till data is transmitted.
		data++;
		}
	}
	ack = I2C_write(*data);
	I2C_stop();
	return 0;
}

#endif  /* EC_I2C_H */
```

Appendix 11: E-Compass Software Driver

```
/*
 * File:   ec_driver.c
 * Author: Jonathan Peterson
 *
```

```
 * Created on February 19, 2020, 9:31 PM
 */

#include <stdio.h>
#include <stdlib.h>
#include <xc.h>
#include "sourcecode/configbits.h"
#include "EC_I2C.h"



//I2C
#define I2C_ERROR -1
#define I2C_OK 0

// FXOS8700CQ I2C address
#define FXOS8700CQ_SLAVE_ADDR 0x1E // with pins SA0=0, SA1=0

// FXOS8700CQ internal register addresses
#define FXOS8700CQ_STATUS 0x00
#define FXOS8700CQ_WHOAMI 0x0D
#define FXOS8700CQ_XYZ_DATA_CFG 0x0E
#define FXOS8700CQ_CTRL_REG1 0x2A
#define FXOS8700CQ_M_CTRL_REG1 0x5B
#define FXOS8700CQ_M_CTRL_REG2 0x5C
#define FXOS8700CQ_WHOAMI_VAL 0xC7

// number of bytes to be read from the FXOS8700CQ
#define FXOS8700CQ_READ_LEN 13 // status plus 6 channels = 13 bytes

typedef struct
{
          int16_t x;
          int16_t y;
          int16_t z;
}SRAWDATA;

unsigned char accel_mag_config();
int16_t ReadAccelMagnData (SRAWDATA *pAccelData, SRAWDATA *pMagnData);


int main(int argc, char** argv) {

          I2C1_init(400000UL); //initialize
```

```
        //Configure E-Compass
        unsigned char result;
        result = accel_mag_config();

        //Read out Data
        SRAWDATA pAccelData;
        SRAWDATA pMagnData;

        SRAWDATA *accel_ptr = &pAccelData;
        SRAWDATA *mag_ptr = &pMagnData;

        int16_t result2;

        result2 = ReadAccelMagnData(accel_ptr, mag_ptr);

        while(1);

        return (EXIT_SUCCESS);
}

// function configures FXOS8700CQ combination accelerometer and magnetometer sensor
unsigned char accel_mag_config() {
        unsigned char databyte;
        // read and check the FXOS8700CQ WHOAMI register
        databyte = I2C_ec_read_byte(FXOS8700CQ_SLAVE_ADDR,
        FXOS8700CQ_WHOAMI);
        if (databyte != FXOS8700CQ_WHOAMI_VAL)
        {
                return (I2C_ERROR);
        }
        /* write 0000 0000 = 0x00 to accelerometer control register 1 to
        * place FXOS8700CQ into standby
        *  [7-1] = 0000 000
        * [0]: active=0 */
        unsigned char write_result;
        databyte = 0x00;
        write_result = I2C_ec_write_byte(FXOS8700CQ_SLAVE_ADDR,
FXOS8700CQ_CTRL_REG1,databyte);
        if (write_result == -1)
        {
                return (I2C_ERROR);
        }
```

```
 // write 1001 1111 = 0x1F to magnetometer control register 1
// [7]: m_acal=1: auto calibration enabled
// [6]: m_rst=0: no one-shot magnetic reset
// [5]: m_ost=0: no one-shot magnetic measurement
// [4-2]: m_os=111=7: 8x oversampling (for 200Hz) to reduce
//magnetometer noise
// [1-0]: m_hms=11=3: select hybrid mode with accel and
//magnetometer active
databyte = 0x9F;
write_result = I2C_ec_write_byte(FXOS8700CQ_SLAVE_ADDR,
FXOS8700CQ_M_CTRL_REG1,databyte);
if (write_result == -1)
{
        return (I2C_ERROR);
}
// write 0010 0000 = 0x20 to magnetometer control register 2
// [7]: reserved
// [6]: reserved
// [5]: hyb_autoinc_mode=1 to map the magnetometer registers to follow the
// accelerometer registers
// [4]: m_maxmin_dis=0 to retain default min/max latching even
// though not used
// [3]: m_maxmin_dis_ths=0
// [2]: m_maxmin_rst=0
// [1-0]: m_rst_cnt=00 to enable magnetic reset each cycle
databyte = 0x20;
write_result = I2C_ec_write_byte(FXOS8700CQ_SLAVE_ADDR,
         FXOS8700CQ_M_CTRL_REG2,databyte);
if (write_result == -1)
{
        return (I2C_ERROR);
}
// write 0000 0001= 0x01 to XYZ_DATA_CFG register
// [7]: reserved
// [6]: reserved
// [5]: reserved
// [4]: hpf_out=0
// [3]: reserved
// [2]: reserved
// [1-0]: fs=01 for accelerometer range of +/-4g range with 0.488mg/LSB
databyte = 0x01;
write_result = I2C_ec_write_byte(FXOS8700CQ_SLAVE_ADDR,
FXOS8700CQ_XYZ_DATA_CFG,databyte);
if (write_result == -1)
{
```

```
                return (I2C_ERROR);
        }
        // write 0000 1101 = 0x0D to accelerometer control register 1
        // [7-6]: aslp_rate=00
        // [5-3]: dr=001 for 200Hz data rate (when in hybrid mode)
        // [2]: lnoise=1 for low noise mode
        // [1]: f_read=0 for normal 16 bit reads
        // [0]: active=1 to take the part out of standby and enable sampling
        databyte = 0x0D;
        write_result = I2C_ec_write_byte(FXOS8700CQ_SLAVE_ADDR,
FXOS8700CQ_CTRL_REG1,databyte);
        if (write_result == -1)
        {
                return (I2C_ERROR);
        }
        //normal return
        return (I2C_OK);
}

// read status and the three channels of accelerometer and
//magnetometer data from
// FXOS8700CQ (13 bytes)

int16_t ReadAccelMagnData (SRAWDATA *pAccelData, SRAWDATA *pMagnData)
{
        uint8_t Buffer[FXOS8700CQ_READ_LEN]; // read buffer
        // read FXOS8700CQ_READ_LEN=13 bytes (status byte and the six
        // channels of data)
        unsigned char *rdptr;
        rdptr = Buffer;
        unsigned char read_result;
        read_result = I2C_ec_read_multi(FXOS8700CQ_SLAVE_ADDR,
                FXOS8700CQ_STATUS, FXOS8700CQ_READ_LEN,rdptr);
        if (read_result == 0)
        {
                // copy the 14 bit accelerometer byte data into 16 bit words
                pAccelData->x = (int16_t)(((Buffer[1] << 8) | Buffer[2]))>> 2;
                pAccelData->y = (int16_t)(((Buffer[3] << 8) | Buffer[4]))>> 2;
                pAccelData->z = (int16_t)(((Buffer[5] << 8) | Buffer[6]))>> 2;
                // copy the magnetometer byte data into 16 bit words
                pMagnData->x = (Buffer[7] << 8) | Buffer[8];
                pMagnData->y = (Buffer[9] << 8) | Buffer[10];
                pMagnData->z = (Buffer[11] << 8) | Buffer[12];
        }
        else
```

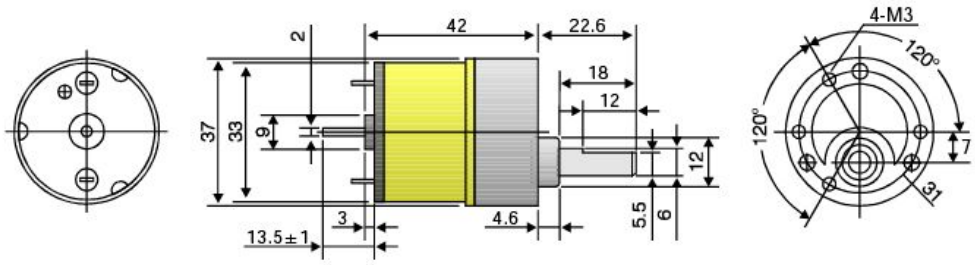
```
        {
                // return with error
                return (I2C_ERROR);
        }
        // normal return
        return (I2C_OK);
}
```

Appendix 12: Motor Data Sheet

Data sheet for:
GHM-16
12vdc 30:1 200rpm
6mm shaft

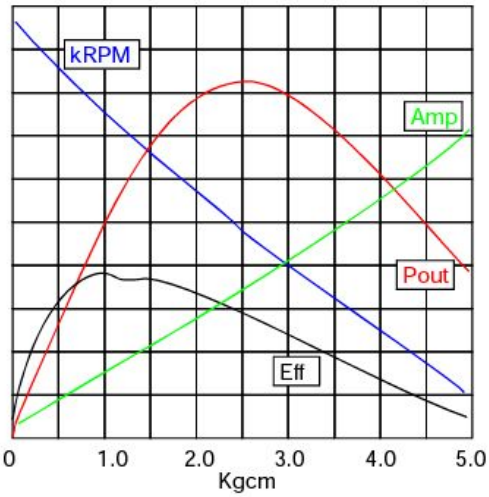
www.lynxmotion.com

I. OUTER DIMENSIONS



II. DRAWING OF CURVES

Pout	Amp	Eff	kRPM
3.0	2.0	1.0	0.20
2.7	1.8	0.9	0.18
2.4	1.6	0.8	0.16
2.1	1.4	0.7	0.14
1.8	1.2	0.6	0.12
1.5	1.0	0.5	0.10
1.2	0.8	0.4	0.08
0.9	0.6	0.3	0.06
0.6	0.4	0.2	0.04
0.3	0.2	0.1	0.02
0.0	0.0	0.0	0.00



III. SPECIFICATIONS

Type: HN-GH35GB
Model: HN-GH12-1634T - 30:1

1. Testing Conditions:
   Temp: 25° Celsius
   Humidity: 60%
   Motor Orientation: Horizontal
2. Rated Voltage: 12vdc
3. Voltage Operating Range: 6-12vdc
4. Rated Load at 12vdc: 0.78Kg-cm
   Do not exceed rated load. Damage may occur!
5. No Load Speed at 12vdc: 200 RPM +/- 10%
6. Speed at Rated Load (0.78Kg-cm): 163 RPM +/- 10%
7. No Load Current at 12vdc: < 115mA
8. Current at Rated Load (0.78Kg-cm): < 285mA
9. Shaft End-Play: Maximum 0.8m/m
10. Insulation Resistance: 10M ohm at 300vdc
11. Withstand Voltage: 300vdc for 1 Second
12. The gear motor is not intended for instant reverse. The gear motor must be stopped before reversing.
13. The gear motor does not include protection from water or dust etc.

Appendix 13: PIC32MX Configuration Bits

```
/*
 * File:   configbits.h
 * Author: Jonathan Peterson
 * Date: February 2019
 * VaLET PIC32MX MCU
 */

#ifndef CONFIGBITS_H
#define          CONFIGBITS_H

/*
 using internal FRC (80 MHz)
 peripheral clock = at 40 MHz (80 MHz/8)
 */

#pragma config FNOSC = FRCPLL // Oscillator selection, FRC + PLL
#pragma config POSCMOD = OFF // Primary oscillator mode
#pragma config FPLLIDIV = DIV_2 // PLL input divider (8 -> 4)
#pragma config FPLLMUL = MUL_20 // PLL multiplier  ( 4x20 = 80)
#pragma config FPLLODIV = DIV_1 // PLL output divider (80/1 = 80MHz)
#pragma config FPBDIV = DIV_2 // Peripheral bus clock divider (80/2 = 40 mhz)
#pragma config FSOSCEN = OFF // Secondary oscillator enable
/* Clock control settings
*/
#pragma config IESO = OFF // Internal/external clock switchover
#pragma config FCKSM = CSECME // Clock switching (CSx)/Clock monitor (CMx)
#pragma config OSCIOFNC = ON // Clock output on OSCO pin enable
/* USB Settings
*/
//#pragma config UPLLEN = OFF // USB PLL enable
//#pragma config UPLLIDIV = DIV_2 // USB PLL input divider
//#pragma config FVBUSONIO = OFF // VBUS pin control
//#pragma config FUSBIDIO = OFF // USBID pin control
/* Other Peripheral Device settings
*/
#pragma config FWDTEN = OFF // Watchdog timer enable
#pragma config WDTPS = PS4096 // Watchdog timer post-scaler
#pragma config FSRSSEL = PRIORITY_7 // SRS interrupt priority
#pragma config DEBUG = ON

#pragma config          ICESEL  = ICS_PGx1      // ICE pin selection
#endif  /* CONFIGBITS_H */
```

Appendix 14: 3.3V MCU Board Regulator

LD1117 Data Sheet

Appendix 15: Python Functions for Vision System

```
### subsystem_PiCam.py
### VALET's subsystem demo for the end of Semester 1 in University of Notre Dame's EE 41430: Senior Design Class
### Code designed to detect QR Code for final handoff of VALET delivery between robot and customer, as well as detection major obstructions
### Major obstruction detection used as a redundant sensor for our distance sensor. Operates by taking the standard deviation of all pixel values on screen
###     Author: Alden Kane

import cv2 as cv2
from pyzbar import pyzbar

def get_Camera():
  # Get video stream from Webcam
  cam = cv2.VideoCapture(0)

  # Read from camera
  retval, img = cam.read()

  # Rescale if too large
  res_scale = 0.5
  img = cv2.resize(img, (0, 0), fx=res_scale, fy=res_scale)

  # Scope Variables
  barcodeData = "NULL"

  ################################################
  ### Section 2: QR Code Recognition
  ################################################
  # Find + decode barcodes
  barcodes = pyzbar.decode(img)

  # Iterate over barcodes
  for barcode in barcodes:
    # Get bounding box
    (x, y, w, h) = barcode.rect

    # Get information on barcodes
    barcodeData = barcode.data.decode("utf-8")
    barcodeType = barcode.type

    # Only interested in QRCODE recognition
```

```
        if (barcodeType == 'QRCODE'):
            # Draw bounding box, and put text next to it
            cv2.rectangle(img, (x, y), (x + w, y + h), (0, 0, 255), 2)
            text = "{} ({})".format(barcodeData, barcodeType)
            cv2.putText(img, text, (x, y - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)

    ################################################
    ### Section 3: Obstruction Detection
    ################################################
    # Calculate mean and standard deviation of image
    mean, std = cv2.meanStdDev(img)

    # Obstruction Calculation
    if std[0] < 30 or std[1] < 30 or std[2] < 30:
        obstruction = 1
    else:
        obstruction = 0

    obstruction_text = "Obstruction Value ({})".format(obstruction)

    # Put Text
    cv2.putText(img, str(mean), (20, 20), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)
    cv2.putText(img, str(std), (20, 40), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)
    cv2.putText(img, obstruction_text, (20, 60), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)

    # Imshow for GUI Debug
    cv2.imshow("VALET", img)
    key = cv2.waitKey(3000)

    return obstruction, barcodeData

def get_Camera_Headless():
    # Get video stream from Webcam
    cam = cv2.VideoCapture(0)

    # Read from camera
    retval, img = cam.read()

    # Rescale if too large
    res_scale = 0.5
    img = cv2.resize(img, (0, 0), fx=res_scale, fy=res_scale)

    # Scope Variables
    barcodeData = "NULL"

    ################################################
    ### Section 2: QR Code Recognition
    ################################################
    # Find + decode barcodes
    barcodes = pyzbar.decode(img)

    # Iterate over barcodes
    for barcode in barcodes:
```

```
        # Get bounding box
        (x, y, w, h) = barcode.rect

        # Get information on barcodes
        barcodeData = barcode.data.decode("utf-8")
        barcodeType = barcode.type

    ################################################
    ### Section 3: Obstruction Detection
    ################################################
    # Calculate mean and standard deviation of image
    mean, std = cv2.meanStdDev(img)

    # Obstruction Calculation
    if std[0] < 30 or std[1] < 30 or std[2] < 30:
        obstruction = 1
    else:
        obstruction = 0

    return obstruction, barcodeData

### get_LiDAR.py
### Code to fetch distance to object in FOV of Garmin LiDAR Lite v4
###     Author: Alden Kane

# Improvements
    # Return a flag that says whether or not a distance was picked up
    # Get this to not be so buggy

import smbus

# distance_LiDAR_1 = 'No Value Yet'

def get_LiDAR():
    # Declare Globals and Device Parameters
    CHANNEL = 1
    DEVICE_ADDRESS = 0x62

    ACQ_COMMANDS = 0x00
    STATUS = 0x01
    FULL_DELAY_LOW = 0x10
    FULL_DELAY_HIGH = 0X11

    RESET_BYTE = 0x00
    BIASED_DISTANCE = 0X04

    # Initialize Global
    # global distance_LiDAR_1

    # Declare Bus w/ SMbus, Intialize I2C
    bus = smbus.SMBus(CHANNEL)

    # Initialize LiDAR by Writing to Correction Distance Mode to ACQ Register
    bus.write_byte_data(DEVICE_ADDRESS, ACQ_COMMANDS, BIASED_DISTANCE)
```

```
    # Read the STATUS Register
    r_1 = bus.read_byte_data(DEVICE_ADDRESS, STATUS)

    # Check to see if distance can be read from device
    if r_1 == 1:
        # Had a try statement here
        distance_LiDAR_1 = bus.read_byte_data(DEVICE_ADDRESS, FULL_DELAY_LOW)
        #distance_LiDAR_2 = bus.read_i2c_block_data(DEVICE_ADDRESS, FULL_DELAY_LOW, 2)
    else:
        distance_LiDAR_1 = 'Null - No Signal Acquired'

    return distance_LiDAR_1
```

Appendix 16: Raspberry Pi Model 3b+ Datasheet

Raspberry Pi Model 3b+ Datasheet

Appendix 17: Garmin LiDAR Lite v4 Datasheet

Garmin LiDAR Lite v4 Datasheet

Appendix 18: Raspberry Pi Pinout

Interactive Raspberry Pi Pinout

Appendix 19: Logitech c615 Webcam

Logitech c615 Webcam

Appendix 20: Python Functions for Wi-Fi System

```
# user_Info_From_JSON_Online.py
# A script to parse GPS and user info from an online JSON file
# For Notre Dame EE Senior Design 2020 - VALET
#  Author: Alden Kane

import urllib
import requests

def get_User_Info(target):
    # Declare Target URL for User Info
```

```
  target_url = target

  # Ensure We Can Access URL, Error Handling
  try:
     urllib.request.urlopen(target_url)
  except:
     exit()

  # Fetch JSON Using Requests
  data = requests.get(target_url).json()

  # Parse Data
  firstName = data["firstName"]
  lastName = data["lastName"]
  userKey = data["userKey"]
  lat = data["latitude"]
  long = data["longitude"]

  return firstName, lastName, userKey, lat, long
```

Appendix 21: Python System File to Run Vision and Wi-Fi Subsystems on Raspberry Pi

```
system.py
# VALET's Raspberry Pi System, with Functions Imported
# Combines functions of:
  # Fetching JSON from URL
  # Reading Data from PiCamera for QR Code
  # Obstruction Detection
  # Accessing Lidar
  # Putting Data over SPI
# Future Improvements, as of 03/05/2020 at 2:26 AM
  # Tune up LiDAR for More Consistent Functionality
  # Putting Data over SPI
  # More advanced camera functions
  # User matching functions
  # Moving average for LiDAR
  # Timeout functions!!!
# Author: Alden Kane

##########################################
# Section 1: Import Libraries and Functions, Declare Globals
##########################################

from get_User_Info import get_User_Info
from get_Camera import get_Camera_Headless
from get_LiDAR import get_LiDAR

target_URL = 'http://seniordesign.ee.nd.edu/2020/Design%20Teams/valet/users.json'
distance_LiDAR_1 = 0

##########################################
```

```
# Section 2: Call One-Time Functions
##########################################
firstName, lastName, userKey, lat, long = get_User_Info(target_URL)

while True:
  ##########################################
  # Section 3: Call Repeat Read Functions
  ##########################################
  obstruction, barcodeData = get_Camera_Headless()
  distance_LiDAR_1 = get_LiDAR()

  ##########################################
  # Section 4: Print for Debug
  ##########################################
  # Print for Debug
  print("system.py Block")
  print("--------------------")
  print("Web Fetched User Key: " + str(userKey))
  print("Target Delivery Latitude: " + lat)
  print("Target Delivery Longitude: " + long)
  print("QR Encoded userKey: " + barcodeData)
  print("Vision Based Obstruction Found: " + str(obstruction))
  print("LiDAR Detection Distance: " + str(distance_LiDAR_1) + " cm")
```

Appendix 22: Link to GitHub Repository w/ all Code for Raspberry Pi, Vision, and Wi-Fi Systems

Link to GitHub Repository