RAPID BATTERY EXCHANGE AUTOMATION

BY
ANDREA EVERSON

SENIOR PROJECT

COMPUTER ENGINEERING DEPARTMENT
CALIFORNIA POLYTECHNIC STATE UNIVERSITY
SAN LUIS OBISPO

June, 2016

TABLE OF CONTENTS

Section .... Page

TABLE OF CONTENTS 2
- List of Figures 3
- List of Tables 3

Abstract 5

Introduction 6

Chapter 2 7
- 2.1 Customer Needs Assessment 7
- 2.2 Requirements and Specifications 7

Functional Decomposition 9
- 3.1 Level 0 Function Decomposition 9
- 3.2 Level 1 Function Decomposition 10
- 3.3 Level 2 Function Decomposition 11

Design 14
- Check Inputs Function 17
- Check Serial Function 21

References 23

Appendix A: Analysis of Senior Project Design 24

Appendix B: Error Codes 28

Appendix C: Ramp Code 30
- Ramp 30
- Van 44

Appendix D: Pin Out 56

List of Figures

Figure 1: Level 0 Block Diagram For Rapid Battery Exchange System.......................... 9

Figure 2: Level 1 Block Diagram For Rapid Battery Exchange System........................ 10

Figure 3: Level 2 Block Diagram For Rapid Battery Exchange System........................ 11

Figure 4: Finite State Machine (FSM) for Ramp System.. 15

Figure 5: Finite State Machine (FSM) for Van System.. 16

Figure 6: The Ramp CheckInputs Byte Representation... 17

Figure 7: The Van CheckInputs Byte Representation... 18

List of Tables

Table 1: Rapid Battery Exchange System requirements and specifications [3]............. 7

Table 2: Rapid Battery Exchange Deliverables [3].. 8

Table 3: Level 0 Functionality Table Of Rapid Battery Exchange System.................... 9

Table 4: Level 1 Functionality Table Of Rapid Battery Exchange System................. 10

Table 5: Level 2 Functionality Table Of Rapid Battery Exchange System................. 11

Table 6: Ramp Input Constants and Their Values... 17

Table 7: Van Input Constants and Their Values... 18

Table 8: Valid Inputs for the States in the Van's System.. 19

Table 9: Valid Inputs for the States in the Ramp's System... 20

Table 10: XBee Signals Ramp Sends to Van.. 21

Table 11: XBee Signals Van Sends to Ramp.. 22

Table 12: Original Estimated Total Component Costs.. 25

Table 13: Actual Total Component Costs.. 25

Table 14: Error Codes for Ramp.. 28

Table 15: Error Codes for Van.. 29

TABLE 16: RAMP PIN ASSIGNMENTS FOR ATMEGA32U4 56

TABLE 17: VAN PIN ASSIGNMENTS FOR ATMEGA32U4 57

Abstract

With the drastic threat of exhausting the world's fossil fuels, society has been scrambling to find alternative forms of energy for transportation. One of these viable options is an electric vehicle powered by batteries. While this idea is fairly viable, consumers are looking to have the same experience with an electric car than they would with a gas-powered vehicle. One of their major concerns is the lack of range in an electric vehicle.

This project aims to present a solution that would extend the range by allowing users to quickly go from low battery to fully charged battery by performing a battery exchange. When an electric vehicle needs to be recharged, the driver would simply drive up onto the ramp, and initiate the exchange. The total exchange time for an exchange is around 60 seconds, allowing the driver to get back on the road in a shorter time than required to fill up a car with gas.

Introduction

While business is booming for the electric automobile industry, many consumers are frightened about the range limit on the vehicle. This is a major concern since the average fully electric vehicle can only go 60 to 120 miles before needing a charge and the charge may take anywhere from 4 to 8 hours to get back to 100% [1]. The car with the longest range is the Tesla Model S 85D which offers a much longer range of 270 miles, but at a tag price of over $75,000 [2].

This project aims to solve this problem by allowing electric vehicles to extend their range by swapping batteries when needed. So instead of having to find a particular place to go in order to charge the car, consumers could simply drive into a gas station where this system is operational and exchange the battery faster than the time it takes to fill up with gas. This system is called the Rapid Battery Exchange (RBX) system and has been a a club project for Cal Poly's Electric Vehicle Engineering Club (EVEC) for the last 15 years. My specific portion of this project is to automate and ensure at each state the system is in a valid and safe state.

This project consists of 2 separate systems, a GMC electric G-van and a ramp, which communicate though Xbee wireless communication. The G-van has been modified from it's original state in order with a retrofitted battery pack chassis that allows quick and easy access to the battery pack. In order to implement this system on other types of vehicles on the market today, the company would need to implement a standard battery pack that allows for easy access.

This report will explain the project, discuss the wireless communication between the systems and discuss how valid states and inputs are determined. In addition, the flow of the system will be explained with flow charts of both systems.

Chapter 2

2.1 Customer Needs Assessment

The consumer of this product would be an electric vehicle owner hoping to extend the driving range of their vehicle without also extending their travel time. Table 1 below shows the list of the customer needs.

2.2 Requirements and Specifications

TABLE 1:
RAPID BATTERY EXCHANGE SYSTEM REQUIREMENTS AND SPECIFICATIONS [3]

Marketing Requirements	Engineering Specifications	Justification
1, 3	Should complete an exchange within 1 – 2 minutes.	Exchanging the battery pack is similar to refueling a car, so the time requirement should be about equivalent.
3-5	Should provide easy to understand documentation in case of error.	Providing this documentation is beneficial when debugging system or implementing additional features.
3,5	Should provide easy to understand instructions for operability.	Normal individuals would be using the system, and therefore it must be straightforward and intuitive. Providing flashing buttons with labels would be sufficient enough.
2, 4-5	Should provide a manual mode that allows to re-adjust system when alignment is off or an error has occurred.	By integrating the option for manual mode, if a user sees a dangerous situation arising, they would have the change to quickly and safely stop and reinitialize the system. Also, this feature will help when adding additional features or testing automation code.
1,3,5	Should constantly check for unsafe or unexpected situations. Should go into an error state and stop all movement.	Due to the fact each battery pack holds 216 volts and is lifted up, accidents involving the lift and carts could be disastrous.
1-3	Should be fully automated from start of exchange (after driver hits start button).	This project requires the exchange be fully automated since this is the main purpose of this project – to easily and quickly swap batteries with no assistance required.

Marketing Requirements

1. Fast
2. Automated
3. Easy Utilization
4. Easy Maintenance
5. Safety

TABLE 2:
RAPID BATTERY EXCHANGE DELIVERABLES [3]

Delivery Date	**Deliverable Description**
2/17/2016	Design Review
4/15/2016	Open House Demo
5/20/2016	Progress Report
5/27/2016	Sr. Project Expo Demo
6/10/2016	CPE 462 Report

Functional Decomposition

3.1 Level 0 Function Decomposition

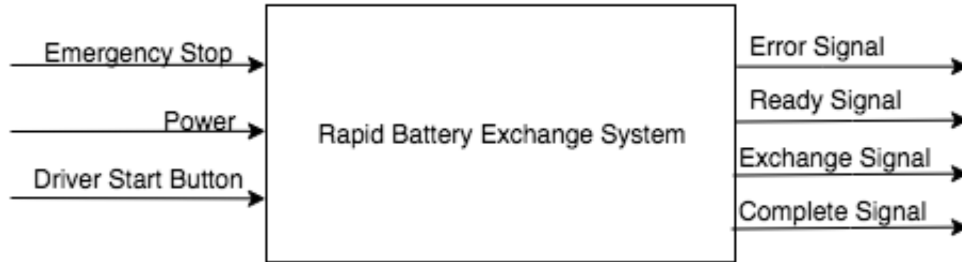

FIGURE 1:
LEVEL 0 BLOCK DIAGRAM FOR RAPID BATTERY EXCHANGE SYSTEM

TABLE 3:
LEVEL 0 FUNCTIONALITY TABLE OF RAPID BATTERY EXCHANGE SYSTEM

Module	Rapid Battery Exchange
Input	• Emergency Stop Button • Power • Driver Start Button
Output	• Error Signal • Ready Signal • Exchange Signal • Complete Signal
Function	Reports the state of the automated system to the user. This allows the user to know when to drive off the ramp since the complete signal will be on. The user, however, doesn't need to see any of the internal communication signals between the van and the ramp.

3.2 Level 1 Function Decomposition

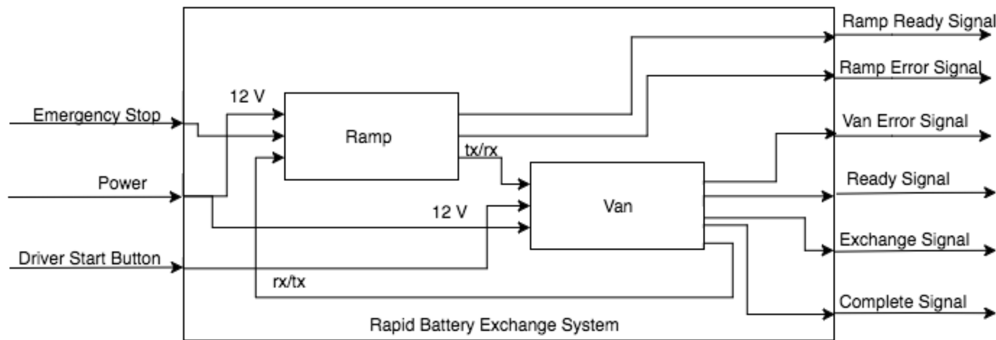

FIGURE 2:
LEVEL 1 BLOCK DIAGRAM FOR RAPID BATTERY EXCHANGE SYSTEM

TABLE 4:
LEVEL 1 FUNCTIONALITY TABLE OF RAPID BATTERY EXCHANGE SYSTEM

Module	Ramp
Input	• Power – 12 V • Emergency Stop • Rx Signal – Receives Van's Xbee signals
Output	• Tx Signal – Sends Xbee signals to the van • Error Signal – Outputs a specific error code[1] based on the ramp's POV. • Ready Signal
Function	Uses inputs to determine when and what to send over the transmission line to the van. These signals would include a signal to notify the van the actuators should be removed or that there has been an error in the ramp, so the van should also go into error mode. Also lights up the ready and error LEDs in appropriate cases.

Module	Van
Input	• Power – 12 V • Driver Start Button • Rx Signal – Received signals from the ramp
Output	• Tx Signal – Sends Xbee signals to the ramp • Error Signal – Outputs a specific error code[1] based on the van's POV. • Ready Signal

[1] Ramp and Van error codes are found in appendix B

	• Exchange Signal • Complete Signal
Function	Uses inputs to determine when and what to send to the ramp over the transmission line. An example of this would be once the ramp has notified the van it will be raising the lift, the van needs to notify the ramp that the lift has reached the top and to stop the lift. Also lights up ready, error, exchange and complete LEDs in appropriate cases.

3.3 Level 2 Function Decomposition

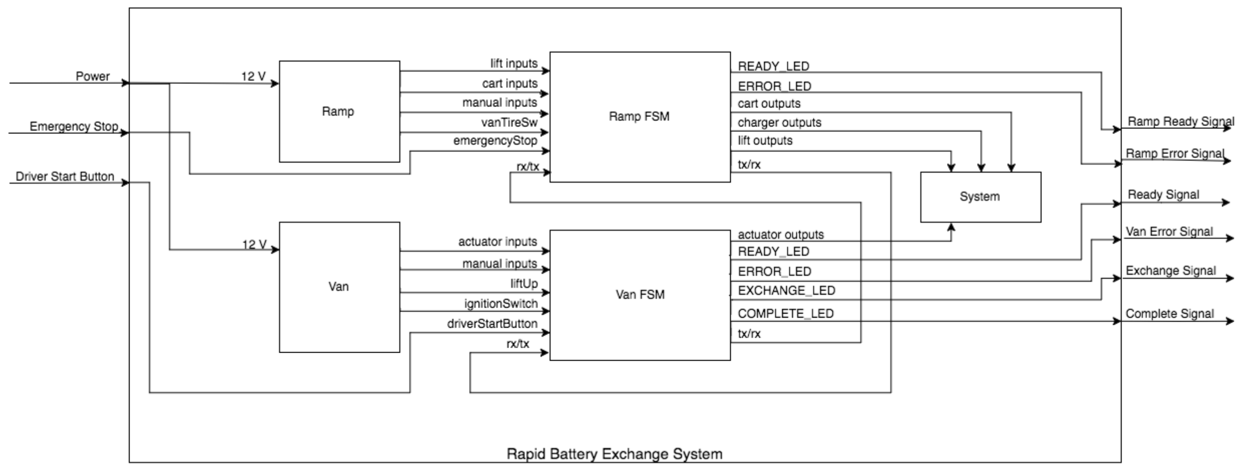

FIGURE 3:
LEVEL 2 BLOCK DIAGRAM FOR RAPID BATTERY EXCHANGE SYSTEM

TABLE 5:
LEVEL 2 FUNCTIONALITY TABLE OF RAPID BATTERY EXCHANGE SYSTEM

Module	Ramp
Input	• Power – 12 V
Output	• Lift inputs • Cart inputs • Manual inputs • Van Tire Switch
Function	This module generates all of the button inputs that will be used to generate the automation code.

Module	Van
Input	• Power – 12 V
Output	• Actuator Inputs • Manual Inputs • Lift Up • Ignition Switch
Function	This module generates all of the button inputs that will be used to generate the automation code. The actuator inputs here have already been analyzed to generate a Boolean value

Module	Ramp FSM
Input	• Lift Inputs • Cart Inputs • Manual Inputs • Van Tire Switch • Emergency Stop • Rx Signal – Receives Van's Xbee signals
Output	• READY_LED • ERROR_LED • Cart Outputs • Charger Outputs • Lift Outputs • Tx Signal – Outputs the transmission signal to send to the Van
Function	This module uses all of the inputs to determine the next state of the finite state machine (FSM). Uses the state to determine the motor and LED outputs and sends this to the ramp system. Also outputs any transmission signals to the van that communicate an important state change.

Module	Van FSM
Input	• Actuator Inputs • Manual Inputs • Lift Up • Ignition Switch • Driver Start Button • Rx Signal – Receives Ramp's Xbee signals
Output	• Actuator Outputs • READY_LED

	• ERROR_LED • EXCHANGE_LED • COMPLETE_LED • Tx Signal – Outputs the transmission signal to send to the Ramp
Function	This module uses all of the inputs to determine the next state of the finite state machine (FSM). Uses the state to determine the motor and LED outputs and sends this to the van system. Also outputs any transmission signals to the ramp that communicate an important state change.

Module	Van FSM
Input	• Actuator Signals • Cart Signals • Lift Signals • Charger Signals
Output	
Function	This module simply represents the physical van and ramp system. The inputs represent the signals going to raise and lower the lift, move the carts forward and back, turn the front and back chargers on and off and move the actuators in and out.

Design

The design for this project consists of two separate systems, the van and the ramp, connected only through Xbee wireless communication. The idea is that once one system can determine the state through their specific inputs, they send a signal through the Xbee in order to communicate that state change to the other system. The van used in this project is a GMC electric G-Van that has been retrofitted with a battery receptacle and actuator pins to allow for quick access to the battery pack.

The overall flow of the system is as follows:

- Van will drive onto ramp until tire button is pressed.
- Ramp will notify the van that the battery exchange is ready.
- The driver hits the button to initiate the battery exchange.
- The lift will lift up the battery pack within the van until no weight remains on the actuator pins.
- The actuators will disengage from the battery pack unlocking the pack from the receptacle.
- The lift will lower the battery pack back into the cart in the ramp.
- The carts will move and the battery pack that was just removed will be placed underneath a charger which will start charging that battery pack.
- Once the battery pack starts charging, the new battery pack will be lifted up into the van receptacle.
- Once at the top of the receptacle, the actuators will engage and lock in the battery pack.
- The lift will slowly lower and put the wait of the battery pack on the actuator pins.
- Once the lift is safely at the bottom of the lift, the exchange is complete and the van is free to drive off the ramp.

While this quickly explains the idea of the project, the Finite State Machines below in Figures 4 and 5 describe the logic flow very clearly. For these diagrams, the legend can be found in the lower right corner detailing what each symbol represents. For example, the pink signals represent the serial inputs that the Xbee radios read in from the other system. Each state is assumed to also link to the error state (not shown for clarity of diagram). Each state has expected inputs that is checked through the *checkCorrect()* function. This function checks the current input, *currInput*, against the correct input, *correctInput*. *correctInput* is a variable that gets changed throughout the FSM in order to reflect the valid inputs. This allows an easy way to check the inputs for validity without having to hardcode numbers. This also makes it easy to determine what the exact error is if the inputs do not match the valid input. The states and their corresponding valid inputs are shown below in Tables 8 and 9. The constants used in these tables can be found in Tables 6 and 7. The function used to obtain these constants is explained below tables 6 and 7.

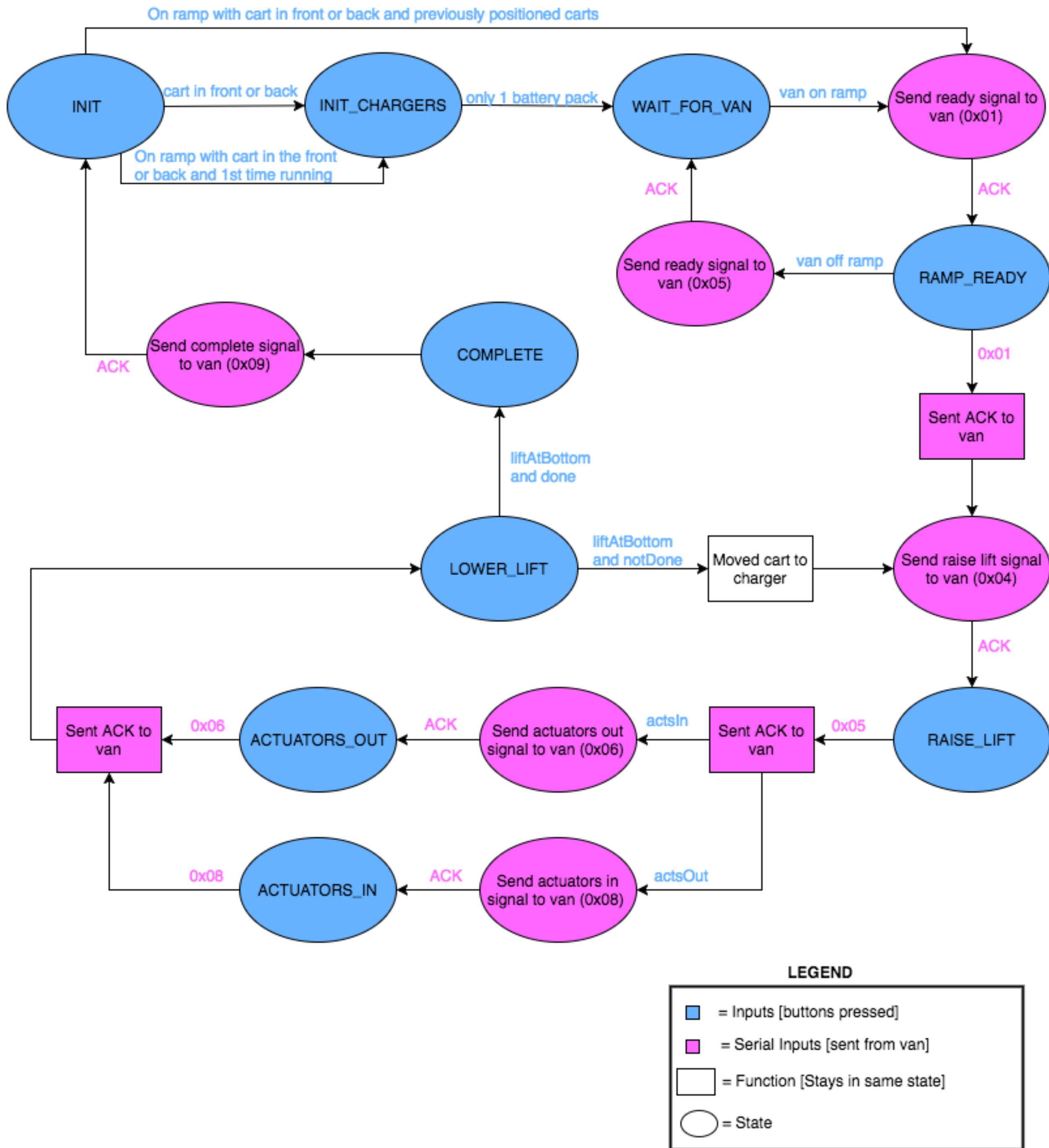

FIGURE 4:
FINITE STATE MACHINE (FSM) FOR RAMP SYSTEM

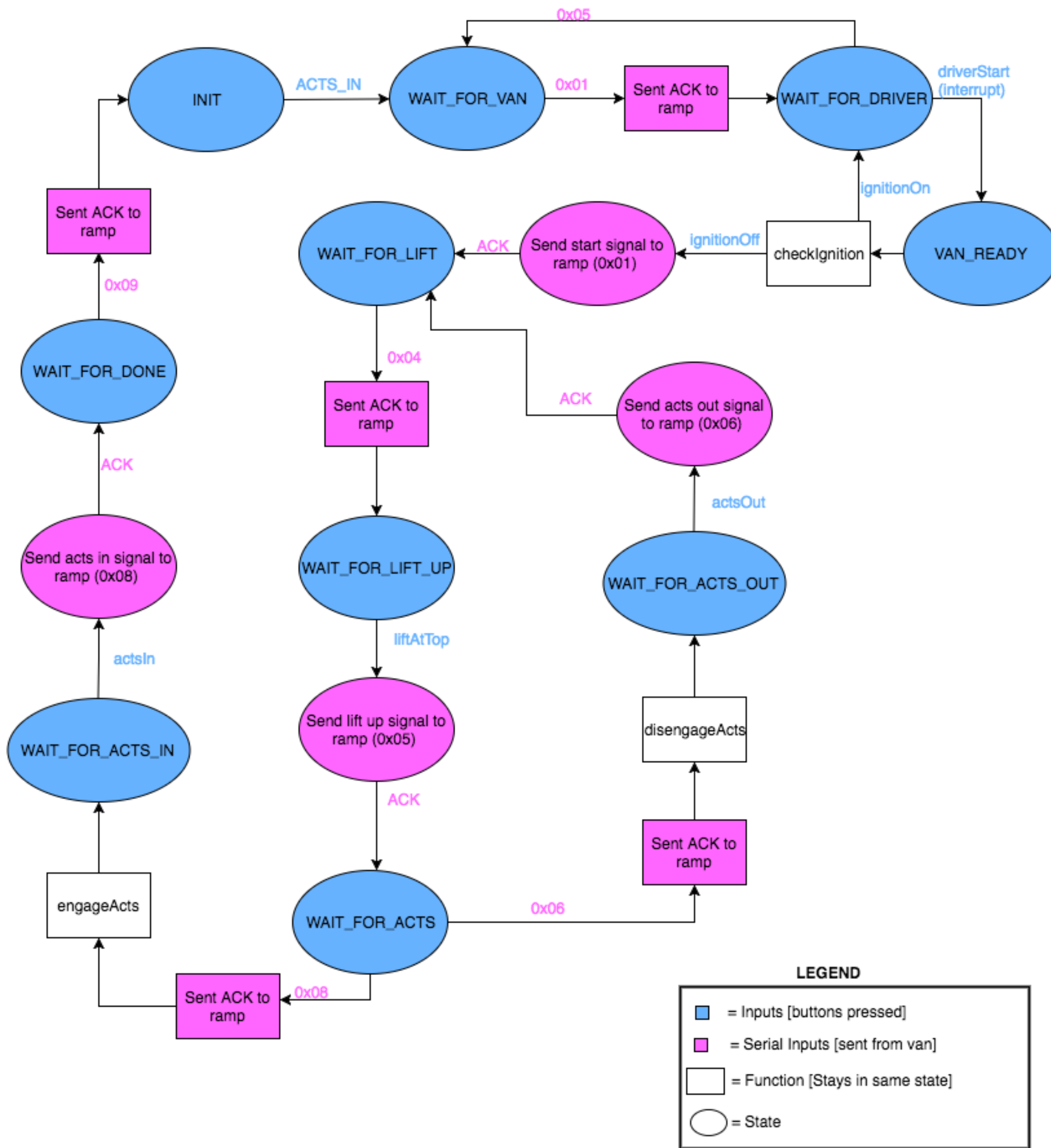

FIGURE 5:
FINITE STATE MACHINE (FSM) FOR VAN SYSTEM

Check Inputs Function

TABLE 6:
RAMP INPUT CONSTANTS AND THEIR VALUES

Constant	Value
CART_FWD	0x04
CART_BACK	0x08
LIFT_DOWN	0x40
VAN_ON	0x80
VAN_OFF_RAMP	0x7F

The constants shown in table 6 were derived from the checkInputs function. The checkInputs function simply takes each input, reads it in as a Boolean (or in the case of the actuators which are analog values, generates a Boolean value by evaluating if the actuators are fully engaged or not). The function then takes this value, offsets it by the position in the byte it is located at and ORs it to the byte. This byte is then returned from the function. For an example, if the cartAtFront button was pressed, and the lift was at the bottom, both inputs would evaluate to true (i.e. 1). The cartAtFront is supposed to end up in bit 2 and would be offset by 2 and would end up as 0x04. The liftAtBottom is supposed to end up in bit 6 and would be offset by 6, ending up as 0x40. These two inputs together would result in an output of 0x44. As shown below, even though the manual inputs manCartFwd, manCartBack, manLiftUp and manLiftDown have a place in the byte, they are all set to 0. Since this is an automated system, manual inputs are not important until an error is encountered and switches to manual mode to give control to the user. In that case, the manInputs function would be used and manual inputs would be evaluated just like the other inputs.

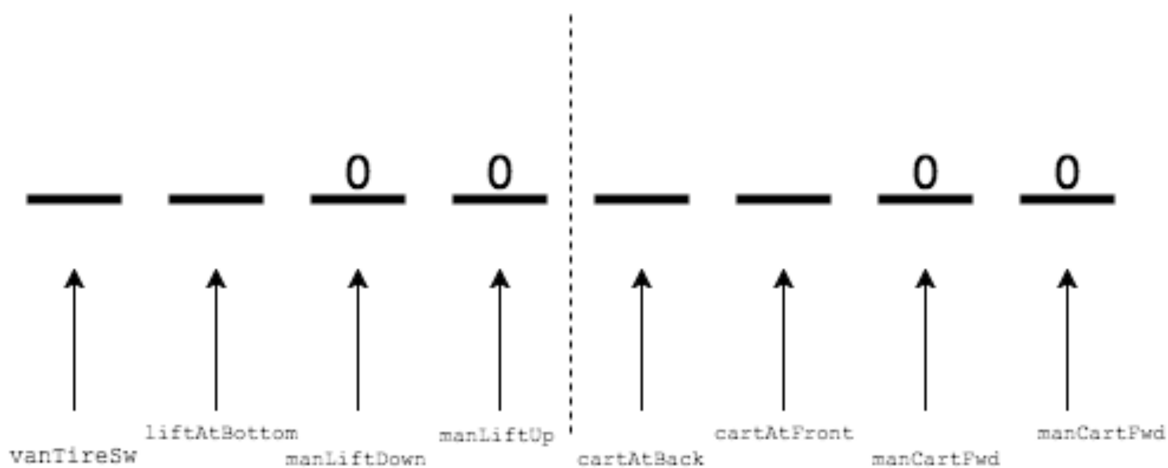

FIGURE 6:
THE RAMP CHECKINPUTS BYTE REPRESENTATION

TABLE 7:
VAN INPUT CONSTANTS AND THEIR VALUES

Constant	Value
ACTS_IN	0x04
ACTS_OUT	0x08
LIFT_AT_TOP	0x10
LIFT_NOT_UP	0xEF

The byte representation returned by the checkInputs function for the van is shown below in figure 7. This checkInputs function works the same as the ramp's checkInputs does. The only difference is that the actuators are not a digital input but rather analog inputs. In order for the checkInput function to work the same for the ramp and the van, the analog input needs to be converted to a Boolean input. Actuators are represented by analog values since actuators are pins that measure their position as an integer value rather than a binary value. For this application, the actuators need to be all the way in/out to be engaged/disengaged. The position values that generate those results are 750 for engaged and 35 for disengaged. There are 2 separate actuators pins, and in order for both of them to verify the actuators are engaged/disengaged requires that both position values be above or below those limits.

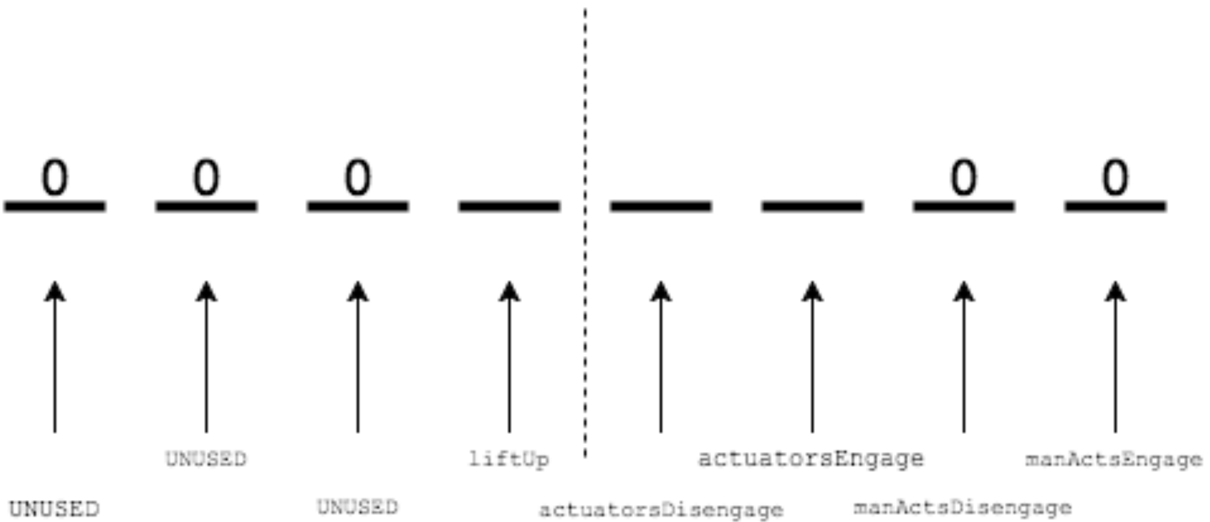

FIGURE 7:
THE VAN CHECKINPUTS BYTE REPRESENTATION

TABLE 8:
VALID INPUTS FOR THE STATES IN THE VAN'S SYSTEM

State	Valid Inputs for this State
INIT	ACTS_IN
WAIT_FOR_VAN	ACTS_IN
WAIT_FOR_DRIVER	ACTS_IN
VAN_READY	ACTS_IN
WAIT_FOR_LIFT	ACTS_IN
	ACTS_OUT
	ACTS_IN | LIFT_UP[2]
	ACTS_OUT | LIFT_UP[3]
WAIT_FOR_LIFT_UP	ACTS_IN
	ACTS_IN | LIFT_UP[3]
	ACTS_OUT
	ACTS_OUT | LIFT_UP[4]
WAIT_FOR_ACTS	ACTS_IN | LIFT_UP
	ACTS_OUT | LIFT_UP
WAIT_FOR_ACTS_IN	ACTS_OUT | LIFT_UP
	ACTS_IN | LIFT_UP[4]
WAIT_FOR_ACTS_OUT	ACTS_IN | LIFT_UP
	ACTS_OUT | LIFT_UP[4]
WAIT_FOR_DONE	ACTS_IN
	ACTS_IN | LIFT_UP

[2] Once the current input has changed and no longer has the lift up, change the correct input to reflect this change since the lowerLift function has just been called.
[3] These inputs represent the correct input in order to move onto the next state.

TABLE 9:
VALID INPUTS FOR THE STATES IN THE RAMP'S SYSTEM

State	Valid Inputs for this State
INIT	CART_FWD | LIFT_DOWN
	CART_BACK | LIFT_DOWN
	VAN_ON | CART_FWD | LIFT_DOWN
	VAN_ON | CART_BACK | LIFT_DOWN
RAMP_READY	VAN_ON | CART_FWD | LIFT_DOWN
	VAN_ON | CART_BACK | LIFT_DOWN
RAISE_LIFT	VAN_ON | CART_FWD | LIFT_DOWN
	VAN_ON | CART_BACK | LIFT_DOWN
	VAN_ON | CART_FWD | LIFT_NOT_DOWN
	VAN_ON | CART_BACK | LIFT_NOT_DOWN
ACTUATORS_OUT	VAN_ON | CART_FWD | LIFT_NOT_DOWN
	VAN_ON | CART_BACK | LIFT_NOT_DOWN
LOWER_LIFT	VAN_ON | CART_BACK | LIFT_NOT_DOWN
	VAN_ON | CART_BACK | LIFT_DOWN[4]
	VAN_ON | CART_FWD | LIFT_DOWN[4]
	VAN_ON | CART_FWD | LIFT_NOT_DOWN
ACTUATORS_IN	VAN_ON | CART_FWD | LIFT_NOT_DOWN
	VAN_ON | CART_BACK | LIFT_NOT_DOWN
COMPLETE	VAN_ON | CART_FWD | LIFT_DOWN
	VAN_ON | CART_BACK | LIFT_DOWN
INIT CHARGERS	CART_FWD | LIFT_DOWN
	CART_BACK | LIFT_DOWN
	VAN_ON | CART_FWD | LIFT_DOWN
	VAN_ON | CART_BACK | LIFT_DOWN
WAIT_FOR_VAN	VAN_ON | CART_FWD | LIFT_DOWN
	VAN_ON | CART_BACK | LIFT_DOWN
	CART_FWD | LIFT_DOWN[4]
	CART_BACK | LIFT_DOWN[5]

One of the most difficult aspects of this project is testing for safety. Since this system will not always run in the exact same way, the inputs that are valid for a particular state may vary

[4] These inputs would cause the state to change back to RAMP_READY since the van drove off.

depending on the previous states and inputs. If the system would only check for valid inputs regardless of the previous inputs, this could be dangerous since both the cart at front and cart at back are valid for all states, but not throwing an error if this changed randomly within a state could be very dangerous. This is why instead of checking each state for whether the input is valid as determined from the above tables, the system will instead check to see if the current inputs are consistent with the expected inputs based on the previous states. This way, once the CART_FWD case has been determined, randomly changing to CART_BACK would throw an error as expected.

Check Serial Function

Another essential feature of this project is the Xbee radio communication. Since wireless communication is not entirely reliable, after some frustrating testing, the EVEC team and I determined the communication should be modeled like TCP. TCP, like this project, sends out a signal and then waits for an acknowledgement that the system received the signal sent before it continues. The system will keep sending the signal (a max of 5 times) until it receives the acknowledgement signal (i.e. ACK). If the system does not receive the ACK by then, it will timeout and send an error signal to the other system. Then, the system will go into error and print out the error message as MISSED_SIGNAL.

The function that checks to see if a signal has been sent is the *checkSerial* function. This function is called in every state and checks to see if there was a signal sent. If there is any signal waiting to be read, it reads it in, then checks to see what the signal is and if the state is a valid state to receive that signal in. The tables 10 and 11 below show which signals are sent from which system and the reasons for sending each signal.

TABLE 10:
XBEE SIGNALS RAMP SENDS TO VAN

Signal Sent	Value
Ramp_Ready	0x01
ACK	0x02
Raise Lift	0x04
Take Actuators Out	0x06
Put Actuators In	0x08
Exchange Complete	0x09

TABLE 11:
XBEE SIGNALS VAN SENDS TO RAMP

Signal Sent	Value
Start Exchange	0x01
ACK	0x02
Lift At Top	0x05
Taking Actuators Out	0x06
Putting Actuators In	0x08

References

[1] "All-Electric Vehicles." *All-Electric Vehicles*. US Department of Energy, Energy Efficiency and Renewable Energy, n.d. Web. 15 May 2016.

[2] Schaal, Eric. "10 Electric Vehicles With the Best Range in 2015." *The Cheat Sheet*. N.p., 12 Nov. 2015. Web. 20 May 2016.

[3] R. Ford and C. Coulston, *Design for Electrical and Computer Engineers*, McGraw-Hill, 2007, p. 37, 92-93, 205.

[4] *IEEE Std 1233, 1998 Edition*, p. 4 (10/36), DOI: 10.1109/IEEESTD.1998.88826

Appendix A: Analysis of Senior Project Design

Project Title: Rapid Battery Exchange Automation

Student's Name: Andrea Everson

Student's Signature: ________________________

Advisor's Name: Art MacCarley

Advisor's Initials: _________

Date: ____________________

Summary of Functional Requirements

The Rapid Battery Exchange system replaces drained battery packs with freshly charged pack on a GMC electric G-Van. This must be done autonomously and without error while also checking all of the inputs to ensure safety. This project uses Xbee radios (IEEE 802.15.4) to provide communication between the separate van and ramp systems.

Primary Constraints

A big issue that was encountered during this project was the reliability of the Xbee radio communication. When the system was first tested, sometimes the ACK (acknowledgement) signals would not be properly received by the other system and would throw errors. In order to debug this issue easier, two Raspberry Pi were added to the system as an interface between the xBee radios and the microcontrollers. This way, if errors occurred, the system could print out exactly which signals were received and which signals had been sent to that system.

Economic

Since my portion of this project was the automation code, the only real costs that would be generated throughout the lifecycle would be the ATMega32U4 microcontrollers as well as the Xbee radios. For the entire project, the costs would be a lot more since the system and batteries would need to be maintained. This would require replacing battery packs after the batteries have been used past their lifetime cycles. The initial cost would also be a lot to build the entire system. On top of the equipment costs that this project requires, if this project were to be manufactured,

it would require a lot of labor costs. While my part of the project was writing the automation code that did not cost much in terms of the equipment necessary, I would estimate I spent around 80 - 110 hours writing, testing and debugging the code. If the employee working on this had a hourly basis of $20/hr, that would cost from $1600 - $2200.

Initially, I believed that this project would take no more than 50 hours to complete. After I started working, however, I quickly realized how much time would be required. On one day alone I spent 8 – 10 hours working on debugging the system with the rest of the EVEC team. I realized how hard it is to estimate how much time will be required from a project until you are completely done. I have also added a lot more safety features than I had thought of when I first started.

Table 10 below shows the original estimated total cost for the components needed for this project. Table 11 shows the actual total costs necessary for this portion of the Rapid Battery Exchange System. In the end, we needed to get 4 microcontrollers so we could use 2 microcontrollers in the testing suitcase and the other 2 microcontrollers could be used in the van and the ramp system.

TABLE 12:
ORIGINAL ESTIMATED TOTAL COMPONENT COSTS

Component	Quantity	Cost	Total Cost
ATMega32U4 breakout board	2	$20	$40
Xbee Kit	1	$100	$100
Total			$140

TABLE 13:
ACTUAL TOTAL COMPONENT COSTS

Component	Quantity	Cost	Total Cost
ATMega32U4 breakout board	4	$20	$80
Xbee Kit	1	$100	$100
Total			$180

Environmental

This project would have a huge impact on the environment. This project could persuade consumers to only own just an electric vehicle instead of needing a gas-powered vehicle for

longer trips and just using the electric vehicle for every day driving to work, the grocery store and in town. If consumers were to only own an electric vehicle, this would drastically cut down on fuel consumption and smog pollution. Implementing this system nationwide could also encourage more individuals to switch to electric vehicles since now driving long distance wouldn't be as big of a hassle.

Manufacturability

While the ultimate goal of this project is to manufacture this system and make it available for consumer use, this product is a ways away from the final state. This project has been in the works for more than a decade and as a result, the structure of the system is not ideal and has been modified over the years. Ideally, since this project is now in a working state, new Cal Poly students would join the Electric Vehicle Engineering Club (EVEC) and continue this project by designing it from the ground up as if it were to be manufactured by a company. If this were to occur, this project could be used around campus with campus vehicles like SLO Safe Ride.

Sustainability

This project has recently been upgraded to include a water-resistant enclosure to stop the rain from damaging the equipment. Before this system could be trusted to withstand the weather, these enclosures would have to be upgraded and would need to be sealed at every possible point.

Ethical

If this system were to be used incorrectly, there are possible situations in which someone could get hurt or killed. There are safety measures in place to mitigate some situations, but since this project has 216 volt battery packs that move around via motors and hydraulic lifts, even without the 216 volt potential, someone could be injured by the moving parts.

Health and Safety

As stated above in the ethical section, there is a chance that someone could be injured or killed if the system were used improperly.

Social and Political

This could help the electric vehicle industry by providing a solution to a problem that these companies have been trying to solve for the last decade. These companies, if they jumped on board to create a battery pack that could be easily removed for this system, could see a huge spike in sales. Additionally, those companies who do not implement this solution could see a decrease in sales that could potential put them out of business if they did not adjust to the current market.

Development

In this class, I have learned a lot about wireless communication and the problems that go along with this technology. Because of this, I learned how TCP works and implemented a similar solution in order to acknowledge if a signal was received. Once this was implemented, the system worked more reliably.

Appendix B: Error Codes

The following tables detail the specific error codes returned from the corresponding van and ramp systems. Both the van and the ramp should display an error, but the reasoning for this error may be different. For instance, if the van encounters an unacceptable input for a specific state, the van would declare the reason for the error and send an ERROR_SIGNAL using the Xbee. When the ramp receives this, they classify the error as a VAN_ERROR.

TABLE 14:
ERROR CODES FOR RAMP

Number of Blinks	Error	Reason for Error
1	EMERGENCY_BUTTON	The emergency button was pressed.
2	LIFT_UP	Lift was up when not supposed to be.
3	CARTS_TIMED_OUT	Carts were moving forward/back and took to long before the cart hit the front or back button.
4	PACK_NOT_AT_EITHER_SIDE	Neither the front or back cart button is pressed.
5	TOO_MANY_BATTERY_PACKS	The carts have two battery packs.
6	NO_BATTERY_PACKS	The carts have no battery packs.
7	VAN_ERROR	The error came from the van.
8	WRONG_INPUT	Checked the inputs and found unwanted inputs.
9	MISSED_SIGNAL	Expected to receive an ACK, but timed out before receiving it.

TABLE 15:
ERROR CODES FOR VAN

Number of Blinks	Error	Reason for Error
1	LIFT_UP	Lift was up when not supposed to be.
2	LIFT_DOWN	Lift was down when not supposed to be.
3	ACTUATORS_DISENGAGED	Actuators out when they were supposed to be engaged.
4	ACTUATORS_ENGAGED	Actuators in when they were supposed to be disengaged.
5	WRONG_INPUT	Input did not match expected input, but wasn't able to identify what specific error it belonged to.
6	RAMP_ERROR	The error came from the ramp.
7	MISSED_SIGNAL	Expected to receive an ACK, but timed out before receiving it.

Appendix C: Ramp Code

Ramp

```
/*
 Bit 0 = manCartFwd
 Bit 1 = manCartBack
 Bit 2 = cartAtFront
 Bit 3 = cartAtBack
 Bit 4 = manLiftUp
 Bit 5 = manLiftDown
 Bit 6 = liftAtBottom
 Bit 7 = vanTireButtonPressed
 Bit 8 = unassigned
 Bit 9 = unassigned
 Bit 10 = unassigned
 Bit 11 = unassigned
 Bit 12 = unassigned
 Bit 13 = unassigned
 Bit 14 = unassigned
 Bit 15 = unassigned
 input: <man> <Obj> <Back/Front | up/down>V
 output: <Obj> At <Where>
 */

/*
OUTPUTS:
  1. moveCart[FWD/Back]
  2. motorOn
  3. moveLiftUp
  4. moveLiftDown
*/

/* Serial Signals */
const int ERROR_SIGNAL = 0xFF;
const int ACK = 0x02;

/* Constants */
const int CART_FWD = 0x04;        // Signal for when cartAtFront (front button pressed)
const int CART_BACK = 0x08;       // Signal for when cartAtBack  (back button pressed)
const int LIFT_DOWN = 0x40;       // Signal for if LiftAtBottom
const int VAN_ON = 0x80;          // Signal for if Van on ramp
const int VAN_OFF_RAMP = 0x7F;    // MASK to remove VAN_ON from correctInput.
const int LIFT_NOT_DOWN = 0xBF;   // Mask to remove LIFT_AT_BOTTOM from correctInput.

const int LOST_SIGNAL = 1000;     // If signal not received within this time limit, signal was lost.
const int DEBOUNCE_TIME = 10;     // Amount of time input needs to remain steady for debounce input

// Offsets of signals in checkInputs/manInputs
const int MAN_FWD_OFFSET = 0;
const int MAN_BACK_OFFSET = 1;
const int FWD_STOP_OFFSET = 2;
const int BACK_STOP_OFFSET = 3;
const int MAN_LIFT_UP_OFFSET = 4;
const int MAN_LIFT_DOWN_OFFSET = 5;
const int LIFT_DOWN_OFFSET = 6;
const int VAN_TIRE_SW = 7;


// Emergency Stop Input
const int emergencyStop = 2; //D1

// 4 inputs for carts
const int manCartFwd = 16;   //B2
const int manCartBack = 14;  //B3
const int cartAtFront = 4;   //D4
const int cartAtBack = 10;   //B6

// 3 inputs for lift
```

```
const int manLiftUp = 15;    //B1
const int manLiftDown = 17;  //B0
const int liftAtBottom = 9;  //B5

// 1 input for van switch
const int vanTireSw = 3;  //D0

//4 outputs
const int moveCartFwd = 13;   //C7
const int moveCartBack = 13;  //C7
const int motorOn = 5;        //C6
const int moveLiftUp = 6;     //D7
const int moveLiftDown = 12;  //D6

// LED Outputs
const int ERROR_PIN = 7;    //D5
const int READY_PIN = 11;     //B7

//outputs for checking battery location
const int rearChargerSelect = 21; //F4
const int frontChargerSelect = 22; //F1
const int energizeCharger = 20; //F5

//input for checking battery location
const int frontBatteryCheck = 19; //F6
const int backBatteryCheck = 18;//F7

const int BATTERY_VOLTAGE_MIN_DIFFERENCE = 2;

//enum for battery location
typedef enum {FRONT, BACK} batteryLocation;

const int DEBOUNCE_COUNT = 10;   // number of millis/samples to consider before declaring a debounced in

void setup() {
    Serial.begin(9600);
    Serial1.begin(9600);

    // Set ramp signals as inputs
    pinMode(manCartFwd, INPUT);
    pinMode(manCartBack, INPUT);
    pinMode(cartAtFront, INPUT);
    pinMode(cartAtBack, INPUT);
    pinMode(vanTireSw, INPUT);
    pinMode(emergencyStop, INPUT);

    // Set lift signals as inputs
    pinMode(manLiftUp, INPUT);
    pinMode(manLiftDown, INPUT);
    pinMode(liftAtBottom, INPUT);

    // Set output pins:
    pinMode(moveCartFwd, OUTPUT);
    pinMode(moveCartBack, OUTPUT);
    pinMode(motorOn, OUTPUT);
    pinMode(moveLiftUp, OUTPUT);
    pinMode(moveLiftDown, OUTPUT);
    pinMode(READY_PIN, OUTPUT);
    pinMode(ERROR_PIN, OUTPUT);

    // Outputs for checking battery location
    pinMode(rearChargerSelect, OUTPUT);
    pinMode(frontChargerSelect, OUTPUT);
    pinMode(energizeCharger, OUTPUT);

    // Input for battery checks
    pinMode(frontBatteryCheck, INPUT);
    pinMode(backBatteryCheck, INPUT);

    // Set the inputs as high (since active low)
    digitalWrite(manCartFwd, HIGH);
```

```
    digitalWrite(manCartBack, HIGH);
    digitalWrite(cartAtFront, HIGH);
    digitalWrite(cartAtBack, HIGH);
    digitalWrite(vanTireSw, HIGH);
    digitalWrite(emergencyStop, HIGH);

    digitalWrite(manLiftUp, HIGH);
    digitalWrite(manLiftDown, HIGH);
    digitalWrite(liftAtBottom, HIGH);

    // don't set optoIsolator high [it is done in checkForBattery()]

    // Set outputs as low initially
    digitalWrite(ERROR_PIN, LOW);
    digitalWrite(moveCartFwd, LOW);
    digitalWrite(moveCartBack, LOW);
    digitalWrite(motorOn, HIGH);
    digitalWrite(moveLiftUp, LOW);
    digitalWrite(moveLiftDown, LOW);
    digitalWrite(READY_PIN, LOW);

    digitalWrite(rearChargerSelect, LOW);
    digitalWrite(frontChargerSelect, LOW);
    digitalWrite(energizeCharger, LOW);
    attachInterrupt(digitalPinToInterrupt(emergencyStop), StopISR, FALLING);
}

/* This function takes all of the inputs and adds it into
  the state to be returned (currInput). All inputs are
   active low.
   * Bit 0 = UNUSED (0) [Don't want to include manual inputs here.]
   * Bit 1 = UNUSED (0) [Don't want to include manual inputs here.]
   * Bit 2 = cartAtFront
   * Bit 3 = cartAtBack
   * Bit 4 = UNUSED (0) [Don't want to include manual inputs here.]
   * Bit 5 = UNUSED (0) [Don't want to include manual inputs here.]
   * Bit 6 = liftAtBottom
   * Bit 7 = vanTirePressed
*/
uint16_t checkInputs() {
  uint8_t tempState = 0x00;

  tempState |= (!debouncePin(cartAtFront)) << FWD_STOP_OFFSET;
  tempState |= (!debouncePin(cartAtBack)) << BACK_STOP_OFFSET;
  tempState |= (!debouncePin(liftAtBottom)) << LIFT_DOWN_OFFSET;
  tempState |= (!debouncePin(vanTireSw)) << VAN_TIRE_SW;
  return tempState;
}

/* This function is similar to checkInputs, but also include the manual
 *  inputs (i.e. pressing button to move carts, and buttons to move lift).
 *  Bit 0 = manCartFwd
 *  Bit 1 = manCartBack
 *  Bit 2 = cartAtFront
 *  Bit 3 = cartAtBack
 *  Bit 4 = manLiftUp
 *  Bit 5 = manLiftDown
 *  Bit 6 = liftAtBotom
 *  Bit 7 = vanTirePressed
`*/

uint16_t manInputs(){
  uint8_t tempState = 0x00;

  tempState |= (!digitalRead(manCartFwd)) << MAN_FWD_OFFSET;
  tempState |= (!digitalRead(manCartBack)) << MAN_BACK_OFFSET;
  tempState |= (!digitalRead(cartAtFront)) << FWD_STOP_OFFSET;
  tempState |= (!digitalRead(cartAtBack)) << BACK_STOP_OFFSET;
  tempState |= (!digitalRead(manLiftUp)) << MAN_LIFT_UP_OFFSET;
  tempState |= (!digitalRead(manLiftDown)) << MAN_LIFT_DOWN_OFFSET;
  tempState |= (!digitalRead(liftAtBottom)) << LIFT_DOWN_OFFSET;
```

```
  return tempState;
}

/* This function will debounce the input and ensure that the input
 * remains in a stable state for certain period of time.
`*/
int debouncePin(int pin) {
  int temp = digitalRead(pin);
  delay(DEBOUNCE_TIME);
  int temp2 = digitalRead(pin);
  if (temp2 == temp && temp == LOW)
    return LOW;
  else
    return HIGH;
}

enum ErrorState { NONE, EMERGENCY_BUTTON, LIFT_UP, CARTS_TIMED_OUT, PACK_NOT_AT_EITHER_SIDE,
TOO_MANY_BATTERY_PACKS, NO_BATTERY_PACKS, VAN_ERROR, WRONG_INPUT, MISSED_SIGNAL};
ErrorState errState = NONE;

enum State { INIT, RAMP_READY, STOP, RAISE_LIFT, LOWER_LIFT, ACTUATORS_OUT, ACTUATORS_IN,
COMPLETE,INIT_CHARGERS, WAIT_FOR_VAN};
State currState = INIT;
State prevState = INIT;
bool actuatorPosition = true;
bool exchangeDone = false;
bool packInFrontCart = false;
bool packInBackCart = false;
bool frontChecked = false;
bool backChecked = false;
bool vanReady = false;
int frontBatteryVoltage = 0;
int backBatteryVoltage = 0;

/* Variables to hold the current and previous states */
uint16_t currInput = 0x0000;
uint16_t manInput = 0x0000;
uint16_t correctInput = 0x00;

void loop() {
  while (currState != STOP) {
      printStateChange();
      switch (currState) {
        case INIT: {
          actuatorPosition = true;
          exchangeDone = false;
          currInput = checkInputs();
          checkSerial();

          /* Check if system started with emergency Stop pressed.*/
          if(digitalRead(emergencyStop) == LOW) {
            Serial.println("System started with Emergency Stop Button Pressed. ERROR!");
            Serial1.write(ERROR_SIGNAL);
            errState = EMERGENCY_BUTTON;
            error();
          }

          /* Van is not on ramp yet. */
          if(currInput == (CART_FWD | LIFT_DOWN) || currInput == (CART_BACK | LIFT_DOWN)) {
            currState = INIT_CHARGERS;
            correctInput = currInput;
            Serial.println("Van off ramp. Going to INIT_CHARGERS from INIT.");
          }

          /* Ramp is ready for exchange to begin */
          else if(currInput == (VAN_ON | CART_FWD | LIFT_DOWN)  || currInput == (VAN_ON | CART_BACK |
LIFT_DOWN)) {
            /* First time running. Need to initialize ramp. */
            correctInput = currInput;
            if(!packInFrontCart && !packInBackCart) {
              currState = INIT_CHARGERS;
```

```
          Serial.println("Van on Ramp. Going to INIT_CHARGERS from INIT.");
        }
        /* Have already initialized system. Go to RAMP_READY. */
        else {
          if(packInFrontCart)
              fwdChargerOn();

          else if(packInBackCart)
              backChargerOn();

          /* Tell the van that ramp ready. */
          if(sendMessage(0x01) != ACK) {
            Serial1.write(ERROR_SIGNAL);
            errState = MISSED_SIGNAL;
            error();
          }
          currState = RAMP_READY;
          Serial.println("Van on ramp and already positioned carts correctly. Going to RAMP_READY from
INIT.");
        }
      }

      /* Ramp in a non-valid position. Check if lift up... */
      else if(digitalRead(liftAtBottom) == HIGH) {
        Serial.println("LIFT UP!!");
        Serial1.write(ERROR_SIGNAL);
        errState = LIFT_UP;
        error();
      }

      /* Otherwise something else wrong. Possibly pack_not_at_either_side?? */
      else {
          Serial.println("Error in INIT!");
          Serial.print("currInput = ");
          Serial.println(currInput, HEX);
          Serial1.write(ERROR_SIGNAL);
          errState = WRONG_INPUT;
          error();
      }
    }
    break;

    case RAMP_READY: {
      digitalWrite(READY_PIN, HIGH);
      currInput = checkInputs();
      checkSerial();

      /* Check if van drove off ramp... */
      if(currInput == (correctInput & VAN_OFF_RAMP)) {
        currState = WAIT_FOR_VAN;
        correctInput = currInput;
        digitalWrite(READY_PIN, LOW);

        /* Sending signal to van letting know that van drove off ramp. */
        if(sendMessage(0x05) != ACK) {
          Serial1.write(ERROR_SIGNAL);
          errState = MISSED_SIGNAL;
          error();
        }
      }
      /* Make sure otherwise ramp in valid condition */
      else {
        checkCorrect();
      }
    }
    break;

    case RAISE_LIFT: {
      /* Wait until signal sent saying lift at top */
      checkSerial();
      currInput = checkInputs();
```

```
    /* Checking that either current inputs are correct inputs (w/ lift down) or correct inputs w/ lift up */
    if(currInput != correctInput && (currInput != (LIFT_DOWN | correctInput))) {
      Serial.print("Not correct inputs! CurrInput = ");
      Serial.println(currInput);
      Serial1.write(ERROR_SIGNAL);
      errState = WRONG_INPUT;
      error();
    }
  }
  break;

  case ACTUATORS_OUT: {
    currInput = checkInputs();
    checkCorrect();
    checkSerial();
 }
 break;

case LOWER_LIFT: {
  checkSerial();
  currInput = checkInputs();
  if(currInput != correctInput || currInput != (correctInput | LIFT_DOWN)) {
    Serial.print("Not correct inputs! CurrInput = ");
    Serial.println(currInput);
    Serial1.write(ERROR_SIGNAL);
    errState = WRONG_INPUT;
    error();
  }
  else if(currInput == (correctInput | LIFT_DOWN)) {
    stopLift();
    correctInput = currInput;
    if (!exchangeDone) {
      exchangeDone = true;
      moveToCharger();
      currState = RAISE_LIFT;
      raiseLift();
    }
    else {
      exchangeDone = false;
      currState = COMPLETE;
    }
  }
}
break;

case ACTUATORS_IN: {
  currInput = checkInputs();
  checkCorrect();
  checkSerial();
}
break;

case COMPLETE: {
  Serial.println("COMPLETE: ");
  if (sendMessage(0x09) == ACK) {
    currState = INIT;
    exchangeDone = true;
    Serial1.flush();
  }
  else {
    Serial1.write(ERROR_SIGNAL);
    errState = MISSED_SIGNAL;
    error();
  }
}
break;

case INIT_CHARGERS: {
  Serial.println("INIT_CHARGERS: ");
```

```
  if (!frontChecked && (digitalRead(cartAtFront) == LOW)) { // Carts at Front
      Serial.println("Cart Forward");
      frontBatteryVoltage = analogRead(frontBatteryCheck);
      frontChecked = true;
      if (!backChecked) {
        movRev();
      }
    }
    else if (!backChecked && (digitalRead(cartAtBack) == LOW)) {
      Serial.println("cart back");
      backBatteryVoltage = analogRead(backBatteryCheck);
      backChecked = true;
      if (!frontChecked) {
        movFwd();
      }
    }
    else if (frontChecked && backChecked) {
      if ((backBatteryVoltage > BATTERY_VOLTAGE_MIN_DIFFERENCE) &&
       (frontBatteryVoltage > BATTERY_VOLTAGE_MIN_DIFFERENCE)) {
        Serial1.write(ERROR_SIGNAL);
        errState = TOO_MANY_BATTERY_PACKS;
        error();
      }
      else if ((backBatteryVoltage <= BATTERY_VOLTAGE_MIN_DIFFERENCE)
       && (frontBatteryVoltage <= BATTERY_VOLTAGE_MIN_DIFFERENCE)) {
        Serial1.write(ERROR_SIGNAL);
        errState = NO_BATTERY_PACKS;
        error();
      }
      else if (backBatteryVoltage > frontBatteryVoltage) {
        packInBackCart = true;
        packInFrontCart = false;
        if(digitalRead(cartAtFront) == LOW) { //At front, move Rev
          movRev();
        }
        backChargerOn();
        correctInput = 0x06 | ((!digitalRead(vanTireSw)) << VAN_TIRE_SW);
        currState = WAIT_FOR_VAN;
      }
      else if (frontBatteryVoltage > backBatteryVoltage) {
        packInFrontCart = true;
        packInBackCart = false;
        if(digitalRead(cartAtBack) == LOW) {  //At back, move Fwd
          movFwd();
        }
        fwdChargerOn();
        correctInput = (CART_FWD | LIFT_DOWN) | ((!digitalRead(vanTireSw)) << VAN_TIRE_SW);
        currState = WAIT_FOR_VAN;
      }
      else {
        Serial.println("BattCheckError");
        Serial1.write(ERROR_SIGNAL);
        error();
      }
   }
}
break;

case WAIT_FOR_VAN: {
  Serial.println("WAIT_FOR_VAN: ");
  currInput = checkInputs();
  checkSerial();

  // Check to see if it is valid input (off ramp) or on ramp.
  if(currInput != correctInput && (currInput != (correctInput | VAN_ON))) {
    Serial1.write(ERROR_SIGNAL);
    errState = WRONG_INPUT;
    error();
  }
  else if(currInput == (correctInput | VAN_ON)) {
    Serial.println("van on ramp now!.");
```

```
          correctInput = currInput;
          currState = RAMP_READY;

          /* Send signal to van saying Van_Ready. */
          if(sendMessage(0x01) != ACK) {
            Serial1.write(ERROR_SIGNAL);
            errState = MISSED_SIGNAL;
            error();
          }
        }
      }
      break;

      }
    }
}

void checkCorrect() {
  currInput = checkInputs();
  if(currInput != correctInput) {
    Serial1.write(ERROR_SIGNAL);
    errState = WRONG_INPUT;
    error();
  }
}

void manControl() {
   digitalWrite(motorOn, HIGH);
   while(1) {
    // Manual mode!!!
    manInput = manInputs();
    Serial.print("Manual State: ");
    Serial.println(manInput, HEX);
    switch(manInput) {
      /* Lift not all the way down and not moving lift. */
      case 0x01:
      case 0x02:
      case 0x04:
      case 0x05:
      case 0x06:
      case 0x08:
      case 0x09:
      case 0x0A:
      break;

      case 0x40: //lift is down, not moving
      case 0x44: //fwd end on, down on, not moving
      case 0x48: //back end on, down on, not moving
        // Stop carts
        digitalWrite(moveCartBack, LOW);
        stopLift();
      break;

      case 0x41: //Move fwd switch is on
      case 0x49: //Move fwd, back end is on
        // Move cart fwd
        digitalWrite(moveCartFwd, HIGH);
        delay(500);
        digitalWrite(moveCartFwd, LOW);
      break;

      case 0x45: //move fwd, but fwd end is on
      case 0x4A: //move backward, but back end is on
        // Stop carts
        digitalWrite(moveCartBack, LOW);
      break;

      case 0x42: //move backward switch is on
      case 0x46: //move backward, fwd end is on
        // Moving cart back.
        digitalWrite(moveCartBack, HIGH);
```

```
        DELAY(500);
        DIGITALWRITE(MOVECARTBACK, LOW);
      BREAK;

      CASE 0X14: //MANUAL RAISE LIFT IS ON, CARTS IN FRONT
      CASE 0X18: //MANUAL RAISE LIFT IS ON, CARTS IN BACK
      CASE 0X54: //MANUAL RAISE LIFT IS ON, LIFT IS AT BOTTOM, CARTS IN FRONT.
      CASE 0X58: //MANUAL RAISE LIFT IS ON, LIFT IS AT BOTTOM, CARTS IN BACK.
        MOVUP();
        WHILE(DIGITALREAD(MANLIFTUP) == LOW);
        STOPLIFT();
      BREAK;

      CASE 0X24: //MOVE DOWN SWITCH IS ON, LIFT IS NOT AT BOTTOM, CARTS IN FRONT.
      CASE 0X28: //MOVE DOWN SWITCH IS ON, LIFT IS NOT AT BOTTOM, CARTS IN BACK.
        MOVDOWN();
        WHILE((DIGITALREAD(MANLIFTDOWN) == LOW) && DIGITALREAD(LIFTATBOTTOM) == HIGH);
        STOPLIFT();
        BREAK;

        CASE 0X64: //MOVE DOWN, BUT DOWN ENDSTOP IS ON, CARTS ARE IN FRONT.
        CASE 0X68: //MOVE DOWN, BUT DOWN ENDSTOP IS ON, CARTS IN BACK
          STOPLIFT();
        BREAK;

        DEFAULT: //LOL GG DONE MESSED UP
          //ERROR();
        BREAK;
      }
    }
}

/* PRINT OUT IF THE STATE HAS CHANGED. */
VOID PRINTSTATECHANGE() {
   IF(PREVSTATE != CURRSTATE) {
      SERIAL.PRINT("CHANGING STATE. PREVIOUS STATE = ");
      SERIAL.PRINTLN(PREVSTATE);
       SERIAL.PRINT("NEW STATE = ");
      SERIAL.PRINTLN(CURRSTATE);
    }
    PREVSTATE = CURRSTATE;
}

VOID MOVFWD() {
  INT COUNT = 0;
  WHILE(DIGITALREAD(CARTATFRONT) == HIGH && COUNT < 5) {
    IF(DIGITALREAD(CARTATBACK) == LOW) {
      COUNT++;
      MOVECART();
    }
  }
  IF(COUNT >= 5) {
    SERIAL1.WRITE(ERROR_SIGNAL);
    ERRSTATE = CARTS_TIMED_OUT;
    ERROR();
  }
}

VOID STOPCARTS() {
  DIGITALWRITE(MOTORON, LOW);
  DIGITALWRITE(MOVECARTBACK, LOW);
}

VOID MOVREV() {
    INT COUNT = 0;
    WHILE(DIGITALREAD(CARTATBACK) == HIGH && COUNT < 5) {
      IF(DIGITALREAD(CARTATFRONT) == LOW) {
        COUNT++;
        MOVECART();
      }
    }
```

```
    if(count >= 5) {
      Serial1.write(ERROR_SIGNAL);
      errState = CARTS_TIMED_OUT;
      error();
    }
}

void moveCart() {
  digitalWrite(moveCartFwd, HIGH);
  delay(1000);
  digitalWrite(moveCartFwd, LOW);
  delay(2000);
}

void signalExchange() {
  Serial.println("Start exchange. Turn off READY_PIN!");
  digitalWrite(READY_PIN, LOW);
  chargersOff();
}

void moveToCharger() {
  if (packInFrontCart) {
    movRev();
    packInFrontCart = false;
    packInBackCart = true;
    correctInput = CART_BACK | VAN_ON | LIFT_DOWN;

  }
  else if (packInBackCart) {
    movFwd();
    packInBackCart = false;
    packInFrontCart = true;
    correctInput = CART_FWD | VAN_ON | LIFT_DOWN;
  }
  else {
    Serial.println("Pack not in Back Cart or in Front Cart!!");
    Serial1.write(ERROR_SIGNAL);
    errState = PACK_NOT_AT_EITHER_SIDE;
    error();
  }
}

void actuatorsOut() {
  int message = sendMessage(0x06);
  if (message == ACK) {
    Serial.println("act going out");
    currState = ACTUATORS_OUT;
  }
  else {
    Serial1.write(ERROR_SIGNAL);
    errState = MISSED_SIGNAL;
    error();
  }
}

void actuatorsIn() {
  if(sendMessage(0x08) != ACK) {
    Serial1.write(ERROR_SIGNAL);
    errState = MISSED_SIGNAL;
    error();
  }
}

void raiseLift() {
  Serial.println("raise lift");
  int message = sendMessage(0x04);
  if (message == ACK) {
    Serial.println("going up");
    Serial1.flush();
    digitalWrite(moveLiftDown, LOW);
    digitalWrite(moveLiftUp, HIGH);
```

```
  }

  else {
    Serial.println("Didn't receive signal!");
    Serial1.write(ERROR_SIGNAL);
    errState = MISSED_SIGNAL;
    error();
  }
}

void lowerLift() {
    Serial.println("lower lift");
    digitalWrite(moveLiftUp, LOW);
    digitalWrite(moveLiftDown, HIGH);
}

void movUp() {
  digitalWrite(moveLiftDown, LOW);
  digitalWrite(moveLiftUp, HIGH);
}

void stopLift() {
  digitalWrite(moveLiftDown, LOW);
  digitalWrite(moveLiftUp, LOW);
}

void movDown() {
  digitalWrite(moveLiftUp, LOW);
    digitalWrite(moveLiftDown, HIGH);
}

void fwdChargerOn() {
  digitalWrite(frontChargerSelect, HIGH);
    digitalWrite(energizeCharger, HIGH);
}

void backChargerOn() {
  digitalWrite(rearChargerSelect, HIGH);
    digitalWrite(energizeCharger, HIGH);
}

void chargersOff() {
  digitalWrite(rearChargerSelect, LOW);
  digitalWrite(frontChargerSelect, LOW);
  digitalWrite(energizeCharger, LOW);
}

void error() {
      Serial.println("ERROR: ");
      chargersOff();
      stopLift();
      stopCarts();
      digitalWrite(READY_PIN, LOW);
      Serial.println(currInput, HEX);
      printErrorMessage();
      digitalWrite(ERROR_PIN, HIGH);
      manControl();
}

/* Send a message to the van and recieve the message back. */
uint8_t sendMessage(uint8_t message) {
  int missCount = 0;

  bool response = false;
  uint8_t packate = 0x00;
  Serial1.write(message);
  unsigned long currTime = millis();
  while(!response) {
    if(missCount > 8){
      return ERROR_SIGNAL;
    }
```

```
    if(Serial1.available() > 0) {
      response = true;
      packate = Serial1.read();
      Serial.println(packate);
    }
    if((millis() - currTime) > LOST_SIGNAL) {
      /*RESEND*/
      Serial1.write(message);
      missCount++;
      currTime = millis();
      //response = true;
      //packate = ERROR_SIGNAL;
    }
  }
  return packate;
}

/* Prints out the error message */
void printErrorMessage() {
  switch(errState) {

    // Blink on/off 1 time
    case EMERGENCY_BUTTON:
      for(int i = 0; i < 3; i++) {
        blinkError();
        delay(2000);
      }
    break;

    // Blink on/off 2 times
    case LIFT_UP:
      for(int i = 0; i < 3; i++) {
        blinkError();
        delay(500);
        blinkError();
        delay(2000);
      }
    break;

    // Blink on/off 3 times
    case CARTS_TIMED_OUT:
      for(int i = 0; i < 3; i++) {
        blinkError();
        delay(500);
        blinkError();
        delay(500);
        blinkError();
        delay(2000);
      }
    break;

    // Blink on/off 4 times
    case PACK_NOT_AT_EITHER_SIDE:
      for(int i = 0; i < 3; i++) {
        blinkError();
        delay(500);
        blinkError();
        delay(500);
        blinkError();
        delay(500);
        blinkError();
        delay(2000);
      }
    break;

    // Blink on/off 5 times
    case TOO_MANY_BATTERY_PACKS:
      for(int i = 0; i < 3; i++) {
        blinkError();
        delay(500);
        blinkError();
```

```
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(2000);
  }
BREAK;

// BLINK ON/OFF 6 TIMES
CASE NO_BATTERY_PACKS:
  FOR(INT I = 0; I < 3; I++) {
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(2000);
  }
BREAK;


// BLINK ON/OFF 7 TIMES
CASE VAN_ERROR:
  FOR(INT I = 0; I < 3; I++) {
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(2000);
  }
BREAK;

 // BLINK ON/OFF 8 TIMES
CASE WRONG_INPUT:
  FOR(INT I = 0; I < 3; I++) {
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(500);
    BLINKERROR();
    DELAY(2000);
  }
BREAK;
```

```
    // Blink on/off 9 times
    case MISSED_SIGNAL:
      for(int i = 0; i < 3; i++) {
        blinkError();
        delay(500);
        blinkError();
        delay(500);
        blinkError();
        delay(500);
        blinkError();
        delay(500);
        blinkError();
        delay(500);
        blinkError();
        delay(500);
        blinkError();
        delay(500);
        blinkError();
        delay(500);
        blinkError();
        delay(2000);
      }
    break;

    default:
      for(int i = 0; i < 3; i++) {
        digitalWrite(ERROR_PIN, HIGH);
        delay(1000);
        digitalWrite(ERROR_PIN, LOW);
        delay(2000);
      }
    break;
    }
}

/* Will blink the error light once */
void blinkError() {
  digitalWrite(ERROR_PIN, HIGH);
  delay(500);
  digitalWrite(ERROR_PIN, LOW);
}

/* Receives all of the signals from the van. */
void checkSerial() {
  uint8_t temp = 0x00;
    if(Serial1.available() > 0) {
      temp = Serial1.read();
      if(temp == 0x01) {
        vanReady = true;
        if (currState == RAMP_READY) {
            Serial1.write(ACK);
            /* Change LEDS to reflect starting exchange. */
            signalExchange();
            currState = RAISE_LIFT;
            /* Send signal and start raising lift*/
            correctInput = correctInput & LIFT_NOT_DOWN;
            raiseLift();
        }
      }
      if(temp == 0x05) {
        if(currState == RAISE_LIFT) {
          /* Got signal that lift at top. Stop lift! */
          Serial1.write(ACK);
          stopLift();
          if(actuatorPosition) {
            actuatorPosition = false;
            currState = ACTUATORS_OUT;
            /* Send signal to van to pull acts out */
            actuatorsOut();
            /* Want to have correctInput here that has lift up. */
          }
```

```
          else {
            actuatorPosition = true;
            currState =  ACTUATORS_IN;
            /* Send signal to van to put acts in */
            actuatorsIn();
          }
        }
      }
      if(temp == 0x06) {
        /* Van sent signal that ACTS out, and missed ACK signal. Send again. */
        if(currState == LOWER_LIFT) {
          Serial1.write(ACK);
        }
        else if(currState == ACTUATORS_OUT) {
          currState = LOWER_LIFT;
          Serial1.write(ACK);
          lowerLift();
        }
      }
      if(temp == 0x08) {
        /* Van sent signal that ACT in, and missed ACK signal. Send again. */
        if(currState == LOWER_LIFT) {
          Serial1.write(ACK);
        }
        else if(currState == ACTUATORS_IN) {
          currState = LOWER_LIFT;
          Serial1.write(ACK);
          lowerLift();
        }
      }
      if (temp == ERROR_SIGNAL) {
        errState = VAN_ERROR;
        error();
      }
    }
}

void StopISR() {
  Serial.println("Emergency Stop Button Pressed!");
  Serial1.write(ERROR_SIGNAL);
  errState = EMERGENCY_BUTTON;
  error();
}
```

Van

```
/*
INPUTS:
  Bit 0 = manActuatorsEngage
  Bit 1 = manActuatorsDisengage
  Bit 2 = actuatorsEngaged
  Bit 3 = actuatorsDisengaged
  Bit 4 = liftUp
  Bit 5 = NOT USED
  Bit 6 = NOT USED
  Bit 7 = NOT USED
OTHER INPUTS:
  driverStartButton [Interrupt]
  ignitionSwitchPin
*/

/*lift
OUTPUTS:
  1. movEngageActuators (actuatorsIn)
  2. movDisengageActuators (actuatorsOut)
OTHER OUTPUTS:
  ERROR_PIN
  READY_PIN
  EXCHANGE_PIN
```

```
  COMPLETE_PIN
*/

/* Serial Signals */
const int ERROR_SIGNAL = 0xFF;
const int ACK = 0x02;

/* Constants */
const int MAN_ACTS_IN = 0x01;     // Signal for when manual Actuators Engaged
const int MAN_ACTS_OUT = 0x02;    // Signal for when manual Actuators Disengaged
const int ACTS_IN = 0x04;        // Signal for when ActsIn (engaged)
const int ACTS_OUT = 0x08;        // Signal for when ActsOut (disengaged)
const int LIFT_AT_TOP = 0x10;     // Signal for when liftAtTop
const int LIFT_NOT_UP = 0xEF;     // Mask to remove LIFT_AT_TOP from correctInput.

const int LOST_SIGNAL = 1000;     //If signal not received within this time limit, signal was lost.
const int DEBOUNCE_TIME = 10;     //Amount of time input needs to remain steady for debounce input

volatile unsigned long last_micros;

//Offsets of signals in checkInputs/manInputs
const int MAN_ACTUATORS_ENGAGE_OFFSET = 0;
const int MAN_ACTUATORS_DISENGAGE_OFFSET = 1;
const int ACTUATORS_ENGAGED_OFFSET = 2;
const int ACTUATORS_DISENGAGED_OFFSET = 3;
const int LIFT_OFFSET = 4;

//Threshholds for actuator placement
const int ACTUATORS_DISENGAGED_THRESHHOLD = 35;
const int ACTUATORS_ENGAGED_THRESHHOLD = 750;
const int ACTUATOR_RANGE = 10;

//Inputs for van
const int manActuatorsEngage = 16;  //B2
const int manActuatorsDisengage = 15;  //B1
const int frontActuatorLocationPin = 0;  //F7
const int rearActuatorLocationPin = 1;  //F6
const int ignitionSwitchPin = 21; //F4
const int driverStartButton = 7; //E6 [Interrupt]
const int liftUp = 14;//B3

//Outputs for van
const int movActuatorsEngage = 6;  //D7
const int movActuatorsDisengage = 12;  //D6

//LED Outputs
const int ERROR_PIN = 22;  //F1
const int READY_PIN = 5; //C6
const int EXCHANGE_PIN = 13;//C7
const int COMPLETE_PIN = 23; //F0

//Current State
uint16_t currInput = 0x00;
uint16_t manState = 0x0000;
bool rampReady = false;
bool actsIn = true;
uint16_t correctInput = 0x00;

void setup() {
    Serial.begin(9600);
    Serial1.begin(9600);

    //Set actuator signals as inputs
    pinMode(manActuatorsEngage, INPUT);
    pinMode(manActuatorsDisengage, INPUT);
    pinMode(frontActuatorLocationPin, INPUT);
    pinMode(rearActuatorLocationPin, INPUT);

    //Set other inputs
    pinMode(ignitionSwitchPin, INPUT);
    pinMode(driverStartButton, INPUT);
```

```
    pinMode(liftUp, INPUT);

    //Set output pins
    pinMode(ERROR_PIN, OUTPUT);
    pinMode(READY_PIN, OUTPUT);
    pinMode(EXCHANGE_PIN, OUTPUT);
    pinMode(COMPLETE_PIN, OUTPUT);
    pinMode(movActuatorsEngage, OUTPUT);
    pinMode(movActuatorsDisengage, OUTPUT);

    //Set digital inputs as high (since active low)
    digitalWrite(manActuatorsEngage, HIGH);
    digitalWrite(manActuatorsDisengage, HIGH);
    digitalWrite(driverStartButton, HIGH);
    digitalWrite(liftUp, HIGH);

    //Start ignition pin as low
    digitalWrite(ignitionSwitchPin, LOW);

    //Initialize outputs to low
    digitalWrite(ERROR_PIN, LOW);
    digitalWrite(READY_PIN, LOW);
    digitalWrite(EXCHANGE_PIN, LOW);
    digitalWrite(COMPLETE_PIN, LOW);
    digitalWrite(movActuatorsEngage, LOW);
    digitalWrite(movActuatorsDisengage, LOW);

    attachInterrupt(digitalPinToInterrupt(driverStartButton), StartISR, FALLING);
}

/*This function takes all inputs and ORs them onto the proper
    position on the state to be returned (currInput). Digital inputs
    are active low, so they are inverted. Analog inputs are compared
    to their threshholds and set accordingly.
    * Bit 0 = UNUSED (0) [Don't want to include manual inputs here.]
    * Bit 1 = UNUSED (1) [Don't want to include manual inputs here.]
    * Bit 2 = actuatorsEngaged (In)
    * Bit 3 = actuatorsDisengaged (Out)
    * Bit 4 = liftUp
*/
uint8_t checkInputs() {
  uint8_t tempState = 0x00;
  tempState |= ((analogRead(frontActuatorLocationPin) < (ACTUATORS_DISENGAGED_THRESHHOLD +
ACTUATOR_RANGE))
                && (analogRead(rearActuatorLocationPin) < (ACTUATORS_DISENGAGED_THRESHHOLD +
ACTUATOR_RANGE)))
                << ACTUATORS_ENGAGED_OFFSET;

  tempState |= ((analogRead(frontActuatorLocationPin) > (ACTUATORS_ENGAGED_THRESHHOLD - ACTUATOR_RANGE))
                && (analogRead(rearActuatorLocationPin) > (ACTUATORS_ENGAGED_THRESHHOLD -
ACTUATOR_RANGE)))
                << ACTUATORS_DISENGAGED_OFFSET;

  tempState |= ((!debouncePin(liftUp)) << LIFT_OFFSET);
  return tempState;
}

/* This function is similar to checkInputs, but also includes the manual
    inputs (i.e. pressing button to disengage/engage actuators).
    * Bit 0 = manActuatorsEngage (In)
    * Bit 1 = manActuatorsDisengage (Out)
    * Bit 2 = actuatorsEngaged (In)
    * Bit 3 = actuatorsDisengaged (Out)
*/
uint8_t manInputs() {
  uint8_t tempState = 0x00;
  tempState |= (!digitalRead(manActuatorsEngage)) << MAN_ACTUATORS_ENGAGE_OFFSET;
  tempState |= (!digitalRead(manActuatorsDisengage)) << MAN_ACTUATORS_DISENGAGE_OFFSET;

  tempState |= ((analogRead(frontActuatorLocationPin) < ACTUATORS_DISENGAGED_THRESHHOLD)
              && (analogRead(rearActuatorLocationPin) < ACTUATORS_DISENGAGED_THRESHHOLD))
```

```
            << ACTUATORS_ENGAGED_OFFSET;

  tempState |= ((analogRead(frontActuatorLocationPin) > ACTUATORS_ENGAGED_THRESHHOLD)
            && (analogRead(rearActuatorLocationPin) > ACTUATORS_ENGAGED_THRESHHOLD))
            << ACTUATORS_DISENGAGED_OFFSET;

  //tempState != ((!debouncePin(liftUp)) << LIFT_OFFSET);
  return tempState;
}


/* This function will debounce the input and ensure that the input
 * remains in a stable state for certain period of time.
`*/
int debouncePin(int pin) {
  int temp = digitalRead(pin);
  delay(DEBOUNCE_TIME);
  int temp2 = digitalRead(pin);
  if (temp2 == temp && temp == LOW)
    return LOW;
  else
    return HIGH;
}

enum ErrorState {NONE, LIFT_UP, LIFT_DOWN, ACTUATORS_DISENGAGED, ACTUATORS_ENGAGED, WRONG_INPUT,
RAMP_ERROR, MISSED_SIGNAL};
ErrorState errState = NONE;

enum State {INIT, WAIT_FOR_VAN, WAIT_FOR_DRIVER, VAN_READY, WAIT_FOR_LIFT, WAIT_FOR_LIFT_UP,
WAIT_FOR_ACTS, WAIT_FOR_ACTS_OUT, WAIT_FOR_ACTS_IN, WAIT_FOR_DONE,STOP};
State currState = INIT;
State prevState = INIT;

void loop() {
  while(currState != STOP) {
    printStateChange();
      switch(currState) {
        case INIT: {
          Serial.println("VAN_INIT: ");
          Serial.print("currInput: ");
          Serial.println(currInput, HEX);

          checkSerial();
          correctInput = ACTS_IN;
          checkCorrect();
          currState = WAIT_FOR_VAN;
        }
        break;

        case WAIT_FOR_VAN: {
          checkCorrect();
          checkSerial();
        }
        break;

        case WAIT_FOR_DRIVER: {
          // Just want to wait for ISR to engage OR for van to send signal (Error/not ready).
          EIFR = 0x01;
          Serial.println("VAN_WAIT_FOR_DRIVER: ");
          checkCorrect();
          checkSerial();
        }
        break;

        case VAN_READY: {
          Serial.println("VAN_READY: ");
          checkSerial();
          checkIgnition();
          /* ignition was on. Jump back to switch statement b4 executing code here.*/
          if(currState != VAN_READY)
            break;
```

```
  checkCorrect();
  int message = sendMessage(0x01);
  if(message == ACK) {
    Serial.println("Started Exchange. Go and wait for lift signal");
    digitalWrite(COMPLETE_PIN, LOW);
    digitalWrite(READY_PIN, LOW);
    digitalWrite(EXCHANGE_PIN, HIGH);
    currState = WAIT_FOR_LIFT;
  }
  else {
    Serial.print("Message was: ");
    Serial.println(message, HEX);
    Serial1.write(ERROR_SIGNAL);
    errState = MISSED_SIGNAL;
    error();
  }
}
break;

case WAIT_FOR_LIFT: {
  currInput = checkInputs();
  checkSerial();

  if(currInput != correctInput) {
    /* Check if lift has started to go down. */
    if(currInput == (correctInput & LIFT_NOT_UP)) {
      Serial.println("Just started putting lift down.");
      correctInput = currInput;
    }
    else {
      checkCorrect(); // Call this so can give correct errState.
      /*Serial1.write(ERROR_SIGNAL);
      errState = WRONG_INPUT;
      error();
      */
    }
  }
}
break;

case WAIT_FOR_LIFT_UP: {
  currInput = checkInputs();
  checkSerial();

  /* If lift is up and still correct input*/
  if(currInput == (correctInput | LIFT_AT_TOP)) {
    int message = sendMessage(0x05);
    if(message == ACK) {
      Serial.println("Lift up.");
      correctInput = (correctInput | LIFT_AT_TOP);
      currState = WAIT_FOR_ACTS;
    }
    else {
      Serial1.write(ERROR_SIGNAL);
      errState = MISSED_SIGNAL;
      error();
    }
  }

  else if(currInput != correctInput) {
    checkCorrect();     // Call this so can give correct errState.
    /*Serial1.write(ERROR_SIGNAL);
    errState = WRONG_INPUT;
    error();
    */
  }
}
break;

case WAIT_FOR_ACTS: {
```

```
  Serial.println("WAIT_FOR_ACTS: ");
  checkCorrect();
  checkSerial();
}
break;

/* Can the inputs here be anything but acts in or acts out? */
case WAIT_FOR_ACTS_OUT: {
  checkSerial();
  currInput = checkInputs();

  /* actuators are out */
  if(currInput == correctInput) {
    digitalWrite(movActuatorsDisengage, LOW);
    if(sendMessage(0x06) != ACK) {
      Serial1.write(ERROR_SIGNAL);
      errState = MISSED_SIGNAL;
      error();
    }
    else {
      currState = WAIT_FOR_LIFT;
    }
  }
  else if(digitalRead(liftUp) == HIGH) {
    Serial.println("Lift not up! ERROR!");
    Serial1.write(ERROR_SIGNAL);
    errState = LIFT_DOWN;
    error();
  }
}
break;

/* Can the inputs here be anything but acts in or acts out? */
case WAIT_FOR_ACTS_IN: {
  checkSerial();
  currInput = checkInputs();
  /* actuators are in */
  if(currInput == correctInput) {
    digitalWrite(movActuatorsEngage, LOW);
    if(sendMessage(0x08) != ACK) {
      Serial1.write(ERROR_SIGNAL);
      errState = MISSED_SIGNAL;
      error();
    }
    else {
      currState = WAIT_FOR_DONE;
    }
  }
  else if(digitalRead(liftUp) == HIGH) {
    Serial.println("Lift not up! ERROR!");
    Serial1.write(ERROR_SIGNAL);
    errState = LIFT_DOWN;
    error();
  }
}
break;

case WAIT_FOR_DONE: {
  if(currInput != correctInput) {
    /* Check if lift has started to go down... */
    if(currInput == (correctInput & LIFT_NOT_UP)) {
      Serial.println("Just started putting lift down.");
      correctInput = currInput;
    }
    else {
      checkCorrect();     // Call this so that can give correct errState.
      /*Serial1.write(ERROR_SIGNAL);
      errState = WRONG_INPUT;
      error();
      */
    }
```

```
          }
          checkSerial();
        }
        break;
      }
    }
}

void checkCorrect() {
  currInput = checkInputs();
  if(currInput != correctInput) {
    Serial1.write(ERROR_SIGNAL);
    /* correctInput has ACTS_IN and currInput does not... */
    if(((correctInput & ACTS_IN) == ACTS_IN) && ((currInput & ACTS_IN ) != ACTS_IN)) {
      errState = ACTUATORS_DISENGAGED;
    }
    /* Correct input has ACTS_OUT and currInput does not...*/
    else if(((correctInput & ACTS_OUT) == ACTS_OUT) && ((currInput & ACTS_OUT) != ACTS_OUT)) {
      errState = ACTUATORS_ENGAGED;
    }
    /* Correct input has LIFT_UP and currInput does not... */
    else if(((correctInput & LIFT_AT_TOP) == LIFT_AT_TOP) && ((currInput & LIFT_AT_TOP) != LIFT_AT_TOP))
{
      errState = LIFT_DOWN;
    }
    /* Correct input has LIFT_DOWN and currInput does not... */
    else if(((correctInput | LIFT_NOT_UP) == LIFT_NOT_UP) && ((currInput | LIFT_NOT_UP) != LIFT_NOT_UP))
{
      errState = LIFT_UP;
    }
    else {
      errState = WRONG_INPUT;
    }
    error();
  }
}

void manControl() {
  while(1) {
    manState = manInputs();
    Serial.print("VAN_MANUAL State: ");
    Serial.println(manState, HEX);
    switch(manState) {
      case 0x00: //Everything is off
      case ACTS_IN: //Actuators engaged, not requesting acts in or out
      case ACTS_IN | MAN_ACTS_IN: //Trying to engage actuators, but they are already engaged.
      case ACTS_OUT: //Actuators disengaged, not requesting acts in or out.
      case ACTS_OUT | MAN_ACTS_OUT: //Trying to disengage actuators, but they are already disengaged.
        stopActuators();
      break;

      case MAN_ACTS_IN: //Engaging actuators, currently neither engaged nor disengaged.
      case ACTS_OUT | MAN_ACTS_IN: //Engaging actuators, currently disengaged.
        engageActuators();
      break;

      case MAN_ACTS_OUT: //Disengaging actuators, currently neither engaged nor disengaged.
      case MAN_ACTS_OUT | ACTS_IN: //Disengaging actuators, currently engaged.
        disengageActuators();
      break;

      default: //LOL GG DONE MESSED UP SON
        error();
      break;
    }
  }
}

/* Print out if the state has changed. */
void printStateChange() {
   if(prevState != currState) {
```

```
      Serial.print("Changing state. Previous State = ");
      Serial.println(prevState);
       Serial.print("New State = ");
      Serial.println(currState);
    }
    prevState = currState;
}

/* May need to get rid of while loop? Or not since in the ISR?? */
void checkIgnition() {
  int count = 0;
  bool ignitionOn = false;
  while (count < 5 && !ignitionOn) {
    uint8_t  status = 0x00;
    delay(100);
    status |= digitalRead(ignitionSwitchPin);
    count++;
    if (status != 0x00)
      ignitionOn = true;
  }
  if (ignitionOn) {
    Serial.println("Ignition is on. Must turn off before pressing button.");
    currState = WAIT_FOR_DRIVER;
  }
}

void stopActuators() {
  digitalWrite(movActuatorsEngage, LOW);
  digitalWrite(movActuatorsDisengage, LOW);
}

void engageActuators() {
  digitalWrite(movActuatorsEngage, HIGH);
  digitalWrite(movActuatorsDisengage, LOW);
}

void disengageActuators() {
  digitalWrite(movActuatorsEngage, LOW);
  digitalWrite(movActuatorsDisengage, HIGH);
}

void error() {
  digitalWrite(COMPLETE_PIN, LOW);
  digitalWrite(READY_PIN, LOW);
  digitalWrite(EXCHANGE_PIN, LOW);
  stopActuators();

  /* DEBUGGING! */
  Serial.println("Van ERROR. Power down. ");
  Serial.print("currInput is: ");
  Serial.println(currInput);
  Serial.print("currState is: ");
  Serial.println(currState);

  printErrorMessage();
  digitalWrite(ERROR_PIN, HIGH);
  manControl();
}

void printErrorMessage() {
  switch(errState) {

    // Blink on/off 1 time
    case LIFT_UP:
      for(int i = 0; i < 3; i++) {
        blinkError();
        delay(2000);
      }
    break;

    // Blink on/off 2 times
```

```
case LIFT_DOWN:
  for(int i = 0; i < 3; i++) {
      blinkError();
      delay(500);
      blinkError();
      delay(2000);
  }
break;

// Blink on/off 3 times
case ACTUATORS_DISENGAGED:
  for(int i = 0; i < 3; i++) {
      blinkError();
      delay(500);
      blinkError();
      delay(500);
      blinkError();
      delay(2000);
  }
break;

// Blink on/off 4 times
case ACTUATORS_ENGAGED:
  for(int i = 0; i < 3; i++) {
      blinkError();
      delay(500);
      blinkError();
      delay(500);
      blinkError();
      delay(500);
      blinkError();
      delay(2000);
  }
break;

// Blink on/off 5 times
case WRONG_INPUT:
  for(int i = 0; i < 3; i++) {
      blinkError();
      delay(500);
      blinkError();
      delay(500);
      blinkError();
      delay(500);
      blinkError();
      delay(500);
      blinkError();
      delay(2000);
  }
break;

// Blink on/off 6 times
case RAMP_ERROR:
  for(int i = 0; i < 3; i++) {
      blinkError();
      delay(500);
      blinkError();
      delay(500);
      blinkError();
      delay(500);
      blinkError();
      delay(500);
      blinkError();
      delay(500);
      blinkError();
      delay(2000);
  }
break;

// Blink on/off 7 times
case MISSED_SIGNAL:
```

```
      for(int i = 0; i < 3; i++) {
          blinkError();
          delay(500);
          blinkError();
          delay(500);
          blinkError();
          delay(500);
          blinkError();
          delay(500);
          blinkError();
          delay(500);
          blinkError();
          delay(500);
          blinkError();
          delay(2000);
      }
    break;
  }
}

/* Will blink the error light once */
void blinkError() {
  digitalWrite(ERROR_PIN, HIGH);
  delay(500);
  digitalWrite(ERROR_PIN, LOW);
}


uint8_t sendMessage(uint8_t message) {
  int missCount = 0;

  bool response = false;
  uint8_t packate = 0x00;
  Serial1.write(message);
  unsigned long currTime = millis();
  while(!response) {
    if(missCount > 8) {
      return ERROR_SIGNAL;
    }
    if(Serial1.available() > 0) {
      response = true;
      packate = Serial1.read();
      Serial.println(packate);
    }
    if((millis() - currTime) > LOST_SIGNAL) {
      /*RESEND*/
      Serial1.write(message);
      missCount++;
      currTime = millis();
    }
  }
  return packate;
}

void checkSerial() {
  uint8_t temp = 0x00;
    if(Serial1.available() > 0) {
      temp = Serial1.read();
      Serial.print("checkSerial: temp is: ");
      Serial.println(temp, HEX);

      if(temp == ERROR_SIGNAL) {
        Serial.println("Received error from ramp!");
        errState = RAMP_ERROR;
        error();
      }

      if(temp == 0x01) {
        if(currState == WAIT_FOR_VAN) {
          rampReady = true;
          digitalWrite(READY_PIN, HIGH);
```

```
          Serial1.write(ACK);
          currState = WAIT_FOR_DRIVER;
        }
      }

      if(temp == 0x04) {
        // if we're already in WAIT_FOR_LIFT_UP, the ACK was missed, resend it
        if (currState == WAIT_FOR_LIFT_UP) {
          Serial1.write(ACK);
           currState = WAIT_FOR_LIFT_UP;
        }
        else if(currState == WAIT_FOR_LIFT) {
          currState = WAIT_FOR_LIFT_UP;
          Serial1.write(ACK);
        }
        Serial.println("state to raise");
      }

      if(temp == 0x05) {
        if(currState == WAIT_FOR_DRIVER) {
          Serial1.write(ACK);
          currState = WAIT_FOR_VAN;
          digitalWrite(READY_PIN, LOW);
          digitalWrite(COMPLETE_PIN, LOW);
          digitalWrite(EXCHANGE_PIN, LOW);
        }
        else if(currState == WAIT_FOR_VAN) {
          Serial1.write(ACK); // Ramp may have missed ACK from driving off ramp.
        }
      }

      if(temp == 0x06) {
        if(currState == WAIT_FOR_ACTS_OUT)
          Serial1.write(ACK);
        else if(currState == WAIT_FOR_ACTS) {
          currState = WAIT_FOR_ACTS_OUT;
          /* Checking inputs for validity.. */
          Serial1.write(ACK);
          correctInput = ACTS_OUT | LIFT_AT_TOP;
          disengageActuators();
        }
      }

      if(temp == 0x08) {
        if(currState == WAIT_FOR_ACTS_IN)
          Serial1.write(ACK);
        else if(currState == WAIT_FOR_ACTS) {
          currState = WAIT_FOR_ACTS_IN;
          Serial1.write(ACK);
          correctInput = ACTS_IN | LIFT_AT_TOP;
          engageActuators();
        }
      }

      if(temp == 0x09) {
        if(currState == WAIT_FOR_DONE)
          Serial1.write(ACK);
          digitalWrite(EXCHANGE_PIN,LOW);
          digitalWrite(COMPLETE_PIN, HIGH);
          currState = INIT;
          Serial1.flush();
      }

    }
    Serial.print("currState:");
    Serial.println(currState);
}


void StartISR() {
  bool buttonPressed = true;
```

```
  for(int i = 0; i < 5 && buttonPressed; i++) {
    /* Delay 1/10 second and check if button is pressed still */
    delayMicroseconds(100000);
    if(digitalRead(driverStartButton) == HIGH) {  /* Button no longer pressed */
      buttonPressed = false;
    }
  }

  /* Button was pressed for .5 seconds. Should be safe to continue. */
  if(buttonPressed) {
    if(currState == WAIT_FOR_DRIVER) {
      currState = VAN_READY;
      Serial.println("DRIVER_HIT_START_BUTTON! Go to VAN_READY.");
    }
    else {
      Serial.print("IN ISR! AutoState = ");
      Serial.println(currState);
    }
  }
  else {
    Serial.println("button not pressed down long enough.");
  }
}
```

Appendix D: Pin Out

TABLE 16:
RAMP PIN ASSIGNMENTS FOR ATMEGA32U4

Connection	Pin	Variable in Code
Emergency Stop Button	D1	emergencyStop
Manual Cart Forward	B2	manCartFwd
Manual Cart Backward	B3	manCartBack
Front Cart Button	D4	cartAtFront
Back Cart Button	B6	cartAtBack
Manual Lift Up	B1	manLiftUp
Manual Lift Down	B0	manLiftDown
Lift Down Button	B5	liftAtBottom
Van on Ramp	D0	vanTireSw
Move Cart	C7	moveCartFwd/moveCartBack
Garage Door Motor	C6	motorOn
Move Lift Up	D7	moveLiftUp
Move Lift Down	D6	moveLiftDown
Error LED	D5	ERROR_PIN
Ready LED	B7	READY_PIN
Back Charger	F4	rearChargerSelect
Front Charger	F1	frontChargerSelect
Charger On	F5	energizeCharger
Front Battery Voltage	F6	frontBatteryCheck
Back Battery Voltage	F7	backBatteryCheck

TABLE 17:
VAN PIN ASSIGNMENTS FOR ATMEGA32U4

Connection	Pin	Variable in Code
Manual Actuators In	B2	manActuatorsEngage
Manual Actuators Out	B1	manActuatorDisengage
Front Actuator Location	F7	frontActuatorLocationPin
Rear Actuator Location	F6	rearActuatorLocationPin
Ignition Check	F4	ignitionSwitchPin
Driver Start Button	E6	driverStartButton
Lift At Top	B3	liftUp
Move Actuators In	D7	movActuatorsEngage
Move Actuators Out	D6	moveActuatorsDisengage
Error LED	F1	ERROR_PIN
Ready LED	C6	READY_PIN
Exchange In Progress LED	C7	EXCHANGE_PIN
Exchange Complete LED	F0	COMPLETE_PIN