WATECHPARK

SMART Parking Lot System

Vikas Sharma, George Alexandris, Elias Sabbagh

April 15, 2020
Computer Engineering Technology

Status

/1 Hardware present?

/1 Title Page

/1 Declaration of Joint Authorship

/1 Proposal (500 words)

/1 Executive Summary

Declaration of Joint Authorship

We, Student A (Vikas Sharma), Student B (George Alexandris), and Student C (Elias Sabbagh), confirm that this work submitted is the joint work of our group and is expressed in our own words. Any uses made within it of the works of any other author, in any form (ideas, equations, figures, texts, tables, programs), are properly acknowledged at the point of use. A list of the references used is included. The work breakdown is as follows: Each of us provided functioning, documented hardware for a sensor or effector. Student A provided VCNL4010 Proximity Sensor. Student B provided IR Beam Sensor. Daniel O Donnell provided LSR Camera Sensor/2 Stepper motors. Due to unexpected circumstances, Daniel's involvement with the project was passed to Elias Sabbagh joining the group during the Winter 2020 semester, to ensure successful progression of the project. In the integration effort, Student A is the lead for further development of our mobile application, Student B is the lead for the Hardware, and Student C is the lead for connecting the two via the Database.

Proposal

WatechPark- SMART Parking Lot System Proposal

From: Vikas Sharma, N01160135

Discipline: Computer Engineering Technology

Date: January 15, 2020

Background

This document will outline the software portion of the project in CENG 319 that will be coupled with hardware in CENG 317 for the final integration in CENG 355. Our project is going to be on a SMART parking lot system. Many busy parking lots are often plagued with congestion, drivers competing to find a spot by cruising around and visually finding spots. This is inefficient, time consuming where productivity is lost for consumers and businesses. The system we will be developing will address payment for parking, capacity management and location finding following an IoT approach using hardware and software.

Problem Statement

The problem being addressed includes, time spent searching for a parking spot, increased capacity levels during peak hours. This project is focused on solving these issues by connecting consumers to parking lot owners and providing parking services by using a more convenient, simpler method to retrieve parking lot data seamlessly.

Methodology

Phase 1: Hardware Design/Build

The small physical prototypes that we build are to be small and safe enough to be brought to class every week as well as be worked on at home. In alignment with the space below the tray in the Humber North Campus Electronics Parts kit the overall project maximum dimensions are 12 13/16" x 6" x 2 7/8" = 32.5cm x 15.25cm x 7.25cm.Keeping safety and Z462 in mind, the highest AC voltage that will be used is

16Vrms from a wall adapter from which +/- 15V or as high as 45 VDC can be obtained. Maximum power consumption will not exceed 20 Watts.

Phase 2: System Integration/Connection

This phase will be completed during the final semester of the Computer Engineering Program. The work gathered from both software/hardware courses will be combined and integrated for the final capstone project. The development platforms we will be working with is Android Studio 3.5.2, Raspberry Pi 4 Model B, and Google Firebase database. The mobile application provides key functionality to allow consumers to access parking lot data, view sensor/effector information specific to a location and choose the best parking space during different peak hours of the day. The VCNL4010 Proximity sensor will be used to detect the status of a given parking space at a specific time of access. The IR Beam Sensor will control the gate opening/closing and detect the presence of a vehicle near/far away. The LSR Camera sensor will be used for valid license plate recognition. The 2 stepper motors will control the gate and allow entry/exit based on the sensor data, and status of the lot.

Phase 3: Final Demonstration to Potential Employers

At this stage, we will demonstrate our 2 semester's worth of work to be assessed. Our project description/specifications will be reviewed by, Mike Wrona, ideally an employer in a position to potentially hire once we graduate.

Hypothesis

This project is focused on providing a solution for managing parking lot data, providing a less time-consuming experience with a simple, intuitive interface. This is an opportunity

to showcase our knowledge and understanding to build a collaborative effort for an industry sampled IoT project. I request approval of this project.

Executive Summary

In retrospect, this document outlines both the hardware and software aspects of the project. This project intends to build an IoT design that would help support industry related issues such as capacity management, location-finding by finding ways to reduce the time spent manually searching for parking spots. This document aims to provide insight into the design, development, testing phase of our SMART parking lot system project. In collaboration with our partner at ParkingBoxx, we have gathered our ideas to create a simple, intuitive and user-friendly platform for consumers within the market.

Our product aims to provide the essential needs for both consumers/businesses to view and manage parking lot data. In terms of market use, we believe through the project we will build a product that can be offered from an industry standpoint as well as be marketable to other fields of interest. Through the development of this product, we wanted to reach as many demographics and be able to provide an inexpensive and reliable platform where parking lot information can be retrieved at a glance. We offer users with the ability to be able to add/manage cars, view parking lot data, make on-the go reservations for parking passes, accessible via an online database to send/receive information in real-time, all built-in with a simple, effective interface. Due to these reasons, we believe it will be ideal to be considered to be hired by an investor for employment. This will be an extraordinary opportunity for us to be able present our work, knowledge and skills to promote our product from a marketing perspective.

Table of Contents

List of Figures

1.0 Introduction

This report will outline the development and integration of our final capstone project as part of the Computer Engineering Technology program at Humber. The individual team contribution for this project goes towards George Alexandris, Vikas Sharma, and Elias Sabbagh all 3 of whom were extensively involved with bringing the project to life. The focus was to implement an IoT (Internet of Things) design, where software and hardware interaction would be vital to address an industry related issue and help solve real-world problems. This project consists of a SMART parking lot management platform, which allows consumers/businesses the ability to manage and monitor parking lot data through real-time progression. The goal, being to address problems arising in the parking industry specifically in terms of capacity management, increased manual interference, and the lack of location-finding near/far from an area. The product looks towards determining the challenges in the parking industry today, and provide a platform to navigate to a parking space quicker, and in a more efficient manner. This includes, managing parked users, or monitoring the status of a lot at a given time. The main objective of this undertaking is to provide a more efficient and reliable platform to aid with parking scenarios. In particular, for the purpose of the consumer demographic who's in the market for an alternative parking lot management system. Our focus was to develop a platform, that would be the gateway to support consumers with finding the best parking space during any time, any place or anywhere in the world.

Project Schedule

The following is an overall breakdown of our work schedule for the duration of the entire project, of the two consecutive semesters:

GANTT CHART

	Task Mode	Task Name	Duration	Start	Finish	Predecessors
1		Requirements Gathering and Research	6 days	Tue 9/3/19	Tue 9/10/19	
2		First Meeting with Collaborator	0 days	Tue 9/17/19	Tue 9/17/19	1
3		Gantt and Mockup Design of Program	4 days	Tue 9/24/19	Fri 9/27/19	1,2
4		Distribute Work	1 day	Fri 9/27/19	Fri 9/27/19	2,3
5		SRS Document	3 days	Mon 9/30/19	Wed 10/2/19	3,4
6		Database and Schema Design	11 days	Thu 10/3/19	Thu 10/17/19	5
7		Work on code to implement screens	11 days	Thu 10/3/19	Thu 10/17/19	5
8		Second Meeting with Collaborator	0 days	Tue 10/8/19	Tue 10/8/19	
9		Presentation of Working Code. Submission to github	0 days	Tue 10/15/19	Tue 10/15/19	6,7
10		Implement database functionality with software towards beta release	21 days	Tue 10/15/19	Tue 11/12/19	9
11		Finalize Code	14 days	Wed 11/13/1	Mon 12/2/19	10
12		Meeting with collaborator and present project.	1 day	Tue 12/3/19	Tue 12/3/19	11
13		Final Submission	0 days	Tue 12/10/19	Tue 12/10/19	12

Figure 1 - Gantt Chart (Software)

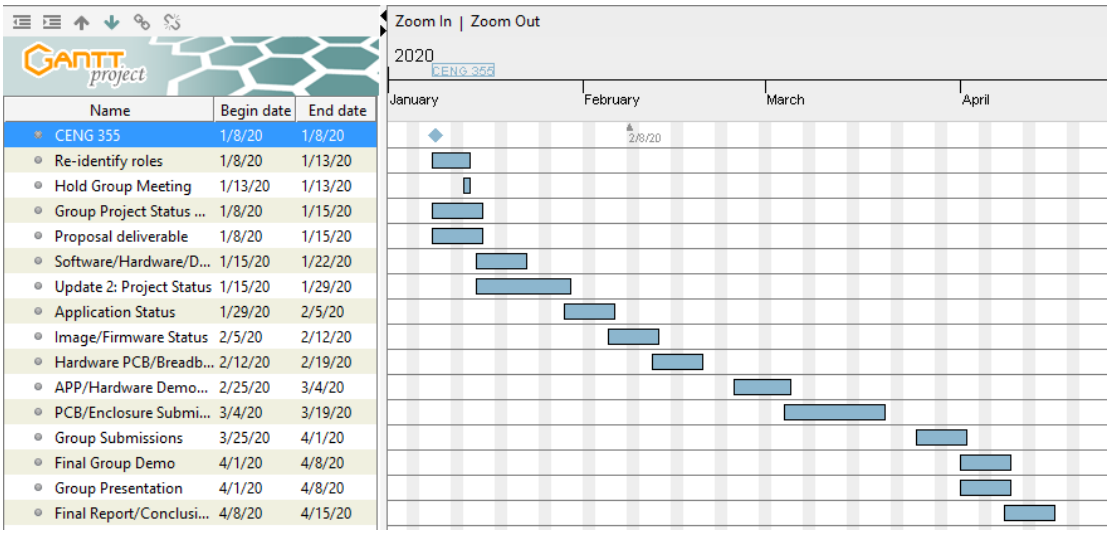


Figure 2 - Gantt Chart (Final Breakdown)

In terms of overall scheduling purposes, the project requires approximately fourteen weeks to complete during the Winter 2020 semester of the Computer Engineering Technology program. During the start of the final semester, the work done from the previous fall 2019 semester from CENG 319 and CENG 317 was carried over to complete the final integration. Most of the software development portion of the mobile application, and the connection to the database was done during the fall semester. Our group also ensured we were on the right track, by ensuring at least one of our team members can connect their sensor data to the database to send/display data to the platform during the fall semester. The core amount of work was accomplished towards the end of the Software Programming course, as well as in the Hardware Production Technology course. Our team followed our planned schedule efficiently throughout the fall semester, and carried the workload into the Winter 2020 semester. The final semester needed the group to finish implementing the sensor data to the Firebase database to be able to read/display data from the actual hardware. At this time, our team will extensively focus on completing the mobile application, and adding its intended features. As well as, re-designing our PCB boards to accumulate all three sensors from each team member into a final PCB design. The final semester will be used appropriately to ensure successful completion, and a polished end product.

1.1 Scope and Requirements

This report will address the key fundamentals of the project, including the software/hardware aspects of the design/specification and the final integration between these applications. It will touch on how the hardware (sensors/effectors) communicate and interact with the software application to allow consumers to track parking lot data.

Further details will be examined in this document, in regards to the hardware sensors used, the main objectives behind choosing the type of hardware and how it correlates to the application being developed. Along with, describing the database structure used to push and retrieve data from the hardware to the mobile application, and vice-versa. In terms of the scope of the project, there were instances where the planned outcomes from the initial stages of the development period had to be modified or re-considered. This includes, having some minor design/features changes and removals mainly due to time constraints or software limitations, to further increase productivity and meet the overall requirements. Some of these changes included, a major overhaul of the main Home screen to provide a smoother, sleeker interface for user interaction, removing Google API login integration, removing the search function from the mobile application after much consideration. A major change, occurred during the Winter 2020 semester, where our development team decided it was in our best interest to not move forward with the development of the business aspect of the application. This feature would have allowed users to track their daily progress and set goals to reach in the mobile application from a marketing standpoint. The team decided to improve on the existing consumer functionality instead, with implementing the proposed camera sensor functionality, and stepper motor gate control. Due to this reason, these features had to be omitted as a result of not meeting the plan, and the overall extent of the project had to be re-evaluated and worked on. Our team followed an agile methodology/procedure where work would be committed and evaluated on a daily basis to ensure the team stays on track. Development was split into distinct sprints of software coding/hardware testing to ensure both applications are met within the due time. During this time, coding

was separated into blocks of development periods where the team focused on completing a single feature, and aiming towards further progression from both sides. The limitations provided our team with an outlet, to focus on the main goal/priority and cut down on excess amount of work that can be evaluated later in terms of a marketing perspective. For example, one of the ideas the team came up with was introducing the business aspect through an online website where users would be able to view financial, user information along with providing identical software/hardware interaction. This SMART parking assistance platform provides real-time proximity measurements, lot detection methods with use of the hardware elements. Such as, the entry/exits of vehicles, detecting parking space movement, gate control.

Development Platform Specification

The following are the specifications of the software side of development. In terms of application use, consumers only need to know how to use a smartphone device. No technical expertise is needed as the platform we develop will be simple to use and gain a grasp of. The following are the list of software requirements vital for the platform to operate as intended:

- Android Studio 3.5.2 development platform
- Java (coding language) used for mobile application
- Software must have bilingual capabilities for English/French language integration
- Internet connection (Wi-fi) is needed to access the mobile application, and its main functions.

Hardware Specification

The following is a list of hardware requirements needed by the user to operate the application, and its functions:

- Raspberry Pi 4 Model B (CPU platform to process sensor data)
- Must support at least 2GB of storage, RAM
- Embedded CPU (Raspberry Pi) device will always need a connection to the server for the purposes of authenticating users, and receiving data.

Android Device Specification

- Must be running Android OS on mobile system
- Mobile APP will run only on Android devices
- API 21(Android Version 5.0 Lollipop) and above (supported roughly over 80% of the Android population)

Database Specification/Protocols

- Google Firebase database for storage purposes, push/retrieve real-time data from sensors to mobile application, and vice -versa.
- User Authentication

Protocols:

- HTTPS/SSL Encryption end to end communication
- TCP/UDP Connection
- Wi-Fi/Cellular connection

Server-Side:

- Email: SMTP (Simple Mail Transfer Protocol)

- Data transfer rates must be capped to not utilize an excess amount of data, depending on connection type (size must be compressed).

Report

/1 Hardware present?

/1 Introduction (500 words)

/1 Scope and Requirements

/1 Background (500 words)

/1 References

2.0 Background

In the industry today, there have many occurrences where parking in general has become a hassle for city residents and parking lot owners. This includes, not possessing the right tools to manage capacity when a parking lot is full, where drivers are struggling to find the best spot to park their vehicles. This can lead to dis-satisfying scenarios, where drivers are unaware of their surroundings, before even entering into the space. Due to this reason, it can lead to congestion in major traffic centric cities, with drivers competing to find a spot. This can be time-consuming, inefficient where productivity is lost for consumers and businesses. This project is focused on helping reduce the impact of this cause, by developing a system that will address payment for parking by taking an advanced and modern approach towards capacity management, and real-time information gathering to keep consumers up to date with their daily occurrences.

The group would like to thank Mike Wrona, installation manager, of Parking Boxx who provided support for this project. The project is a SMART parking lot system that incorporates a phone app to manage a user's tickets, account, and where to park in the parking lot. The idea of this project came up when the group realized that we can develop an easier way to find parking spots, by connecting all the spots to a parking app. We thought about creating an IoT parking lot that can connect to a database and update the database with information about its open/closed parking spots. It will be able to send and retrieve information about the parking lot. The mobile application will allow users to connect to the database and manage user accounts and payments for their tickets. Examples of some existing platforms are Indigo, BestParking, EasyPark and

ParkWhiz. These platforms have reservation capabilities, on the go parking with mobile or web application. What we are going to do differently from these companies is to integrate sensors to help users navigate to a parking spot.

One of the parking companies we looked at is EasyPark(EasyPark, 2016). EasyPark offers monthly payments for its customers to park in the EasyPark parking facilities. The goal for EasyPark is “provide safe, clean, friendly, convenient and affordable parking to the Greater Vancouver community” (EasyPark, 2016). EasyPark offers a phone app (EasyPark, 2016) called EasyPark Parking as well that allows it users to view parking available at its facilities. You can pay for parking on the phone app (Google Play, 2020). The parking app allows its users to register, find EasyPark parking lots close to you, keep track of how long and where your car is parked in the parking facility. Payment is done through the mobile platform with a much simpler method of operation.

We looked at the other parking companies that were mentioned such as Indigo, BestParking, and ParkWhiz. In terms of overall design and interface almost each application had full resemblance to the previous. Indigo offers a map that allows users to choose which parking lot they want to park at. Users can book their parking ticket in advance, reserve the spot for the designated parking facility owned by the company they want to park at, and they can see the rates of the parking lots (Park Indigo Canada Inc, 2019). Apps such as ParkWhiz, offers the options to add cars for verification, pay for monthly parking on the app and their ticket (ParkWhiz, 2019).

3.0 Methodology

The following section will outline and extensively explain our method of operation for the duration of this project. This includes the steps taken to ensure we meet the correct criteria in reaching the end goal. Following an agile methodology, we will keep working diligently to get each part of the application up and running. In the event of any outstanding circumstance, we will review the design and re-assess which parts are causing the issue and work around from there. Development and testing phases of the project will be split into distinct sprints of development. The next few branches of this section, will go into detail in regards to the required resources. Initially, focusing on the parts/components/materials used to complete the project, the manufacturing process for PCB and enclosure, tools/facilities utilized, shipping/duty/taxes and lastly touch on the working time versus lead time needed to complete the final hardware integration for the project.

3.1 Required Resources

Report

/1 Parts/components/materials (500 words)

/1 PCB, case (500 words)

/1 Tools, facilities (500 words)

/1 Shipping, duty, taxes (250 words)

/1 Working time versus lead time (250 words)

3.1.1 Parts, Components, Materials

Parts and Components

This project consists of a variety of parts, components and materials which were utilized during the development, testing and final integration phase of our parking application. These components were essential to different features/functionalities of the parking platform and vital in terms of performing each functionality of our application. The project made use of three primary hardware sensors/effectors. These included, the VCNL4010 Proximity Sensor, IR Break Beam Sensor (Receiver/Transmitter), USB Camera Sensor (YoLuke HD Webcam). The CPU Broadcom Development platform chosen for the project is the Raspberry Pi 4 Model B. The Raspberry Pi 4 is the latest piece of technology that is capable in providing multi-support interfaces, at much higher speeds. Firstly, for the VCNL4010 Proximity sensor, we made use of the following hardware components: 3 (220) ohm variable resistors to limit the current flow through the circuit, a Common Anode RGB (red, green, blue) LED which is used to detect and react to changing states in proximity readings/parking lot status, 20x2 stackable header, to mount the PCB board to the Raspberry Pi, 6 pin stackable header, to mount the sensor to the soldered PCB board, and finally a set of jumper wires(female to male) for the breadboard testing portion. For the IR Break Beam sensor, the components used include the following: 4.7K ohm resistor, 1.0K variable resistor used to detect if a vehicle is in the way of the gate to allow entry/exit when the beam is broken by an object in the way of the transmitter/receiver. The Raspberry Pi 4 Model B microcontroller will provide the key functionalities for the VCNL4010 Proximity sensor, IR Bream Beam sensor, and

the USB Camera sensor alongside the 2 stepper motors used for parking lot gate control.

Furthermore, other components used include a power supply alongside an Ethernet to USB (Universal Serial Bus) adapter cable. These components will provide the interface to connect to the Broadcom Development platform through either remote desktop connection or via VNC Viewer connection to further develop, integrate the sensor/effector functionalities. An SD card will be used to provide the imaging/firmware configuration settings from an Ethernet connection, using the universal serial bus to Ethernet adapter.

Along with these essential parts, multiple jumper wires were used to connect to the GPIO (General Purpose Input/Output) pins of the Raspberry Pi device. The VCNL4010 Proximity sensor, and the IR Beam sensor will be used under a I2C interface. Using a 40-pin stackable header for the GPIO pins of the Raspberry Pi, the VCNL4010 Proximity sensor and the IR Beam sensor will be connected to the Raspberry Pi using its general-purpose input/output pins corresponding to each circuitry design for each sensor. These sensors will then communicate based on the surrounding environment, and its conditions. For example, detect whether a vehicle is in the way of the gate or a space is being occupied/ statuses of the lot. These components will then interface through the real-time database setup through Firebase database to push/retrieve parking lot data/statuses actively. The USB Camera sensor will be used for license plate recognition and capture an image of a valid license plate at entry. The data captured will be sent to the database for further examination. More details in terms of the mobile

application, and how the data is presented will be explained in the Development Platform section under the Mobile Application branch in this report.

Additionally, three stackable headers were needed to mount all three sensors to the PCB (printed circuit board) for the final PCB design process, further detailed in the Manufacturing part of this section. Our team focused on retrieving parts and components that were feasible and inexpensive to our planned budget and would provide the best quality of performance. The parts made use of for the project included the presented ones above, including the added components needed to successfully test and implement each sensor.

Materials

The materials to be used as part of the project, includes the following: laser-cut acrylic for the final enclosure design, to protect all three sensors from potential harm/damage prepared in a suitable housing environment. Other materials included making use of a breadboard to test the hardware sensors and the corresponding circuits that were built. Some other materials used included, the printed circuit board constructed and etched through sheets/layers of copper foil, and glass fiber material (fiberglass epoxy resin). These dielectric materials, such as glass fiber was used as an insulating layer to create the PCB from the top/bottom layers of the printed circuit board. Through the laser-cutting work achieved for the enclosure, acrylic material will be used alongside plastic housing to support the hardware and provide panels for the safety of the Raspberry Pi platform, printed circuit board, and its sensors. Acrylic material is lightweight, superior in quality and provides the best resistance for each sensor housed in the case. The enclosure will be able to offer a clearer insight into the different components and

sensors/effectors and provide high protection for all hardware components assembled. The genera layout of the case will follow an acrylic design and structure to house the sensors/effectors used for the project. All-inclusively, these parts, components and materials gathered will be important in the integration effort to ensure successful completion of the project, following our planned methodology and strategy.

3.1.2 Manufacturing

PCB/Case

In terms of the manufacturing process, most of the components gathered were ordered through the Amazon, BuyAPi, and Adafruit industrial companies. The printed circuit board designed was also manufactured and produced by the prototype lab, through tuition paying students with an approximate cost of $6.65(CAD).

The final PCB and enclosure designs were all discussed in this semester including completed PCB and case designs from last semester. Last semester in the Hardware Production Technology course we had to design PCB’s and enclosures for the sensors we required for the final Parking Lot project in this semester. In this part of the report, we will take a look at the previous designs of the PCBs and cases being used in the project, and how they will be integrated into one final design.

The IR Break Beam Sensor project was designed by George Alexandris. The IR Break Beam sensor had a few design changes over the Hardware course last semester. The first version of the PCB design for the IR Break Beam Sensor did not fit onto the Raspberry Pi 4 Model B. The size of the first version was too large to be put onto the Raspberry Pi. The second size of the PCB was smaller and was able to be fit onto the board successfully.

Below are images of the second version of the PCB in Fritzing and the printed PCB:

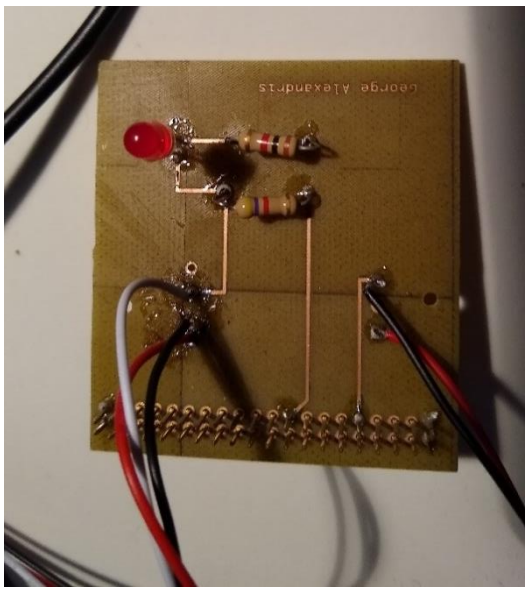


Figure 3: Printed PCB from Fritzing file

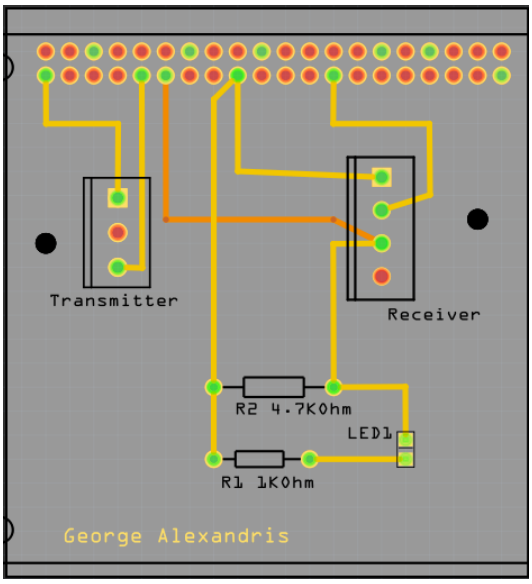


Figure 4: Image of PCB Design in Fritzing

PCB's were all designed in a free program called Fritzing and were printed in the Prototype Lab at Humber College. IR Break Beam sensor is in two parts one is the transmitter which as two connections directly to 3.3V power and ground on the board. The other part is the receiver which has three connections one for 3.3V and ground directly to the Pi while the third connection is to one of the GPIO pins. The GPIO pins can be setup for either input or output but the third wire from the receiver is a digital output so the GPIO pin it is connected to is set up as an input to read the value from the output wire.

The enclosure design for the IR Break Beam sensor project went through many design changes as well. The design for the case had to consider the size of the Raspberry Pi and the PCB for the IR Break beam sensor. The cases were designed in CorelDRAW and laser cut in the prototype lab. Early designs for the case of the IR Break Beam

Sensor Project did not fit the sensor and the Raspberry Pi together in one case. The PCB of the IR sensor was long and the early design for the enclosure was too small in length for the case to fit. The measurements were taken to adjust the size of the case so the PCB and Raspberry Pi can fit together. The new design was longer for the PCB and Raspberry Pi to fit and the final design is shown below.

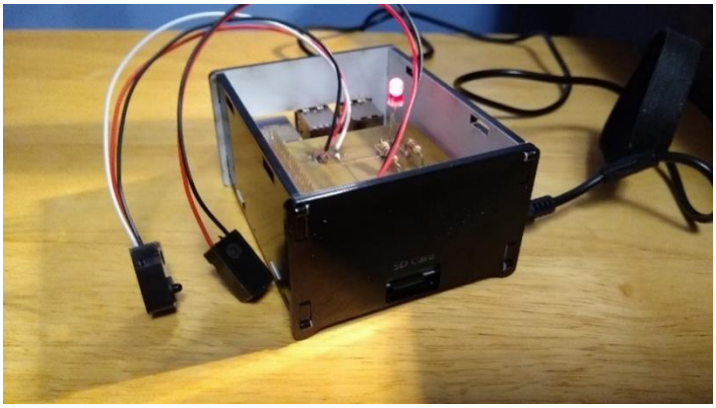


Figure 5: Final Case Design for IR Break Beam Sensor

The second sensor being used for the project is the VCNL4010 Proximity/Light Sensor. The PCB and case for this sensor was designed by Vikas Sharma. This step required use of the Fritzing software, which is an open-source application that allows users to visually create schematic, breadboard designs and finally incorporate their designs into a printed circuit board to be sent for etching and laser-cutting services. Over the course of the semester, Vikas had many different design flaws for the PCB. This was due to connections to the top layer of his PCB. In previous designs there were overlapping connections for the top/bottom layers, so Vikas had to redesign his PCB many times to not overlap his connections. The connection for the GPIO pins has to be on the top

layer because of soldering the PCB to a 40-pin connector for the Raspberry Pi, it is not possible to solder the bottom layer.

The PCB board would require a lot of focus and attention due to the high potential of having connections accidently being joined together, which may result in a redesign of the whole board. Vikas was able to luckily not have to go through that process, while soldering. For his design, there were 4 vias that needed to be soldered. You must thread a single thin wire through the holes, solder it and then cut off the remaining excess wires. The same process, was taken for the resistors where Vikas set up the resistors in their place and had to solder the 2 sides. Then, cut off the excess wire. While soldering the LED, Vikas had to be very cautious as the connections were designed really close to each other, so a lot of focus was needed here. Initially, there is a 6-pin header that comes with the sensor that needed to be soldered. Vikas also soldered the 40 pin GPIO pins on the PI, to keep the header stable while soldering and to create a sturdy connection. After soldering, the board should be ready to be tested, and should look like the image on the left below.

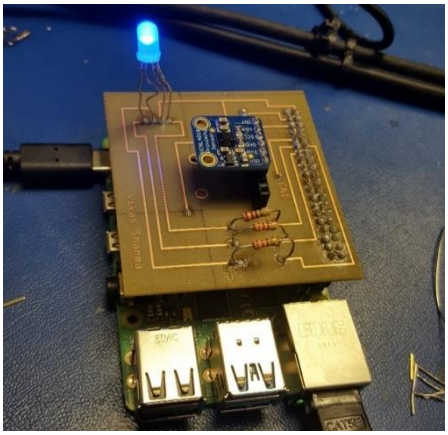


Figure 6: PCB final Design for VCNL4010

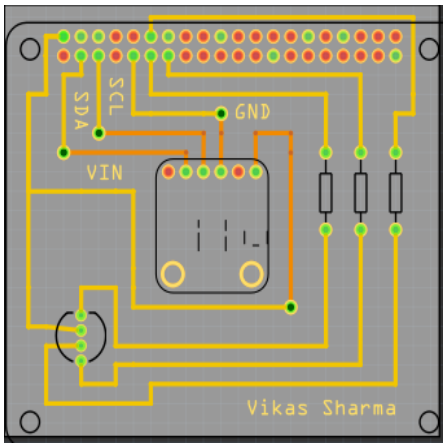


Figure 7: PCB Design for VCNL4010 in Fritzing

Once the PCB board testing phase was successfully complete and Vikas was retrieving/sending data to Firebase and the mobile application, he moved on to the enclosure design . For this, he decided to use the CorelDraw software as it seemed like the best choice to design the case the way you want. Vikas went with a acrylic case design, as the case is to prevent/protect your Raspberry Pi and sensor from potentially being damaged. While making the case, you have to be aware of the dimensions of the Raspberry Pi (height,width from all sides). For the design, he had to make some changes due to using the Raspberry Pi 4, some ports are arranged differently. Such as the LAN (ethernet port), and the 2 micro HDMI ports. For this, Vikas decided to make a wider port/hole to fit both mini HDMI ports into one. Vikas ran into a issue where the hole for the power port was cut too short, for this he had to shave off some excess acrylic to get the hole to be wider, to fit the power cable into the port. Vikas was able to fit it perfectly in the outcome. For the design, Vikas kept the top of the case open with a medium-sized hole, as there needed to be a object in the way of the sensor or far away for it to display the readings.Once all the requirements for the dimensions of the openings were met the final case design was complete.

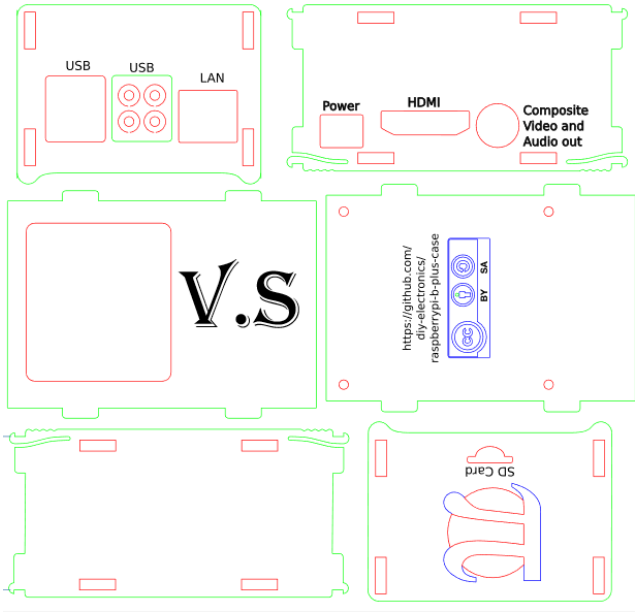


Figure 8: Final Case CorelDRAW design for VCNL4010

Figure 9: Printed Case for VCNL4010

The last sensor, Yoluke HD Camera Sensor, did not have a design for an enclosure and its PCB since it is just a USB camera. The servo motor however, does have a PCB design and will be shown below.

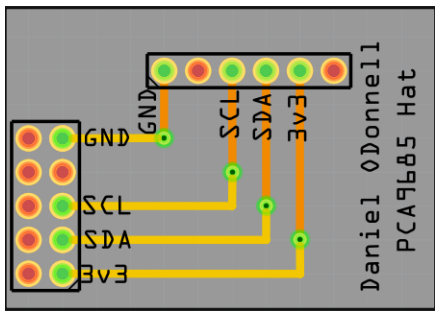


Figure 10: Fritzing PCB design for Servo motor

Taking into account the previous semesters work, the initial plan which will be immediately targeted is to combine all three sensors into a single printed circuit board

entity. This will be done to ensure progression of the project, and meet the requirements for this semester in completing the capstone project. After testing each sensor, and connecting it to the database and the mobile application we will design a new PCB using the Fritzing software. This will enable us to work with all three sensors using only one Raspberry Pi 4 Model B Broadcom Development Platform, connected to a single PCB in the final stage. Elias will be responsible for helping with the hardware implementation and design the final PCB through the open-source software. Along with gaining further support from Vikas in working with the VCNL4010 Proximity sensor, IR Break Beam sensor, and the USB Camera sensor to read/display the values according to the parking lot data/changes in status. Using the I2C interface we will design the final PCB through integrating each sensor into a single design for multiple sensor support. The IR Beam sensor will be extensively tested for the gate control mechanics and George will be accountable to ensure the hardware functions as designed.

Towards the final stages of the project, a re-design of the enclosure will be needed for a cleaner, sleeker and compact design. It will be important to house all three sensors which will be part of the final PCB design, and create a suitable and appropriate design to meet the requirements for each sensor. A collaborative effort will be made from all three members of the WatechPark developer team here, to redesign and refine the casing using the CorelDraw software. The prototype lab will be used for further inspection, and laser-cutting services during this phase of the project. Generally, this section helped explain the design process/experience with the hardware PCB and case. This included, providing an individual look at each sensor, and its printed circuit board design process and the work that was put into each hardware project initially. Finally,

this section addressed the final PCB, casing design and integration that will be the focus for this semester with the criteria mentioned above.

3.1.3 Tools and Facilities

Throughout the duration of the project, for accommodating both the hardware and software side, it was necessary to work with a set of tools and environments to ensure successful completion, and the planned outcome for each hardware component. The tools used to complete the project ranged from the initial design, development, testing and presentation phase of the overall end product. To begin, in terms of the PCB (printed circuit board) process, was the requirement of using copper wire, and the pin headers that came with each sensor, wire strippers to cut off excess wire during soldering the PCB together. The copper wire would be used to solder the components on the printed circuit board together, using a soldering iron provided by the Prototype Lab in Humber College itself. The soldering iron was available from the parts crib and most of the soldering was done in the prototype lab itself. Wire cutters were used to cut off excess wires from soldering the VIAs, or cutting down on extra wires from the components used for the PCB.

The electronics lab facility provides the most necessary components at your disposal. This includes, the PC system itself where most of the configuration was done for each sensor, a soldering iron including the solder itself. Major work was accomplished in the prototype lab, where the solder was received through the parts crib and was used in conjunction with the soldering iron available at the station. The following steps were taken to solder, firstly it is recommended to heat the soldering iron before applying the solder at around the temperature of 360 to 370 degrees Fahrenheit. The solder can

then be applied to the tip of the soldering iron to produce the best results before starting the process. Other tools used in the production of the PCB (printed circuit board) included a helping hand. This tool allows you to reposition the PCB at different angles to solder from different perspectives, to avoid any overlapping connections or accidently joining a connection together. This tool is used to angle the printed circuit board and hold the board while you solder the components as necessary, without having the need to manually re-adjust the board at different times of soldering.

The facilities used in the project, included the prototype lab in Humber College. This facility is the main source of providing the services to etch the PCB board during its final stages of production, as well as provide the laser-cutting services for the final enclosure design. This facility was used to solder the components, sensors together on the PCB and test the final design. Through this environment, we were able to learn the process of soldering, and using the human resources in the facility we were able to make the best decisions, or re-think our strategy to solder the PCB. The facility provided us with more viable options, recommendations and the best way to overcome any issue we were experiencing. This included, designing the PCB board, using the CorelDraw software for housing the sensors in the final enclosure developed. Adjustments were made depending on the different scenarios, and based on the advice of Vlad and Kelly whom were present to help. Along with their support, we were able to come to re-assess, learn and fix our design mistakes to create a more polished end product for both cases. These services were provided by the prototype lab, for both etching and cutting services for the printed circuit board and the acrylic casing, with a laser-cutting machine being available to us aside from our tuition. Overall, these tools and facilities

supported our team and our project in achieving its goals and provided us with the outlet of both human and machine resources. This was accomplished through the help of others in the facilities, or through etching and cutting services provided to meet the project requirements in the end.

3.1.4 Shipping, Duty, Taxes

The budget for the project will be discussed in this section in terms for where we ordered our parts, how much it was shipping, taxes and totals will be shown. Most of the parts were ordered from last semester. George ordered all the parts he needed for the IR Break Beam sensor project, Vikas ordered his components necessary for the VCNL4010 Proximity/Light sensor and the final parts were the 2 Servo Motors and YoLuke HD Camera Sensor.

The components utilized for this project were ordered ahead of time to ensure work can be progressed or to test any faulty hardware piece for a quicker solution. Shipping time for most of the components was divided between 2-5 days, depending on the type of service chosen at the time of the transaction. The shipping options ranged from DHL Express shipping which transits worldwide and internationally. For example, for Vikas's case while ordering the VCNL4010 Proximity sensor and its components at the time of the order, he chose his option of DHL Express shipping. This method of shipping provided, a much faster, reliable form of service with the components arriving roughly 2 days later from when the order was placed. In this case, the shipping cost was divided between two totals as shown in the screenshot below in the red outline. The shipping method and time required was lower than expected with other methods of shipping requiring 5 business days to ship the product from the source to the destination.

Duty was not placed, and avoided at all costs during this time to prevent any further costs from materials being shipped internationally or from other countries. For George's case, most of the products shipped were either from Amazon, or through BuyAPi, with a single purchase for the IR Beam sensor being through Adafruit. The method of shipping was also different here, as with Amazon it provides the option of Amazon prime shipping. This depends upon the location the component is being shipped from of course, and based on that location, the time is calculated to ship the product out. Although, with Amazon Prime shipping most orders placed during a significant time of the day can be processed, and shipped within 2 days and delivered at a much quicker rate. Other hardware parts required a shipping time of 2-5 days with UPS or different forms of shipping requiring higher costs, if the item is needed within a set amount of time. For the most part, the components used in this project were shipped out within 2-5 days, and duty expenses were avoided to ensure an inexpensive project or to avoid any further delays.

Taxes were applied accordingly as well, with US (United States) tax rate being applied to the parts that were purchased through online shopping. Since these rates were in US dollars, we had to convert this value into CAD (Canadian) currency with the rate of 13% being applied instead as displayed below.

Below is the total project cost for the project, in terms for all the parts we all have currently in a BOM (Bill of Materials) format:

Product Name	Supplier	Cost (CAD$)	Number of Units	Tax	Shipping	Subtotals	Totals
Raspberry Pi 4B (4GB RAM)	BuyAPi	74.25	1				
MicroHDMI to HDMI cable	BuyAPi	6.45	1				
Raspberry Pi 15W Power Supply	BuyAPi	10.95	1	13.34	11	115.99	115.99
AmazonBasics USB3.0 to Ethernet	Amazon	20.05	1				
AmazonBasics Ethernet Cat-6 cable	Amazon	6.56	1	3.46	0	30.07	30.07
Raspberry Pi 4 case	Amazon	19.75	1	0	31.99	51.74	51.74
IR Break Beam Sensor	Adafruit	1.95	2	0.51	22.05	26.46	26.46
Total for IR Break Beam Sensor project :							224.26
Raspberry Pi 4B (2GB RAM)	Adafruit	59.65	1				
VCNL4010 Proximity/Light Sensor	Adafruit	9.95	1				
Power Supply	Adafruit	10.54	1	15.88	16.46	112.48	112.48
Jumper Wires	Adafruit	2.59	1				
SD Card	Adafruit	19.82	1				
Ethernet Cable	Adafruit	3.65	1				
MicroHDMI to HDMI cable	Adafruit	11.87	1				
Tri-Color LED	Adafruit	2.65	1				
Stackable Header	Adafruit	3.91	1	27.66	29.25	73.77	73.77
Total for VCNL4010 project :							186.25
CanaKit Raspberry Pi 4 Starter Kit (4GB RAM)		134.99	1	17.55	0	152.54	152.54
AmazonBasics USB2.0 A-Male to Micro B		10.39	1	1.35	0	11.74	11.74
Elegoo MB-102 Breadboard (Bundle of 3)		10.98	1	1.43	0	12.41	12.41
Jumper Wires/Dupont Cable Multicolored(M-M M-F F-F)		9.99	1	1.3	0	11.29	11.29
Adafruit 16-Channel 12-bit PWM/Servo Driver -I2C interface - PCA9685 Createtronic		19.84	1	5.45	22.05	47.34	47.34
Power Adapter Supply DC 5V (2A/2000mA) (Power Servo Driver)		12.98	1	1.69	0	14.67	14.67
ANIMOS Digital High Torque Servo 20 KG Full Metal Gear(2PCS)		25.99	2	3.38	0	58.74	58.74
USB Camera		15	1	1.95	0	16.95	16.95
Total for HD Camera project:							325.68
Total for the full project:							736.19

Figure 11: Final Project Budget

Most parts were ordered last semester and were used to complete their individual project. The total for the IR Break Beam sensor project individually was $224.26 CAD. The total amount for the VCNL4010 Proximity sensor project alone was $186.25 CAD. The total amount for the HD Camera project developed from the previous semester was $325.68 CAD. The projects involved are the IR Break Beam Sensor, VCNL4010 project and the HD Camera Project. They were all calculated from last semester's budget. There will be extra parts ordered for the final project and will be added to the total budget. These extra parts are not taken into consideration, the group is deciding on what extra parts we will use for this project. Such as, potentially ordering 3 more VCNL4010 Proximity sensor to accommodate our final parking lot prototype model with

4 parking spots. We plan on using all the parts ordered from the previous semester in the new semester. Some parts not considered in the budget are the parts being used from the parts kit such as some resistors and LEDs. The parts kit was purchased from the first semester and some of those parts are being used for the final project. This section helped provide a better insight into the shipping, duty and tax rates that went into ordering the components needed to complete the project and the financial state of the project before/after further development.

3.1.5 Time expenditure

Throughout the development of The WatechPark project, our overall time was split between hands on tasks, such as discussions, development, designing, building, debugging and testing, while the rest of our time was spent waiting for external tasks to be completed of a similar kind as parts delivery, laser cutting and PCB printing.

Please note that the time calculated in the following list, represents the total time spent by the three members of the project. For example, when discussing the time spent on designing the first PCB and the finalized PCB, in reality it reflects the collective time spent designing three PCBs for three different sensors and an additional three finalized PCBs for those sensors. This section of the report will touch on the total working time allotted to complete each requirement for the project, as well as address the amount of time needed to manufacturer the component from the total lead time. The following is the total breakdown of each section of both cases.

Total of 105 hours spent as Working Time:

- 12 hours on project discussions and researching the required sensors and controllers
- 6 hours on finding the best vendor in terms or pricing and shipping time, and ordering the required parts, components and materials
- 6 hours on building and assembling breadboard prototypes
- 3 hours on voltage and safety testing for the breadboard connections before connecting our development platforms
- 6 hours on researching and installing the necessary third-party libraries for the sensors
- 6 hours on developing testing programs to be used on bread board circuits
- 6 hours on testing and debugging any anomalies in both the software code and the hardware connections
- 12 hours on designing and submitting both first and the finalized PCBs to be printed
- 6 hours on PCB assembly and soldering the needed components
- 3 hours circuit testing and safety verifications before connecting to the development platform
- 8 hours on designing, laser cutter submission and assembly of the PCB housing cases
- 15 hours on designing and developing the WatechPark android application
- 6 hours on Firebase initialization and filling up with data to be used with our android application

- 10 hours on testing and debugging our android application to ensure expected functionality with simulated sensor data stored in Firebase

Total of 22 days concurrent as Lead Time:

- 5 days on parts to be shipped and delivered
- 10 days on PCBs to be printed
- 7 days on housing case to be laser cut

3.2 Development Platform

Our WatechPark application is centered on three main forms of criteria, this includes the front-end, back-end, and on-site devices used to interact with both the hardware and software aspects of the project. The front-end of our project consists around the use of Android devices/operating systems running Android Version 5.0 (Lollipop) and above. The main software utilized in terms of development, and further progression of the application was done through working with the Android Studio development environment. This was where a large majority of the coding, designing and software implementation was accomplished in terms of the mobile application side, which was then used to interface with the corresponding hardware elements. Coding was done through the use of the Java programming language, for further establishment of dedicated features or desired plans. This interaction was accomplished through our Broadcom Development platform, the Raspberry Pi 4 Model B. This CPU (central processing unit) was utilized to process the sensor data, readings along with the on-site devices, being the sensors/effectors used for our parking lot system. This will all be coupled with the back-end aspect, being the online database. In this case, the Firebase database will be used as a computational/server platform to retrieve real-time parking lot data, and allow the consumer access to parking lot information vital to a specific location. Collectively, these three aspects will serve as the primary connection to our project, and will act as the overall support structure to keep the work progressing through different heaps of development.

Student A (Vikas Sharma) will be addressing the development of the mobile application in this section.

3.2.1 Mobile Application

Memo/ Mobile Application Integration

Initial development of the mobile application, began during the Fall 2019 semester as part of the Software Project class. During this time, our team worked extensively to build a system capable to store and retrieve parking lot data with use of an accompanying online database system. The Android application developed interacts with our Broadcom Development platform and the on-site devices, being the sensors/effectors to relay parking lot data, and monitor lot statuses at different intervals of the day. The goal for the mobile APP itself, being to allow remote access to parking lot locations and discover parking lot statuses in real-time through the working hardware assembled. The task presented for the duration of the semester, was to develop an Android mobile application that helps solve an industrial issue in the real word, working with sensors/effectors and an online database to delegate the data retrieved or sent. Based on our mockup designs created through the Balsamiq software program, we initially intended to focus development on two separate modes, consumer and business. We had planned to develop the consumer application during the Fall 2019 semester in the Software Project class, which we were able to successfully implement, with all proposed features, and some minimal changes/removals due to software limitations or constraints. The plan was to create the consumer side to allow the user to access parking lot data with a majority of the features being available at your disposal. The idea with the business side, was to allow the user to track daily goals, set goals for further achievement in the application, and view the overall history of parking passes purchased, payment details, or vehicles added to the system following an advanced

approach. After much consideration, our team decided it was best to stick with the current consumer application instead, and focus on improving on the features already included, putting aside any business aspirations for potential future considerations. Instead of this idea, we devoted our full attention on the features needed to be implemented for the mobile application, and our parking lot system platform. This included, developing the functionality to allow access to parking lot data in real-time and view/monitor changes in status, rather than the current sample data coupled with the Home (Main Dashboard screen) in the application. This allowed us to focus on what is most important, and not waste valuable time on features that are additional or can be constructed in later development of the project. Therefore, the main reason to drop this idea being to the fact that it would have increased programming complexity and thus created more work to be done than is required.

Currently, our mobile application is capable of providing the key functionalities/features initially proposed during the startup phase of developing the application. This includes, allowing consumer access to a online database to monitor/display details on authenticated users in specific account information, login/register new or current users, make on-the go reservations for particular parking lots based on consumer choice, view parking lot details (in terms of sensor data, status of the lot during different instances), manage/add cars, select parking passes, payment, manage settings, help/about options. This includes, multi-support offerings of English/French language integration. Along with, automatic retrieval of transaction information through the Order History screen, working with the Firebase database to display the data back to the mobile application.

Features not yet deployed for the application, although planned to be implemented during this semester include the following: view a parking lot, monitor statuses of the lot, including overall capacity, gate entry/exit control and will help present a visual representation of the lot and the real-time changes through the use of the sensors/effectors. This feature will be implemented through this upcoming month, and work will commence in the upcoming days to ensure the feature is successfully completed by the set amount of time needed to complete the project. As of right now, our team is currently at pace in terms of continuing development of the mobile application, and actively listening for changes, removals or any potential additions to support the current state of the project. Fortunately, we were able to accomplish, and test each feature implemented from the previous semester at this time to make sure each planned feature works, and the key focus of the application is tackled. The application is designed to provide more functionality, and less cluttering of screens to encourage a further customizable experience, where the user feels comfortable using the application and its UI (User Interface).

Login Activity/Authentication

At the start of opening up the application, the user will be prompted to a screen with options to log-in, register an account, or reset your password using the Forgot Your Password screen or Verify Your Password for different forms of authentication services in-application. We constructed an authentication process, in which we believe is a simple, intuitive interface for storing/checking credentials through secured database access. The following is a portrayal of the Register/Login screen which is automatically prompted to the user at entry of the application.

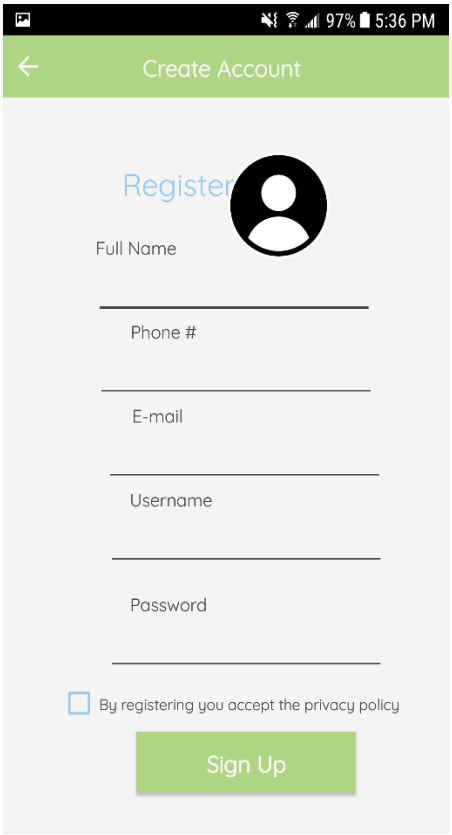


Figure 12 - Register Screen

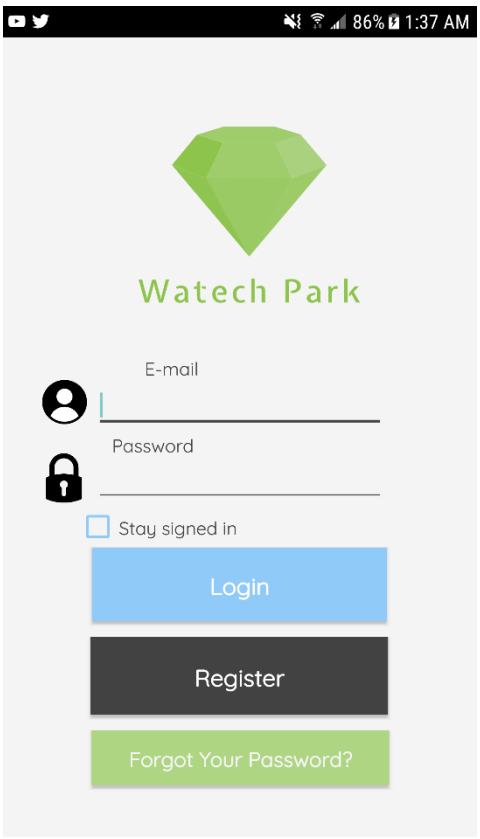


Figure 13 - Login Screen

The authentication part was designed with having a login/register screen separate from each other. The Register screen prompts user input in terms of personal information specific to a user. There is also the option for the user to choose a profile image from a real device through Firebase external storage permission. This account information is submitted onto the Firebase Real-Time database under a data structure, which holds the data. This information incudes, the full name, phone #, e-mail, username, image uploaded. So, once a user registers into the system the data is sent to the Firebase database, and stored to check for validation through login attempts further down the line.

After registering, the next step is to login. To login, we used the Firebase Authentication system for e-mail and password. This is a built-in feature of Firebase, that is used to authenticate a user that exists in the database. So, in the Login screen, the user would sign-in with the “registered” email and password. Firebase checks for valid/invalid credentials based on the information stored in the Firebase database section under that UID (unique identifier). User selections are stored with a “Stay signed in” option. This means, account information is visible after the user leaves the session, or returns to resume the activity. At this time in Firebase, the logged in user would appear in the Authentication section with the corresponding email/password information.

The authentication process continues with the “Forgot Your Password” and “Verify Your Password” screen. The Forgot Your Password screen is used to allow the user to reset their account password. There is the option to use e-mail or phone # authentication. This authentication is done through the Firebase database, where once a consumer selects a service, the other unattended service is not allowed to be accessed. The user enters an email address and through valid checking Firebase will then send a verification email to the corresponding email address. Phone number authentication requires the consumer to enter in a valid phone number using registration data from Firebase. Once the phone number has been validated, and follows the required system format (+1) a verification code is sent through the phone service provider.

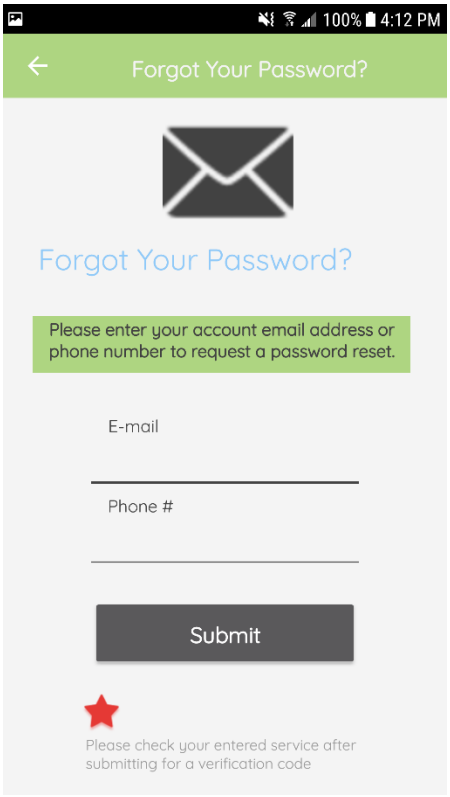


Figure 14 - Forgot Your Password Screen

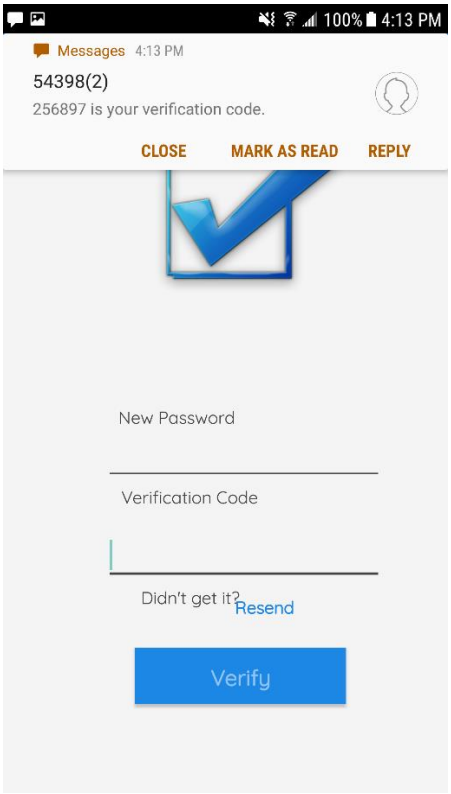


Figure 15 - Verify Your Password Screen

The mobile application also holds a feature, where account information is displayed specific to a user, in this case it would display different details based on the active UID. The screen followed a general layout, a little different from having to fit everything into a fragment view, we decided to use a new separate screen. This screen displays account information from the Register screen. The data is populated and displayed. Such as, the profile image stored from Firebase Storage, phone #, e-mail address, the name and a timestamp. Although, for this semester any current features/screens or designs are subject to change for that matter. We want to focus on the main functionalities of our application, and then move on to what can be added additionally, to further enhance the end product and meet the capstone project requirements.

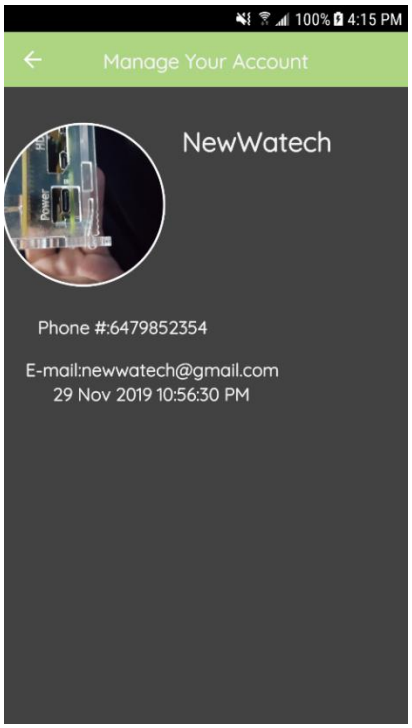


Figure 16 - Manage Your Account Screen

Data Visualization Activity

Once the user has logged in, the main home screen appears. This is where the parking location data is displayed, and the sensor readings are gathered regarding each location with name, distance, and the total price of the parking pass. We believe this is the most streamlined approach taken for the users, and this allows easy and simple access of the data, and the activity all available within one main screen. This includes, proximity readings from the actual hardware, reservation capabilities, parking lot details and the actual lot itself visually represented in real-time. For this screen, we redesigned the main-menu from the proposed version, finally having a much sleeker interface to display each parking lot location, and its information to the screen in a more convenient manner. So, as shown above this screen offers two options to the consumers. This being “View Details and “Reserve”. View Details is where an expandable view pops up

of the lot with an image, cost and the real-time proximity level of the lot at that particular time. At this stage of the project, our intentions are to build on from this feature, to provide access to the real-time parking lot prototype to be constructed during the duration of this semester. Currently, we are at the stage where we have figured out where the data will appear, once our physical parking lot model has been designed, and established. Then the goal here, is to modify this screen to show real data rather than sample data as is currently. The feature will allow further optimization of the hardware sensors/capabilities, with the plan being to provide the user with a visual representation of the application to go along with the physical hardware to be assembled later on in the upcoming month. Such as, display the parking lot and the parking spaces to be the focal point of testing purposes down the line. We have decided to create four main parking spaces, and these hardware aspects have to be mirrored within the mobile application. For example, the application will be able to show real-time changes in each parking space, individually or collectively and display data/readings fetched from the sensors. This includes, statuses of the lot (whether open/full), each parking space and changes viewed from the lot, gate entry/exits and the information will be displayed to the screen accordingly based on the conditions, from the actual hardware and sensors. Some of these conditions we have considered include the following: a vehicle enters the parking lot platform, the camera sensor on-site will immediately capture a image of the license plate, after validation/condition-checking entry is then allowed and during this time the data visible on the application should show four empty parking spaces in the lot. To indicate this, we will use three main colors: green, blue, and red. The light green will be used, when there are no objects/vehicles present in the parking lot. The blue color will

be used to indicate a car is approaching and a message will appear on the application to guide the user to the next status change. The red will be used to indicate when a parking space is being occupied. Once a lot is full, the application will display this data through connecting to the database and receiving each sensor and its contents. So, in this case, entry will not be allowed further, so the application will be displaying a message saying “Lot A is Full”, or vice-versa depending on the situation. Further details on hardware will be discovered in the Breadboard/Independent PCB’s part of this section.

This data through the support of the hardware and the connection to the online database, will be further evaluated and assessed, to then be eligible to be displayed on the mobile application interface. This functionality is yet to be implemented into the final design, and will be the key focus throughout the upcoming month of March, and in Mid-February as it is the backbone/ purpose of what we set out to accomplish with this parking lot management platform. Currently, we have started initial development of the feature, and refactored the previous design into the new one, to further gauge on what is really needed and what features are not required.

Once the spot has been reserved, the corresponding data is sent automatically to the Parking Passes screen. At this time, the parking location that has been reserved on-the go, and its details as represented in the design are sent to Firebase through sample forms of data

If the parking lot has been reserved successfully, a notification pops up on the device presenting the reserved lot and to view the parking passes for the next step. By swiping to the left of the screen, brings up the side navigation drawer that is used to hold the

main/other fragments and features of the application. This allows the user to access every main feature of the application within a gesture, and to avoid having to go through many hurdles or displays to get to their desired destination.

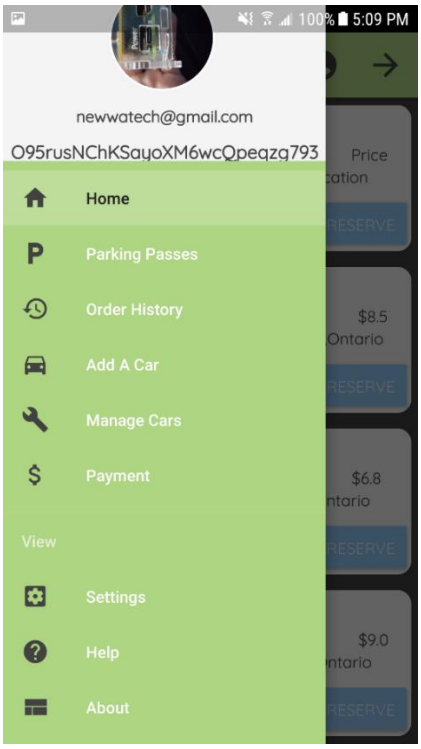


Figure 17 - Main Menu Screen (Sidebar)

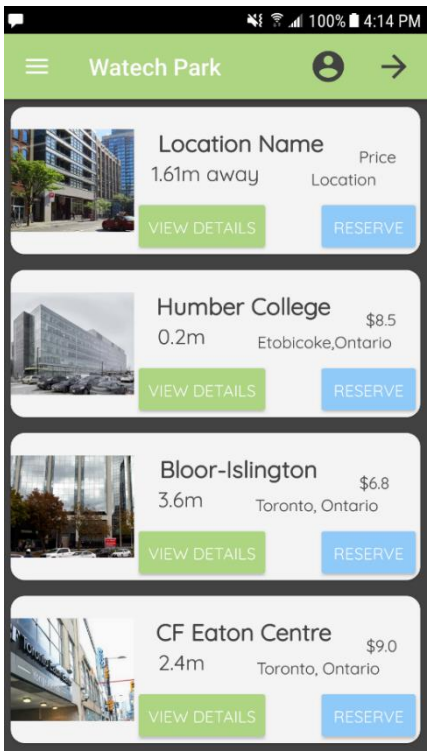


Figure 18 - Main Menu Screen

In the “Add A Car” screen, the user enters in their vehicle details to register a car to the account. The user is prompted to enter information specific to each account, such as the Make, Model, Color, and License Plate #. We have decided to store each added car into the system and send the data according to a specific user to the database. Once a license plate image is taken on-site of a vehicle, the application will assess this image and try to see if it can match it with the added license plate entered by the user

through the mobile application. If the data matches, entry to the lot is allowed and the application would display the gate opening from a visual representation of the lot. This feature is not yet implemented, and will be developed over the current month and finalized by mid-March. There is a button to ADD A CAR, which registers the car to the Firebase database. Once the car is added into the system, and the data is sent to Firebase the consumer can access these details and the registered vehicles in the “Manage Cars” screen.

Manage Cars is where the data is fetched from Firebase database and the information for each “Car” is displayed following a similar format of the main menu. There are 2 options to choose from here: Edit/Delete. Edit allows the user to basically make a change and update the information to the database structure in Firebase. Once the user selects Edit, an inflated view pops up of the fragment prompting to enter in the new information. The user then would tap on the “Apply Changes” button to apply the changes automatically. The changes are visible in real - time through Firebase, once they are set. The Delete option asks the user if they are sure they want to delete the car. Once the user approves, the car is deleted from the real-time database and is removed automatically after the next time you access the Manage Cars section. If the latter is chosen, the action is dropped and cancelled to continue the session.

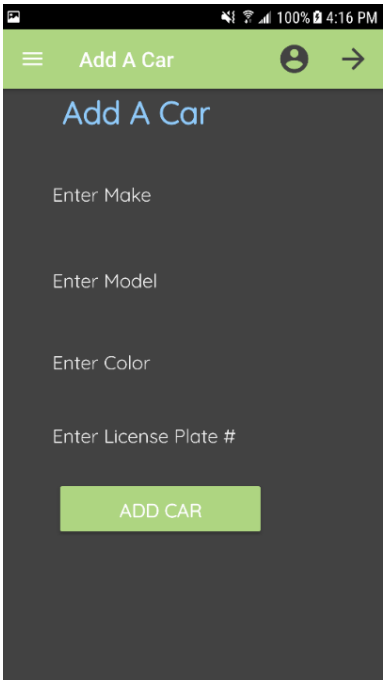


Figure 19 - Add A Car Screen

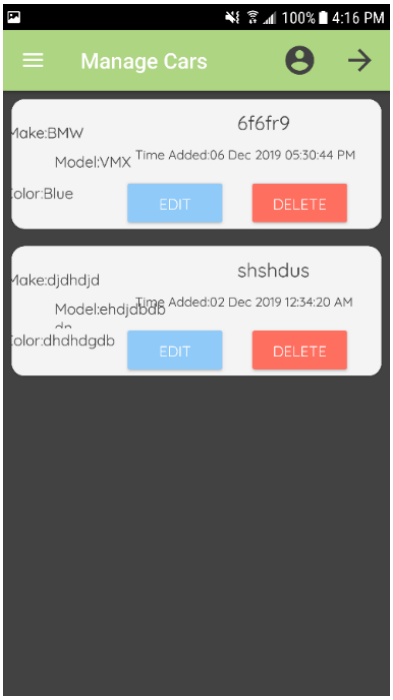


Figure 20 - Manage Your Car Screen

The Parking Passes screen in the application, holds all of the available parking passes for each lot. This includes, the name, location, duration (in hours), validity (time the pass is valid for), type, expiry time, cost, and the account balance before the purchase. There is a button to “SELECT” a parking lot. Once selected the data for that lot is sent to Firebase and stored under the UID of the user. This data is also then sent to the Payment screen, which would be the next step to finalize the reservation through the transaction process.

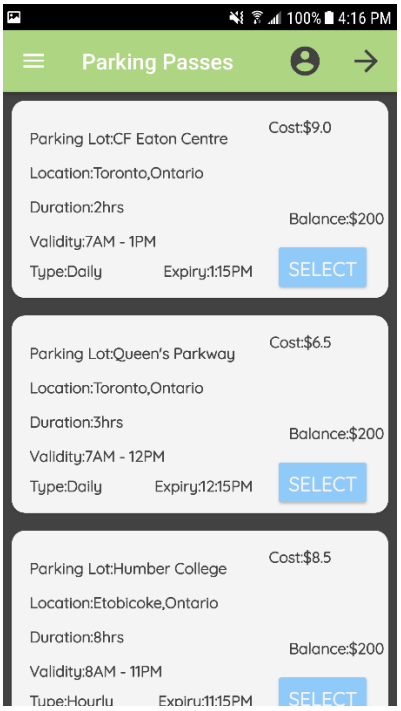


Figure 21 - Parking Passes Screen

On the Payment screen, the selected parking pass is now visible with all the data related to the particular parking lot. On this screen, similar details are displayed, with the addition of an OID (order ID), and e-mail corresponding to the account that is processing the order. Also, the total is calculated for the parking pass with tax and displayed in only a readable form. The balance after the purchase is calculated on the spot and displayed according to the total accumulation. The total would be calculated and based on this set value, the “Generate QR Code” button generates a random QR Code using this value. The user would then tap on the FAB (floating action button) which asks to confirm the purchase. If the order is confirmed then it is successfully been processed. A toast message appears saying “Order has been successfully placed! Please View Order History for more details”.

Order History displays the order's that have been placed using a unique OID (order id) which is the UID used to refer to a specific account. The data is retrieved from Firebase displaying the processed information and a timestamp for when the order confirmation took place.

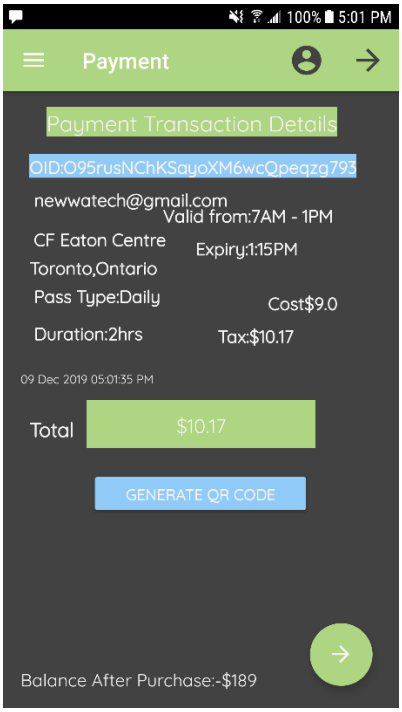


Figure 22 - Payment Screen

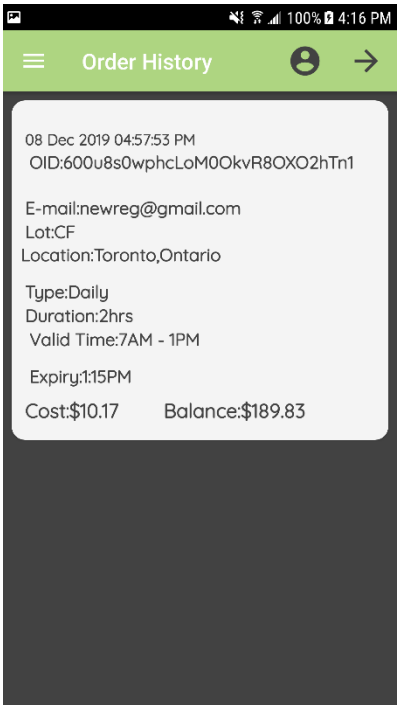


Figure 23 - Order History Screen

The Setting screen provides a localization feature for (English/French integration). The design basically followed a simple UI, with a button to "Select Language. The user checks which one to perform and the languages change state accordingly without the need of re-entering the app. There may be further features added into the final version of the application, towards the final stages of the project which have not been discussed as of yet. The Help screen, displays general help documentation for ways to navigate to

the different screens and use the functionality. The About screen displays project and general mobile application details.

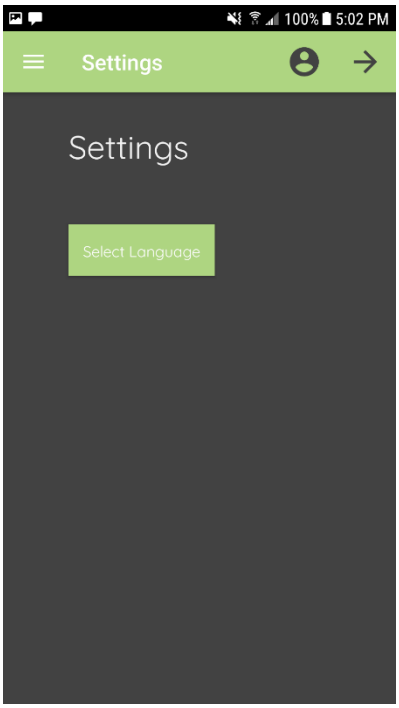


Figure 24 - Setting Screen

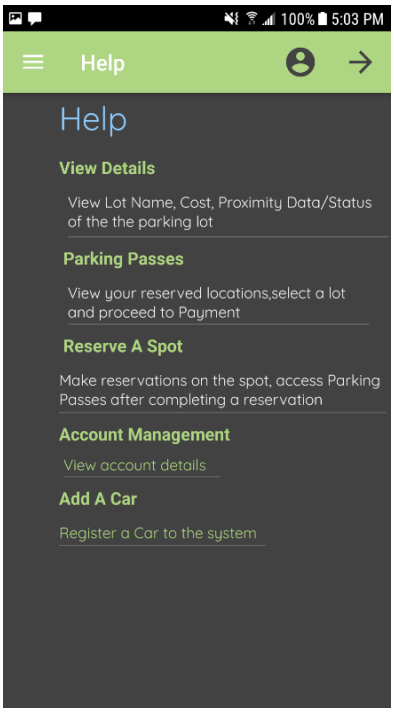


Figure 25 - Help Screen

Figure 26 - About Screen

Action Control Activity

Hardware used for the project must be controlled in some way. This section will explain this process more clearly and how the hardware sensors/effectors, will be used with the mobile application. Each sensor will communicate with the mobile application for different purposes. To begin, the VCNL4010 Proximity sensor will be used to manage/detect when a vehicle has taken up a parking space in the lot. The sensor will be used in terms of sending/displaying proximity data from how far/near the vehicle is from the spot. This information will then be pushed to the online database, where from the application the user will be able to retrieve this data where it is displayed on the APP. The following is the method of operation for controlling the action committed, once

a car has been placed on the parking spot platform, which will be designed/developed during the duration of this semester, the mobile application will detect this change and update the status of the lot in real-time. Therefore, the proximity sensor will only be used to control/detect the distance of an object or vehicle far/near from the actual location. Such as, in a scenario where the parking lot is empty and no cars are visible, the application will relay that data and show all spots “Open” on the application, under a visual manner. A green color will be used along with the RGB LED (red, green, blue) form to accomplish this on the app. In the event, where a vehicle enters the gate and is approaching one of the four spots on-site, the sensors will detect this movement. Then according to this movement, the sensors will know if a car is approaching one of the spots in the location. Our team decided it is best, to only display key data by visually showing the spots in the lot, and only showing which spots are open/occupied or if the lot is full. So, once a car is placed on one of the spots, each proximity sensor on-site will be active, listening and the spot that is taken will update the application to indicate, the spot is currently being occupied(using a red color representation) while all others are in the same state of being open(green). Proximity data/readings will be displayed accordingly based on set conditions, where if a car is approaching and the object is near the sensor the proximity level would be high and therefore the application would indicate an occupied spot. If all spots are taken on-site of the parking lot, then the proximity levels will exceed this amount, therefore the application must display a warning saying “Lot B is Full” and show the remaining spots if any and not allow further entry. This part will be explained further in the Imaging/Firmware branch of this section.

For the IR Break Beam sensor, the main motive behind using this hardware component was to control the gate movements at entry/exit of the parking lot. The sensor will be used to detect when a car is at the gates, and after full validation and error-checking the IR beam sensor will check for an object in the way of the gate. If the car is in the way, the sensor will break the beam and allow the car to enter the parking lot. At exit, the sensor will be used to control the gate as well, to basically allow the car to exit based on distance and movement. Now, once the gates open/close the corresponding data will be sent to the database, and then transferred to the mobile application. This includes, the time of entry, license plate number scanned and the specific UID under which the current user is operating the mobile application. All of this data will be visible for the user, once a specific car assigned to that user has entered/exited the parking lot. On the application, a message will be displayed mentioning “Gate A is opening…” or “Gate A is closing…” once a vehicle is no longer occupying that particular spot, it will return everything to its original state in the data present on the app.

The USB Camera sensor will be used only for the purpose of capturing an image, and license plate recognition. So, once a car comes up to the gates, the primary action to be taken is the camera sensor will automatically capture an image of the license plate, and send this data to the image processing software, and then send it out to the Firebase database. The camera sensor will be placed near the two servo motors and will be the initial action before any decision is made. Depending upon if the license plate image matches with the license plate information stored in the database, entry/exit will be allowed through the two servo motors for that particular user. To allow entry, the license plates must match from on-site of the hardware and the data entered/saved into the

mobile application. The mobile application will not store any sort of images of the license plates in any form, but will only be used to track and match the information with the on-site parking lot through the online database.

The two servo motors being used will only act as the barriers between the entry/exit of a vehicle from the gates. This hardware will only be used to control the actual movement of the gates, during entry/exit. Although, we also are planning to introduce a potential gate override control structure for service vehicles, for admin login purposes. This feature will allow the admin access to the parking lot, and control the hardware servo motors to open the gate in the event of an emergency, or where snow may be blocking the entrance and affecting the parking lot. This feature was considered based on effective feedback provided by the professor for possible use of the servo motors in conjunction with the online database.

Status

/1 Hardware present?

/1 Memo by student A + How did you make your Mobile Application? (500 words)

/1 Login activity

/1 Data visualization activity

/1 Action control activity

3.2.2 Image/firmware

Status

/1 Hardware present?

/1 Memo by student B + How did you make your Image/firmware? (500 words)

/1 Code can be run via serial or remote desktop

/1 Wireless connectivity

/1 Sensor/effector code on repository

Memo and Initial Setup for Imaging/firmware

In the previous semester, we all had to buy individual Raspberry Pi's that were all different versions and models. The model of the Raspberry Pi that is being used for this project is a Raspberry Pi 4 Model B. The Raspbian 10 OS had to be installed for the Raspberry Pi 4 used for this project. Raspbian 10 had to be installed on a 16GB MicroSD card that the Raspberry Pi 4 uses to recognize the OS. The recommended size for this project is 16 to 32GB of external memory storage to store all data and sensor functionality with the best performance possible. The file is located online on the Raspberry Pi website on the Downloads page for Raspbian. Raspbian OS can be also installed with NOOBS (New Out of Box Software) which is an easy operating system installer for the Raspbian environment.

Currently, we have developed and integrated code for each sensor to perform specific functions important to our parking application. Along, with having full functionality of the sensors, we also have connected one of the sensors used in the project to the online database, to send/retrieve data from the Pi to the Firebase database.

During this final semester, our focus will be on modifying the current code for each sensor and create a single applicable program to hold all three sensors and their functionalities through the I2C interface. By combining all three sensors into one large program, it will be easier to access the data simultaneously without having the need for separate connections to each sensor, as is what we currently have at this stage of the project. This phase will be completed in early March, we will multi-process the work to design the final PCB as a single working unit, as well as build the extensive code which will be used in the end to run as one applicable program holding each sensor and its functions.

This section was written by George Alexandris (Student B) who is the lead of the Hardware side of the project.

There were two options mentioned, one is for installing the Raspbian OS image directly onto the Micro SD to insert in the Pi. The second option is to install NOOBS the easy operation system installer that allows users to select what OS version of Raspbian, and what software is desired to be installed onto the Raspberry Pi. The Raspberry Pi image chosen was the direct install for Raspbian OS. First off, you will have to format the SD card with a FAT32 partition. The next step is to extract the files from the zip file downloaded onto the SD card. The SD card is then inserted into the Raspberry Pi. The Raspberry Pi should be setup with the Micro SD card with the image, power adapter for the Raspberry Pi, mouse, keyboard and Micro-HDMI to HDMI cable connected to a TV/monitor. Once you have all these connected to the Pi you can boot the Pi. On boot-up you will see a setup wizard and the Raspberry Pi desktop screen.

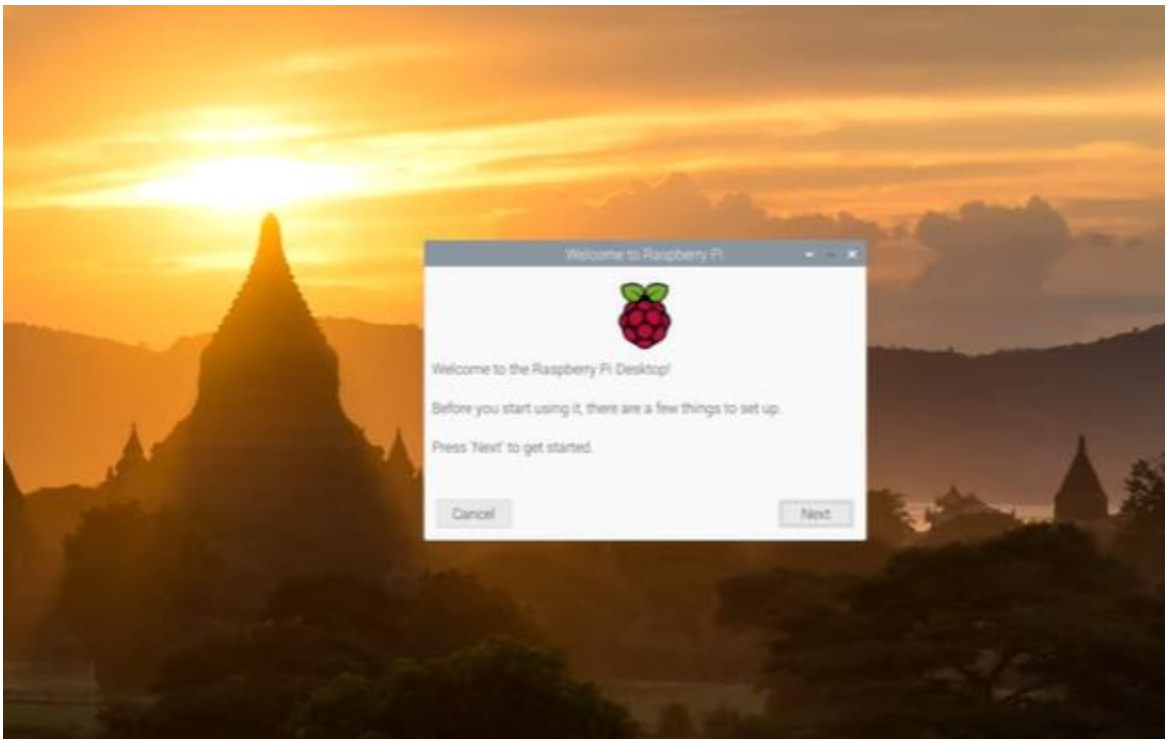


Figure 27 - Raspberry Pi Bootup Screen

After boot-up, the Pi will show a setup screen to select the language, your location, the keyboard version that is being used. The screen will be shown below. You will have these 3 options shown and when done selecting the options you can click next.

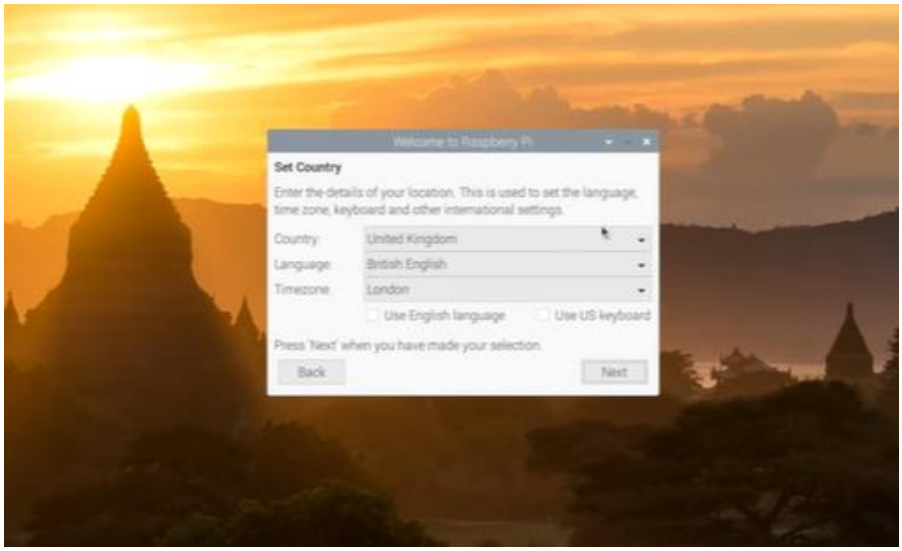


Figure 28 - Raspberry Pi Location/Language Setup

After selecting your language and location the user will have to setup the password for the Raspberry Pi. The image is shown below for the password setup. If the user leaves the new password blank the default password is “raspberry” and user is “pi”. The password is then setup and on login the password is required. The pi user is already setup to have all the permissions available. You can also add another user to the Pi after setup. The pi user is a part of a group called “*sudo”*, which this group is basically the super-users of the system. They have access to all commands, files and directories. Raspbian is a version of Linux and uses the Linux kernel for command specific activities. For permissions to files and directories there are three designated primary permissions. The three permissions are read, write and execute and are assigned to each file for each user, group and other user to access the file. Each user, group and other have permissions on how they can access certain files.

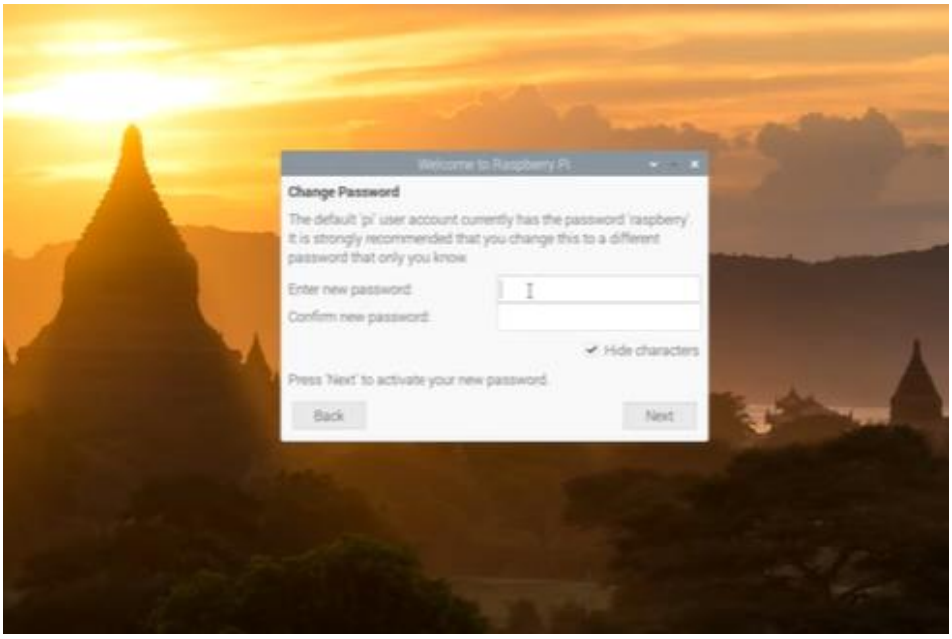


Figure 29 - Raspberry Pi Password Setup

Remote Desktop Connection/Wireless Connectivity

The next step is to setup the network and select the connection through Wi-Fi. Alternatively, the user can connect an Ethernet cable to get an internet connection. After the network is either setup wirelessly or wired you will be able to access the computer fully. The Raspbian image is already setup with everything and you can access everything the OS and Pi have to offer.

In order to access the Raspberry Pi wirelessly, you will want to setup VNC server onto the Pi. To setup the Pi VNC server you will have to launch the terminal and type in *"sudo raspi-config"*. A terminal that shows different options to configure the Raspberry Pi will pop up. Control the menu with the arrow keys and select options with the Enter key. Select the option for "Interfacing Options" then another menu will pop up and give you several options to enable/disable the Camera, SSH, VNC, SPI, I2C, Serial, 1-Wire,

and Remote GPIO. For this project we needed to enable VNC to use the Pi wirelessly, enable I2C for our sensors to be detected by the Raspberry Pi, and lastly, we activated Remote GPIO to use the General-Purpose Input/output pins for us to connect and to interact with our three sensors for the project. Once each one is selected you will have to reboot your Pi for the options to be enabled.

Figure 30 - Command to go to Configuration Menu

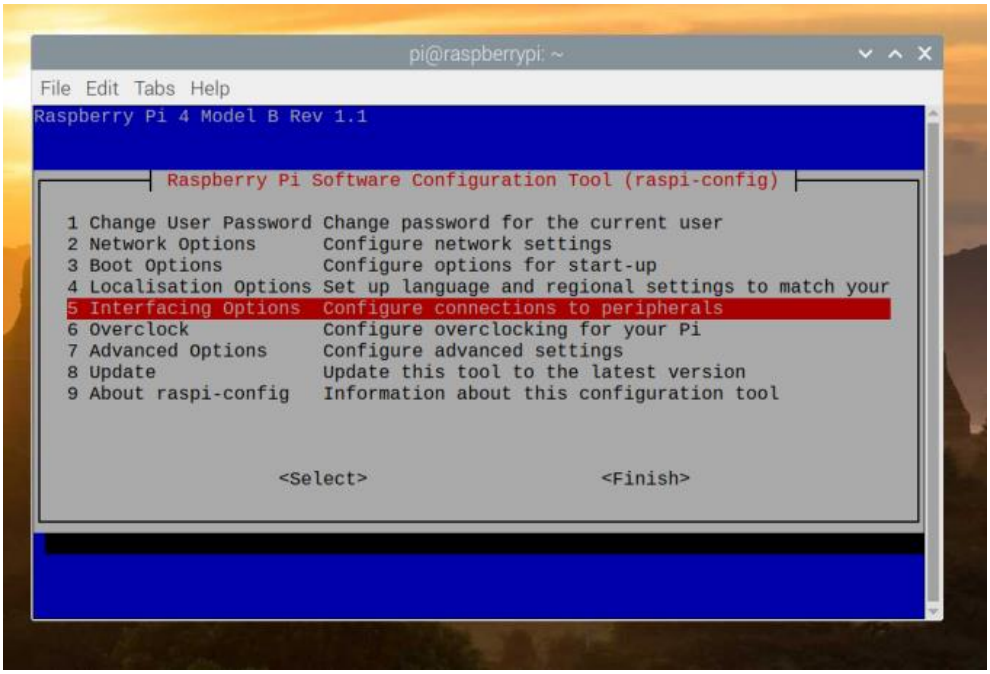


Figure 31 - Options after raspi-config command entered to interact with Pi

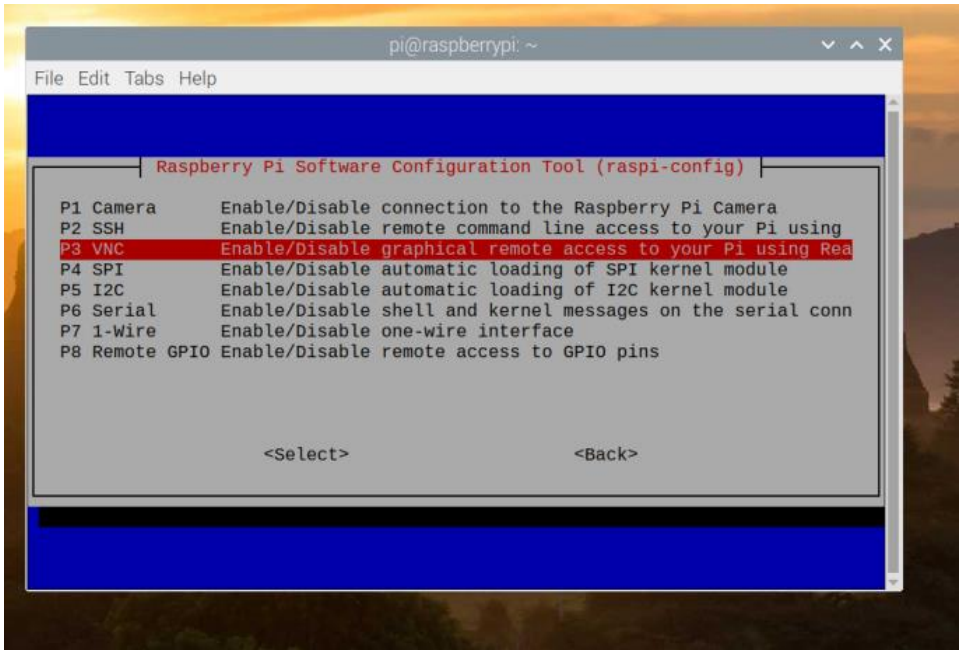


Figure 32 - Enable/Disable Pi Interfacing Menu

All options to work with the Raspberry Pi and sensors are enabled on the Raspberry Pi. Once the Pi setup is complete, you must download VNC Viewer and create a VNC account online through the VNC website. We used a Windows 10 64-bit version so we had to select that download version of VNC Viewer. Subsequently, once the executable is downloaded, install the VNC Viewer program by running the executable. After the setup of VNC Viewer, on your Windows computer you will have to login to the VNC Viewer and then enter the IP address of the VNC server from the Raspberry Pi.

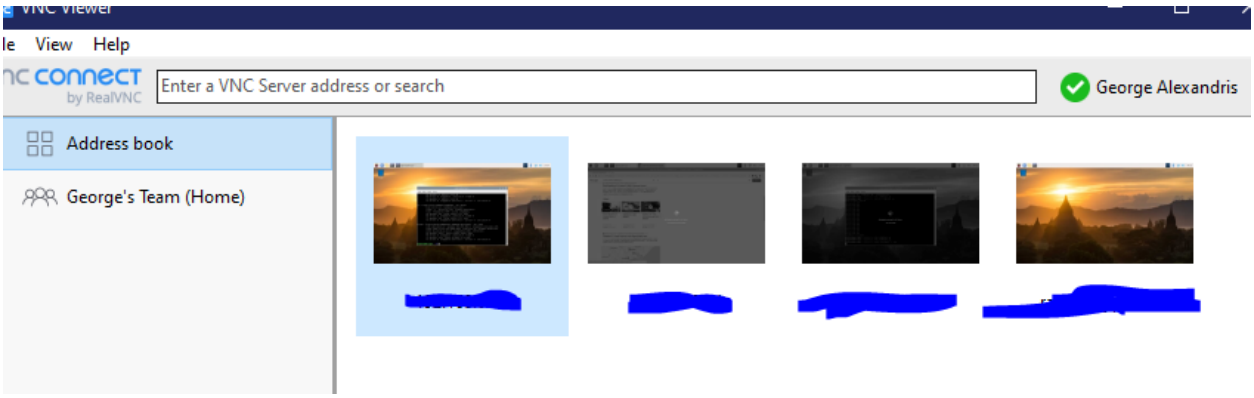


Figure 33 - VNC Viewer Main Screen

To find the IP address of the Raspberry Pi you must enter “ifconfig” into the terminal. After the command is entered you will see an IP address near “wlan0” it will have the IP address beside “inet” enter the IP address in the “Enter a VNC server address” field in the VNC Viewer on the Windows computer. Moreover, the Raspberry Pi will be found and will be able to connect to the Raspberry Pi. When you get a connection, a window will pop up to enter the username and password. Enter in the default “pi” username and the password you created after boot-up. Finally, you are connected to the Raspberry Pi and you can use your personal computer to control the Raspberry Pi wirelessly.

For the purpose of the code developed for each sensor used in the project, we made use of both the remote desktop connection to connect through Wi-Fi settings, and as mentioned above with an alternate route through an Ethernet connection. The Ethernet connection can be established through use of the VNC Viewer software as described above, by connecting the ethernet cable to the Pi’s LAN (Ethernet port) and to a respective electronics system through the USB port such as a laptop, or PC. In that

case, the micro HDM to HDMI cable would not be necessary as the Ethernet connection is all that's needed to access the Pi and the sensors from any location/setting.

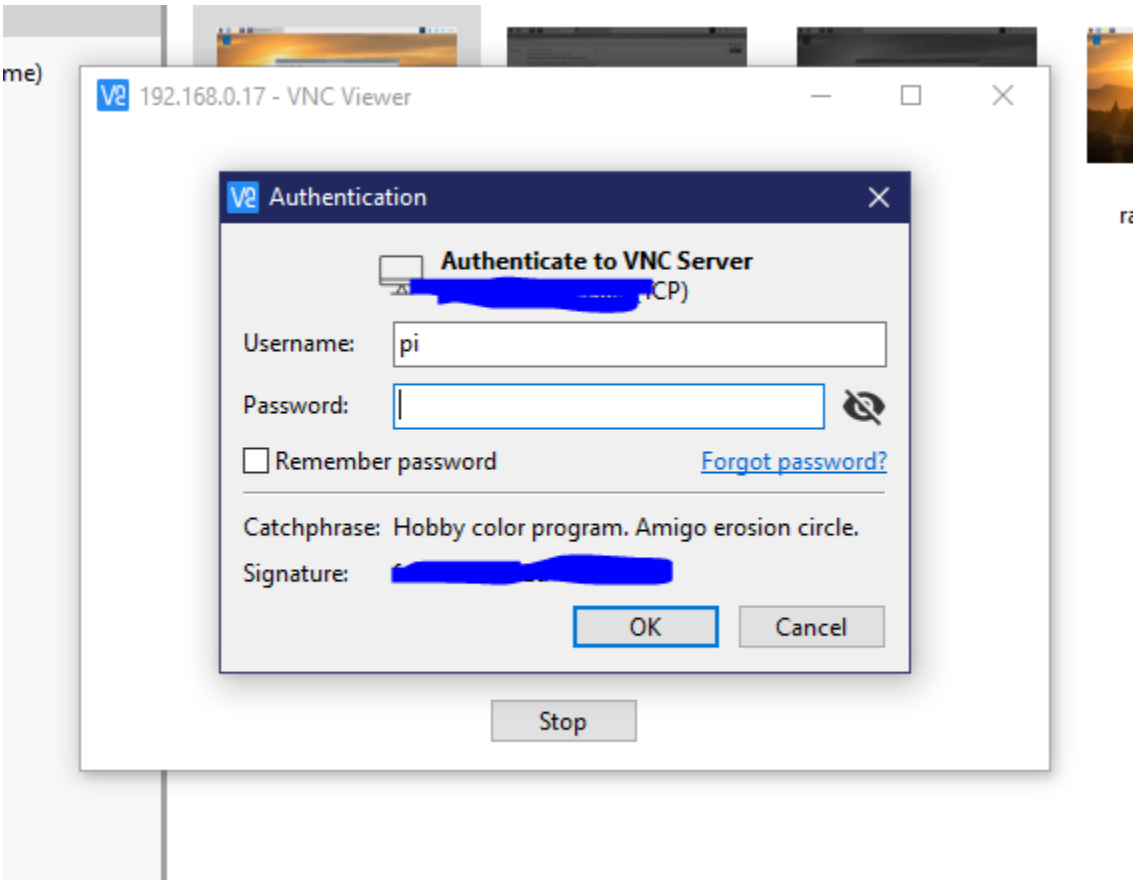


Figure 34 - Login Screen from Pi to Windows

Following the login, you will see the Raspberry Pi desktop and are done setting up your Raspberry Pi, and can control it from your Windows computer. At this stage, please remember the Raspberry Pi IP address and login information you made for your Pi to connect to it wirelessly. You can also remove the HDMI, mouse and keyboard you connected to the Pi and you can use your Windows computer peripherals to control it.

The Raspberry Pi was setup for the project like this in this part of the report. The Pi allows us to now connect our sensors to the 40-pin port which provides 3.3/5V power,

ground and GPIO pins to control the sensors through coding. We decided to use Python programming to control the sensors from the 40-pin port.

I2C Sensor Setup

The sensors we have utilized for this project includes the following: VCNL4010 Proximity Sensor, IR Break Beam Sensor, and the YoLuke's Camera Sensor, which all function as expected through the Raspberry Pi platform. The VCNL4010 Proximity sensor is a I2C device, so you will have to go through the setup for the I2C interface for the sensor. As mentioned in the previous section, there is an option in the Raspberry Pi Configuration Tool menu for enabling I2C modules. To do that once again, you will have to type in the command "sudo raspi-config". Then, select "Interfacing Options" again. The Interfacing Options will be shown for enabling VNC, SSH, Camera, SPI, I2C, Serial, 1-Wire, and Remote GPIO. You will want to enable I2C, confirm to enable the I2C modules. The effect of the I2C modules will start when you reboot your Raspberry Pi, do that by typing in the "sudo reboot" command.

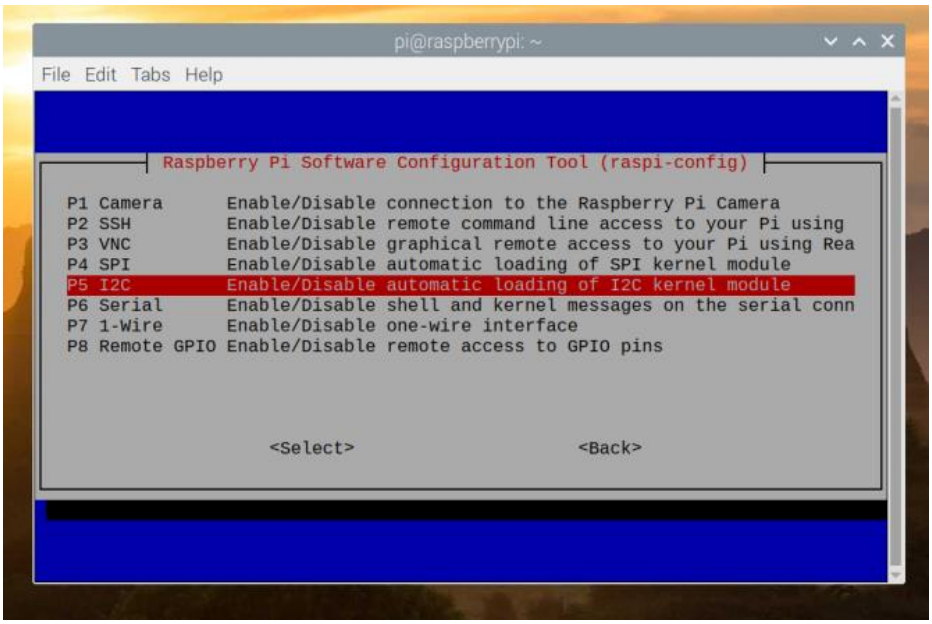


Figure 35 - Enabling I2C Module

Once the Raspberry Pi is rebooted it will have enabled the I2C kernel module and you can interact with any I2C device. After reboot, the Raspberry Pi can detect signals from your I2C module/sensor. You connect your I2C sensor and are then able to read the digital signal. First of all, to get the code to interact with your sensor you will have to detect the address of the I2C sensor. To detect I2C address, the following command “sudo i2cdetect -y 1” can be executed, which in return will output a list of all detected I2C devices as shown in “Figure 36”.

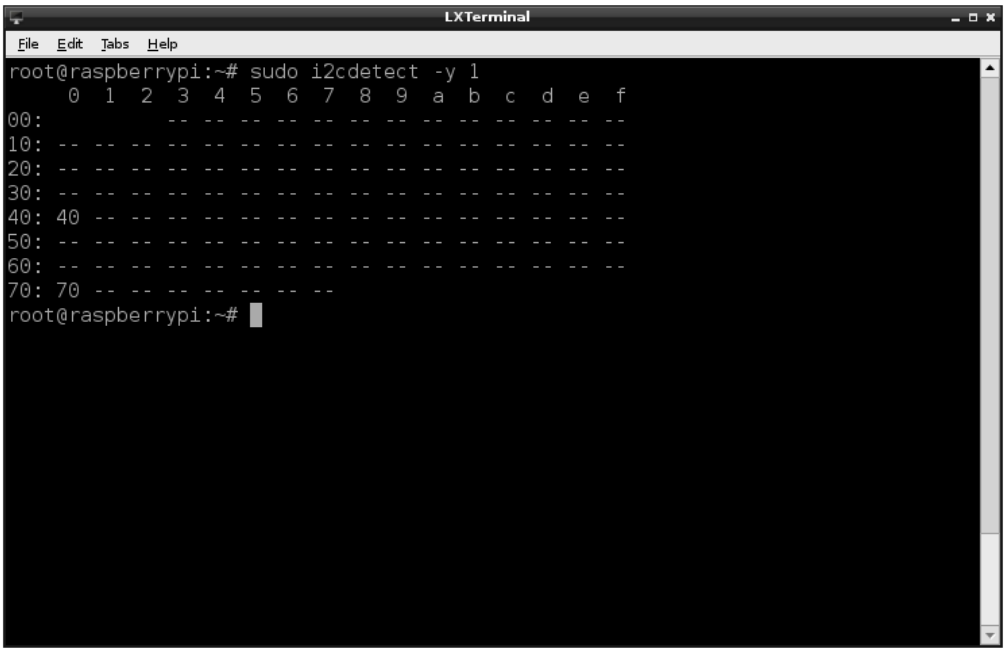


Figure 36 - I2C Command terminal output

In this case, the two addresses that are going to be in use are 0x40 and 0x70 which will be specified in the code. You will be able to get readings from these addresses by stating them in the code.

VCNL4010 Proximity Sensor Setup/Code

The VCNL4010 Proximity sensor allows us to detect if an object approaches the sensor, and relays back proximity data in terms of total distance far/near from an object. In our application, it will be used for the parking spots to detect if a car is about to park in a spot, or if a car is already parked in a parking space. The sensor is connected to our Firebase database, which can send and receive values to and from the database. The mobile application also interacts with the proximity data from the database by receiving the sent values from the Raspberry Pi.

The GPIO pins 17, 18 and 27 are used to output values for the sensors in three states, green for an open spot, blue to detect when a car is approaching the spot and red to indicate when a spot is taken. Further details on how the circuit design and implementation will be touched on in the Breadboard/Independent PCB's section of this report.

The code below shows the strategy that was used to test the hardware + software connection including sending the proximity data to the database and then reading from the data structure *"ProximityData"* and displaying the data on the mobile application.

```
import random
import pyrebase
import time
import math

import smbus
from time import sleep
from gpiozero import LED
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
# Get I2C bus
bus = smbus.SMBus(1)

red = LED(17)
green = LED(18)
blue = LED(27)

def turnOff():
    red.off()
    green.off()
    blue.off()


    global db
    config = {
    "apiKey": "AIzaSyBHz-ZrX8ANSYz3qcVdbjQ_KvpX8Kz3PnU",
    "authDomain": "watechpark.firebaseapp.com",
    "databaseURL": "https://watechpark.firebaseio.com",
    "storageBucket": "watechpark.appspot.com"
    }
    firebase = pyrebase.initialize_app(config)
    db = firebase.database()

try:
    while True:
```

Figure 37 - Initial setup for VCNL4010 Python Code

The first step required to initialize the firebase configuration, with all the project details. In the code it shows setup for the I2C sensor. The smbus library controls the VCNL4010 sensor. Then, the setup for the GPIO (General Purpose Input/Output) interface is done

to control the LED and output and input values for the sensor. This was done through initializing a variable: red, green, blue to store the value of the LED, making use of the three GPIO pins of the Raspberry Pi platform. The “import RPI.GPIO as GPIO” statement was needed to access the GPIO pins on the Pi, and make use of these functions for the LED to process three states, and three separate pins of the Pi as stated above. The time function call was used to allow a delay to occur when necessary in the program, after operation of the sensor is complete of during different changes in states of the LED. A function was also created at this step, to take each value of the LED and turn off the LED after a specific period of time in the code. The function would basically be called later on in the program, to serve its purpose of turning off the LED’s.

```
# VCNL4010 address, 0x13(19)
# Select command register, 0x80(128)
#               0xFF(255) Enable ALS and proximity measurement, LP c
        bus.write_byte_data(0x13, 0x80, 0xFF)
# VCNL4010 address, 0x13(19)
# Select proximity rate register, 0x82(130)
#               0x00(00) 1.95 proximity measeurements/sec
        bus.write_byte_data(0x13, 0x82, 0x00)
# VCNL4010 address, 0x13(19)
# Select ambient light register, 0x84(132)
#               0x9D(157) Continuos conversion mode, ALS rate 2 samp
#bus.write_byte_data(0x13, 0x84, 0x9D)

        time.sleep(0.8)

# VCNL4010 address, 0x13(19)
# Read data back from 0x85(133), 4 bytes
# luminance MSB, luminance LSB, Proximity MSB, Proximity LSB
        data = bus.read_i2c_block_data(0x13, 0x85, 4)

# Convert the data
#luminance = data[0] * 256 + data[1]
        proximity = data[2] * 256 + data[3]


# Output data to screen

#print "Ambient Light Luminance : %d lux" %luminance
        if(proximity<=2500):
            green.on()

            print("\nParking space is available")
            print("Proximity of the Device : %d" %proximity)
            sleep(2)
            turnOff()
        elif(proximity>5000):
            red.blink()
```

Figure 38 - Reading values from VCNL4010 Proximity Sensor

The proximity value from the sensor is taken and used in a while True loop that would print the data while the condition is true. The way this was setup, was based on the different proximity levels of the parking lot at a specific time. The coding aspect was arranged to present different scenarios based on how far/near the vehicle is from the sensor, and basically worked with a range of proximity values. So, if the proximity is less than or equal to a range of 2500, then the green light of the LED will turn on and print a message to say “Parking space is available”, then print the current proximity value, and turn off the led after 2 seconds. At the top as explained before the I2C address was taken from the command “sudo i2cdetect -y 1”. In this case the addresses are 0x13, 0x80, 0x82 and 0x85 to read and write values to and from the sensor.

```
        elif(proximity>5000):
            red.blink()

            print("\nParking space is full")
            print("Gate is closing!")
            print("Proximity of the Device : %d" %proximity)
            sleep(2)
            turnOff()
            #red.off()
        elif(proximity>=3000) or (proximity<=4500):
            blue.on()

            print("\nCar is approaching the parking space")
            print("Gate is opening...")
            print("Proximity of the Device : %d" %proximity)
            sleep(2)
            turnOff()
            #blue.off()

            sleep(2)
            turnOff()

            seconds = time.time()
            a = {"proximity": proximity,
            "timestamp": str(int(math.ceil(seconds)))}

            db.child("ProximityData").push(a)
            print("Proximity data successfully updated to Firebase!\n")
        else:
            print("Failed to push proximity data to Firebase!\n")

finally:

    GPIO.cleanup()


    #GPIO.output(18, 1)
```

Figure 39 - If statements for output of LED color

Then, using an if/else structure to check if the proximity value is greater than 5000 then print out the lot is full, and corresponding data. If it is in the range of 3000 or greater than 4500, display a message saying the “Car is approaching the parking space”. Towards the end of the program, after pushing the data to Firebase the *“GPIO.cleanup()”* function call was used to reset the GPIO pins and basically turn off the LED while not in use.

In the mobile application, how this would work is the data that is sent including the status of the lot would be displayed on the main menu once more details are viewed of the lot. So, while the proximity value changes each time the user chooses a new parking location, the status would also change. On the phone app, it would be indicated by a “Status: Open/Closed/Parked” state. For the purpose of this section, we will only briefly touch on the mobile application integration, as further details have been mentioned previously in the Mobile Application section in this report.

For reference, the code developed for our mobile application for this sensor made use of an original template and was followed and modified to adjust to our parking application, and the overall purpose of the VCNL4010 Proximity sensor in this project. In this case, this template made use of both the proximity/ambient light functions in the sample program, which was not needed for the purpose of our parking application. The template used to support with the coding process can be found here through the following link on GitHub (separate from the project repository) for reference:

https://github.com/ControlEverythingCommunity/VCNL4010/blob/master/Python/VCNL4010.py

The code used and developed of the project for the VCNL4010 Proximity sensor can be found here in the project repository address through the “Sensor Specific Code” folder, on GitHub and is denoted by: **vcnl4010_simpletest.py**

https://github.com/VikasCENG/WatechPark/blob/master/Sensor%20Specific%20Code/vcnl4010_simpletest.py

IR Break Beam Sensor Setup/Code

The IR Break Beam sensor in this application, was utilized to detect a vehicle approaching the gate though either entry/exit terminals, through controlling the gate operations of the parking lot. The main focus of the sensor, being to provide status of the lot at different times of a vehicle movement, including the closer a object is to the gate, the more likely the sensor is able to detect the change. In this case, this sensor would be a simple way to basically detect motion at entrance of the lot. The way this works, is through an emitter side, which sends out an IR beam of light, not physically visible. This is done, through use of a receiver and a transmitter. So, when an object/ or vehicle passes by the gate, and is detected by both the transmitter and receiver, and is not reflected back as IR (infrared) light, then the beam is broken and the gate allows entry into the parking lot. More details of how the sensor works with the circuit design itself will be explained in the Breadboard/Independent PCB’s section in this report.

The coding for this sensor was accomplished through the Python programming language. Initially, the design of the code was based on what was needed from our parking application. Therefore, in this case the sensor needed to detect motion of the gate and the presence of a vehicle through emitting an IR beam light, and checking for transparency with a physical object in the way.

The code for the IR Break Beam sensor is simple all you have to do is set the GPIO pin to input and read the output from the output wire from the receiver. The code is shown on the next page.

```
You're using code navigation to jump to definitions or references.
1  #########
2  #File: irBBtest.py
3  #Date: Sunday Nov 03, 2019   22:32 PM
4  #Author: George Alexandris
5  #Purpose: Going to implement uploading data to the firebase database
6  ########
7
8  import time
9  import RPi.GPIO as GPIO
10
11 Gpio_Pin=10
12
13 #setup GPIO pin as input to read value for HIGH or LOW
14 GPIO.setmode(GPIO.BCM)
15 GPIO.setup(Gpio_Pin,GPIO.IN)
16
17 try:
18         while True:
19                 x = GPIO.input(Gpio_Pin)
20                 if(x==1):
21                         print("Solid,LED OFF")
22                 if(x==0):
23                         print("Beam Broken,LED ON")
24
25                 time.sleep(2.0)
```

Figure 40 - IR Break Beam Python Code

The code reads from the GPIO pin that the receiver output wire is connected to and is set to an input. To set the GPIO pin to input you put in the code *"GPIO.setup(gpio_pin, GPIO.IN)",* then if you want to read the value of the input you enter *"GPIO.input(gpio_pin)"*. Then to output if the beam is cut I setup two if statements that read if the value is HIGH the beam is received if not the beam is broken.

Initially, on the tests that failed the Python code would send results showing that the pin was in a LOW state and then would change to a HIGH state and stay in the HIGH state forever.

Due to the code being set to the wrong pin, the Raspberry Pi was not reading the values from the correct pin. Once the code was set to the right GPIO pin the code was getting the correct values as shown in the screenshot below.

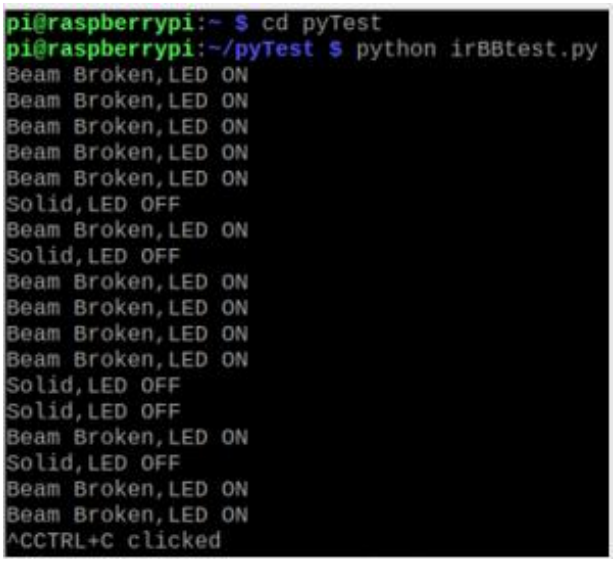


Figure 41 - Image of successful IR Break Beam test

The plan for this semester, is to modify the current code for the IR Break Beam sensor to not only detect a vehicle in the way of the gate, but to also update this data to the online database. We had initially decided to not have the gate entry/exit statuses being uploaded to the Firebase database, but considering our application and what we are seeking to develop, we also believe our users would desire to be informed on the status of the gate through different scenarios in the parking lot. Due to this, the current code for this sensor will be modified to send the data, for when the LED is off or when the

beam is broken(object passes by the sensor) the status of the lot would be denoted by a 1 or 0 (false) or (true) Boolean condition through the Python coding platform. Following these conditions, the status of the lot would be updated each time an entry/exit is made in the lot on Firebase indicating a corresponding message, and that data would then be presented to the user through the mobile application. This part for the sensor and the coding has not been developed yet and will also be the focus in the upcoming weeks along with modifying the current code structures for each sensor.

The code used for the IR Break Beam sensor for this project can be found in the following link to the project repository on GitHub and is denoted by: **irBBtest.py**

https://github.com/VikasCENG/WatechPark/blob/master/Sensor%20Specific%20Code/irBBtest.py

YoLuke USB Camera Sensor Setup/Code

The camera is setup through a USB port on the Raspberry Pi. You must download a command to make the camera take pictures it is called *"fswebcam"*. To install the program and command you must go to the terminal and type in this command *"sudo apt install fswebcam"*. Then, you must add your user to the video group using this command *"sudo usermod -a -G video <username>"* in order to avoid permission errors. To take a picture of an image you enter *"fswebcam <imageName>.jpg"*. Th program will create the file in the current working directory you are in. You can view it by finding it in the directory. You can adjust the resolution, skip frames, and perform other functions to take better pictures based on overall quality and enhancement.

```
1  #!/bin/bash
2
3  DATE=$(date +"%Y-%m-%d_%H%M")
4
5  fswebcam -r 1280x720 -S 7 --no-banner /home/pi/Desktop/image-testing/$DATE.jpg
6
7  echo $(python3 mod_recog.py $DATE.jpg)
```

Figure 42 - carsnaptest.sh file

This is a file to take a picture with the *"fswebcam"* command with options for setting the resolution to 1280x720. Skipping 7 frames in the command, setting the banner to not show. Lastly, in the *"fswebcam"* command it places the image file to the desktop pi in a directory called image-testing. The image file name is set to the current date and time. Then the on line 7, the last line, it runs an echo command to run a file called mod_recog.py inserting the name of the file in the second argument. The Python program acts as our license plate recognition software.

The contents of the mod_recog.py file will be shown and explained in this part of the report. The screenshot below displays the Camera sensor setup through the python coding environment:

```
import sys
import cv2
import imutils
import numpy as np
import pytesseract
from PIL import Image

if len(sys.argv) < 2:
    print("Something went wrong, please pass the name of the image as an argument")
    print("Usage: pyuthon3 License_Plate_Recog.py image.jpg")
    exit(1)

img = cv2.imread(sys.argv[1],cv2.IMREAD_COLOR)

img = cv2.resize(img, (620,480) )


gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) #convert to grey scale
gray = cv2.bilateralFilter(gray, 11, 17, 17) #Blur to reduce noise
edged = cv2.Canny(gray, 30, 200) #Perform Edge detection

# find contours in the edged image, keep only the largest
# ones, and initialize our screen contour
cnts = cv2.findContours(edged.copy(), cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)
cnts = sorted(cnts, key = cv2.contourArea, reverse = True)[:10]
screenCnt = None
```

Figure 43 - Camera Python Setup

This code is for identifying a license plate number from an image. The code above just sets the variables to recognize colors, the contours and isolate the license plate.

```
# loop over our contours
for c in cnts:
        # approximate the contour
        peri = cv2.arcLength(c, True)
        approx = cv2.approxPolyDP(c, 0.018 * peri, True)

        # if our approximated contour has four points, then
        # we can assume that we have found our screen
        if len(approx) == 4:
                screenCnt = approx
                break



if screenCnt is None:
        detected = 0
        print ("No contour detected")
else:
        detected = 1

if detected == 1:
        cv2.drawContours(img, [screenCnt], -1, (0, 255, 0), 3)
```

Figure 44 - For loop to set contour edges

The *"for loop"* traces the contour to isolate the license plate number. It will count the points of the contour and set it to variable *"screenCnt"*. If the *"screenCnt"* is not set to a value no contour was not set. Then a Boolean is "detected" and the value is set to 0 if no contour is detected, and if it was detected then the Boolean value will equal 1. It will draw the contour in the image and highlight the area of the license plate. The purpose of the sensor for our parking application, is to through the use of the image processing software be able to target and only pick out the contents of the license plate number, excluding any other unnecessary details.

```
# Masking the part other than the number plate
mask = np.zeros(gray.shape,np.uint8)
new_image = cv2.drawContours(mask,[screenCnt],0,255,-1,)
new_image = cv2.bitwise_and(img,img,mask=mask)

# Now crop
(x, y) = np.where(mask == 255)
(topx, topy) = (np.min(x), np.min(y))
(bottomx, bottomy) = (np.max(x), np.max(y))
Cropped = gray[topx:bottomx+1, topy:bottomy+1]



#Read the number plate
text = pytesseract.image_to_string(Cropped, config='--psm 11')
print(text)

#cv2.imshow('image',img)
#cv2.imshow('Cropped',Cropped)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure 45 - Code to crop and read image

In the code above it will draw the contour edges and highlight the image. Then it will crop at the highlighted contours. After it will read the image plate, and set it to a variable “text” and print the value of variable “text”.

The following is the code used for the project purposes for the YoLuke USB HD Camera sensor, and can be found in the following project repository address through GitHub and is denoted by: **Licence_Plate_Recog.py**

https://github.com/VikasCENG/WatechPark/blob/master/Sensor%20Specific%20Code/Camera/image-testing/License_Plate_Recog.py

Other testing phases of the camera senor, including captured screenshots and testing results can be found in the “Sensor Specific Code” folder on the repository and accessed through the **Camera/image-testing** folder branched within this folder.

PCA9685 Servo-LED Controller Setup/Code

The servo-motor setup is for gate control and are controlled from the PCA9685 chip. The PCA9685 also made use of python code as well in this project to control the servo motors for the gate. The PCA9685 servo driver can control up to 16 servomotors but we will only use 2 for gate control. The PCA9685 driver makes it a lot easier for the Raspberry Pi to communicate and control the servomotors.

To setup control for the servomotors in the Python programming language is simple. The library for the servo-driver can be found in the Arduino IDE library manager. Firstly, to get the library for the servo-driver you need to install the Arduino IDE by typing in the command *“sudo apt install arduino”*. The Arduino IDE allows the Raspberry Pi to easily manage control of Arduino devices.

```
1  from board import SCL, SDA
2  import busio
3  import time
4
5  from adafruit_pca9685 import PCA9685
6
7  from adafruit_motor import servo
8
9  i2c = busio.I2C(SCL,SDA)
10
11 pca = PCA9685(i2c)
12 pca.frequency = 50
13
14 servo4 = servo.Servo(pca.channels[4])
15 servo1 = servo.Servo(pca.channels[0])
```

Figure 46 - Servomotor Python Setup

The code used for the PCA9685 servo-LED controller in the project can be found here through our project repository address on GitHub and is denoted by: **servotest1.py**

Link to servotest1.py

3.2.3 Breadboard/Independent PCBs

Status

/1 Hardware present?

/1 Memo by student C + How did you make your hardware? (500 words)

/1 Sensor/effector 1 functional

/1 Sensor/effector 2 functional

/1 Sensor/effector 3 functional

Memo and Hardware Creation/Design

This section will address the breadboard and independent PCB's work accomplished throughout the duration of the WatechPark project. This includes, the breadboard /PCB designing, testing phases for each sensor and the progress up to date with each sensor and its overall collaboration with the project. We will be going through each sensor and what led to its initial schematic design, breadboard testing results, and then the PCB which was created for each sensor. Also, including the overall time commitment put into this project for each phase, and the budget accumulated from the final project based on the prototypes developed for each sensor/effector. Initially, the main 3 sensors that will be used in the WatechPark project, were isolated and tested individually by each member of the team, using a breadboard at first and then carrying on with a specific PCB design. The PCB's were designed separately for each sensor to fulfill the requirements of the last semester, but we are currently working on combining all the sensors into one PCB that will be used in our final prototype. A majority of the work in terms of the breadboard and individual PCB's (Printed Circuit Board) design process,

development phase was achieved through use of the Fritzing software. The Fritzing software is an open-source application that allows the user to be able to create, design schematics from scratch and develop physical circuits on a breadboard, which is replicated through its virtual environment. It is highly customizable and easy to use in terms of its overall appeal. This software allowed each of our sensors to be initially designed from a schematic point of view, and then be able to be tested on a physical breadboard. This was based on the breadboard view on the Fritzing software, which is automatically created within the software based on the schematic design of the circuit. From there, the PCB design was available where most of the connections were made with use of the materials needed to create the PCB's and follow through with testing each sensor and its functionalities. The PCB work done for each sensor in this project needs to be proceeded with the utmost care and caution, as through our experience we have learned different techniques to avoid any issues that may arise in the development and designing phases. It is advised and important to reassess your design before sending it into the etching and cutting services provided by Humber College, and its prototype lab.

We have also started work on our physical parking lot model which will work with our mobile application and its sensors, to finally display real-time data based on the occurrences of the parking lot, and the state of each sensor on-site. This will be done through the final PCB integration, and testing of each sensor and its functionalities.

All PCB designs used in this project, Figure 51-52,56, were based on datasheets provided by Adafruit (Adafruit Explore & Learn, 2019), we then built the breadboard layouts using Fritzing, Figure 48,55,60, and then finally assembled and tested each

design to ensure successful testing and retrieval of the expected outcomes based on the results of each sensor.

Student C (Elias Sabbagh) will be addressing the breadboard/independent PCB's status of the hardware side in this section of the report.

VCNL4010 Proximity Sensor Functional

The VCNL4010 is a proximity sensor that utilizes the i2C interface, with a flexible power input between 3.3V – 5V (Adafruit Explore & Learn, 2019), which is perfect because our Raspberry Pi uses 3.3V to handle communication on its i2C interface. Thus, eliminating the need for any voltage controlling parts that would've been necessary between the sensor and the Raspberry Pi if we were forced to use any voltage above 3.3V. Another advantage for using the i2C interface, is that all pins used for this sensor can be extended and used with other i2C devices which would make the final joined PCB a lot simpler.

This is the schematic design which was created for the circuit using the Fritzing software. The design below shows the basic pin-outs used for the sensor. There are 6 pin connections and 4 of the 6 will be used (VIN, GND, SCL SDA). The design shows that there is a sensor in the center utilizing a tri color LED on the side. The sensor is connected to GPIO pins 2 and 3 for SDA and SCL of the Raspberry Pi. VIN of the sensor connects to the 5V GPIO pin on the PI. Ground of the sensor connects to GND on the PI. The 2nd lead of the RGB LED (Common Anode pin) connects to 5V GPIO pin on PI. R1(Resistor 1) connects to GPIO pin 17 of PI, R2 to GPIO pin 18 of PI, R3 to GPIO pin 27 of PI. The tri color LED is connected to ground, then GPIO pins 17, 18 and

27 to output values for the sensors in three states, green for an open spot, blue to indicate when a car is approaching one of the spots and red for when the spot is taken.

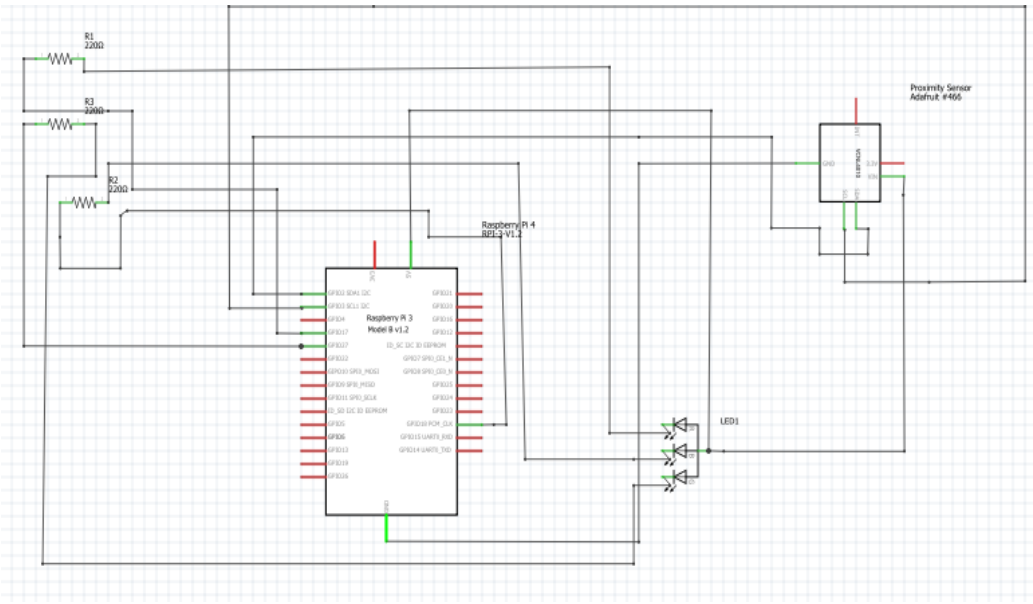


Figure 47 - Fritzing Schematic Design for VCNL4010 Proximity Sensor

The breadboard testing portion required to test the circuitry if it responds to the sensor and if the RPI can detect the sensor and its i2C interface. Using the Fritzing software, here we basically transferred the schematic design to the breadboard. I followed the breadboard design on Fritzing for the circuit, with use of 11 jumper wires. For this sensor, we then decided to use the 5V GPIO pin, designing a voltage divider circuit in the process. This is to ensure there is no possible damage to the circuit, with the amount of current being supplied to the circuit. Based on the breadboard design the sensor used 3.3V out of the 5V which was leftover voltage after the 1st resistor (5V – 1.7V = 3.3V). The voltage was divided between the 3 220-ohm resistors, using roughly

1.7V out of the 5V. The following is the breadboard design created through the Fritzing software.

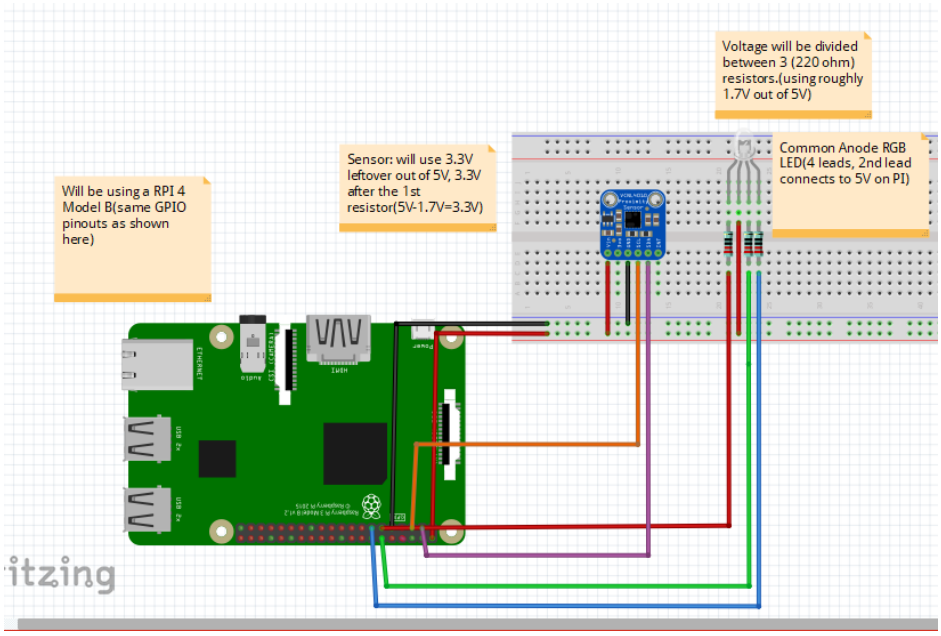


Figure 48 - Fritzing Breadboard Design for VCNL4010 Proximity Sensor

After the breadboard design has been laid out, the next step was to build the actual circuit on a physical breadboard as shown below which show the connections to each part of the sensor and it's accompanying parts.

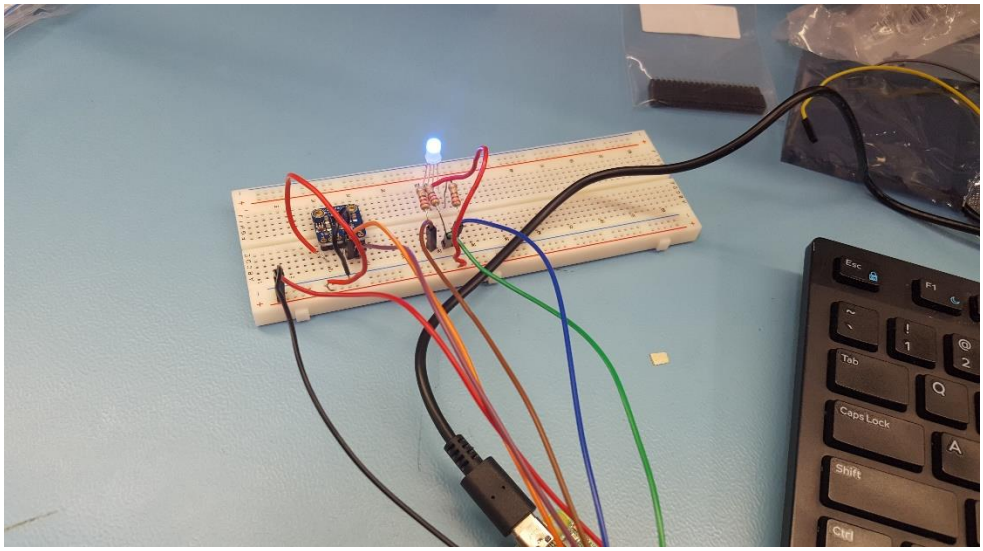


Figure 49 - Breadboard design for VCNL4010 Proximity Sensor

To test the breadboard circuit, we first had to check if the sensor was working as needed with the design. The sensor uses an i2c interface, so we tested using the following command *"sudo i2cdetect -y 1"* to detect if it works and if the i2c connection has been made as shown in the screenshot below:

10

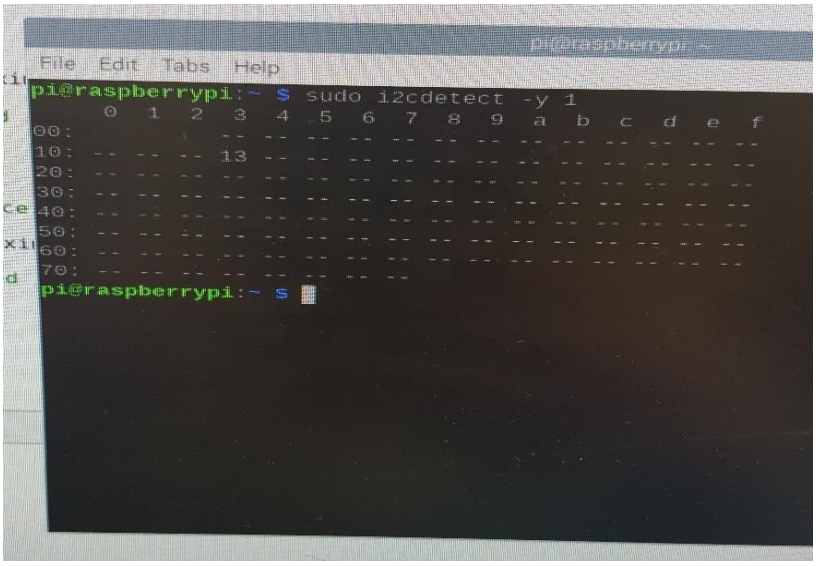


Figure 50 - I2C testing results for VCNL4010 Proximity Sensor

Using this command, you can see the sensor is reading the bytes of data from the i2c interface on address 0x13, showing successful connection of the sensor to the breadboard design.

The next step was to design the PCB board. During this process, we ran into many problems trying to get the right design down, as the board required many changes. The main point being the positioning of the top/bottom layers. After much consideration we were able to successfully complete the PCB design. This is because, all the

connections that go to the GPIO pins had to be in the top layer, and there must be no overlapping connections between the top/bottom layers. So, after redesigning the board, we had to rearrange the top/bottom layers and from there we were able to complete the PCB design as needed. After getting the PCB design finalized, we sent it into the prototype lab to be produced using the laser-cutting services provided, through our tuition.

To solder the PCB board, you would require a lot of focus and attention. We were able to luckily not have to go through that process, while soldering. For this sensor design, there were 4 VIAs that needed to be soldered. You must thread a single thin wire through the holes, solder it and then cut off the remaining excess wires. The same process, was taken for the resistors where we set up the resistors in their place and had to solder the 2 sides. Then, cut off the excess wire. While soldering the LED, we had to be very cautious as the connections were designed really close to each other, so a lot of focus was needed here. Initially, there is a 6-pin header that comes with the sensor that needed to be soldered. Also soldered was, the 40 pin GPIO pins on the PI, to keep the header stable while soldering. The screenshot shown on the right below shows the Fritzing design of the PCB and the end design which was sent into for cutting service to be produced. After soldering, the board should be ready to be tested, and should look like the screenshot shown on the right below.

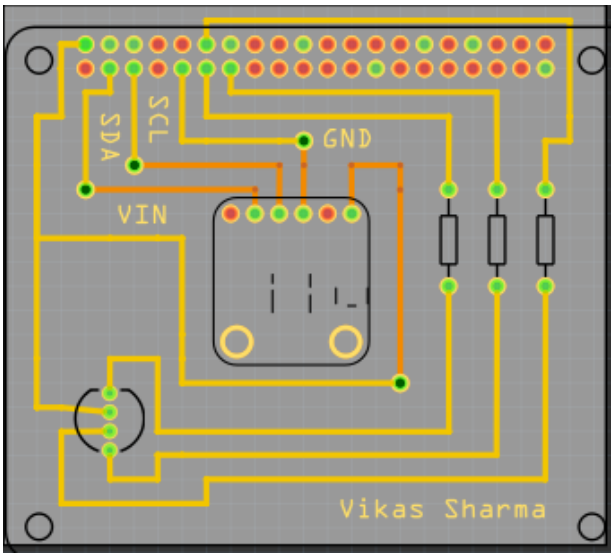


Figure 51 - Fritzing PCB design for VCNL4010

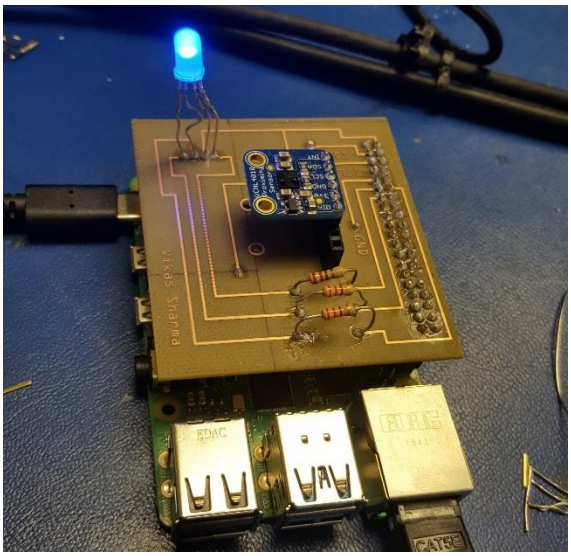


Figure 52 - VCNL4010 Printed Final PCB Design

To test the connections and the functionality, we added an LED to confirm power delivery to the sensor, and tested the functionality by covering the sensor with a piece of paper by reading the values through a testing script on our Raspberry Pi. Next, we tested if the LED functioned as designed for testing purposes. After setting up the circuit, and running the python program created for the sensor and its main application, we were able to get the LED working and tested different values based on the purpose of the sensor in our parking application. We played around with the code by testing different colors to see if we can get all three values of red, green, blue to blink/flash which we were able to do successfully. Using if/else statements to see if the LED can

communicate with the PCB board and the sensor, which we were successfully able to do as shown in the screenshot captured below.

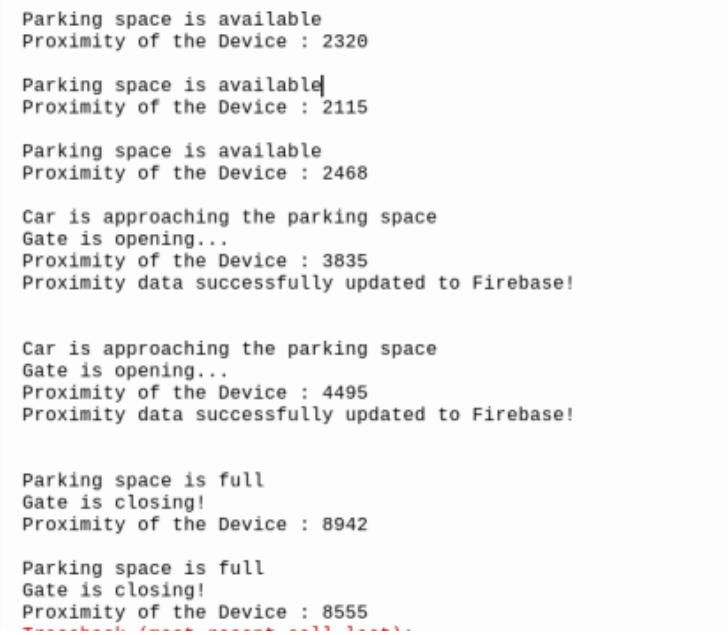


Figure 53 - Breadboard/Testing Results for VCNL4010

IR Break Beam Sensor Functional

This is an analog type of sensor that is split into two parts for it to do its job. This meant that we needed to utilize 3 pins on our Raspberry Pi to use it, 2 pins for power delivery and 1 pin for the analog signal to be read when the IR beam is cut off by an approaching vehicle. Based on the datasheet (Adafruit Explore & Learn, 2019), this meant that there is no need for any additional parts like resistors or transistors to drive this sensor when tested by itself.

The IR Break Beam sensor has two parts to it one the receiver and transmitter. The transmitter part has two wires one for power and the other for ground. Both wires on the transmitter part of the IR break beam sensor can directly connect to power and ground

without resistance. The receiver has 3 wires power and ground as well similar to the transmitter part, but the third wire is a digital output. The power and ground are both directly connected to the power and ground on the Pi. Both parts (receiver and transmitter) of the IR Break Beam sensor each have power and ground, and either can be connected to 3.3V power or 5V power. If the transmitter is connected to the 3.3V it offers shorter range for the Infrared beam to travel but if you connect it to 5V you can send the beam farther. To be safe and to avoid any damage, for the sake of this project both parts of the IR break beam were connected to 3.3V. On the receiver side of the IR Break Beam, the output wire must be connected to a pull-up resistor to read the digital signal. A pull-up resistor of 4.7K Ohm is required, connected to VCC to ensure the known state for a signal. Just like the VCNL4010, we connected an LED and a resistor with the IR break beam sensor to confirm power delivery, and to test functionality we blocked the connection between the sender and the receiver parts of the sensor with an object, and then read the value on the logic output on both the Raspberry Pi and an external multimeter. This LED emits a red light when powered on, it requires a 1K ohm resistor to be lit up to control the voltage and current levels going into it. When the output wire sends the digital signal, it turns on the LED. The LED is used to indicate if the beam is detected or not by the IR sensor. The schematic design for the IR Break Beam will be shown below, produced through the Fritzing software.

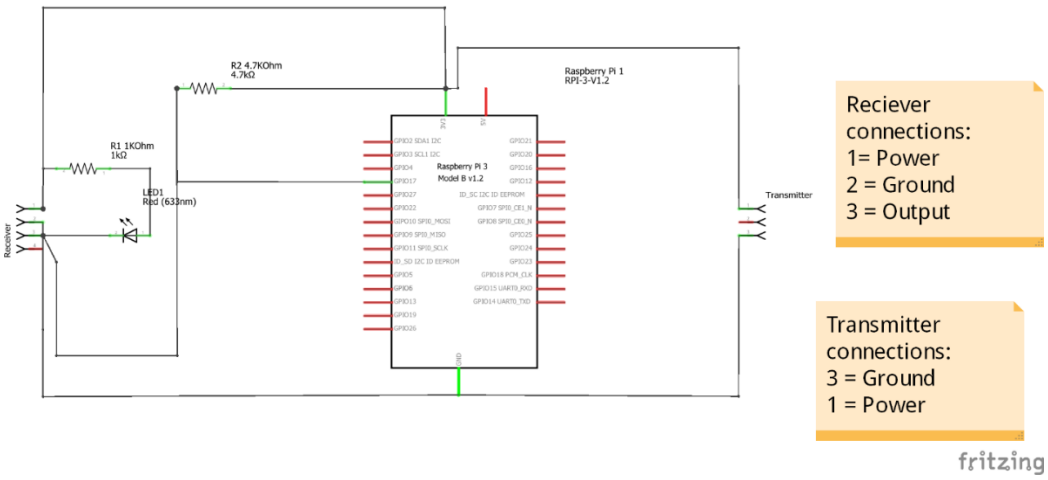


Figure 54 - Fritzing Design for IR Break Beam Sensor

The following screenshot below is the breadboard design, built based on the schematic design using the Fritzing software. It displays all of the physical connections needed for the sensor to function and operate as needed, and to meet the requirements of starting the development of the PCB for the sensor.

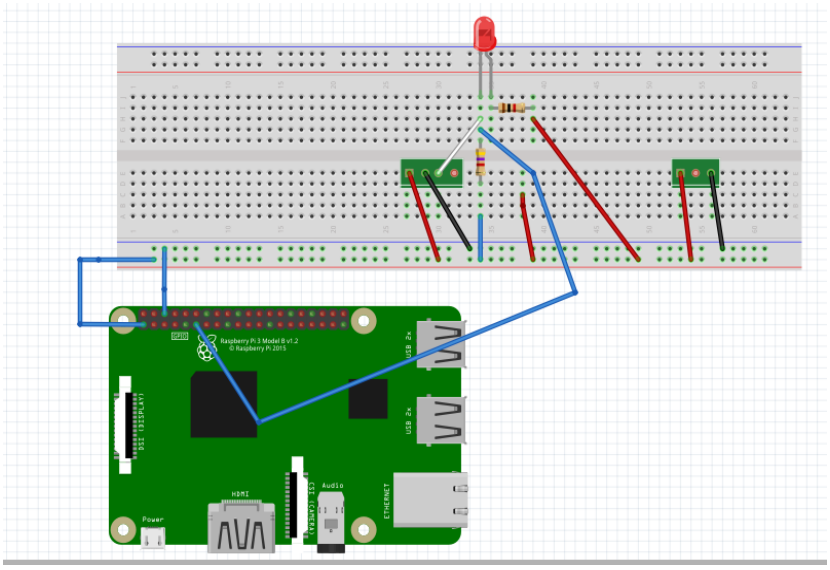


Figure 55 - Fritzing Breadboard Design for IR Break Beam

The PCB designs were all created in a free software called Fritzing as mentioned above. All designs were done in the application from the breadboard design, schematic and then the PCB. The PCB was completed last in the design process. This is to make sure the breadboard tests go well and to see if there can anything being read from the sensor. After the breadboard design and testing is completed, the PCB design needs to be finished. The final PCB design shown here below is made to represent something similar to the breadboard design. Transmitter is connected to both 3.3V power and ground, and the receiver is connected to 3.3V and ground directly, and then its output wire is connected to a 4.7k ohm pull up resistor, LED and then to another pull up resistor that is 1k ohm, also mentioned above. Then the third line from the output wire is connected to the GPIO pin 17.

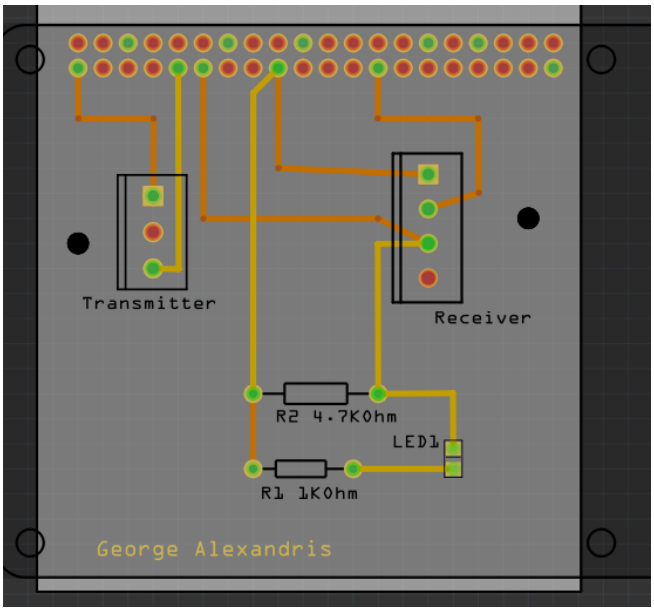


Figure 56 - Fritzing PCB Design for IR Break Beam Sensor

Breadboard testing for the IR Break Beam sensor came from a few different designs, more generally from testing different designs specified from sources. Some were failures and some were generally fault in the code, second guessing the design of the circuit. The testing of the breadboard circuit came from multiple designs just to see if a digital signal can be read from the receiver to the Raspberry Pi's GPIO pins. Some of the designs were faulty due to the code that was tested, or some the pins were not reading the right signal. On the tests that failed, the Python code would send results showing that the pin was in LOW state and then would change to HIGH state and stay in the HIGH state forever. The following screenshot shows the unsuccessful testing of the sensor, which will then be followed up in the next image with the successful testing outcome.

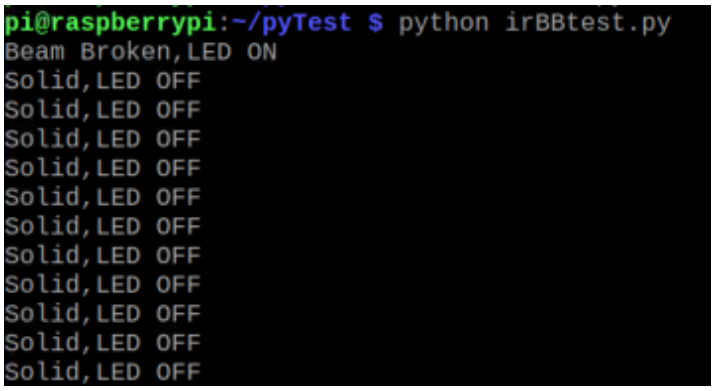


Figure 57 - Image of failed breadboard test of IR Break Beam Sensor

Due to the code being set to the wrong pin, the Raspberry Pi was not reading the values from the correct pin. Once the code was set to the right GPIO pin the code was getting the correct values as shown in the screenshot below, showing successful testing of the breadboard/PCB portion for this sensor and its functionalities.

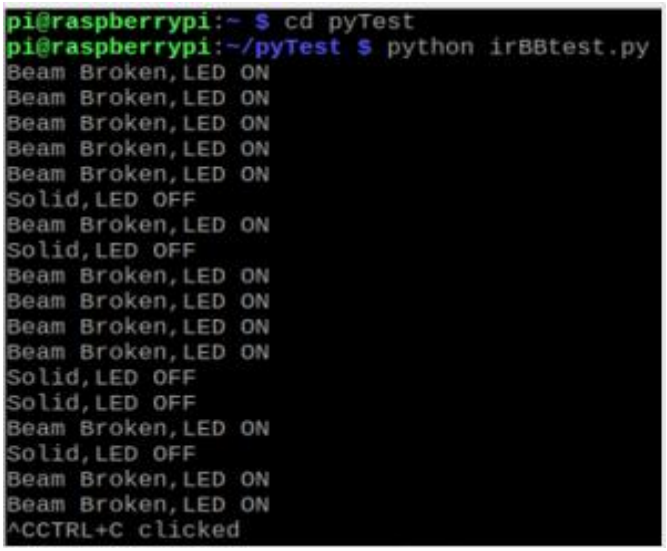


Figure 58 - Image of successful IR Break Beam test

PCA9685 Servo and RGB LED Controller Functional

Just like the VCNL4010, the PCA9685 utilizes the i2c interface and has a flexible voltage input range, additionally, it also has a built-in LED indicator that lights up when the controller is powered on, therefore simplifying not only the breadboard testing design, but the final PCB design as well.

Given that the PCA9685 is a 16-channel servo/PWM controller, an additional external power source is needed to avoid overloading the Raspberry Pi when supplying power to the servo motors and the LEDs. This external power source can be connected directly to the PCA9685 using an isolated power and ground pins. Once the PCA9685 has been powered and connected through the i2C bus, a simple python script can detect it and test its functionality by controlling connected servo motors or changing the color of an

RGB LED. The following screenshot below shows the design and development of the schematic design used for the circuit.

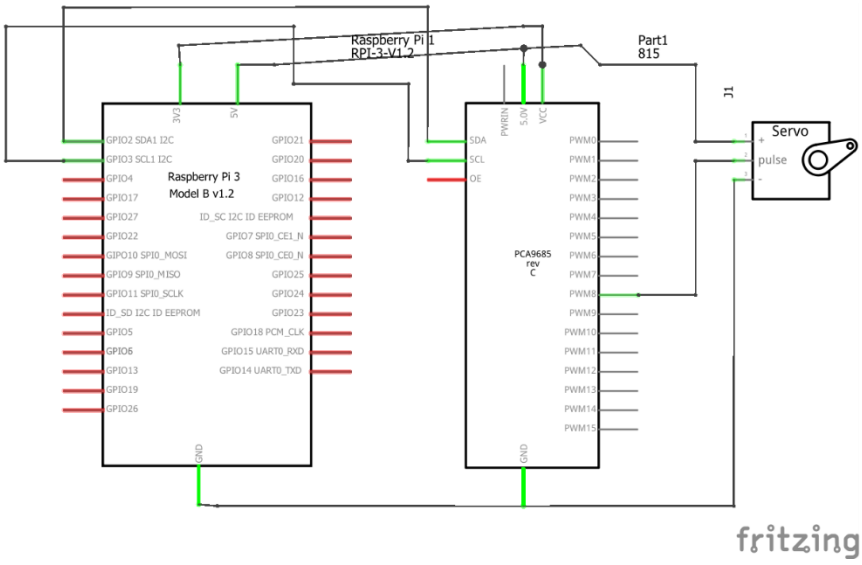


Figure 59 - PCA9685 Fritzing Schematic Design

After designing the schematic layout of the circuit, the breadboard design was developed through the Fritzing software as shown below.

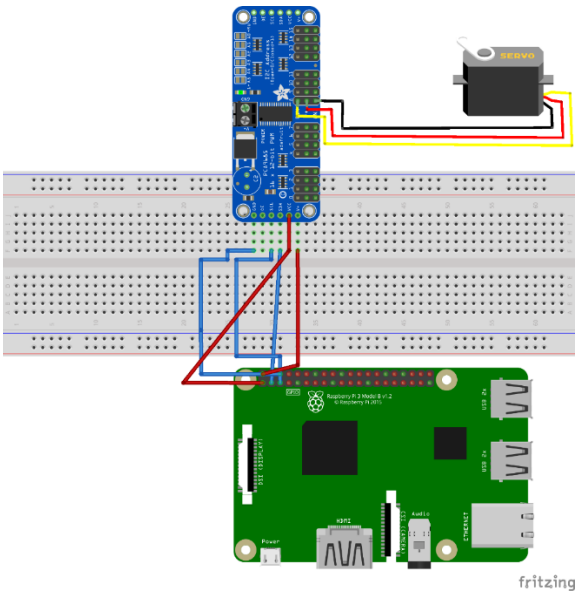


Figure 60 - PCA9685 Fritzing Breadboard Design

Due to the time commitments of this semester, our team decided it was best to not develop a PCB board for this individual sensor/effector and instead go straight to the final PCB design. At this stage of the project, the final PCB is currently being designed and finalized. The individual PCB was not developed for this sensor in the previous semester, as this is a new addition to the project, using a separate controller to hold the RGB LED's and their functions in the project. The use of this controller will help reduce the total voltage supplied to the Raspberry Pi through all 3 sensors, and will avoid any potential damage. As for this PCB design finally, we will make use of only 3.3V of the PI, and reduce the amount of current being used which is a maximum of 50mA, which we believe will be enough to support our project and its sensors.

For the YoLuke USB Camera sensor, a breadboard and PCB design were not needed as the sensor requires a USB connection to the Broadcom Development Platform only, being to the Raspberry Pi itself. We had initially planned to integrate a potential USB

connection on the final PCB design, but that idea was re-assessed and eliminated as the VCNL4010 and PCA9685 only needs a I2C interface and connection. The only difference comes from the IR Break Beam sensor, where it utilizes a SPI interface instead. When building the earlier prototypes, our aim was to test each sensor and effector individually before combining them all together and testing their functionality once joined.

In regards to our time and progress, we have been perfectly in line with our critical path, and have assembled a breadboard to test the connections of our final PCB before sending it to the prototype lab to be printed, once the PCB has been tested and finalized. We will work on the housing case that will be fit onto our custom parking lot design once each sensor and its accompanying functionalities have been established and thoroughly tested. This will be done through both the hardware and mobile application side of the project, with the main focus being the final PCB design and completion in the next phase of the project.

The VCNL4010 Proximity sensor will be used for the 4 valid parking spots on our parking lot system prototype model, which is currently being developed. Therefore, we will make use of an additional 3 VCNL4010 Proximity sensors and extra 3-4 RGB LED’s to fulfil the requirements of our project towards the end of the semester. The total amount for this final semester alone, will be $261.79 including only the materials

needed for the duration of this semester. Here's our current accumulation in a Bill of Materials format.

Product Name	Quantity	Unit Cost	Cost
Raspberry Pi 4b (4GB) Kit with power Supply and SD Card	1	$134.99	$134.99
VCNL4010 Proximity Sensor	4	$9.95	$39.80
IR Break Beam Sensor	2	$1.95	$3.90
PCA9685 PWM\Servo Controller	1	$19.84	$19.84
RGB LED (Pack of 10)	1	$5.95	$5.95
Power Adapter for external power	1	$12.98	$12.98
USB Camera	1	$15.00	$15.00
Micro Servo Motors	2	$5.95	$11.90
Jumper Wires	1	$2.59	$2.59
Resistor Kit	1	$14.89	$14.89
Total			$261.79

Figure 61 - Bill of Materials (all projects combined)

3.2.4 Printed Circuit Board

Demo

/1 Hardware present?

/1 PCB Complete and correct

/1 PCB Soldered wire visible but trim, no holes or vacancies

/1 PCB Tested with multimeter

/1 PCB Powered up

PCB Design process:

This section outlines the final process of combining each student's respective work from the previous semester into a single cohesive PCB unit. This included, integrating the work for the 3 main sensors utilized for the project which includes the VCNL4010 proximity sensor, IR Break Beam sensor, and the PCA9685 Servo Controller device. Mainly speaking, we will be going through the designing/creation, soldering, and testing phase of the final integration for the PCB.

As a start, and before we started designing our prototype PCB, we had to fully test and optimize all the wire connections needed to get all the sensors working together on the breadboard, and that included testing the power load and whether it is distributed safely from our Raspberry Pi or not, along with making any necessary adjustments. Once that was completed, we shifted our attention to the Fritzing software, where we recreated the wire connections that we recently optimized, and started to design the layout of the PCB and position each sensor in its designated spot and ensuring that no wires are

overlapping and most importantly, the design is small and compact enough for our usage.

Once the design was finalized, we sent it to the prototype lab to be printed, and when we received our PCB, we found our first issue, as the PCB was making contact with the Ethernet and USB ports on the Raspberry Pi which prevented the PCB from making full contact with the GPIO headers, that small issue was promptly fixed with a shorter design that accounted for the Ethernet and USB ports.

The following is the repository link to access our final PCB Fritzing design file used as part of the final project which can be accessed in the “electronics” folder:

Shortened PCB V3

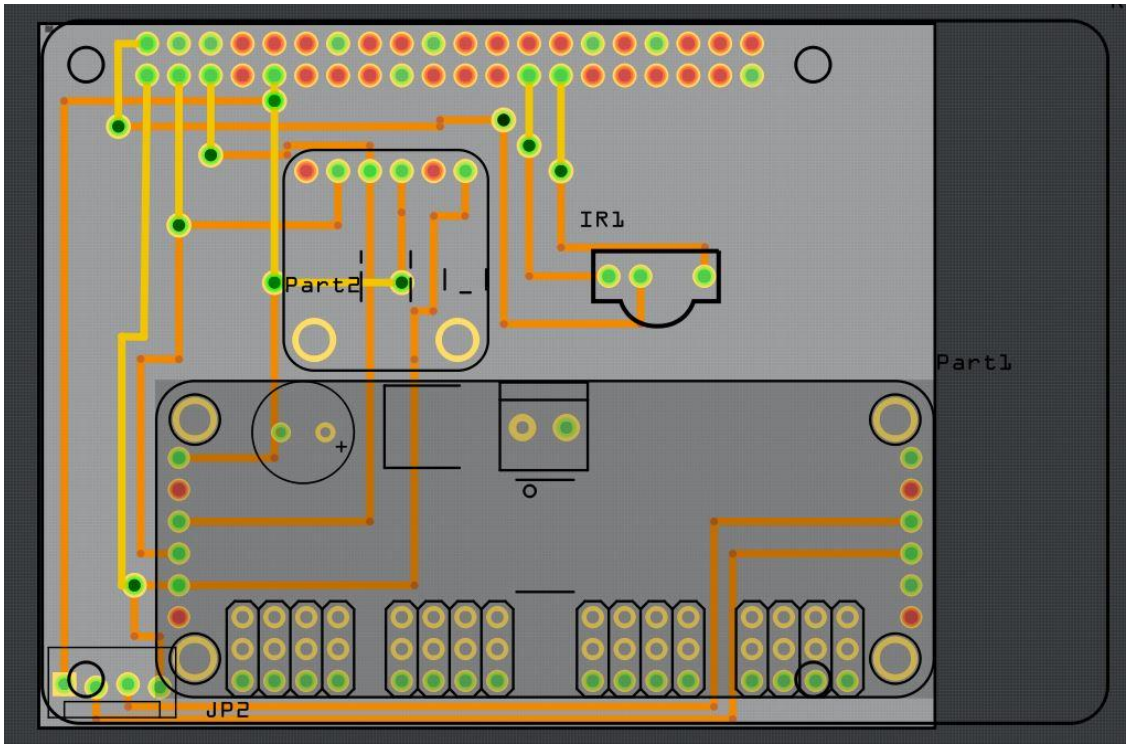


Figure 62 - Shortened Prototype PCB (Fritzing Design)

PCB Soldering

After we received our shortened PCB from the prototype lab, we started soldering the necessary headers onto the PCB and taking care of any VIA holes that needed a wire and solder to connect the two sides of the PCB. We had to ensure at this stage, the connections were not touching each other and avoid any short circuits in the process.

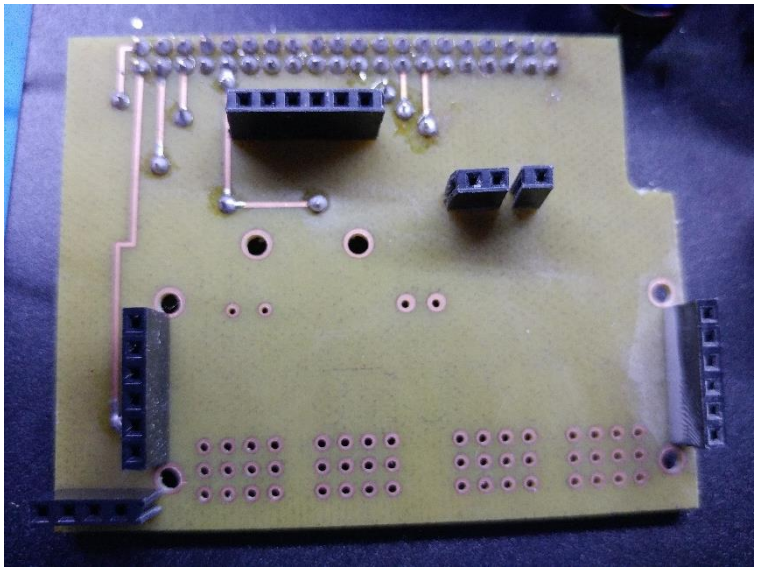


Figure 63 - Soldered PCB 1(initial)

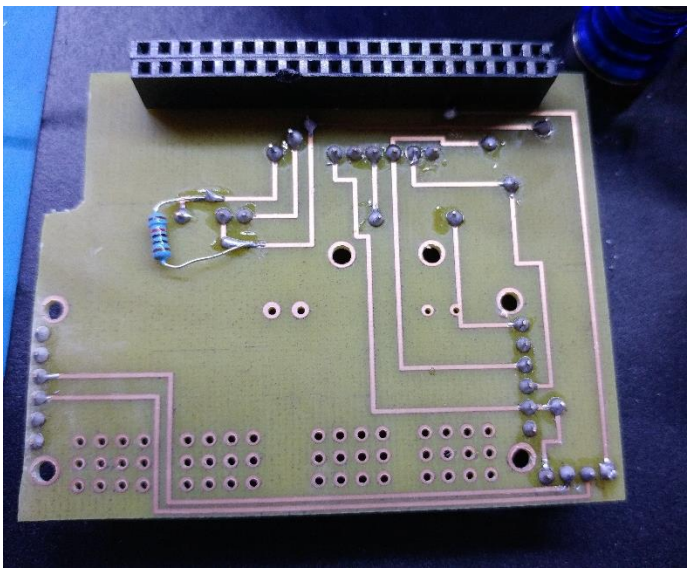


Figure 64 - Soldered PCB 2 (final design)

Multimeter Testing

The next step was to safe test the PCB before connecting the sensors and the Raspberry Pi. Therefore, we started off with a simple connectivity check using the multimeter, and then connected an external power supply to each power line in the PCB and used the multimeter to confirm that power is only present where it needs to be and that it reads the exact voltage provided by the power supply. Another purpose of this testing was to ensure that the correct resistance is being measured, using a resistor already present and being used on the PCB. So, in this case we tested using the addition of a 10K ohm resistor to check for the exact value being read from the multimeter device. In this case, a value close to the range of the original resistor value was read, that being 9.95K ohms when measuring the voltage from the GPIO pins using the 3.3V and 5V lines. So, based on the results we were able to ensure that the connections are going into the correct GPIO pins of the Raspberry Pi as the value matched as expected. Also, there was no interference with the voltage supplied and ground connections on the Raspberry Pi platform, which we were able to successfully avoid. This check was basically performed to analyze any misplaced connections for the GPIO pins on the Raspberry Pi to avoid any possible damage to any extent, and prevent overlapping of connections being too close to each other or touching a connection going to the GPIO pins of the Raspberry Pi causing the potential of further problems to arise.

The following screenshots show the process taken for the Multimeter testing portion measuring both resistance and voltage readings with a use of 10K ohm resistor.

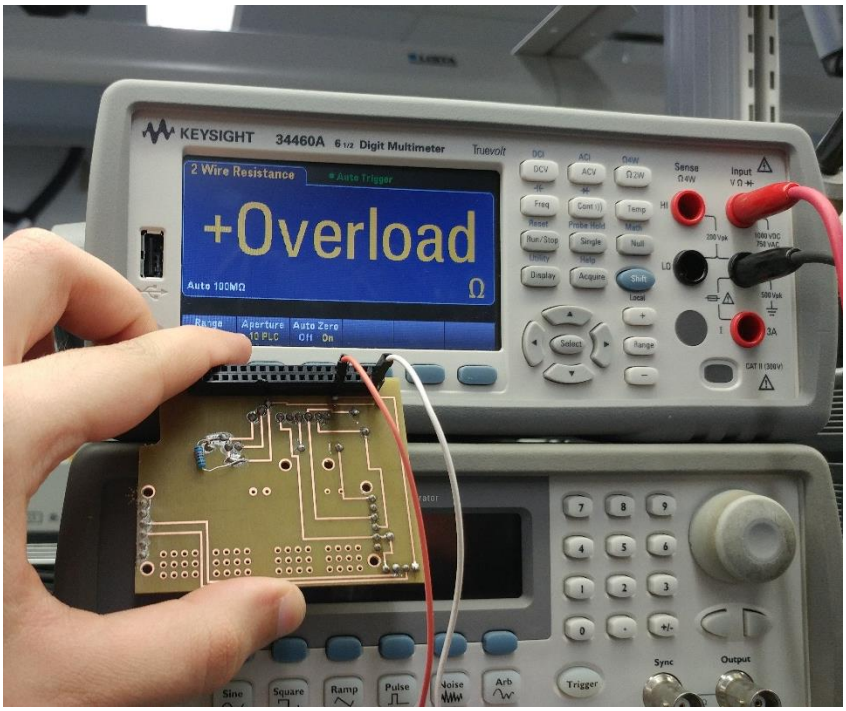


Figure 65 - 5V Resistance Test

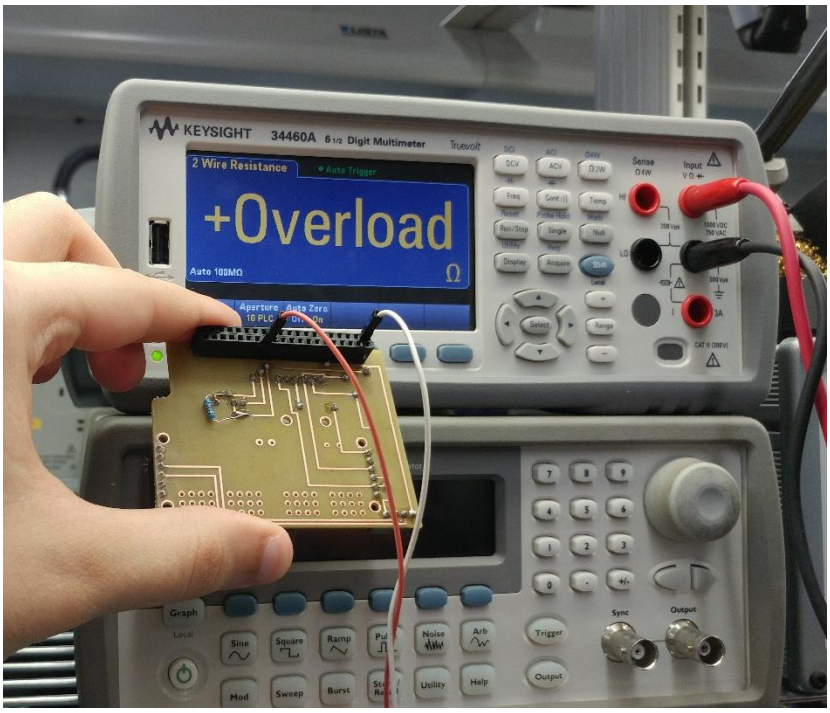


Figure 66 - 3.3V Resistance Test

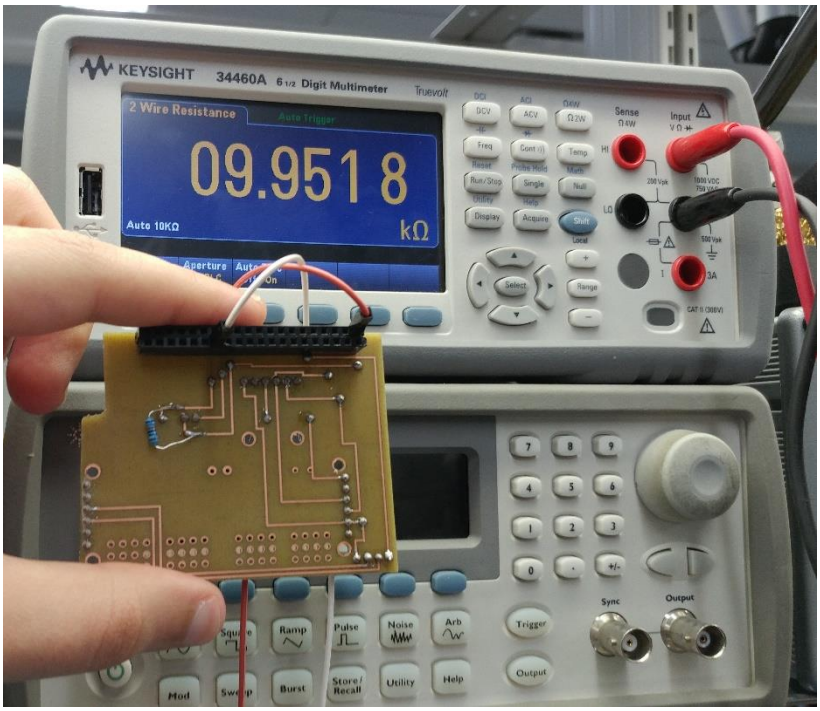


Figure 67 - 10K Resistance Test

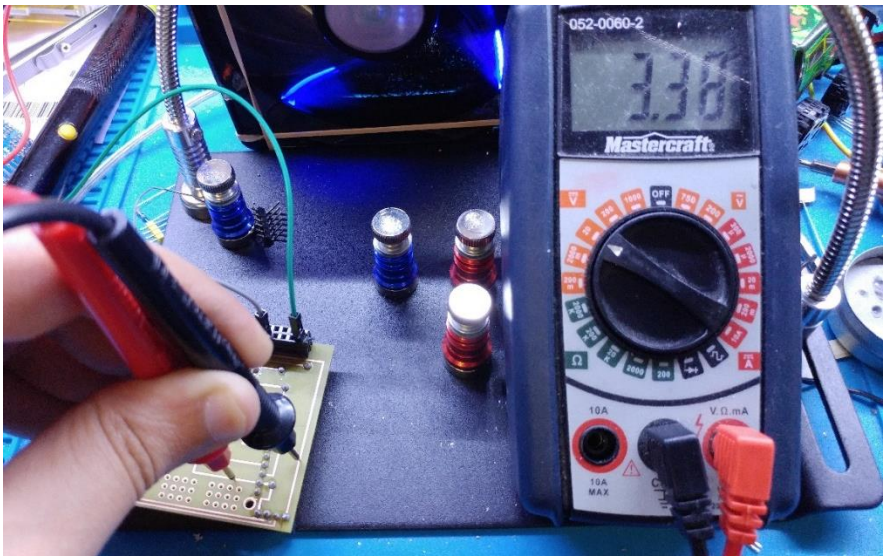


Figure 68 - Multimeter Test 1

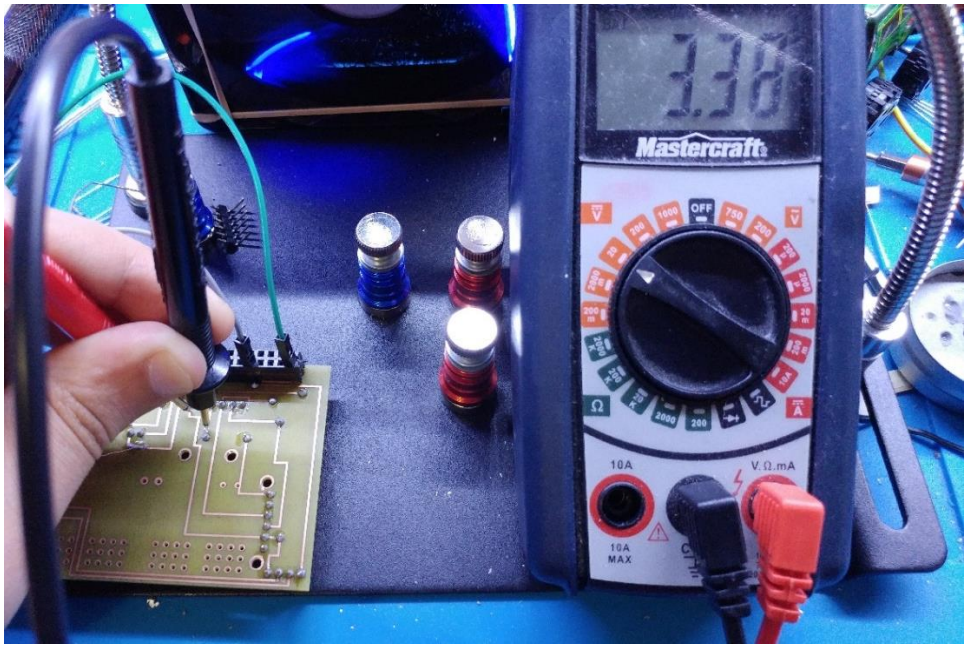


Figure 69 - Multimeter Test 2

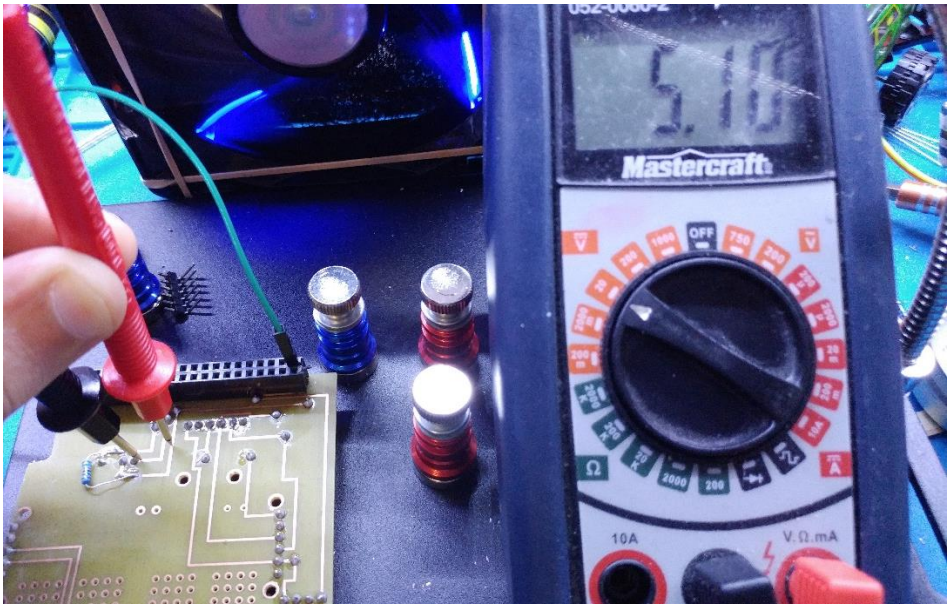


Figure 70 - Multimeter Test 3

Assembly and Confirmation

Finally, and after passing all the safety checks, we connected all the sensors to their designated headers and plugged in the PCB onto the 40 pin GPIO header of the Raspberry Pi and powered it up.

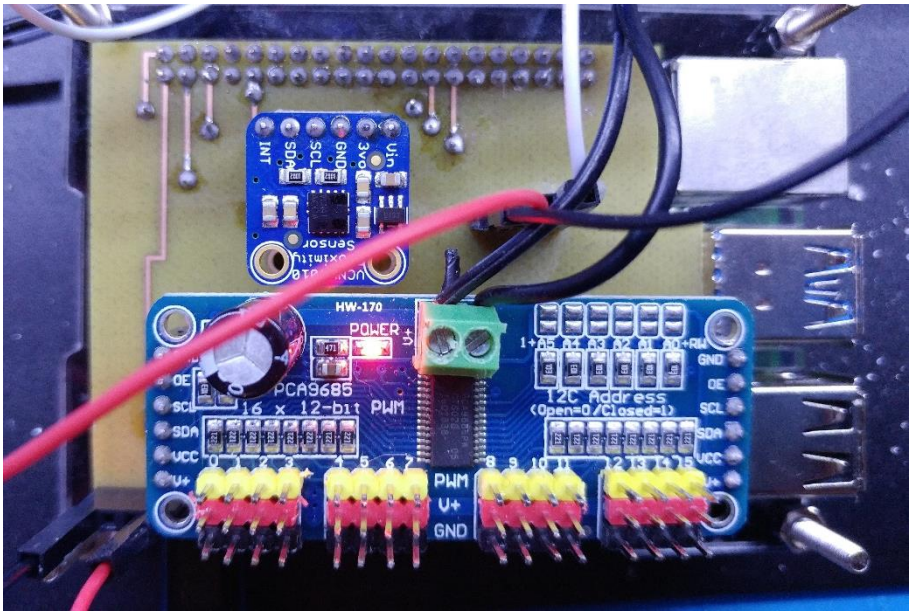


Figure 71 - PCB Connected

To test and verify that everything is working as it should, once the Raspberry Pi has booted, we opened up the terminal and issued the *"sudo i2cdetect –y 1"*, and verified that all sensors are detected, as for the IR Break Beam, which does not use the I2C interface, for that we used a simple script that checked the input on the GPIO pin connected to the IR Break Beam to confirm that it's working when the beam is broken or solid.

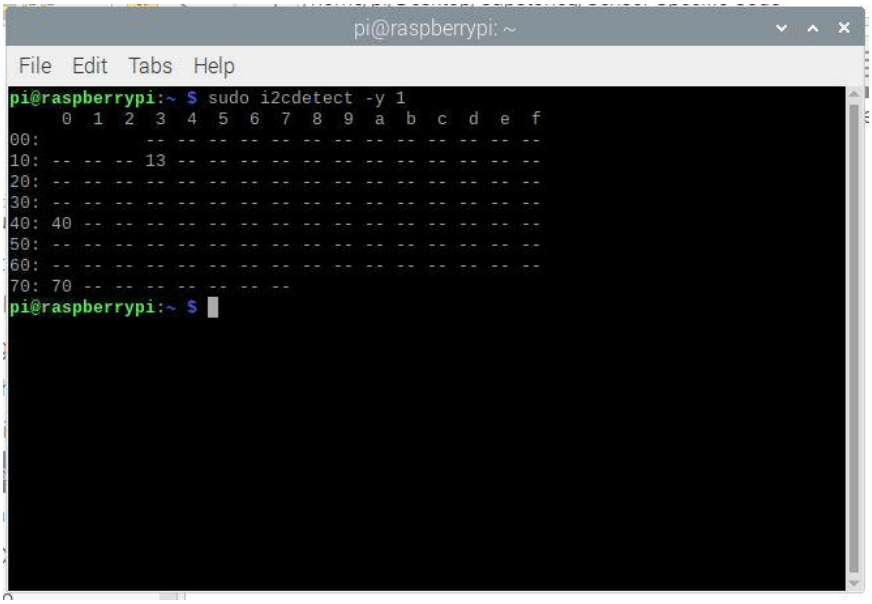


Figure 72 - I2C Confirmation

Future work for the PCB would include a possible smaller size and more GPIO connections to accommodate and additional IR Break Beam sensor for the exit gate, but as of right now, the current version of the PCB is confirmed to have no problems and behave as expected when we test against the projects main code.

3.2.5 Enclosure

Demo

/1 Hardware present?

/1 Case encloses development platform and custom PCB.

/1 Appropriate parts securely attached.

/1 Appropriate parts accessible.

/1 Design file in repository, photo in report.

The enclosure would have been set to have a parking lot prototype along with the Raspberry Pi/PCB attached to the center of the entrance and exit. This design would have been used to house the Raspberry Pi platform/PCB components, serving as solid protection from any outside harm and ensuring the safety of the project assembled. The following visuals below, showcase the end design of the SMART parking lot prototype and the final enclosure to hold the Raspberry Pi/PCB unit.

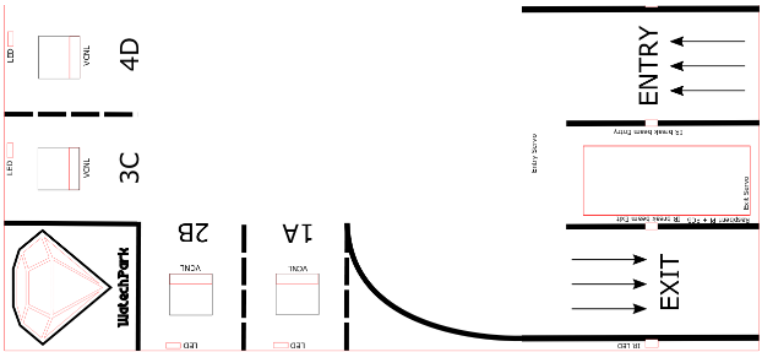


Figure 73 - Parking Lot Prototype Design (Inkscape)

The following link leads to the main repository to access the pdf file of the enclosure developed above. This file can be accessed through the “mechanical” folder as well as the svg file format. Provided below is the pdf format, as well as the svg format used with the Inkscape software.

PDF file format:

https://github.com/VikasCENG/WatechPark/blob/master/mechanical/WatechParkCaseDesignLaser.pdf

SVG File format (using Inkscape):

https://github.com/VikasCENG/WatechPark/blob/master/mechanical/Enclosure%20Design(semi%20Final).svg

As shown above, between the entrance and exit there is available space for the Raspberry Pi and the sensors attached. The sensors/effectors would go through under the parking lot and connect back to the space allotted. The IR Break-Beam sensors are connected to the entry and exit. The VCNL4010 proximity sensor is located at Slot 1A, which would be connected through a wall and attached facing the parking spot to detect the presence of a car approaching the spot. The camera would have been attached to a bar over the entry to scan the license plate on top of the car. The 3D printed barrier would have been attached to the sides of the servo motors horns, designed using the Inkscape software along with the case as shown below. The 3D printed barrier was discovered through the Thingiverse platform, which allows any shape/size of 3D printed objects to be downloaded in an instant for project purposes. The enclosure would clip onto the SMART parking lot model and hold the Raspberry Pi/PCB in place. The

following screenshots below showcase our planned designs for the acrylic enclosure to house the sensors/effectors. As well as, the 3D printed component that was planned to be connected to our parking lot prototype.

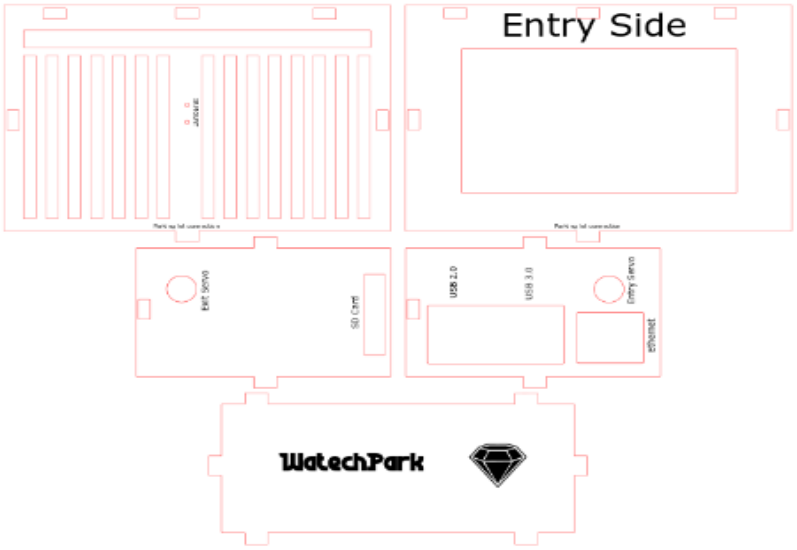


Figure 74 - Final Enclosure Design (Inkscape)

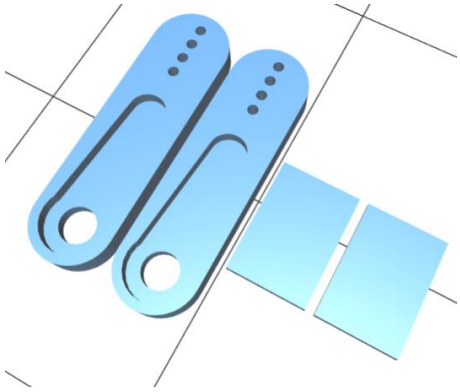


Figure 75 - 3D Printed SG90 Servo Horn Extender (used as barrier)

This is the link to access the prototype file as well as the STL file for the 3D printed barrier component. The file representing the prototype design can be found in the main repository under the “Parking Lot Design” folder. The 3D printed component can be

found in the “mechanical” folder along with the laser-cut enclosure file as mentioned above.

Parking Lot Prototype PDF File Access:

Link to WatechParkLotDesignLaser.pdf

SVG file format:

Link to WatechParkLotDesignLaser.svg

3D Printed (SG90 Servo Motor Horn Extender reference STL file):

Link to WatechParkDesign3D.stl

3.3 Integration

3.3.1 Enterprise Wireless Connectivity

Connecting our SMART Parking lot prototype wirelessly to the database is done through the Raspberry Pi 4 Model B. The connection between the Raspberry Pi and the WiFi is done through a configuration file on the Pi. The configuration file is found in the /etc/ directory. This directory holds all the system-wide configuration files and holds the /etc/wpa_supplicant/wpa_supplicant.conf. WPA is an acronym for WiFi Protected Access this is essentially a security certification program to secure wireless host networks. This file holds all the information about networks configured on your device. We configured the Raspberry Pi for our prototype by connecting it to Humber's network named "eduroam". To configure the device to connect to eduroam we had to edit the wpa_supplicant.conf and add a network line like this:

```
network={
        ssid="eduroam"
        key_mgmt=WPA-EAP
        auth_alg=OPEN
        eap=PEAP
        identity="n########@humber.ca"          #humber account number
        password="HumberAccPassword"            #humber account password
        phase2="auth=MSCHAPV2"
        priority=999
        proactive_key_caching=1
}
```

We then restarted the RPi and were able to connect remotely to the device from VNC Viewer. The Raspberry Pi is now allowed to be remotely accessed from home using an open Wi-fi spot, and work on the Raspberry Pi is done wirelessly instead of connecting all the peripherals physically.

3.3.2 Database Configuration

The online database is configured based on essential criteria needed to access the mobile application, hardware and vice-versa. The main source of delegating data is done through the Google Firebase database. SQL scripts are used to design these tables through the Firebase structure and data is populated based on a SQL format.

The parking lot prototype establishes its connection through setting up and initializing the firebase/pyrebase credentials, to gain access to the Firebase API. This includes, adding the API Key to send/retrieve data. Pyrebase is essentially a wrapper class used with python programming, to allow access to Firebase and manipulate the data. Following this, the dependencies were installed using the “sudo pip install pyrebase” command for each sensor to make the connection to the database.

There are five main data structures used in the project, along with four sub-siding tables used for the purpose of the mobile application, and other intended functionality of our parking application.

The ‘TestUsers’ data structure stores registration details specific to the user. The UID (user ID) acts as the primary key identifying each existing user and its registered account information.

Figure 76 - TestUsers table

The ‘ProximityData’ structure stores raw proximity values sent from the VCNL4010 hardware device, and is retrieved by the mobile application to display the real-time proximity levels of the lot.

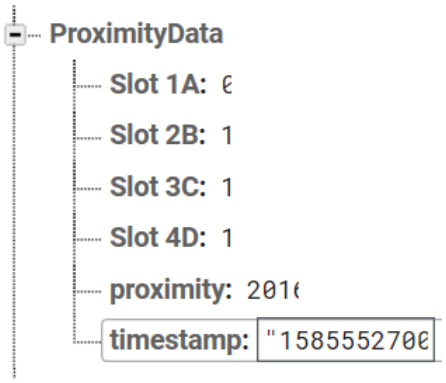


Figure 77 - ProximityData table

The ‘ParkingLocations’ table stores parking lot data under the specific UID of the current logged in user.

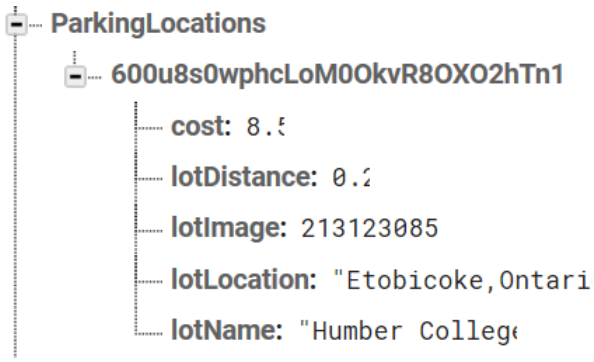


Figure 78 - ParkingLocations table

The ‘ParkingLocation’ table stores the reserved lot information. Data is retrieved through the use of foreign keys from the ‘ParkingLocations’ table. This includes the cost, location, duration, validity hours, type of pass, and the expiry time.

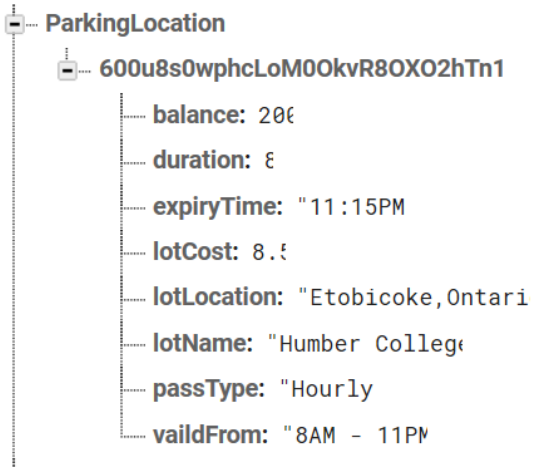


Figure 79 - ParkingLocation table

The ‘Orders’ table stores the payment processing details of the user after a parking pass purchase has been made. An OID is used in reference as a foreign key to each existing UID located in the ‘TestUsers’ table, working in accordance to identify each order based on a logged in/existing user.

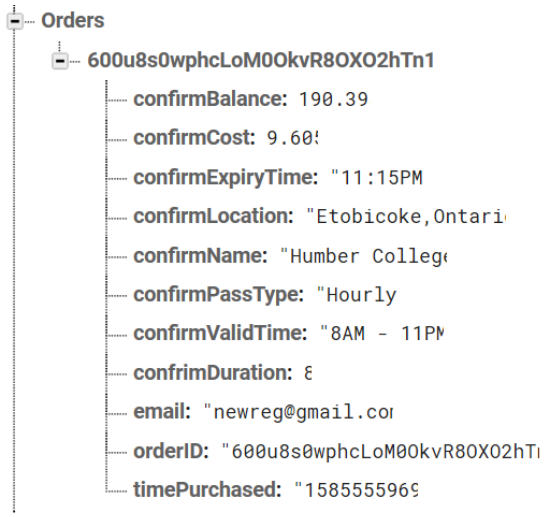


Figure 80 - Orders table

The ‘GateStatus’ table stores IR Break Beam entry/exit status and a timestamp to indicate the exact time an action is performed.

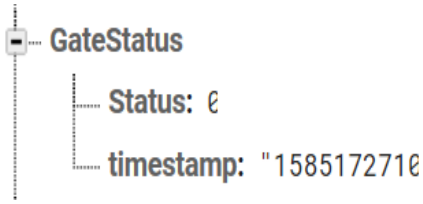


Figure 81 - GateStatus table

The ‘EntryStatus’ table stores a value of 0 each time the lot is full to notify the database of invalid entry.

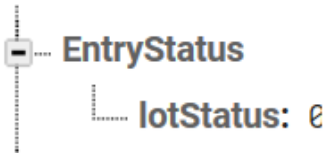


Figure 82 - EntryStatus table

The ‘Cars’ table stores the make, model, color, license plate number of the vehicle.

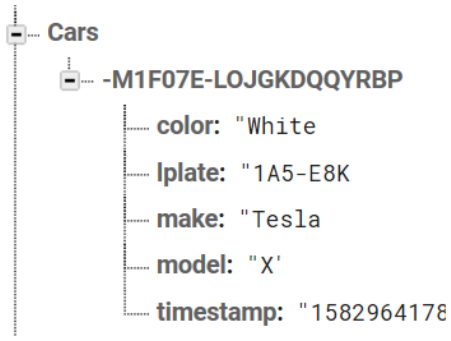


Figure 83 - Cars table

The ‘AdminControl’ table stores the status of the gate and sends a value of 1 or 0 to indicate an opening or closing.

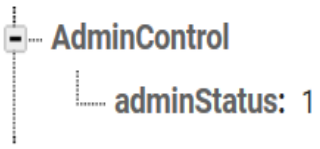


Figure 84 - AdminControl table

3.3.3 Security

Security is a vital feature in our parking application. The main goal of this feature being to prevent unwanted attacks, or potential harm to personal or sensitive data where data can be manipulated or abused. In our database setup, we have our read/write permissions set to limited access to outside users who may be trying to access the WatechPark database.

The read permission is set to “true” which allows any user to have access to the database, but for viewing purposes only. The write permission is set to “auth! = null” which means only authenticated users have access to modify the database.

Sensitive data such as passwords or payment information essential to a user are not stored in Firebase, through any form of encrypted credentials. This eliminates the possibility of a breach of data from any anonymous source that may serve as a threat, and ensures data integrity.

3.3.4 Unit Testing/Production Testing

Tests completed on individual parts will be described here. This includes the mobile application and Raspberry Pi parking lot prototype testing. Along with, the necessary steps taken after deployment of the project for checking for any potential bugs, crashes, and the overall production testing procedure.

WatechPark Android App

Test #1: Register

- Tested if entered registration data can be sent to Firebase and pushed to 'TestUsers' table

Test #2: Forgot Your Password

- Tested email and phone verification through Firebase provided functions
- Tested if email gets sent for new password request, notification is received through phone verification
- Tested for bugs, crashes after using email verification method, observed each method for any glitches, or possible crashes through the "debug" and logcat feature in Android Studio.

Test #3: Verify your Password

- Tested verification code process, if received notification with code to retrieve a new password
- Test if new password entered is updated for the user, and passwords match at Login.

Test #4: Login

- Test if email exists in Firebase data structure ‘TestUsers’, check for valid credentials
- If email exists in ‘Authentication’ section of Firebase and password credentials match go to Home screen, else display an error message

Test #5: Home Screen

- Tested if ‘Humber College’ parking lot is available in CardView structure along with a ‘View Details’ and ‘Reserve’ button, re-designed the menu to display a single lot choice, allowing a much faster, sleeker UI(user interface).
- Test when View Details button is pressed, information is displayed about the parking lot,
- Test when Reserve is pressed then it will allow you to get a parking pass, check Firebase if values have been added into ‘ParkingLocations’ table using the UID
- Check if user received notification on phone when Parking Pass is available.

Test #6: Parking Passes

- Check if data is sent and displayed from ‘ParkingLocation’ table and can access the reserved lot

- Test if ‘SELECT’ button sends the data to ‘ParkingLocation’ table and can be accessed through the Payment Screen

Test #7: Order History

- After parking pass is bought check if it appears on the Order History Screen using the ‘Orders’ table

Test #8: Payment

- Test if data is displayed based on the selected parking pass from Parking Passes screen using the ‘ParkingLocation’ table
- Check ‘Orders’ table on Firebase to make sure order went through once payment is processed

Test #9: Add a Car

- Test if users can add a car by entering make, model, color, and license plate number.
- Check if the data is sent to the Firebase data structure “Cars”

Test #10: Manage Cars

- Check if added cars from the ‘Cars’ table appear here.
- Test if user can edit or delete cars by pressing the coordinating buttons

Test #11: View Details Button (Parking Lot Data Screen)

- Test if ‘GateStatus’ value on Firebase is changed for entry and exit and colors change on app accordingly, along with a timestamp

- Test proximity value reading from Firebase using the ‘ProximtityData’ table, and that values is updated in real-time
- Test admin control sending values for gate control through ‘AdminControl’ table

SMART Parking Lot Prototype

Test #1: IR Break Beam

- Test if the sensor can detect if an object is blocking IR beam, and if receiver can detect the beam being broken
- Test if a HIGH value is received when the beam is detected, noEntry is sent to close gate, and LOW value when beam is not detected, entry is set to open gate using ‘GateStatus’

Test #2: VCNL Proximity

- Test if proximity values are sent based on if a vehicle approaches the sensor. Report the distance to Firebase under ‘ProximityData’ table, database should be updated according to sensor value
- Check if Slot 1A is available if the sensor reads a value less than or equal to 2500. If the value is greater than 5000 the space is taken. If value is between 2500 and 5000, car is approaching the space.

Test #3: Servo-Motor

- Once the IR beam receiver detects interference with IR beam it will raise the servo motor arm and allow entry/exit

- Check if ‘GateStatus’ on Firebase has changed value for when gate is allowing “entry” (1) for true condition or “noEntry” (0) for false condition. Gate will lower arm after 2 seconds and status will change to noEntry.

April 14/15 Results and Discussion, Conclusion, Prescreening, Report Mechanics and Structure Checklist: (section 10 of https://www.oacett.org/getmedia/9f9623ac-73ab-4f99-acca-0d78dee161ab/TR_GUIDELINES_Final.pdf.aspx)

your repository/Documentation/GroupNameConclusionandChecklists.pdf

4.0 Results and Discussions

Our SMART parking assist platform, acts as an alternative method to support everyday consumer occurrences and provide the tools necessary to make parking easier, and more portable through forms of digital appeal by a mobile application for handy pocket pick-up use, and by offering parking data monitoring functionalities all bundled in one location. We have developed a parking lot prototype which works as a small-scale model to mirror our parking application with interactivities to offer to the consumer demographic/audiences worldwide. This application is developed with the consumer in mind, with ideas designed around the types of features a user may desire and crave from a business standpoint, as well as figuring out ways of how to present parking through a much more advanced, modern day approach in return making consumer/driver lives easier and more manageable. In terms of the end outcome of our prototype, we believe we have tackled our planned objectives for the project in a suitable and manageable manner for what is possible with the tools, services we have access to during development, and the final integration of all hardware/software components.

Due to the circumstances, we were faced with many challenges in overcoming how we would be able to meet the end requirements, including having all planned parts

functioning and working in the end. There is always room for further improvement, after any kind of deployment and we believe collectively in terms of the hardware side/footprint of the prototype there can be much needed changes, modifications and further improvements to the existing software infrastructure. This will be further discussed in the Conclusion section under the “Recommendations/Future Steps” branch. We believe if given the opportunity to complete the project in its entirety we would have a much more efficient, and working product with all pieces brought to life and physically assembled. This was because, in the end we were unable to have our parking lot prototype designs come to life, and physically assembled with the other hardware pieces such as the enclosure design and sensors/effectors. Thus, due to the situation we focused mainly on completing all aspects of the project and parking application, without the physical assembly but give our full attention to refining the mobile application, testing the hardware sensors/effectors and adding the planned functionality which would have been showcased with a connected parking lot prototype. Although, in the end we are proud of what we set out to do from the initial development in the previous Fall 2019 semester, and what we were able to accomplish during this final Winter 2020 semester. As mentioned, there is always room for improvement in any type of project or application, and we believe as with anything nothing can be in-stone perfect.

Future directions can lead to a much stabilized and finished product with having the physical assembly also potentially completed with lead way to time and integrated for demonstration and showcase purposes. The work accomplished on the software end we believe is outstanding from both the firmware and mobile application sides. In the

end, we were able to include all functionalities of each sensor into the firmware code and build upon features for the mobile application, refine the work done for testing and deployment down the line. Due to time limitations and limited access to vital services, we were forced to work with the tools we had at our disposal near the end of the term and focused on providing a finished end product with all working features from the hardware/software side. Although, being unable to demonstrate our accomplishments with the hardware sensors/effectors functioning intact with the mobile application and active database setup in person but virtually.

This project taught us a variety of essential skills and tactical strategies to utilize from both industrial and technical standpoints, many of our hurdles along the way allowed our group to remain patient and face any challenges and overcome them. This includes, working in a team environment, which in the future for each of us would be important and informative to being able to share ideas, bounce off design choices or development strategies working together as opposed to individually. This project taught us how to be vigilant with our work, and put in the most effort by making sure the work is divided evenly and each member along the journey contributes an equal amount in the end. This skill as well as experience would be an ideal choice to consider when building a resume, as we were able to cooperate as a team and be active in doing so.

From an employer’s standpoint, this would be an outstanding achievement and would help showcase our skills we learned from this capstone project environment. We learned how to schedule events, manage our time wisely using management tools such as the Gantt chart, being able to track our status based on planned milestones from the start of development. We learned how to take a schematic design, and transfer the

circuit to a breadboard and then finally create individual PCB's, followed up with a combined unit holding each respective sensor/effector with use of the Fritzing software, and having access to services from campus such as the prototype lab. This also taught us how to design enclosures, PCB's in the most effective way and correctly with avoiding major mistakes down the line in production, which may be costly to us in the future for testing purposes. We learned how to use development tools/graphic applications such as CorelDraw and Inkscape to create 2D enclosure/parking lot prototype designs and use the tools provided to build a 3D representation of what was required from our parking application. Through this, we learned how the laser-cutting machines function and produce an acrylic design and have that come to life with safety measures taken and considerations taken for the types of material being available at our disposal in the prototype lab.

In this process, we remained patient in getting the work done, with having in mind there may be the need of multiple print-outs of the enclosure to fit the dimensions of the parking lot platform, or make enough space to hold the Raspberry Pi/PCB and the sensors/effectors attached to these hardware units. For the most part, we had to go through multiple designs, re-adjust and test which designs fit our hardware and work with those designs. We did not get the chance to take our case designs and physically assemble one, but we had the correct measurements taking into consideration each side and hardware piece. We learned how to work with an online database and figure out ways to communicate with a mobile application, and work with a development platform. We learned how to take individual hardware projects/units and re-design these units into a single, compact unit holding all sensors/effectors taking into consideration

the hardware requirements for each sensor/effector, and any hardware limitations. Throughout this process, we learned it is not always possible to have planned objectives work out in the end, and there should always be backup plans or alternative ways put into place to achieve a goal. For example, facing the problem with the VCNL4010 device and being unable to support all 4 spots on the prototype due to hardware limitations on the I2C device end.

Collectively, we really enjoyed working on this project as through the ups and downs we remained loyal and continued developing and moving forward with the pieces needed to assemble the final project. There were some doubts from the previous semester, which may have derailed the project due to a former member leaving the group but fortunately allowed us to put in even more effort, and complete the project whether individually as from last semester, or in the end as a three-member team. We are proud with what we had set out to accomplish and the end outcome, as we completed the project in a finished fashion with a majority of the planned features along with some minor adjustments along the way. In the end, our parking lot prototype was able to achieve and perform its overall purpose for the consumer, and we believe we have demonstrated this efficiently to help create a parking lot prototype that mirrors our software as well as helps in visualizing a SMART parking assist platform. As mentioned, there is always room for improvement in the case of our prototype as it is not perfect, but this product accomplishes our key goals and requirements for the project which was our target in the end. We will be actively monitoring our current prototype, and continue to update the hardware if needed through further advancements based on the technology world. Our SMART parking system targets consumer lives in support of

everyday parking occurrences, and helps provide the gateway to parking from a global perspective.

A sincere thank you to Professor Kristian Medri and Professor Austin Tian, for helping shape this project and providing active feedback to further improve on key topics in support with the overall development and final integration.

5.0 Conclusions

The WatechPark IoT capstone project was developed to address the consumer demographic by determining an alternative method in regards to payment for parking, capacity management, and real-time information gathering. During the course of fifteen weeks, we have worked extensively on designing, developing, and integrating hardware/software components in preparation of unit testing, mass production testing phases, and overall refinement. Therefore, through the combined effort of all team members, we have successfully achieved our proposed objective by creating an alternative parking lot management system, to assist with everyday consumer occurrences and parking situations.

Recommendations/Future Steps

Future steps may include, a more compact PCB design to accompany the use of the two servo motors. Thus, reducing the overall footprint size of the project by a more considerable amount and allowing further room for improvement in terms of hardware use/capabilities. Other possible additions, may include adding support for each available parking spot on our prototype model. This would require the use of a I2C multiplexer to allow individual VCNL4010 proximity sensors to be positioned at each parking space, rather than the current single VCNL4010 device. Additionally, to appeal to the mass market a web application may be developed to allow a wide range of public users the ability to view advanced metrics/live parking lot data. The application would mirror the mobile application and allow the consumer to interact through a global reach.

Thus, in return gaining further popularity to contend as a major alternative SMART parking IoT platform from an investor's standpoint.

In regards to mass production of the project, including the parking lot prototype model we have each individual file associated to the hardware/software components of the project present in our project repository for replication purposes. This includes, the hardware designs of the PCB assembled in the end of the project, as well as the design files for the enclosure developed through CorelDraw, 3D printing pieces to accompany our small-scale parking lot model. This would also depend on the location the sensors/effectors are ordered from, and the ideal timeframe the parts can be delivered. Along with no major interruptions in the shipping and delivering of the parts, an individual can gather all parts in roughly two days and begin working on the PCB and electronic components. As all of the major development files are available on our repository, this includes the main code to run our mobile application an individual would only need to have access to an Android Studio development platform and run the software through an emulator, or a hardware device. Through the documentation files uploaded, it would be easier for anyone without a technical expertise to be able to replicate the project in due time with the mobile application, and using the files readily available without having the need to re-create from scratch.

Ideally, to mass produce between 1000 to 100,000 units of the prototype and hardware elements such as the PCB and enclosure, we would need to have access to a printing services for laser-cutting acrylic as well as help re-produce the PCB design through the Fritzing application and make use of the Gerber files for mass-production of these parts. In terms of overall production of the hardware prototype, an excess amount of

acrylic/3D material would be needed to complete the enclosure, replicate the 3D printed components as well as the parking lot prototype in a small-scale atmosphere. This also includes, ordering the sensors/effectors which would be costly and stock status would need to be considered, and the available locations with these parts. An increased amount of tools/materials would be required for mass-production phases and extended marketing use with the sensors/effectors being in high demand and gathered through online access, including shipping/potential duty costs. The parking lot prototype would also require the use of a laser-cutting machine to be able to bring the model to life and replicate the project in its entirety. For testing procedures, access to a multimeter would be ideal to allow for testing of the sensors/effectors used in the project and test each sensors/effector thoroughly before beginning the next production phase for any upcoming units. This incudes, testing connections once soldering the sensors/effectors to the PCB, which we have also included documentation work/procedures to support any individual attempting to solder equipment, parts with an ideal temperature range to work with, without combining connections or creating a short in the process which can lead to possible damage to the hardware.

In the case, of production failure we would assess each part that may be causing the issue and have a backup plan for each hardware component. This can include, re-using some of the material such as the acrylic material for the enclosure or parking lot prototype. This would ensure, before mass-producing the project, that it would be capable to function as originally designed and developed following the steps provided through our main project repository and cut-down on excess waste for manufacturing purposes in the future. Other fundamentals required, would be making use of the

firmware code available to run each sensor appropriately and have each part perform its necessary functionalities though the code and for the purpose of our parking application. Through these considerations, it would be ideal for the project to potentially be replicated over the course of two weekends. This includes, having access to the parts/materials laser-cutting/3D printing services and a soldering station that can used for completing the hardware, and testing the code for each sensor. Due to this, for commercialization purposes the product itself would require the cost of shipping, duty and taxes which as mentioned would include an excess amount of printing material, accumulated costs through designing, producing and manufacturing; would also be included when considering an ideal marketable cost for a complete project with all working parts for the potential of 1000 or more units being readily available for consumer purchase.

6.0 References

EasyPark. (2016). Retrieved from EasyPark Mobile Parking App: https://www.easypark.ca/products-services/mobile-parking-app

EasyPark. (2016). Retrieved from City Parking Vancouver, Public Parking, EasyPark Mission: https://www.easypark.ca/about-easypark/mission

EasyPark. (2016). Retrieved from History: https://www.easypark.ca/about-easypark/history

Google Play. (2020). Retrieved from EasyPark Parking - Apps on Google Play: https://play.google.com/store/apps/details?id=ca.easypark2.app&hl=en

Park Indigo Canada Inc. (2019). Indigo. Retrieved from https://ca.parkindigo.com/en

ParkWhiz. (2019). Find and Book Parking Anywhere. Retrieved from https://www.bestparking.com/

7.0 Appendix

Modified Code Files(Firmware code)

VCNL4010_simpletext.py

```
# Distributed with a free-will license.
# Use it any way you want, profit or free, provided it fits in the licenses of its associated works.
# VCNL4010
# This code is designed to work with the VCNL4010_I2CS I2C Mini Module available from ControlEverything.com.
# https://www.controleverything.com/content/Light?sku=VCNL4010_I2CS#tabs-0-product_tabset-2

import random
import pyrebase
import time
import math

import smbus
from time import sleep
from gpiozero import LED
import RPi.GPIO as GPIO
GPIO.setmode(GPIO.BCM)
# Get I2C bus
bus = smbus.SMBus(1)

red = LED(17)
green = LED(18)
blue = LED(27)

# Sensor 1 Status
open1 = 0
occupied1 = 1
# Sensor 2 Status
open2 = 0
occupied2 = 1
# Sensor 3 Status
open3 = 0
occupied3 = 1
# Sensor 4 Status
open4 = 0
occupied4 = 1
```

```
def turnOff():
    red.off()
    green.off()
    blue.off()


try:
    while True:

# VCNL4010 address, 0x13(19)
# Select command register, 0x80(128)
#               0xFF(255)     Enable ALS and proximity measurement, LP oscillator
        bus.write_byte_data(0x13, 0x80, 0xFF)
# VCNL4010 address, 0x13(19)
# Select proximity rate register, 0x82(130)
#               0x00(00)       1.95 proximity measeurements/sec
        bus.write_byte_data(0x13, 0x82, 0x00)
# VCNL4010 address, 0x13(19)
# Select ambient light register, 0x84(132)
#               0x9D(157)     Continuos conversion mode, ALS rate 2 samples/sec
#bus.write_byte_data(0x13, 0x84, 0x9D)

        time.sleep(0.8)

# VCNL4010 address, 0x13(19)
# Read data back from 0x85(133), 4 bytes
# luminance MSB, luminance LSB, Proximity MSB, Proximity LSB
        data = bus.read_i2c_block_data(0x13, 0x85, 4)

# Convert the data
#luminance = data[0] * 256 + data[1]
        proximity = data[2] * 256 + data[3]

        global db
        config = {
        "apiKey": "AIzaSyBHz-ZrX8ANSYz3qcVdbjQ_KvpX8Kz3PnU",
        "authDomain": "watechpark.firebaseapp.com",
        "databaseURL": "https://watechpark.firebaseio.com",
        "storageBucket": "watechpark.appspot.com"
        }
        firebase = pyrebase.initialize_app(config)
        db = firebase.database()
```

```
        seconds = time.time()

        # Sensor Data
        a = {"proximity": proximity,"Slot 1A": occupied1,"Slot 2B": occupied2, "Slot 3C":
occupied3, "Slot 4D": occupied4,
        "timestamp": str(int(math.ceil(seconds)))}
        b = {"proximity": proximity,"Slot 1A": open1,"Slot 2B": occupied2, "Slot 3C":
occupied3, "Slot 4D": occupied4,
        "timestamp": str(int(math.ceil(seconds)))}

# Output data to screen

#print "Ambient Light Luminance : %d lux" %luminance
        if(proximity<=2500):
            green.on()
            print("\nSlot 1A is available")
            print("Proximity of Sensor 1 : %d" %proximity)
            print(bool(open1))
            print("\nSlot 2B is occupied")
            print(bool(occupied2))
            print("\nSlot 3C is occupied")
            print(bool(occupied3))
            print("\nSlot 4D is occupied")
            print(bool(occupied4))
            db.child("ProximityData").set(b)
            print("Proximity data successfully updated to Firebase!\n")
            sleep(2)
            turnOff()
        elif(proximity>5000):
            red.blink()

            print("\nSlot 1A is currently occupied")
            print("Proximity of Sensor 1 : %d" %proximity)
            print(bool(occupied1))
            print("\nSlot 2B is currently occupied")
            print(bool(occupied2))
            print("\nSlot 3C is currently occupied")
            print(bool(occupied3))
            print("\nSlot 4D is currently occupied")
            print(bool(occupied4))
            db.child("ProximityData").set(a)
            print("Proximity data successfully updated to Firebase!\n")
            sleep(2)
            turnOff()
            #red.off()
        elif(proximity>=3000) or (proximity<=4500):
```

```
            blue.on()

            print("\nCar is approaching the parking space")
            print("Proximity of Sensor 1 : %d" %proximity)
            sleep(2)
            turnOff()
            #blue.off()

            sleep(2)
            turnOff()


            db.child("ProximityData").set(a)
            print("Proximity data successfully updated to Firebase!\n")
        else:
            print("Failed to push proximity data to Firebase!\n")


finally:

    GPIO.cleanup()


    #GPIO.output(18, 1)
    #green.on()
```

This sensor code will react to status changes on Slot 1A of the parking lot and send the proximity data gathered to the database and through to the mobile application.

For reference: Link to the contents of the VCNL4010_simpletest.py

irBBnewtest.py

```
##########
#File: irBBtest.py
#Date: Sunday Nov 03, 2019   22:32 PM
#Author: George Alexandris
#Purpose: Going to implement uploading data to the firebase database
#########

import time
import RPi.GPIO as GPIO
import pyrebase
```

```
Gpio_Pin=10

trueCheck = 0
falseCheck = 1

global db
config = {
"apiKey": "",
"authDomain": "",
"databaseURL": "",
"storageBucket": ""
}
firebase = pyrebase.initialize_app(config)
db = firebase.database()


seconds = time.time()

# Sensor Data
noEntry = {"Status": falseCheck, "timestamp": str(int(math.ceil(seconds)))}
entry = {"Status": trueCheck, "timestamp": str(int(math.ceil(seconds)))}

#setup GPIO pin as input to read value for HIGH or LOW
GPIO.setmode(GPIO.BCM)
GPIO.setup(Gpio_Pin,GPIO.IN)

try:
        while True:
                x = GPIO.input(Gpio_Pin)
                if(x==1):
                print("Solid,LED OFF")
                print("Vehicle currently not detected at entry")
                print(bool(falseCheck))
                db.child("GateStatus").set(noEntry)
            print("Gate status successfully updated to Firebase!\n")
                if(x==0):
                print("Beam Broken,LED ON")
                print("Vehicle detected at entry")
                print(bool(trueCheck))
                db.child("GateStatus").set(entry)
            print("Gate status successfully updated to Firebase!\n")
                time.sleep(2.0)

except KeyboardInterrupt:
        print("CTRL+C clicked")
```

```
except:
        print("some error occurred")

finally:
        GPIO.cleanup()
```

This sensor code will react to gate changes when a vehicle approaches entry, and when a vehicle comes in the way of the sensor it will trigger the camera sensor to take a screenshot of the license plate on the car and begin its validation process. It sends a boolean 1 or 0(true or false) value to a data structure in Firebase, staying active at all times in the presence of the vehicle at the entrance.

For reference: Link to the contents of the irBBnewtest.py

servotest1.py

```
from board import SCL, SDA
import busio
import time

from adafruit_pca9685 import PCA9685

from adafruit_motor import servo

i2c = busio.I2C(SCL,SDA)

pca = PCA9685(i2c)
pca.frequency = 50

servo4 = servo.Servo(pca.channels[4])
servo1 = servo.Servo(pca.channels[0])

for i in range(180):
    servo4.angle = i
    servo1.angle = i
    #time.sleep(0.1)
time.sleep(5)
for i in range(180):
    servo4.angle = 180 - i
    servo1.angle = 180 - i
```

pca.deinit()

This sensor code controls the entry/exit gates of the parking lot and provides rotation of the motors for a vehicle to enter the parking lot. The sensor uses a I2C interface much like the VCNL4010 proximity sensor.

For reference: Link to the contents of the servotest1.py

watech(MP).py(Main Program integrating all 3 sensors) – with multiprocessing

```
import pyrebase
import time
import RPi.GPIO as GPIO
import math
import random
import busio
import smbus
import board
import multiprocessing
import threading
import theonewhoshallnotbenamed


#Sensor Specific Imports
import adafruit_vcnl4010
from adafruit_pca9685 import PCA9685
from adafruit_motor import servo

#Firebase initialization
global db
db = theonewhoshallnotbenamed.get_db()

#function to handle the entry gate
def entGate():
    ent_srv = servo.Servo(pca.channels[7])
    for i in range(90):
        ent_srv.angle = i
        #gate is open at this point, updating database:
    db.child("GateStatus").set(noEntry)
    while(GPIO.input(ent_ir)!=1):
          time.sleep(0.5)
    time.sleep(2.0)
```

```
    for i in range(90):
        ent_srv.angle = 90 - i

#function to handle the entry gate
def extGate():
    ext_srv = servo.Servo(pca.channels[3])
    for i in range(90):
        ext_srv.angle = i
    while(GPIO.input(ext_ir)!=1):
          time.sleep(0.5)
    time.sleep(2.0)
    for i in range(90):
        ext_srv.angle = 90 - i


def camSen():
    return "1A6-E81"

def licCheck():
    incommingCar = camSen()
    getstat = db.child("Cars").get()
    for user in getstat.each():
        userid = user.key()
        platVal = user.val()["lplate"]
        #print(platVal)
        if(platVal==incommingCar):
            return 1
    return 0

#Variable Initialziations
ent_ir = 0
ext_ir = 24

trueCheck = 1
falseCheck = 0

seconds = time.time()

i2c = busio.I2C(board.SCL,board.SDA)
#might need to use this instead i2c = busio.I2C(board.SCL,board.SDA)

#Sensor variabl declaration
global pca
pca = PCA9685(i2c)
pca.frequency = 60
vcnl = adafruit_vcnl4010.VCNL4010(i2c)
```

```
#GPIO Setup
GPIO.setmode(GPIO.BCM)
GPIO.setup(ent_ir,GPIO.IN)
GPIO.setup(ext_ir,GPIO.IN)

# IR Sensor Data
global noEntry
noEntry = {"Status": falseCheck, "timestamp": str(int(math.ceil(seconds)))}
global entry
entry = {"Status": trueCheck, "timestamp": str(int(math.ceil(seconds)))}


# Sensor 1A Status
open1 = 0
occupied1 = 1
# Sensor 2 Status
open2 = 0
occupied2 = 1
# Sensor 3 Status
open3 = 0
occupied3 = 1
# Sensor 4 Status
open4 = 0
occupied4 = 1


#main code:
def adminControl():
    print("this is adminControl running")
    try:
        while True:
            adminC = db.child("AdminControl").get()
            if(adminC.val()["adminStatus"]==1):
                entGate()
                db.child("AdminControl").set({"adminStatus": 0})
                time.sleep(1)
    except:
        print("Admin Control is down")

    finally:
        pca.deinit()
        GPIO.cleanup()

def vcnlFunc():
```

```
    print("this is VCNL running")
    try:
        while True:
            prox = vcnl.proximity
            #VCNL variables for future usage
            a = {"proximity": prox,"Slot 1A": occupied1, "Slot 2B": occupied2, "Slot 3C":
occupied3, "Slot 4D": occupied4,
            "timestamp": str(int(math.ceil(seconds)))}
            b = {"proximity": prox,"Slot 1A": open1,"Slot 2B": occupied2, "Slot 3C":
occupied3, "Slot 4D": occupied4,
            "timestamp": str(int(math.ceil(seconds)))}
            if(prox<=2500):
                #car is on spot condition
                print("\nSlot 1A is available")
                print("Proximity of Sensor 1 : %d" %prox)
                print(bool(open1))
                #TODO: Pair with an if statement to avoid updating the database when there
is no change
                db.child("ProximityData").set(b)
            elif(prox>=5000):
                #no car is around
                print("\nSlot 1A is currently occupied")
                print("Proximity of Sensor 1 : %d" %prox)
                print(bool(occupied1))
                #TODO: Pair with an if statement to avoid updating the database when there
is no change
                db.child("ProximityData").set(a)
            time.sleep(5)
    except:
        print("Something went wrong with VNCL function, exiting")

def irbFunc():
    try:
        while True:
            #entry sensor part
            entG = GPIO.input(ent_ir)
            eStat = db.child("EntryStatus").get()
            if(entG==1):
                print("Entry Gate Clear")
            if(eStat.val()["lotStatus"]==0):
                print("Lot is full, all spots are occupied!")
            if(entG==0 and (licCheck()==1)):
                print("Vehicle detected at entry")
                entGate()
                db.child("GateStatus").set(entry)
```

```
#            #exit sensor part
#            extG = GPIO.input(ext_ir)
#            if(extG==1):
#                print("Exit is clear")
#            if(extG==0):
#                print("Vehicle at exit, opening exit")
#                extGate()
            time.sleep(1)
    except:
        print("IR beam down")

if __name__ == "__main__":
    print("waking up")
    #p1 = Process(target=adminControl)
    #p2 = Process(target=vcnlFunc)
    Horcrux1 = threading.Thread(target=adminControl)
    Horcrux2 = threading.Thread(target=vcnlFunc)
    Horcrux3 = threading.Thread(target=irbFunc)
    Horcrux1.start()
    print("admin control should be active now")
    Horcrux2.start()
    print("VCNL should up now")
    Horcrux3.start()
    print("all should be activated")
```

After the code files have been tested individually with each sensor, both the VCNL4010 and the IR Break Beam files have to be further tested by connecting the sensors to the database and confirming the expected behavior. As of now, all code files have been joined into one file “**watech.py**”, after they have been stripped from any calls to the database just until proper joint functionality and PCB testing has been confirmed.
The VCNL4010 code has been simplified and made easier to read and use, and the PCA9685 code has been optimized to be used for both servos and RGB LEDs, and has been placed in the form of a separate function that is only called when needed, while the rest of the sensor code is always running in an infinite while loop. Further work will be accomplished, towards completing and testing the database connection, and

configuring each sensor to be able to say connected to the online database and push data to the Firebase interface. The VCNL4010 code will also be tweaked later on with the additions of the fixed spots for simulation purposes. As of right now, Slot 1A is the only spot on the parking lot with changing values and status. The rest of the spots will be set to a fixed value in the code, with a value of 1 denoting a “occupied” spot and must not be changed, as there is no physical hardware data being read from those 3 locations on the lot. This will mainly be done for simulation purposes along with the single VCNL4010 device on-site at the parking lot. The function for the Camera senor has also been set, with having a fixed string value for the license plate number be compared through the database for valid entry. Further code will be added to check for a match or a mismatch, depending on the value of the license plate number entered by the user through the mobile application to the database, and whether it equals to the expected fixed string value. This will be done once the database connection has been fully tested with the final PCB.

For reference this is the link to the main code for our project (as shown above):

https://github.com/VikasCENG/WatechPark/blob/master/Sensor%20Specific%20Code/watech(MP).py

Modified Code Files (Mobile Application)

ParkingLocationAdapter.java (main modification file for parking lot feature)

```
package com.example.watechpark;

import android.annotation.SuppressLint;
import android.app.Activity;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
```

```
import android.content.Context;
import android.content.Intent;
import android.content.res.ColorStateList;
import android.os.Build;
import android.os.Bundle;
import android.os.CountDownTimer;
import android.os.Handler;
import android.text.TextUtils;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;

import androidx.annotation.NonNull;
import androidx.annotation.RequiresApi;
import androidx.appcompat.app.AlertDialog;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.NotificationCompat;
import androidx.core.content.ContextCompat;
import androidx.fragment.app.Fragment;
import androidx.fragment.app.FragmentManager;
import androidx.fragment.app.FragmentTabHost;
import androidx.fragment.app.FragmentTransaction;
import androidx.recyclerview.widget.RecyclerView;

import com.google.android.gms.tasks.OnCompleteListener;
import com.google.android.gms.tasks.Task;
import com.google.android.material.snackbar.Snackbar;
import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.auth.FirebaseUser;
import com.google.firebase.database.DataSnapshot;
import com.google.firebase.database.DatabaseError;
import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;
import com.google.firebase.database.ValueEventListener;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;
import java.util.Timer;
import java.util.TimerTask;
```

```
/*
* Project: WatechPark
* Modified code file: ParkingLocationAdpater.java for adding in the parking lot status
feature(planned for Winter 2020)
* Modified by: Vikas Sharma(Student A)
* Group Members: Elias Sabbagh, George Alexandris
* Course: CENG 355
 */


public class ParkingLocationAdapter extends
RecyclerView.Adapter<ParkingLocationAdapter.ParkingLocationViewHolder> {

    private Context context;
    private List<ParkingLocation> parkingLocationList;

    private FirebaseAuth mAuth;
    private FirebaseUser user;
    FirebaseAuth.AuthStateListener mAuthListener;
    private DatabaseReference mDatabase;

    private DatabaseReference mDatabase1;

    private DatabaseReference mDatabase2;


    private FirebaseDatabase database;

    private long trueCond = 1;
    private long falseCond = 0;

    // variable to be used later on to indicate exit status
    private int status0;



    public ParkingLocationAdapter(Context context, List<ParkingLocation>
parkingLocationList) {
        this.context = context;
        this.parkingLocationList = parkingLocationList;
    }


    @NonNull
    @Override
```

```
    public ParkingLocationAdapter.ParkingLocationViewHolder
onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
      View view =
LayoutInflater.from(parent.getContext()).inflate(R.layout.fragment_home, parent, false);
      return new ParkingLocationViewHolder(view);

    }

    @Override
    public void onBindViewHolder(@NonNull final
ParkingLocationAdapter.ParkingLocationViewHolder holder, final int position) {
        // get the data for the main lot and the corresponding details from the
ParkingLocation data structure
        final ParkingLocation parkingLocation = parkingLocationList.get(position);
        holder.lotName.setText(parkingLocation.getLotName());
        holder.lotLocation.setText(parkingLocation.getLotLocation());
        holder.lotDistance.setText(String.valueOf(parkingLocation.getLotDistance() +
context.getString(R.string.m)));
        holder.lotCost.setText(context.getString(R.string.dol3) +
parkingLocation.getCost());


holder.lotImage.setImageDrawable(context.getResources().getDrawable(parkingLocatio
n.getLotImage(), null));


        // Firebase initialization
        mAuth = FirebaseAuth.getInstance();
        user = mAuth.getCurrentUser();


        // View Details button
        holder.viewDetails.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // read the data from each respective data structure in Firebase for further use
using a DatabaseReference object
                mDatabase =
FirebaseDatabase.getInstance().getReference().child("ProximityData");
                mDatabase1 = FirebaseDatabase.getInstance().getReference().child("Cars");
                mDatabase2 =
FirebaseDatabase.getInstance().getReference().child("GateStatus");
                // listen for active changes in the ProximityData structure
                mDatabase.addValueEventListener(new ValueEventListener() {
```

```
                @RequiresApi(api = Build.VERSION_CODES.N)
                @Override
                public void onDataChange(@NonNull DataSnapshot dataSnapshot) {

                    // read and store the data inside the ProximityData structure into a object
of the class
                    final ProximityData proximityData =
dataSnapshot.getValue(ProximityData.class);
                    // get the data stored in the Cars data structure and store it into a Cars
object


                    Cars cars = dataSnapshot.getValue(Cars.class);
                    // use a intent to send the name of the chosen lot from the main menu to
the "View Details" pop-up(data screen)
                    Intent intent = new Intent();
                    intent.putExtra(context.getString(R.string.put_name),
parkingLocationList.get(position).getLotName());
                    // retrieve the name of the lot and store it into a String variable
                    String lotName =
intent.getStringExtra(context.getString(R.string.getname));

                    //String proximity1 = ds.getValue(ProximityData.class).getProximity();

                    // create a new AlertDialog box
                    final AlertDialog.Builder dialog = new AlertDialog.Builder(context);
                    // inflate the view for the layout from the res folder
                    final View view =
LayoutInflater.from(context).inflate(R.layout.activity_lot_detail, null);
                    TextView text = view.findViewById(R.id.textView36);
                    // set the textview to the lot name using the lotName variable from the
previous step
                    text.setText(lotName);
                    TextView status = view.findViewById(R.id.textView37);
                    final EditText adminPass = view.findViewById(R.id.editText16);
                    // get the value entered by the user in the EditText field, store into a
string variable
                    final String plate = adminPass.getText().toString().trim();
                    //adminPass.setText("admin")

                    Button adminBtn = view.findViewById(R.id.button3);
                    // open gate button
                    adminBtn.setOnClickListener(new View.OnClickListener() {
                        @Override
                        public void onClick(View v) {
```

```
                    // check if what is entered by the user equals "admin"
                    if (adminPass.getText().toString().equals("admin")) {

                        // get a new object of the AdminControl class
                        AdminControl status = new AdminControl();
                        // if it equals/matches the "admin" string value then present a
success messgae and open the gate
                        Toast.makeText(context, "Admin Access Activated: Gate is
OPEN", Toast.LENGTH_SHORT).show();

                        // get a instance or reference of the AdminControl class and push
the value of 1(open gate) to the AdminControl structure

FirebaseDatabase.getInstance().getReference("AdminControl").child("adminStatus").set
Value(status.getAdminEntry());
                        ImageView adminEntry = view.findViewById(R.id.imageEntry);
                        // set the color of the entry arrow to blue(indicate the gate is
opened through admin access)

adminEntry.setColorFilter(context.getResources().getColor(R.color.colorBlue));
                    } else {
                        AdminControl status = new AdminControl();
                        // else don't allow access and present a toast for unsuccessful
entry
                        Toast.makeText(context, "Unauthorized Admin Access! Invalid
Credentials", Toast.LENGTH_SHORT).show();
                        // set the value as false and push a 0 to the AdminControl
structure in the adminStatus child

FirebaseDatabase.getInstance().getReference("AdminControl").child("adminStatus").set
Value(status.getAdminEntry0());
                        ImageView adminEntry = view.findViewById(R.id.imageEntry);
                        // set the color to red for the entry  arrow to indicate entry is not
allowed

adminEntry.setColorFilter(context.getResources().getColor(R.color.colorRed));

                    }


                }
            });


            TextView prox = view.findViewById(R.id.textView42);
            //get the proximity from the ProximityData structure and set the TextView
```

```
                prox.setText("RT Proximity: " +
String.valueOf(proximityData.getProximity()));

                TextView s1 = view.findViewById(R.id.textSlot1);
                String slot1 = String.valueOf(proximityData.getSlot1A());
                TextView s2 = view.findViewById(R.id.textSlot2);
                String slot2 = String.valueOf(proximityData.getSlot2B());
                TextView s3 = view.findViewById(R.id.textSlot3);
                String slot3 = String.valueOf(proximityData.getSlot3C());
                TextView s4 = view.findViewById(R.id.textSlot4);
                String slot4 = String.valueOf(proximityData.getSlot4D());


                // reccomended way by Android Studio to basically get the long value of
the proximity
                int value = 0;
                if (android.os.Build.VERSION.SDK_INT >=
android.os.Build.VERSION_CODES.N) {
                    value = Math.toIntExact(proximityData.getProximity());
                }

                // if the proximity value is less than or equal to 2500
                if (value <= 2500) {
                    ImageView en = view.findViewById(R.id.imageEntry);
                    // set the color of the entry arrow to green

en.setColorFilter(context.getResources().getColor(R.color.colorGreen));
                    ImageView spot1 = view.findViewById(R.id.imageSpot1);
                    spot1.setBackgroundResource(R.color.colorGreen);
                    ImageView spot2 = view.findViewById(R.id.imageSpot2);
                    spot2.setBackgroundResource(R.color.colorRed);
                    ImageView spot3 = view.findViewById(R.id.imageSpot3);
                    spot3.setBackgroundResource(R.color.colorRed);
                    ImageView spot4 = view.findViewById(R.id.imageSpot4);
                    spot4.setBackgroundResource(R.color.colorRed);

                    // set the text to each corresponding status
                    s1.setText("Open");
                    s2.setText("Occupied");
                    s3.setText("Occupied");
                    s4.setText("Occupied");
                    // update the overall status of the lot
                    status.setText(context.getString(R.string.setstatus2) + " Slot 1A is
available -- (1/4) ENTRY IS ALLOWED!");
                    //Toast.makeText(context, "LOT HC: ENTRY IS ALLOWED!",
Toast.LENGTH_SHORT).show();
```

```
                spot1.setOnClickListener(new View.OnClickListener() {
                    @Override
                    public void onClick(View v) {
                        Snackbar.make(view, "Do you want to reserve this slot: Slot
1A?", Snackbar.LENGTH_LONG)
                                .setAction("CANCEL", new View.OnClickListener() { // leave
the reservation
                                    @Override
                                    public void onClick(View v) {


                                    }
                                }).setAction("RESERVE", new View.OnClickListener() {
                            @Override
                            public void onClick(View v) {

                                final String name = parkingLocation.getLotName();
                                final String location = parkingLocation.getLotLocation();
                                final double distance =
Double.parseDouble(String.valueOf(parkingLocation.getLotDistance()));
                                final double cost =
Double.parseDouble(String.valueOf(parkingLocation.getCost()));
                                final int image =
Integer.parseInt(String.valueOf(parkingLocation.getLotImage()));

                                mDatabase =
FirebaseDatabase.getInstance().getReference("ParkingLocations");

                                ParkingLocation parkingLocation1 = new
ParkingLocation(name,location,distance,cost,image);

//mDatabase.child(user.getUid()).setValue(parkingLocation1);

mDatabase.child(FirebaseAuth.getInstance().getCurrentUser().getUid())

.setValue(parkingLocation1).addOnCompleteListener(new
OnCompleteListener<Void>() {

                                    @Override
                                    public void onComplete(@NonNull Task<Void> task) {

                                        if (task.isSuccessful()) {
                                            Toast.makeText(context, R.string.success_notif,
Toast.LENGTH_SHORT).show();
```

```
                                    Notification notification = new
Notification.Builder(context)
                                            .setTicker(context.getString(R.string.not))

.setContentTitle(context.getString(R.string.reserved_aspot) + name)

.setContentText(context.getString(R.string.check_pass))
                                            .setSmallIcon(R.drawable.logo)
                                            .build();

                                    notification.flags =
Notification.FLAG_AUTO_CANCEL;
                                    NotificationManager notificationManager =
(NotificationManager)context.getSystemService(Context.NOTIFICATION_SERVICE);
                                    notificationManager.notify(0, notification);
                                } else {
                                    Toast.makeText(context, R.string.data_err,
Toast.LENGTH_SHORT).show();
                                }
                            }

                        });


                        AdminControl status = new AdminControl();
                        TextView s1 = view.findViewById(R.id.textSlot1);
                        // set the status of Slot 1A as occupied
                        s1.setText("Occupied");
                        // update the visual spots and arrow colors
                        ImageView spot1 = view.findViewById(R.id.imageSpot1);
                        spot1.setBackgroundResource(R.color.colorRed);
                        ImageView en = view.findViewById(R.id.imageEntry);

en.setColorFilter(context.getResources().getColor(R.color.colorRed));
                        ImageView ex2 = view.findViewById(R.id.imageExit);

ex2.setColorFilter(context.getResources().getColor(R.color.colorGreen));
                        TextView status2 = view.findViewById(R.id.textView37);
                        // update the overall status to indiate all spots are taken
                        status2.setText(context.getString(R.string.setstatus2) + " All
slots are occupied! -- (4/4)");
                        // push a 0 to the AdminControl data strcuture

FirebaseDatabase.getInstance().getReference("AdminControl").child("adminStatus").set
Value(status.getAdminEntry0());
```

```
                        }
                    }).show();

                }
            });
            // check if proximity value is greater than 5000
        } else if (value > 5000) {
            // update the slot color changes
            ImageView en2 = view.findViewById(R.id.imageEntry);

en2.setColorFilter(context.getResources().getColor(R.color.colorRed));
            ImageView exit = view.findViewById(R.id.imageExit);

exit.setColorFilter(context.getResources().getColor(R.color.colorGreen));
            ImageView spot1 = view.findViewById(R.id.imageSpot1);
            spot1.setBackgroundResource(R.color.colorRed);
            // Slot 2B,3C,4D will remain occupied(with color red as the
background)
            ImageView spot2 = view.findViewById(R.id.imageSpot2);
            spot2.setBackgroundResource(R.color.colorRed);
            ImageView spot3 = view.findViewById(R.id.imageSpot3);
            spot3.setBackgroundResource(R.color.colorRed);
            ImageView spot4 = view.findViewById(R.id.imageSpot4);
            spot4.setBackgroundResource(R.color.colorRed);
            s1.setText("Occupied");
            s2.setText("Occupied");
            s3.setText("Occupied");
            s4.setText("Occupied");
            // if the user clicks on Slot1A when all spots are occupied show a toast
message
            spot1.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    Toast.makeText(context," All slots are occupied currently! Please
wait a while for a opening...",Toast.LENGTH_SHORT).show();
                }
            });
            // update the overall status
            status.setText(context.getString(R.string.setstatus2) + " All slots are
occupied! -- (4/4) ENTRY IS NOT ALLOWED!");
            //Toast.makeText(context, "LOT HC: ENTRY IS NOT ALLOWED! LOT
IS FULL", Toast.LENGTH_SHORT).show();

            // initialize a new EntryStatus object
            EntryStatus stat = new EntryStatus();
```

```
                    // get the status and store into a variablee
                    status0 = stat.getEntry();

                    // set this value to the EntryStatus structure for the lotStatus

FirebaseDatabase.getInstance().getReference("EntryStatus").child("lotStatus")
                            .setValue(status0);
                    // check if proximity value is in between the range of open or occupied
                } else if (value > 2500 && value < 5000) {
                    ImageView en3 = view.findViewById(R.id.imageEntry);

en3.setColorFilter(context.getResources().getColor(R.color.colorGreen));
                    // have the Slot 1A change backgrund color to blue
                    ImageView spot1 = view.findViewById(R.id.imageSpot1);
                    spot1.setBackgroundResource(R.color.colorBlue);
                    // update the status
                    status.setText(context.getString(R.string.setstatus2) + " Vehicle is
approaching the parking space");
                    // set the TextView of Slot1A to targeted(to indicate vehicle is
approaching to Slot1A)
                    s1.setText("Targeted");
                    s2.setText("Occupied");
                    s3.setText("Occupied");
                    s4.setText("Occupied");


                }

                // set the view of the dialog to the layout

                dialog.setView(view);


                // show the dialog box
                AlertDialog alertDialog = dialog.create();
                alertDialog.show();


            }


            @Override
            public void onCancelled(@NonNull DatabaseError databaseError) {


            }
```

```
            });

            // read active changes occuring at the GateStatus data structure(of the IR
Break Beam Sensor)
            mDatabase2.addValueEventListener(new ValueEventListener() {
                @Override
                public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
                    long gate = dataSnapshot.child("Status").getValue(Long.class);
                    GateStatus g = dataSnapshot.getValue(GateStatus.class);
                    final View view =
LayoutInflater.from(context).inflate(R.layout.activity_lot_detail, null);
                    final TextView status = view.findViewById(R.id.textView37);
                    // check if value is equal to 1
                    if (gate == 1) {
                        final Long time = System.currentTimeMillis() / 1000;
                        final String newTime = time.toString();
                       // status.setText(context.getString(R.string.setstatus2) + "License Plate
# verification completed!");
                        // retrieve the status of entry and the timestamp from the GateStatus
data structure
                        Toast.makeText(context, "HC LOT: 1A5-E8K verified! Car has
ENTERED the lot! Time: " + convertTimestamp(g.getTimestamp()),
Toast.LENGTH_SHORT).show();
                    }




                }

                @Override
                public void onCancelled(@NonNull DatabaseError databaseError) {

            }
            });
        }
    });

    // if user clicks on anywhere on the Humber College(Parking Lot) CardView in the
main meenu
    holder.itemView.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
```

```
        // read the data from each respective data structure in Firebase for further use
using a DatabaseReference object
        mDatabase =
FirebaseDatabase.getInstance().getReference().child("ProximityData");
        mDatabase1 = FirebaseDatabase.getInstance().getReference().child("Cars");
        mDatabase2 =
FirebaseDatabase.getInstance().getReference().child("GateStatus");
        // listen for active changes in the ProximityData structure
        mDatabase.addValueEventListener(new ValueEventListener() {


            @RequiresApi(api = Build.VERSION_CODES.N)
            @Override
            public void onDataChange(@NonNull DataSnapshot dataSnapshot) {

                // read and store the data inside the ProximityData structure into a object
of the class
                final ProximityData proximityData =
dataSnapshot.getValue(ProximityData.class);
                // get the data stored in the Cars data structure and store it into a Cars
object


                Cars cars = dataSnapshot.getValue(Cars.class);
                // use a intent to send the name of the chosen lot from the main menu to
the "View Details" pop-up(data screen)
                Intent intent = new Intent();
                intent.putExtra(context.getString(R.string.put_name),
parkingLocationList.get(position).getLotName());
                // retrieve the name of the lot and store it into a String variable
                String lotName =
intent.getStringExtra(context.getString(R.string.getname));

                //String proximity1 = ds.getValue(ProximityData.class).getProximity();

                // create a new AlertDialog box
                final AlertDialog.Builder dialog = new AlertDialog.Builder(context);
                // inflate the view for the layout from the res folder
                final View view =
LayoutInflater.from(context).inflate(R.layout.activity_lot_detail, null);
                TextView text = view.findViewById(R.id.textView36);
                // set the textview to the lot name using the lotName variable from the
previous step
                text.setText(lotName);
                TextView status = view.findViewById(R.id.textView37);
                final EditText adminPass = view.findViewById(R.id.editText16);
```

```
                // get the value entered by the user in the EditText field, store into a
string variable
                final String plate = adminPass.getText().toString().trim();
                //adminPass.setText("admin")

                Button adminBtn = view.findViewById(R.id.button3);
                // open gate button
                adminBtn.setOnClickListener(new View.OnClickListener() {
                    @Override
                    public void onClick(View v) {

                        // check if what is entered by the user equals "admin"
                        if (adminPass.getText().toString().equals("admin")) {

                            // get a new object of the AdminControl class
                            AdminControl status = new AdminControl();
                            // if it equals/matches the "admin" string value then present a
success messgae and open the gate
                            Toast.makeText(context, "Admin Access Activated: Gate is
OPEN", Toast.LENGTH_SHORT).show();

                            // get a instance or reference of the AdminControl class and push
the value of 1(open gate) to the AdminControl structure

FirebaseDatabase.getInstance().getReference("AdminControl").child("adminStatus").set
Value(status.getAdminEntry());
                            ImageView adminEntry = view.findViewById(R.id.imageEntry);
                            // set the color of the entry arrow to blue(indicate the gate is
opened through admin access)

adminEntry.setColorFilter(context.getResources().getColor(R.color.colorBlue));
                        } else {
                            AdminControl status = new AdminControl();
                            // else don't allow access and present a toast for unsuccessful
entry
                            Toast.makeText(context, "Unauthorized Admin Access! Invalid
Credentials", Toast.LENGTH_SHORT).show();
                            // set the value as false and push a 0 to the AdminControl
structure in the adminStatus child

FirebaseDatabase.getInstance().getReference("AdminControl").child("adminStatus").set
Value(status.getAdminEntry0());
                            ImageView adminEntry = view.findViewById(R.id.imageEntry);
                            // set the color to red for the entry  arrow to indicate entry is not
allowed
```

```
adminEntry.setColorFilter(context.getResources().getColor(R.color.colorRed));

                        }


                    }
                });


                TextView prox = view.findViewById(R.id.textView42);
                //get the proximity from the ProximityData structure and set the TextView
                prox.setText("RT Proximity: " +
String.valueOf(proximityData.getProximity()));

                TextView s1 = view.findViewById(R.id.textSlot1);
                String slot1 = String.valueOf(proximityData.getSlot1A());
                TextView s2 = view.findViewById(R.id.textSlot2);
                String slot2 = String.valueOf(proximityData.getSlot2B());
                TextView s3 = view.findViewById(R.id.textSlot3);
                String slot3 = String.valueOf(proximityData.getSlot3C());
                TextView s4 = view.findViewById(R.id.textSlot4);
                String slot4 = String.valueOf(proximityData.getSlot4D());


                // reccomended way by Android Studio to basically get the long value of
the proximity
                int value = 0;
                if (android.os.Build.VERSION.SDK_INT >=
android.os.Build.VERSION_CODES.N) {
                    value = Math.toIntExact(proximityData.getProximity());
                }

                // if the proximity value is less than or equal to 2500
                if (value <= 2500) {
                    ImageView en = view.findViewById(R.id.imageEntry);
                    // set the color of the entry arrow to green

en.setColorFilter(context.getResources().getColor(R.color.colorGreen));
                    ImageView spot1 = view.findViewById(R.id.imageSpot1);
                    spot1.setBackgroundResource(R.color.colorGreen);
                    ImageView spot2 = view.findViewById(R.id.imageSpot2);
                    spot2.setBackgroundResource(R.color.colorRed);
                    ImageView spot3 = view.findViewById(R.id.imageSpot3);
                    spot3.setBackgroundResource(R.color.colorRed);
                    ImageView spot4 = view.findViewById(R.id.imageSpot4);
```

```
                    spot4.setBackgroundResource(R.color.colorRed);

                    // set the text to each corresponding status
                    s1.setText("Open");
                    s2.setText("Occupied");
                    s3.setText("Occupied");
                    s4.setText("Occupied");
                    // update the overall status of the lot
                    status.setText(context.getString(R.string.setstatus2) + " Slot 1A is
available -- (1/4) ENTRY IS ALLOWED!");
                    //Toast.makeText(context, "LOT HC: ENTRY IS ALLOWED!",
Toast.LENGTH_SHORT).show();

                    spot1.setOnClickListener(new View.OnClickListener() {
                       @Override
                       public void onClick(View v) {
                          Snackbar.make(view, "Do you want to reserve this slot: Slot
1A?", Snackbar.LENGTH_LONG)
                                  .setAction("CANCEL", new View.OnClickListener() { // leave
the reservation
                                     @Override
                                     public void onClick(View v) {


                                     }
                                  }).setAction("RESERVE", new View.OnClickListener() {
                              @Override
                              public void onClick(View v) {


                                  mDatabase =
FirebaseDatabase.getInstance().getReference("ParkingLocations");

                                  final String name = parkingLocation.getLotName();
                                  final String location = parkingLocation.getLotLocation();
                                  final double distance =
Double.parseDouble(String.valueOf(parkingLocation.getLotDistance()));
                                  final double cost =
Double.parseDouble(String.valueOf(parkingLocation.getCost()));
                                  final int image =
Integer.parseInt(String.valueOf(parkingLocation.getLotImage()));

                                  ParkingLocation parkingLocation1 = new
ParkingLocation(name, location, distance, cost, image);
                                  mDatabase.child(user.getUid()).setValue(parkingLocation1);
```

```
mDatabase.child("ParkingLocations").child(FirebaseAuth.getInstance().getCurrentUser(
).getUid())

.setValue(parkingLocation1).addOnCompleteListener(new
OnCompleteListener<Void>() {
                                    @Override
                                    public void onComplete(@NonNull Task<Void> task) {
                                        if (task.isSuccessful()) {
                                            Toast.makeText(context, R.string.success_notif,
Toast.LENGTH_SHORT).show();



                                            // send a notification to the user if the lot information
is sent successfully
                                            Notification notification = new
Notification.Builder(context)


                                                    .setTicker(context.getString(R.string.not))

.setContentTitle(context.getString(R.string.reserved_aspot) + name)
                                                    .setContentText("Slot 1A booked! Please
proceed to Payment after selecting your Parking Pass...")
                                                    .setSmallIcon(R.drawable.logo)
                                                    .build();


                                            notification.flags =
Notification.FLAG_AUTO_CANCEL;
                                            NotificationManager notificationManager =
(NotificationManager) context.getSystemService(Context.NOTIFICATION_SERVICE);
                                            notificationManager.notify(0, notification);


                                        } else {
                                            Toast.makeText(context, R.string.data_err,
Toast.LENGTH_SHORT).show();
                                        }
                                    }
                                });

                                AdminControl status = new AdminControl();
                                TextView s1 = view.findViewById(R.id.textSlot1);
                                // set the status of Slot 1A as occupied
```

```
                                s1.setText("Occupied");
                                // update the visual spots and arrow colors
                                ImageView spot1 = view.findViewById(R.id.imageSpot1);
                                spot1.setBackgroundResource(R.color.colorRed);
                                ImageView en = view.findViewById(R.id.imageEntry);

en.setColorFilter(context.getResources().getColor(R.color.colorRed));
                                ImageView ex2 = view.findViewById(R.id.imageExit);

ex2.setColorFilter(context.getResources().getColor(R.color.colorGreen));
                                TextView status2 = view.findViewById(R.id.textView37);
                                // update the overall status to indiate all spots are taken
                                status2.setText(context.getString(R.string.setstatus2) + " All
slots are occupied! -- (4/4)");
                                // push a 0 to the AdminControl data strcuture

FirebaseDatabase.getInstance().getReference("AdminControl").child("adminStatus").set
Value(status.getAdminEntry0());


                            }
                        }).show();

                    }
                });
                // check if proximity value is greater than 5000
            } else if (value > 5000) {
                // update the slot color changes
                ImageView en2 = view.findViewById(R.id.imageEntry);

en2.setColorFilter(context.getResources().getColor(R.color.colorRed));
                ImageView ex5 = view.findViewById(R.id.imageExit);

ex5.setColorFilter(context.getResources().getColor(R.color.colorGreen));
                ImageView spot1 = view.findViewById(R.id.imageSpot1);
                spot1.setBackgroundResource(R.color.colorRed);
                // Slot 2B,3C,4D will remain occupied(with color red as the
background)
                ImageView spot2 = view.findViewById(R.id.imageSpot2);
                spot2.setBackgroundResource(R.color.colorRed);
                ImageView spot3 = view.findViewById(R.id.imageSpot3);
                spot3.setBackgroundResource(R.color.colorRed);
                ImageView spot4 = view.findViewById(R.id.imageSpot4);
                spot4.setBackgroundResource(R.color.colorRed);
                s1.setText("Occupied");
                s2.setText("Occupied");
```

```
                    s3.setText("Occupied");
                    s4.setText("Occupied");
                    // if the user clicks on Slot1A when all spots are occupied show a toast
message
                    spot1.setOnClickListener(new View.OnClickListener() {
                        @Override
                        public void onClick(View v) {
                            Toast.makeText(context," All slots are occupied currently! Please
wait a while for a opening...",Toast.LENGTH_SHORT).show();
                        }
                    });
                    // update the overall status
                    status.setText(context.getString(R.string.setstatus2) + " All slots are
occupied! -- (4/4) ENTRY IS NOT ALLOWED!");
                    //Toast.makeText(context, "LOT HC: ENTRY IS NOT ALLOWED! LOT
IS FULL", Toast.LENGTH_SHORT).show();

                    // initialize a new EntryStatus object
                    EntryStatus stat = new EntryStatus();
                    // get the status and store into a variablee
                    status0 = stat.getEntry();

                    // set this value to the EntryStatus structure for the lotStatus

FirebaseDatabase.getInstance().getReference("EntryStatus").child("lotStatus")
                            .setValue(status0);
                    // check if proximity value is in between the range of open or occupied
                } else if (value > 2500 && value < 5000) {
                    ImageView en3 = view.findViewById(R.id.imageEntry);

en3.setColorFilter(context.getResources().getColor(R.color.colorGreen));
                    // have the Slot 1A change backgrund color to blue
                    ImageView spot1 = view.findViewById(R.id.imageSpot1);
                    spot1.setBackgroundResource(R.color.colorBlue);
                    // update the status
                    status.setText(context.getString(R.string.setstatus2) + " Vehicle is
approaching the parking space");
                    // set the TextView of Slot1A to targeted(to indicate vehicle is
approaching to Slot1A)
                    s1.setText("Targeted");
                    s2.setText("Occupied");
                    s3.setText("Occupied");
                    s4.setText("Occupied");


                }
```

```
                // set the view of the dialog to the layout

                dialog.setView(view);


                // show the dialog box
                AlertDialog alertDialog = dialog.create();
                alertDialog.show();


            }


            @Override
            public void onCancelled(@NonNull DatabaseError databaseError) {


            }
        });

        // read active changes occuring at the GateStatus data structure(of the IR
Break Beam Sensor)
        mDatabase2.addValueEventListener(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
                long gate = dataSnapshot.child("Status").getValue(Long.class);
                GateStatus g = dataSnapshot.getValue(GateStatus.class);
                final View view =
LayoutInflater.from(context).inflate(R.layout.activity_lot_detail, null);
                final TextView status = view.findViewById(R.id.textView37);
                // check if value is equal to 1
                if (gate == 1) {
                    final Long time = System.currentTimeMillis() / 1000;
                    final String newTime = time.toString();
                    // status.setText(context.getString(R.string.setstatus2) + "License
Plate # verification completed!");
                    // retrieve the status of entry and the timestamp from the GateStatus
data structure
                    Toast.makeText(context, "HC LOT: 1A5-E8K verified! Car has
ENTERED the lot! Time: " + convertTimestamp(g.getTimestamp()),
Toast.LENGTH_SHORT).show();
                }
```

```
            }

            @Override
            public void onCancelled(@NonNull DatabaseError databaseError) {

            }
        });

    }
});

holder.reserveButton.setOnClickListener(new View.OnClickListener() {

    @Override

    public void onClick(View v) {

        final String name = parkingLocation.getLotName();
        final String location = parkingLocation.getLotLocation();
        final double distance =
Double.parseDouble(String.valueOf(parkingLocation.getLotDistance()));
        final double cost =
Double.parseDouble(String.valueOf(parkingLocation.getCost()));
        final int image =
Integer.parseInt(String.valueOf(parkingLocation.getLotImage()));

        mDatabase =
FirebaseDatabase.getInstance().getReference("ParkingLocations");

        ParkingLocation parkingLocation1 = new
ParkingLocation(name,location,distance,cost,image);
        //mDatabase.child(user.getUid()).setValue(parkingLocation1);
        mDatabase.child(FirebaseAuth.getInstance().getCurrentUser().getUid())
                .setValue(parkingLocation1).addOnCompleteListener(new
OnCompleteListener<Void>() {

            @Override
            public void onComplete(@NonNull Task<Void> task) {

                if (task.isSuccessful()) {
                    Toast.makeText(context, R.string.success_notif,
Toast.LENGTH_SHORT).show();
                    Notification notification = new Notification.Builder(context)
                            .setTicker(context.getString(R.string.not))
                            .setContentTitle(context.getString(R.string.reserved_aspot) +
name)
```

```
                        .setContentText(context.getString(R.string.check_pass))
                        .setSmallIcon(R.drawable.logo)
                        .build();

                    notification.flags = Notification.FLAG_AUTO_CANCEL;
                    NotificationManager notificationManager =
(NotificationManager)context.getSystemService(Context.NOTIFICATION_SERVICE);
                    notificationManager.notify(0, notification);
                } else {
                    Toast.makeText(context, R.string.data_err,
Toast.LENGTH_SHORT).show();
                }
            }

        });

    }

});

}

// return the size of the array list for the parking location displayed on the main menu
 @Override
 public int getItemCount() {
    return parkingLocationList.size();
 }

 // convert the long format of the timestamp into a readable string form
 private String convertTimestamp(String timestamp){
    long yourSeconds = Long.valueOf(timestamp);
    Date mDate = new Date(yourSeconds * 1000);
    DateFormat df = new SimpleDateFormat(context.getString(R.string.date_for));
    return df.format(mDate);

 }


 public class ParkingLocationViewHolder extends RecyclerView.ViewHolder {

    ImageView lotImage;
    TextView lotName, lotLocation, lotDistance, lotCost;
    Button viewDetails, reserveButton,openGate;

    TextView proximity, timestamp;
```

```
    public ParkingLocationViewHolder(@NonNull View itemView) {
        super(itemView);

        lotImage = itemView.findViewById(R.id.image_card);
        lotName = itemView.findViewById(R.id.text_locname);
        lotLocation = itemView.findViewById(R.id.textView39);
        lotDistance = itemView.findViewById(R.id.text_dist);
        lotCost = itemView.findViewById(R.id.textView40);
        viewDetails = itemView.findViewById(R.id.button11);
        reserveButton = itemView.findViewById(R.id.button10);
        openGate = itemView.findViewById(R.id.button3);


}
}
}
```

For reference this is the link to the modified ParkingLocationAdapter.java code for our

project (as shown above):

https://github.com/VikasCENG/WatechPark/blob/master/WatechPark(APP)/app/src/main/java/com/example/watechpark/ParkingLocationAdapter.java

MainMenu.java

```
package com.example.watechpark;

import android.content.DialogInterface;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;

import com.bumptech.glide.Glide;
import com.bumptech.glide.request.RequestOptions;
import com.example.watechpark.ui.AddACarFragment;
import com.example.watechpark.ui.Home.HomeFragment;
import com.example.watechpark.ui.Settings.PaymentFragment;
import com.google.android.gms.auth.api.signin.GoogleSignIn;
import com.google.android.gms.auth.api.signin.GoogleSignInAccount;
import com.google.android.gms.tasks.OnSuccessListener;
import com.google.android.material.bottomappbar.BottomAppBar;
import com.google.android.material.bottomnavigation.BottomNavigationView;
import com.google.android.material.floatingactionbutton.FloatingActionButton;
```

```
import com.google.android.material.snackbar.Snackbar;

import android.view.MenuItem;
import android.view.View;

import androidx.annotation.NonNull;
import androidx.appcompat.app.AlertDialog;
import androidx.appcompat.widget.SearchView;
import androidx.fragment.app.Fragment;
import androidx.fragment.app.FragmentManager;
import androidx.fragment.app.FragmentTransaction;
import androidx.navigation.NavController;
import androidx.navigation.Navigation;
import androidx.navigation.ui.AppBarConfiguration;
import androidx.navigation.ui.NavigationUI;

import com.google.android.material.navigation.NavigationView;
import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.auth.FirebaseUser;
import com.google.firebase.storage.FirebaseStorage;
import com.google.firebase.storage.StorageReference;
import com.squareup.picasso.Picasso;

import androidx.drawerlayout.widget.DrawerLayout;

import androidx.appcompat.app.AppCompatActivity;
import androidx.appcompat.widget.Toolbar;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

import android.view.Menu;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ImageButton;
import android.widget.ImageView;
import android.widget.TextView;
import android.widget.Toast;

import java.util.ArrayList;
import java.util.List;

import de.hdodenhof.circleimageview.CircleImageView;

public class MainMenu extends AppCompatActivity {
```

```
    private AppBarConfiguration mAppBarConfiguration;

    RecyclerView recyclerView;
    ParkingLocationAdapter adapter;
    List<ParkingLocation> parkingList;


    private ImageView i1,i2,i3;
    private Button b1,b2,b3;
    private TextView t1,t2,t3,t4,t5,t6,t7,t8,t9,t10, t11, userName, userEmail;

     FirebaseAuth mAuth;
     FirebaseUser user;

    private FirebaseStorage firebaseStorage;

    protected Button addACar;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_menu);

        parkingList = new ArrayList<>();

        recyclerView = findViewById(R.id.recyclerView2);
        recyclerView.setHasFixedSize(true);

        recyclerView.setLayoutManager(new LinearLayoutManager(this));

        parkingList.add(new ParkingLocation(getString(R.string.queens_parkway),
getString(R.string.loc0), 1.6, 6.50 , R.drawable.queens));
        parkingList.add(new ParkingLocation(getString(R.string.humber),
getString(R.string.loc1), 0.2, 8.50 , R.drawable.humber2));



        adapter = new ParkingLocationAdapter(this, parkingList);
        recyclerView.setAdapter(adapter);

       // userName = findViewById(R.id.nav_user);
       // userEmail = findViewById(R.id.nav_email);

        Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
```

```
        mAuth = FirebaseAuth.getInstance();
        user = mAuth.getCurrentUser();

        firebaseStorage = FirebaseStorage.getInstance();
        StorageReference storageReference = firebaseStorage.getReference();


        getSupportActionBar().setTitle(getString(R.string.home_title));


        DrawerLayout drawer = findViewById(R.id.drawer_layout);
        NavigationView navigationView = findViewById(R.id.nav_view);




        // Passing each menu ID as a set of Ids because each
        // menu should be considered as top level destinations.
        mAppBarConfiguration = new AppBarConfiguration.Builder(
                R.id.nav_home, R.id.nav_passes, R.id.nav_orderhistory, R.id.nav_addacar,
R.id.nav_manage,
                R.id.nav_payment, R.id.nav_settings, R.id.nav_help, R.id.nav_about)
                .setDrawerLayout(drawer)
                .build();
        NavController navController = Navigation.findNavController(this,
R.id.nav_host_fragment);
        NavigationUI.setupActionBarWithNavController(this, navController,
mAppBarConfiguration);
        NavigationUI.setupWithNavController(navigationView, navController);




        displayHeaderInfo();
    }


    @Override
    public boolean onOptionsItemSelected(@NonNull MenuItem item) {

        int id = item.getItemId();
        if(id == R.id.action_acc){
            Intent i = new Intent(getApplicationContext(), AccountManagement.class);
            startActivity(i);

        }else if(id == R.id.action_logout){
```

```
            logOutUser();
        }

        return super.onOptionsItemSelected(item);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main_menu, menu);
        return true;
    }



    @Override
    public boolean onSupportNavigateUp() {
        NavController navController = Navigation.findNavController(this,
R.id.nav_host_fragment);
        return NavigationUI.navigateUp(navController, mAppBarConfiguration)
                || super.onSupportNavigateUp();
    }

    private void logOutUser(){
        AlertDialog.Builder builder = new AlertDialog.Builder(this);
        builder.setTitle(getString(R.string.confirm_out));
        builder.setMessage(getString(R.string.log_outconf));
        builder.setPositiveButton(getString(R.string.y), new
DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                mAuth.signOut();
                startActivity(new Intent(MainMenu.this, LoginRegister.class));
                dialog.dismiss();
            }
        });

        builder.setNegativeButton(getString(R.string.n), new
DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                dialog.cancel();
            }
        });
        AlertDialog alert = builder.create();
        alert.show();
```

```
    }

    private void displayHeaderInfo() {
        NavigationView navigationView = findViewById(R.id.nav_view);
        View headerView = navigationView.getHeaderView(0);
        final TextView userName = headerView.findViewById(R.id.nav_user);
        final TextView userEmail = headerView.findViewById(R.id.nav_email);
        final CircleImageView navImage =
headerView.findViewById(R.id.imageHeader);

        final ParkingPassInfo parkingPassInfo = new ParkingPassInfo();

        //userName.setText(user.getDisplayName());
        //userEmail.setText((user.getEmail()));

        //Picasso.get().load(user.getPhotoUrl()).fit().centerCrop().into(navImage);

//Glide.with(this).load(user.getPhotoUrl()).apply(RequestOptions.circleCropTransform()).
into(navImage);

        firebaseStorage = FirebaseStorage.getInstance();
        StorageReference storageReference = firebaseStorage.getReference();

storageReference.child(user.getUid()).child(getString(R.string.prof_2)).getDownloadUrl(
).addOnSuccessListener(new OnSuccessListener<Uri>() {
            @Override
            public void onSuccess(Uri uri) {
                Picasso.get().load(uri).fit().centerCrop().into(navImage);

                userEmail.setText(user.getEmail());
                userName.setText(user.getUid());
            }
        });


//Glide.with(this).applyDefaultRequestOptions(RequestOptions.circleCropTransform()).a
sBitmap().load(user.getPhotoUrl()).into(navImage);
    }


}
```

For reference this is the link to the modified MainMenu.java code for our project (as

shown above):

https://github.com/VikasCENG/WatechPark/blob/master/WatechPark(APP)/app/src/main/java/com/example/watechpark/MainMenu.java

ParkingPassesFragment.java

```
package com.example.watechpark;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

import androidx.annotation.Nullable;
import androidx.annotation.NonNull;
import androidx.fragment.app.Fragment;
import androidx.fragment.app.FragmentManager;
import androidx.lifecycle.Observer;
import androidx.lifecycle.ViewModelProviders;
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

import com.example.watechpark.R;
import com.example.watechpark.ui.Home.HomeFragment;
import com.google.firebase.database.DataSnapshot;
import com.google.firebase.database.DatabaseError;
import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;
import com.google.firebase.database.ValueEventListener;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ParkingPassesFragment extends Fragment {

    private ParkingPassesViewModel parkingPassesViewModel;

    RecyclerView recyclerView;
    ParkingPassesAdapter adapter;
    List<ParkingPassInfo> parkingPassInfoList;

    private DatabaseReference databaseReference;

    public View onCreateView(@NonNull LayoutInflater inflater,
                             ViewGroup container, Bundle savedInstanceState) {
```

```
        parkingPassesViewModel =
                ViewModelProviders.of(this).get(ParkingPassesViewModel.class);
        View root = inflater.inflate(R.layout.parking_passes_fragment, container, false);


        parkingPassInfoList = new ArrayList<>();


        recyclerView = root.findViewById(R.id.recylerView3);
        recyclerView.setHasFixedSize(true);

        recyclerView.setLayoutManager(new LinearLayoutManager(getContext()));
        databaseReference =
FirebaseDatabase.getInstance().getReference("ParkingLocation");
        databaseReference.addValueEventListener(new ValueEventListener() {
            @Override
            public void onDataChange(@NonNull DataSnapshot dataSnapshot) {
                if(dataSnapshot.exists()){
                    for(DataSnapshot ds: dataSnapshot.getChildren()){
                        ParkingPassInfo p = ds.getValue(ParkingPassInfo.class);
                        parkingPassInfoList.add(p);

                        parkingPassInfoList.add(new ParkingPassInfo(getString(R.string.hc),
getString(R.string.lot_loc1), 8.50,getString(R.string.pass2),8,getString(R.string.valid2),
getString(R.string.t2), 200));



                    }

                    adapter = new ParkingPassesAdapter(getContext(), parkingPassInfoList);
                    recyclerView.setAdapter(adapter);
                }
            }

            @Override
            public void onCancelled(@NonNull DatabaseError databaseError) {


            }
        });

        return root;
    }
}
```

For reference this is the link to the modified ParkingPassesFragment.java code for our

project (as shown above):

https://github.com/VikasCENG/WatechPark/blob/master/WatechPark(APP)/app/src/main/java/com/example/watechpark/ParkingPassesFragment.java

AddACarFragment.java

```
package com.example.watechpark.ui;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.view.WindowManager;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import android.widget.Toast;

import androidx.annotation.Nullable;
import androidx.annotation.NonNull;
import androidx.fragment.app.Fragment;
import androidx.lifecycle.Observer;
import androidx.lifecycle.ViewModelProviders;

import com.example.watechpark.Cars;
import com.example.watechpark.R;
import com.google.android.gms.tasks.OnCompleteListener;
import com.google.android.gms.tasks.Task;
import com.google.firebase.auth.AuthResult;
import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.auth.FirebaseUser;
import com.google.firebase.database.DataSnapshot;
import com.google.firebase.database.DatabaseError;
import com.google.firebase.database.DatabaseReference;
import com.google.firebase.database.FirebaseDatabase;
import com.google.firebase.database.ValueEventListener;

public class AddACarFragment extends Fragment {

    private AddACarViewModel addACarViewModel;

    private FirebaseAuth mAuth;
```

```
private FirebaseUser user;
FirebaseAuth.AuthStateListener mAuthListener;
private DatabaseReference mDatabase;

private TextView label;
private EditText edit_make, edit_model, edit_number, edit_color;

Task<AuthResult> task = FirebaseAuth.getInstance().signInAnonymously();


public View onCreateView(@NonNull LayoutInflater inflater,
                         ViewGroup container, Bundle savedInstanceState) {
    addACarViewModel =
            ViewModelProviders.of(this).get(AddACarViewModel.class);
    View root = inflater.inflate(R.layout.add_acar_fragment, container, false);

    mAuth = FirebaseAuth.getInstance();
    user = mAuth.getCurrentUser();
    mDatabase = FirebaseDatabase.getInstance().getReference();


    edit_make = root.findViewById(R.id.edit_make);
    label = root.findViewById(R.id.textView32);
    edit_model = root.findViewById(R.id.edit_model);
    edit_number = root.findViewById(R.id.edit_number);
    edit_color = root.findViewById(R.id.edit_color);
    Button button = (Button) root.findViewById(R.id.button_add);


    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            addCar();
        }
    });


    return root;


}

private void addCar() {
    final String make = edit_make.getText().toString().trim();
    final String model = edit_model.getText().toString().trim();
    final String color = edit_color.getText().toString().trim();
```

```
        final String licensePlate = edit_number.getText().toString().trim();
        final Long time = System.currentTimeMillis() / 1000;
        final String timestamp = time.toString();

        if (make.isEmpty()) {
            edit_make.setError(getString(R.string.make));
            edit_make.requestFocus();
        } else if (model.isEmpty()) {
            edit_model.setError(getString(R.string.model));
            edit_model.requestFocus();
        } else if (color.isEmpty()) {
            edit_color.setError(getString(R.string.color));
            edit_color.requestFocus();
        } else if (licensePlate.isEmpty()) {
            edit_number.setError(getString(R.string.lic));
            edit_number.requestFocus();
        } else {

            Cars cars = new Cars(make, model, color, licensePlate, timestamp);
            mDatabase.child("Cars")
                    .push().setValue(cars).addOnCompleteListener(new
OnCompleteListener<Void>() {
                @Override
                public void onComplete(@NonNull Task<Void> task) {
                    if (task.isSuccessful()) {
                        Toast.makeText(getContext(), R.string.car_added,
Toast.LENGTH_SHORT).show();
                    } else {
                        Toast.makeText(getContext(), R.string.car_failed,
Toast.LENGTH_SHORT).show();
                    }
                }
            });


        }
    }
}
```

For reference this is the link to the modified AddACarFragment.java code for our project

(as shown above):

https://github.com/VikasCENG/WatechPark/blob/master/WatechPark(APP)/app/src/main/java/com/example/watechpark/ui/AddACarFragment.java

AdminControl.java (for Servo motors)

```
package com.example.watechpark;

public class AdminControl {

    private int adminEntry = 1;
    private int adminEntry0 = 0;

    public AdminControl(int adminEntry, int adminEntry0) {
        this.adminEntry = adminEntry;
        this.adminEntry0 = adminEntry0;
    }

    public AdminControl() {
    }

    public int getAdminEntry() {
        return adminEntry;
    }

    public void setAdminEntry(int adminEntry) {
        this.adminEntry = adminEntry;
    }

    public int getAdminEntry0() {
        return adminEntry0;
    }

    public void setAdminEntry0(int adminEntry0) {
        this.adminEntry0 = adminEntry0;
    }
}
```

For reference this is the link to the added AdminControl.java class for our project (as shown above):

https://github.com/VikasCENG/WatechPark/blob/master/WatechPark(APP)/app/src/main/java/com/example/watechpark/AdminControl.java

EntryStatus.java(for IR Break Beam sensor)

```
package com.example.watechpark;

public class EntryStatus {

    private int entry = 0;

    public EntryStatus(int entry) {
        this.entry = entry;
    }

    public EntryStatus() {
    }

    public int getEntry() {
        return entry;
    }

    public void setEntry(int entry) {
        this.entry = entry;
    }
```

For reference this is the link to the added EntryStatus.java class for our project (as

shown above):

https://github.com/VikasCENG/WatechPark/blob/master/WatechPark(APP)/app/src/mai

n/java/com/example/watechpark/EntryStatus.java

GateStatus.java(for IR Break Beam Sensor)

```
package com.example.watechpark;

public class GateStatus {

    private long gStatus;
    private String timestamp;

    public GateStatus(long gStatus, String timestamp) {
        this.gStatus = gStatus;
        this.timestamp = timestamp;
    }
```

```
    public GateStatus() {
    }

    public long getgStatus() {
        return gStatus;
    }

    public void setgStatus(long gStatus) {
        this.gStatus = gStatus;
    }

    public String getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }
}
}
```

For reference this is the link to the added GateStatus.java class for our project (as

shown above):

https://github.com/VikasCENG/WatechPark/blob/master/WatechPark(APP)/app/src/main/java/com/example/watechpark/EntryStatus.java

ProximityData.java (for VCNL4010 Proximity Sensor)

```
package com.example.watechpark;

public class ProximityData {

    private long proximity;
    private String timestamp;
    private long slot1A;
    private long slot2B;
    private long slot3C;
    private long slot4D;

    public ProximityData() {
    }
```

```
    public ProximityData(long proximity, String timestamp, long slot1A, long slot2B, long
slot3C, long slot4D) {
        this.proximity = proximity;
        this.timestamp = timestamp;
        this.slot1A = slot1A;
        this.slot2B = slot2B;
        this.slot3C = slot3C;
        this.slot4D = slot4D;


    }

    public long getProximity() {
        return proximity;
    }

    public void setProximity(long proximity) {
        this.proximity = proximity;
    }

    public String getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }

    public long getSlot1A() { return slot1A; }

    public void setSlot1A(long slot1A) {
        this.slot1A = slot1A;
    }

    public long getSlot2B() {
        return slot2B;
    }

    public void setSlot2B(long slot2B) {
        this.slot2B = slot2B;
    }

    public long getSlot3C() {
        return slot3C;
    }
```

```
    public void setSlot3C(long slot3C) {
        this.slot3C = slot3C;
    }

    public long getSlot4D() {
        return slot4D;
    }

    public void setSlot4D(long slot4D) {
        this.slot4D = slot4D;
    }
}
```

For reference this is the link to the modified ProximityData.java class for our project (as shown above):

https://github.com/VikasCENG/WatechPark/blob/master/WatechPark(APP)/app/src/main/java/com/example/watechpark/ProximityData.java

7.1 Firmware code

This status update will address the firmware code and integration status of the project up to date. This also includes, the financial state of the project, progress accomplished for each specific hardware device and its relating firmware/code. As well as, the modified code files in the Appendix, and then followed up with the link to the complete code to the repository on GitHub. This includes, the final code integration for each sensor/effector utilized in the project into a main central python program,

Demo

/1 Hardware present?

/3 Code runs concurrently for all sensors/effectors

/1 Project repository contains integrated code

Status

/1 Memo including updates

/1 Financial update

/1 Progress update

/1 Modified Code Files in Appendix

/1 Link to Complete Code in Repository

Memo and Updates

At this stage of the project, we have completed most of our set objectives from the initial designing, development phase of both the hardware and software aspects of our parking application. Currently, we are using this present time to support us with identifying any internal or external issues and find quick, reliable solutions to any problems we may face in the future. In terms of the firmware status for all three sensors being utilized for the project, we are pleased to announce we have successfully integrated each individual sensor into a single program to run all sensors concurrently with each other to display readings, and gather real-time data. This includes, displaying the proximity data from the VCNL4010 Proximity sensor working alongside the IR Break Beam sensor to detect the presence of a vehicle at the gates. The plan in the previous Imaging/Firmware section was to implement this firmware integration in early March, and we have successfully done that through the use of breadboard testing and extensive coding. The firmware is set at this stage for each sensor to provide its functionalities to our application, and corresponding readings to be supported with our online database. This is the next step of the project we are currently working on, to have

the mobile application work with the online database through a real-time data retrieval structure. Now, we are able to have each sensor talk to each other depending on our needs and desired outputs from the firmware side of the code, and through successful breadboard testing we can ensure we will be able to complete both the mobile application, and wireless database configuration in the upcoming milestones.

The code finalized at this stage represents all 3 sensors along with their successful integration into one main program. The database configuration has been included into the code provided in the upcoming parts, but due to unfortunate circumstances currently we are unable to have the database connection working with the final firmware code assembled. This issue is in regards, to the VCNL4010 Proximity sensor where once the code for the database was tested and a value based on the level of the proximity is sent to Firebase, due to some unexpected circumstances this caused the program to crash. The issue is being looked at this time, and will be solved with immediate care to ensure we meet all requirements for the project. Due to this reason, we decided it was best to conform with the firmware code we have successfully tested right now for all sensors/effectors, and move to the database connection once we have established and make sure that the integration can be made. This will be focused on for the upcoming week, along with further work on the mobile application will be established.

During the reading week, we used the time to work on the firmware code and complete the final code to be demoed and presented in the week of March 3rd. Although, during this time we also ran into an issue regarding our physical parking lot model prototype and our intended firmware code that was assembled for the project. We found out that the VCNL4010 Proximity sensor only supports a fixed I2C address of 0x13, which

means for the intended 4 spots on our parking lot prototype, we would be unable to have each proximity sensor stationed at each parking space to use a unique I2C address. This is because, this value is fixed and therefore cannot be changed in any manner. Due to this issue, our team had to re-assess and think critically based on the options we had available to us. Therefore, we came to a consensus to only have one spot on the parking lot be used with only one VCNL4010 Proximity sensor sending the data from the sensor to the online database, and then displaying the status changes on the mobile application. The remaining spots which include the following: Slot 2B, Slot 3C, Slot 4D will be simulated to a set value of “occupied” in the parking lot at all times. This will reduce the work needed for each spot, and the code will flow much efficiently for all sensors.

Fortunately, this will also allow us to engage into what other features may be applicable or desired to be used by our consumer demographic. We also considered a use of the TCA9548A I2C multiplexer device to have each sensor utilize different I2C addresses, allowing access to multiple VCNL4010 devices. Due to time commitments, and our current status of the project we decided to have this be the last option if there is enough time available to us at the end of development of the project.

As a replacement, we decided it was best to move on with only one spot on the parking lot, which is slot 1A and to be accompanied with a single VCNL4010 Proximity sensor to solve the problem. If there is enough time provided at the end of development, the I2C multiplexer may be an extra addition to the project but is not the focus at this time. Another minor change, resulted from the RGB LED use, the 3 Common Anode LED’s were initially planned to be used. Through testing with the servo controller, it was

evident to us that the LED's are using an increased amount of current which may serve as a risk, and potentially blow out one of the LED's. This caused the color "red" on the LED to produce a very bright appearance even, when turned off/or dimmed. As of right now, we have decided to instead acquire Common Cathode RBG LED's to reduce the amount of current utilized by the LED's at each parking space on the parking lot. This in return, will create a much simpler connection to the PCA9685 servo controller using its's built in resistors to limit the current used for each color (red, green, blue), avoiding the need of any other external connections.

The YoLuke's USB Camera sensor, required a major change as well due to software issues with testing the image processing software for license plate detection. The intended purpose, was to have the sensor capture a screenshot of the license plate, and have this screenshot go through the image processing software, retrieving the single license plate number details. After much consideration and testing, we were unable to get the software working with the Camera sensor. Due to this reason, we had to come up with an alternative solution that works through simulation of the license plate detection method. Now, when the Camera sensor takes a screenshot of the license plate, instead of having to go through the image processing, we created a function in the firmware/code to store a fixed string value of the license plate number which will be called immediately upon entry. The firmware/code had to be re-adjusted for this reason, and the image processing software code portion was dropped as a result. The Camera sensor code will be added into the final code while the database connection is made and will be tested for the purpose to capture an image of the license plate only.

Financial Update

As of March 3rd, there has not been a major financial change in the project's cost. In our current situation and due to time limitation, the use of more than one VNCL4010 sensor has been dropped and thus reducing the overall cost by $22.5 USD, but on the other hand, if we had no time constraints then the original overall cost would've been increased by $6.95 USD as we would've required a TCA9548A I2C multiplexer board to allow us to communicate with multiple VCNL4010 sensors. We also had to switch from a common anode RGB LEDs to common cathode RGB LEDs, which has a very minor impact on the project financially.

Progress Update

As we currently stand, the progress for the project is continuing gradually with some challenges due to software limitations. Development is progressing from the hardware and software side, with minimal changes to overcome. These changes had to be re-evaluated for long-term benefits in relation to the original goal of the project, and whether certain features/parts were needed or unnecessary in different scenarios. The final code right now represents the three sensors running and reacting to changes in the parking lot, and will be coupled with the mobile application and online database in the following 1-2 weeks. The code right now utilizes each sensor and provides it's intended functionality through our application. This includes, the initialization for all 3 sensors/effectors, and since each hardware type is a I2C device the code assembled for the firmware was modified to be much simpler, and effective. The IR Break Beam sensor and the VCNL4010 Proximity sensor work together at the same time, rather than

the planned implementation of having each part of the code work in a sequence. Now, the VCNL4010 Proximity sensor does not have to wait for an excess amount of time as it runs concurrently with the IR Break Beam sensor. The code segment for the IR Break Beam is now fully functional with the entry/exit scenarios, based on the status of the gate. The servo motor code has also been integrated into the code for the PCA9685 for both entry and exit situations on the physical parking lot, to move the motors up and down in a 90 degrees rotation. The firmware/code for this sensor will also work with the IR Break Beam sensor to sense the opening/closing of the gates. These functions developed for this sensor will be called in the code, once the database connection has been fully established later on.

Our primary focus in the upcoming final weeks of the semester, will be to complete the mobile application integration along with the wireless database configuration/connection with having the firmware thoroughly tested. The code will then be tested with the final PCB design and tweaked based on any changes we may feel need to be adjusted. Currently, the final PCB design has been sent to the prototype lab to be etched and produced and our plan is to solder, and test the final code of the firmware over this upcoming weekend, for each sensor with the board. Further details on the final PCB integration is described in the PCB (Printed Circuit Board) section of this report in section 3.2.4 above. Therefore, based on our current progress, the project is on the right track and we are confident we will be able to follow-up with all requirements for the upcoming milestones in the coming weeks as the coding and firmware status nears completion.

Link to Complete Code in Repository

The following is the link to the complete code in the repository, which includes the modified individual sensor/effectors files, and the final integration of the 3 sensors into one main cohesive python program (without the database configuration currently) which is denoted by: **watech.py**

https://github.com/VikasCENG/WatechPark/tree/master/Sensor%20Specific%20Code

7.2 Application code

This section of the report will present the final integrated mobile application in its entirety and the end product assembled throughout the duration of the Winter 2020 semester. We will be addressing the current state of the mobile application as the project nears completion, along with the finalized features of the login/authentication activity, various screens available to the consumer which will be further represented through the data visualization activity screens implemented throughout the application. Finally, through the action control activity we will address and present the final implemented parking lot feature proposed previously through our mobile application status update. Towards the end of this section, will be the modified code files organized into the Appendix, along with the link to the complete mobile application code to be accessed through our project repository.

Demo

/1 Hardware present?

/1 Memo by student A

/1 Login activity

/1 Data visualization activity

/1 Action control activity

Report

/1 Login activity

/1 Data visualization activity

/1 Action control activity

/1 Modified Code Files in Appendix

/1 Link to Complete Code in Repository

Memo for Final Mobile Application

At this stage of the project as it nears completion, our team has fulfilled all of the planned objectives initially stated throughout our previous status updates for the mobile application. Along with some minor improvising which had to be made due to time limitations, or software limitations to meet the requirements of our application in its entirety. Therefore, we are proud to announce that we have fully completed the mobile application and have met our goals stated for our parking application.

Development of the mobile application began in the previous Fall 2019 semester and was carried into the Winter 2020 semester where we had to integrate the hardware with the mobile application including each sensor/effector, to be able to talk back and forth from the hardware to the mobile application and vice-versa. Currently, we have completed the mobile application with all proposed features from the initial status update, along with some minor tweaks and adjustments that needed to be made to meet the requirements of the project. The mobile application works alongside an online database structure through the Firebase database and on-site devices which include the VCNL4010 Proximity sensor, IR Break Beam Sensor, and the 2 servo motors running alongside the PCA9685 servo controller.

Similar to previously, the application follows a login and authentication structure and is designed with various screens in mind to support our consumer application. This

includes options to add a car, manage added cars, view real-time data specific to a chosen parking lot, and provide status updates/changes through the supporting sensors/effectors and the data sent and retrieved by the online database. Other features include consumer abilities to reserve a spot in a parking lot, select a parking pass, and be guided to payment services and view order history/transactions entered by the user. As well as access settings, customize language preferences and other in-app options.

Although, the main goal of this semester and for our team in completing the mobile application was to integrate the proposed data visualization feature of the parking lot, provide real-time data through status updates/changes, monitor and track statuses of individual spots during different intervals and provide capacity management techniques. This includes, gate control through entry/exit with use of the hardware. In terms of these planned tasks, we have now finally deployed the parking lot data feature vital to our parking application, along with other key developments and additions added to enhance the final end product.

Development for the mobile application began in early January and was gradually being worked on and ended on March 20, 2020, in line with our planned schedule of events/milestones to be met from the beginning of the semester. Now, we have successfully integrated each sensor to do its job in displaying and replicating the data sent from the sensors/effectors to the online database and have retrieved the data to be displayed through the mobile application. This includes the following, data sent by the VCNL4010 Proximity sensor is sent to the online database through proximity readings and manipulated through the application code. This data is used to display the parking

lot changes, provide status updates in real-time based on the occurrences of the on-site devices physically there on the parking lot model. For example, the added data implementation feature now portrays a visual representation of the lot, and four parking spots to indicate the open or occupied slots with one active spot listening to proximity changes and status updates denoted by Slot 1A.

We also made use of the IR Break Beam sensor to show the status changes on entry/exit of the gate and have that data be replicated and tracked to be notified to the consumer. In addition, with the support of the professor we also were able to successfully implement an admin feature, adding another layer of user capability and access to our application. Based on the advice received, our team decided it was best to move forward and implement an admin control feature with an option to be able to control the hardware utilized at entry from the mobile application, without having the need of having to design separate screens. This in return, also helped reduce the amount of extra work and provided us an outlet of focusing on the major requirements we needed at the end of the project.

Overall, the goal of the mobile application is to provide a sleeker, intuitive interface coupled with various types of services and features. Keeping this in mind, the final data feature was designed on a single screen for easy consumer access, with all of the main data features available at your disposal for the chosen parking lot based on consumer selection. Minor tweaks and adjustments were also made to only tackle what was needed and cut down on excess details that was not needed, as sample data was replaced with real sensor data for the purpose of our parking application.

This section is discussed by Student A (Vikas Sharma), the lead for the mobile application development and integration, along with a group effort by each team member for presenting this final mobile application milestone.

Login Activity

The final version of the mobile application did not require any major changes to the login/authentication side as most of the features stayed intact for the consumer application and its overall purpose. Similar to before, at the start of opening up the application, the user will be prompted to a screen with options to log-in, register an account, or reset your password using the Forgot Your Password screen or Verify Your Password for different forms of authentication services in-application. We constructed an authentication process, in which we believe is a simple, intuitive interface for storing/checking credentials through secured database access. The following is a portrayal of the Register/Login screen which is automatically prompted to the user at entry of the application.

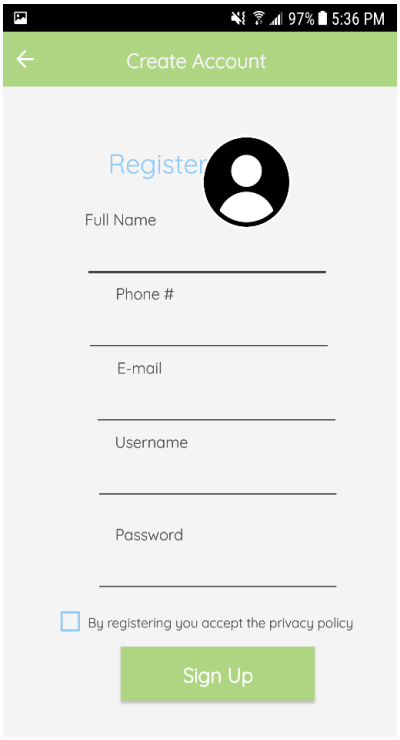


Figure 85 - Register Screen (FINAL)

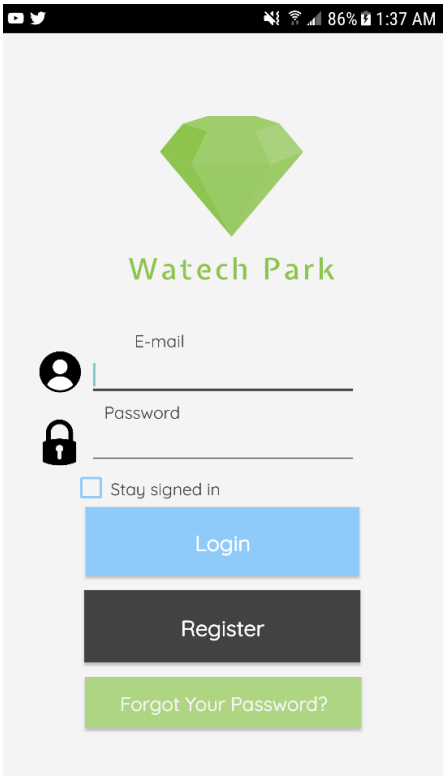


Figure 86 - Login Screen(FINAL)

The authentication part was designed with having a login/register screen separate from each other. The Register screen prompts user input in terms of personal information specific to a user. There is also the option for the user to choose a profile image from a real device through Firebase external storage permission. This account information is submitted onto the Firebase Real-Time database under a data structure, which holds the data. This information incudes, the full name, phone #, e-mail, username, image uploaded. So, once a user registers into the system the data is sent to the Firebase database, and stored to check for validation through login attempts further down the line.

After registering, the next step is to login. To login, we used the Firebase Authentication system for e-mail and password. This is a built-in feature of Firebase, that is used to authenticate a user that exists in the database. So, in the Login screen, the user would sign-in with the “registered” email and password. Firebase checks for valid/invalid credentials based on the information stored in the Firebase database section under that UID (unique identifier). User selections are stored with a “Stay signed in” option. This means, account information is visible after the user leaves the session, or returns to resume the activity. At this time in Firebase, the logged in user would appear in the Authentication section with the corresponding email/password information.

The authentication process continues with the “Forgot Your Password” and “Verify Your Password” screen. The Forgot Your Password screen is used to allow the user to reset their account password. There is the option to use e-mail or phone # authentication. This authentication is done through the Firebase database, where once a consumer selects a service, the other unattended service is not allowed to be accessed. The user enters an email address and through valid checking Firebase will then send a verification email to the corresponding email address. Phone number authentication requires the consumer to enter in a valid phone number using registration data from Firebase. Once the phone number has been validated, and follows the required system format (+1) a verification code is sent randomly through the phone service provider.

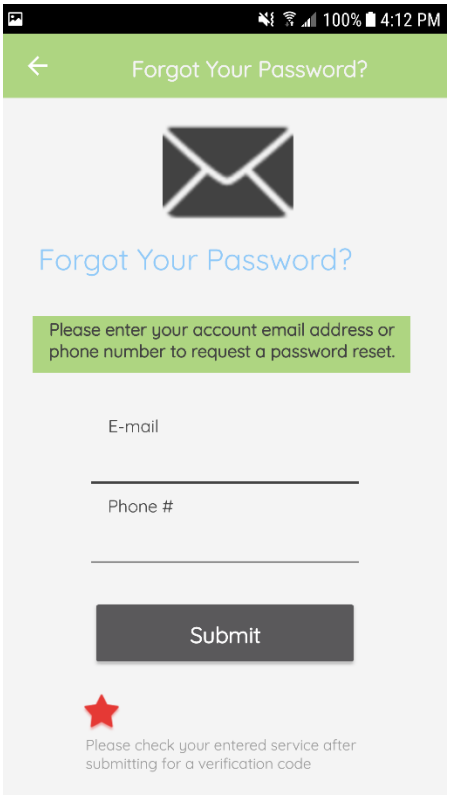


Figure 87 - Forgot Your Password Screen(FINAL)

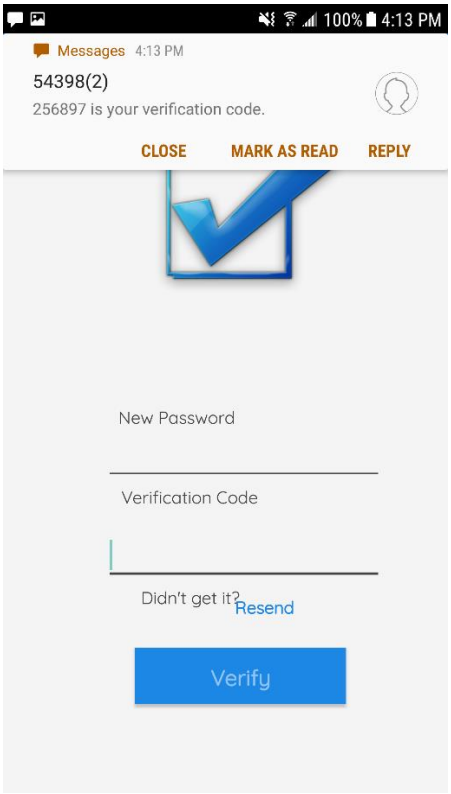


Figure 88 - Verify Your Password Screen(FINAL)

The mobile application also holds a feature, where account information is displayed specific to a user, in this case it would display different details based on the active UID. The screen followed a general layout, a little different from having to fit everything into a fragment view, we decided to use a new separate screen. This screen displays account information from the Register screen. The data is populated and displayed. Such as, the profile image stored from Firebase Storage, phone #, e-mail address, the name and a timestamp. In the final version of the application, there were no changes committed to the login or authentication side and the account screen remained as is.

Figure 89 - Manage Your Account Screen (FINAL)

Data Visualization Activity

The data relevant to the parking lot available appears bundled on the main home screen and once the login authentication process is complete this is the following step the user will encounter. On this screen, a single parking lot can be found which only had text as a placeholder rather than a row of selections. In this case, we decided to use “Humber College” as the only parking lot visible on the screen, and eliminated the rest of the parking lots which were previously only used for sample and visual purposes from the previous semester. We believe this is the most streamlined approach taken for the users, and this allows easy and simple access of the data, and the activity all available within one main screen. This includes, proximity readings from the actual hardware, reservation capabilities, parking lot details and the actual lot itself visually represented in real-time. For this screen, we redesigned the main-menu from the proposed version,

finally having a much sleeker interface to display only one parking lot location, and its information to the screen in a more convenient manner. So, as shown above this screen offers two options to the consumers. This being “View Details and “Reserve”. The View Details screen was the main modification point for the duration of this semester. There is an expandable view which pops up under an AlertDialog box and in this screen is where the parking lot is visually represented, and the sensor data is displayed along with real-time status changes of each spot.

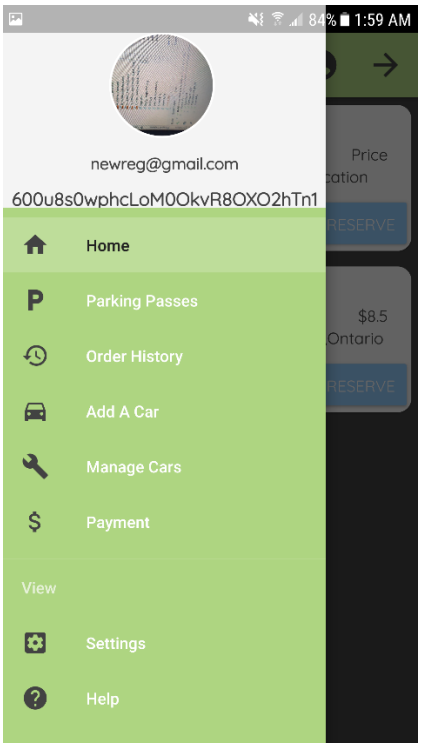
The idea was to build from this feature and specific screen, which we were able to implement with some modifications and removals due to hardware limitations on the VCNL4010 Proximity sensor end. The reason for this was as previously mentioned in this document, we came across an issue and found out that the VCNL4010 Proximity sensor only supports a fixed I2C address of 0x13. Due to this reason, we were unable to support all four sensors with the originally four planned spots, each positioned with an individual proximity sensor, and have that data be fetched and displayed on the mobile application. Our team extensively debated whether it would be applicable to implement a I2C Multiplexer at the end of the project, but we finally came to a consensus to not follow through with that idea as it may be costly in financial terms, and lengthy in the time space we were provided with. Also, if we had decided to implement a I2C multiplexer, we may not have been able to implement other additional features such as the admin control task for the entry servo motor due to time commitments. In addition, we also would need to redesign our current final PCB unit to work with the device, which again would’ve hindered our ability to focus on the essential tasks needed to be completed.

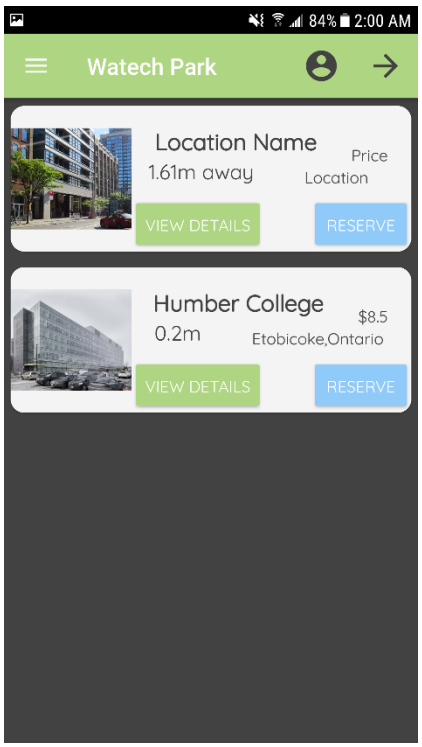
After continuous development, now this screen portrays the data retrieved through the online database which gets sent by the real sensor/effector data through different data types and instances of the code either through raw proximity values or Boolean values which are then converted to String forms to be readable by the mobile application interface. Thus, instead of having the mobile application talk directly to the sensors, we collectively decided to have the Broadcom Development Platform being the Raspberry Pi 4 act as the sender, working in touch with the online database which is the supporting bridge structure acting as the gateway to handle and store the data sent by the hardware for each device. The mobile application in this case acts as the receiver, where it basically retrieves the sensor data and displays this data on this screen. Along with manipulating the data sent, to work with our main features on the “View Details” screen which are behind the scenes tasks performed through the application code.

The data displayed includes the following, four main spots of the parking lot in reference to the data and status of the VCNL4010 Proximity sensor, entry/exit arrows which denote the gate status through the IR Break Beam Sensor. The screen also features the addition of displaying the real-time proximity levels of the VCNL4010 Proximity sensor and is viewed in-app as “RT Proximity” followed by the proximity value fetched through the online database. To indicate the status changes on each respective parking slot, we designed this screen to show relatively three main colors being light green, red, and blue. Based on the different scenarios of the VCNL4010 Proximity sensor this data is then updated through “open” or “occupied” instances of Slot 1A with the other three spots being Slot 2B, Slot 3C, Slot 4C acting as simulated spots set with a value of 0(occupied) for the duration of the data sending/retrieving session. The data is

displayed and organized with the proximity value at the bottom of the screen, fetched from the database, and cooperates with the VCNL4010 Proximity sensor to send the data. More details into how this works will be described in-depth in the upcoming Action Control Activity branch of this section, including how the application code was designed to work with the hardware in the end and vice-versa.

The other option visible is to “Reserve” a spot in the parking lot. Once the user chooses to reserve a spot, the corresponding data is sent automatically to the Parking Passes screen for that parking location and the user is notified in-app to access the payment service. For example, a user reserves a spot for the Humber College parking lot, and once the parking lot has been reserved successfully, a notification pops up in-app presenting the reserved lot and to view the parking passes for the next step. By swiping to the left of the screen, brings up the side navigation drawer that is used to hold the other fragments and features of the application. This allows the user to access every main feature of the application within a gesture, and to avoid having to go through many hurdles or displays to get to their desired destination. A representation of how the data screen and its overall design is shown on the far-right side below of Figure 77.





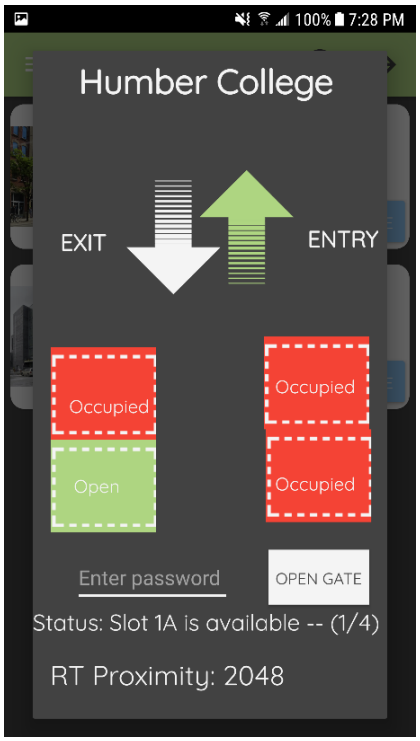


Figure 90 - Main Sidebar Screen (FINAL)

Figure 91 - Main Menu Screen (FINAL)

Figure 92 - Parking Lot

In the “Add A Car” screen, the user enters in their vehicle details to register a car to the account. The user is prompted to enter information specific to each account, such as the Make, Model, Color, and License Plate #. We have decided to store each added car into the system and send the data according to a specific user to the database. Once a license plate image is taken on-site of a vehicle, the application will instead run a function holding a fixed license plate string and try to see if it can match it with the added license plate entered by the user through the mobile application. If the data matches, entry to the lot is allowed and the application displays the gate opening from a visual representation through the emergence of a vehicle and a color change.

The plan was to implement this feature by the timeframe of Mid-March, which we can now say we were successfully able to add this feature as part of the data screen and

have the changes be visible to the app. This was accomplished by basically mirroring the hardware data to a visual form on the application. The ADD A CAR button registers the car to the Firebase database. Once the car is added into the system, and the data is sent to Firebase the consumer can access these details and the registered vehicles in the “Manage Cars” screen. To adjust to the change of having only one parking lot present for the user to choose, and view the data we also had to modify the code to only set a value in the database, and not have it push new values each time a new car is added into the system. This was done, to ensure there are no conflicts that may arise when the search for the license plate number is committed through the Raspberry Pi and to avoid multiple entries by the same user. This is because, at a time the individual is logged in as the “consumer” and this user is capable to add multiple entries based on a single account to serve the purpose of this feature. In the code, initially we pushed the new values and in the final version of the mobile application we set a new value corresponding to the logged-in user

Manage Cars is where the data is fetched from the Firebase database and the information for each “Car” is displayed following a similar format of the main menu. There are 2 options to choose from here: Edit/Delete. Edit allows the user to basically make a change and update the information to the database structure in Firebase. Once the user selects Edit, an inflated view pops up of the fragment prompting to enter in the new information. The user then would tap on the “Apply Changes” button to apply the changes automatically. The changes are visible in real - time through Firebase, once they are set. The Delete option asks the user if they are sure they want to delete the car. If the user approves, the car is deleted from the real-time database and is removed

automatically after the next time you access the Manage Cars section. If the latter is chosen, the action is dropped and cancelled to continue the session.

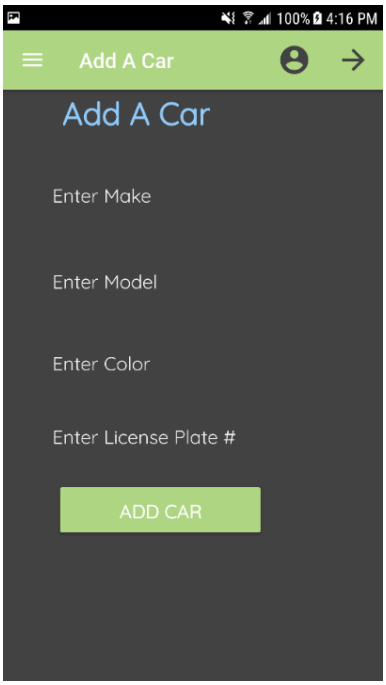


Figure 93 - Add A Car Screen (FINAL)

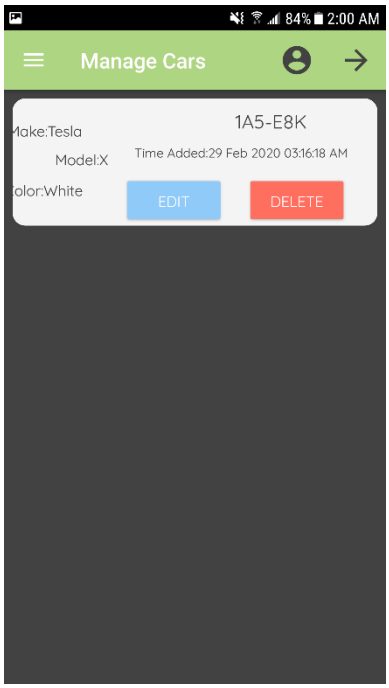


Figure 94 - Manage Your Car Screen (FINAL)

The Parking Passes screen in the application, holds all of the available parking passes for each lot. This includes, the name, location, duration (in hours), validity (time the pass is valid for), type, expiry time, cost, and the account balance before the purchase. There is a button to “SELECT” a parking lot. We also had to modify this screen, to show only one possible parking lot location of “Humber College” as this was the only parking lot available to the user when they login to the application. All other forms of sample data added into the application code was removed, and only the parking lot to hold the real data stayed intact. Once selected, the data for that lot is sent to Firebase and stored

under the UID of the user. This data is also then sent to the Payment screen, which would be the next step to finalize the reservation through the transaction process.

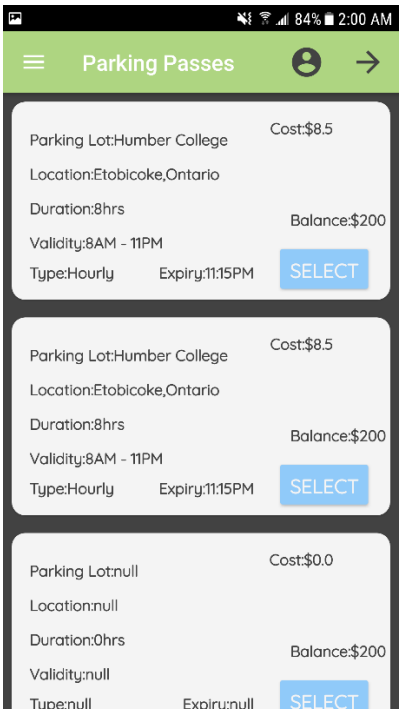


Figure 95 - Parking Passes Screen (FINAL)

On the Payment screen, the selected parking pass is now visible with all the data related to the particular parking lot. On this screen, similar details are displayed, with the addition of an OID (order ID), and e-mail corresponding to the account that is processing the order. Also, the total is calculated for the parking pass with tax and displayed in only a readable form. The balance after the purchase is calculated on the spot and displayed according to the total accumulation and implemented through a in-app QR Code generator. The total would be calculated and based on this set value, the “Generate QR Code” button generates a random QR Code using this value. The user would then tap on the FAB (floating action button) which asks to confirm the purchase. If

the order is confirmed then it has been successfully processed. A toast message appears saying “Order has been successfully placed! Please View Order History for more details”.

Order History displays the order’s that have been placed using a unique OID (order id) which is the UID used to refer to a specific account. The data is retrieved from Firebase displaying the processed information and a timestamp for when the order confirmation took place.

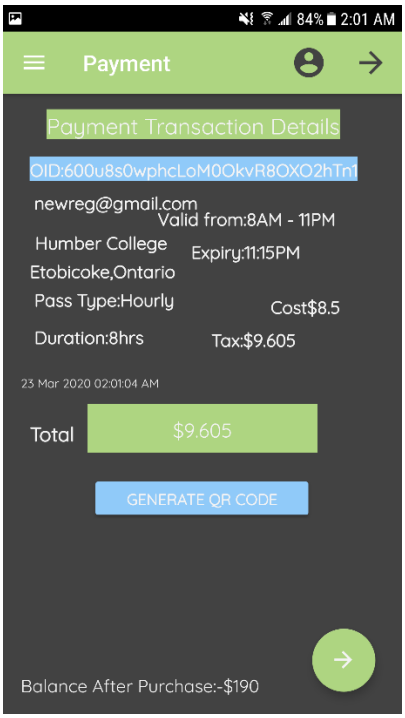


Figure 96 - Payment Screen (FINAL)

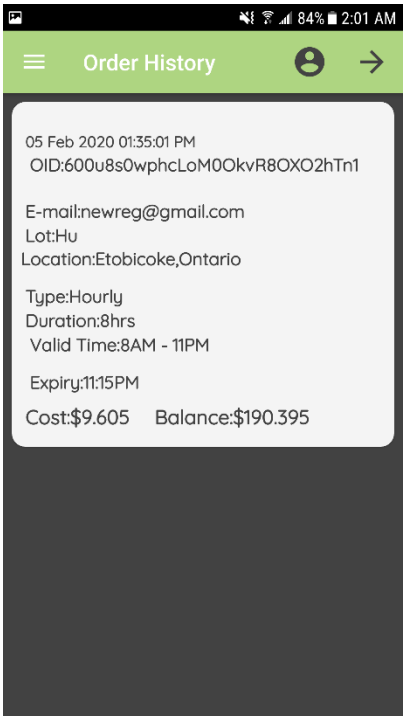


Figure 97 - Order History Screen (FINAL)

As the focus of the semester was to only work on the important features of displaying the parking lot and its status updates/changes in accordance to the hardware, in the end of development there were no added features/modifications to the in-app settings or

options. In this case, as before the Setting screen provides a localization feature for (English/French integration). The design basically followed a simple UI, with a button to “Select Language. The user checks which one to perform and the languages change state accordingly without the need of re-entering the app. The Help screen, displays general help documentation for ways to navigate to the different screens and use the functionality. The About screen displays project and general mobile application details.

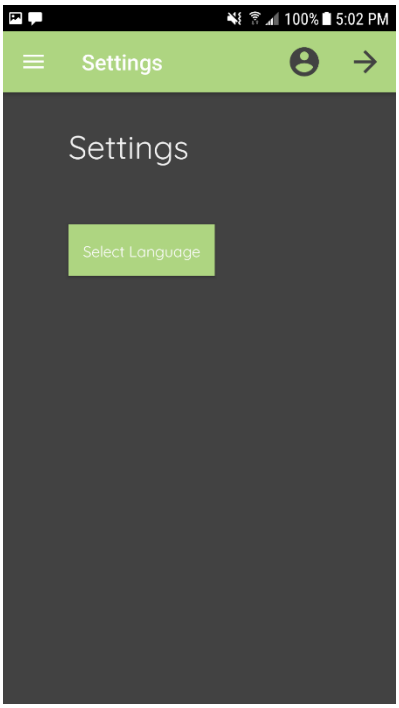


Figure 98 - Setting Screen (FINAL)

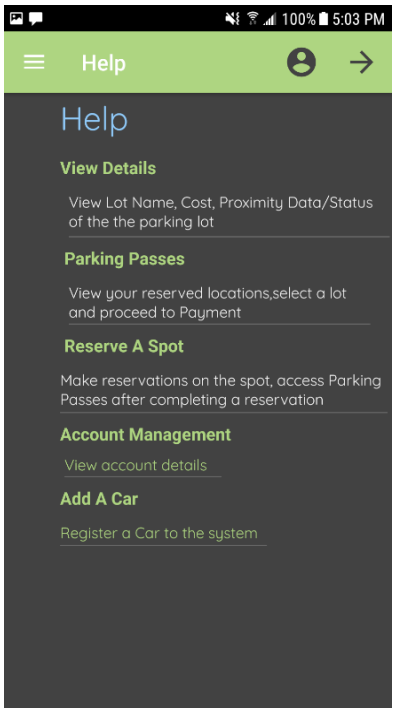


Figure 99 - Help Screen (FINAL)

Figure 100 - About Screen (FINAL)

Action Control Activity

This section will address how the hardware communicates alongside the software, and the process behind having the mobile application control the sensors/effectors, and vice-versa. It is important to have the hardware for the project be able to talk with the application code and allow the user to track and monitor data essential to our parking

application. We will be going through each individual sensor, and their main use for the mobile application for different purposes in the project. Previously, we had discussed the implementation of the three main sensors for the project into the mobile application. This includes, the VCNL4010 Proximity sensor, IR Break Beam sensor, and the 2 servo motors functioning alongside the PCA9685 servo controller. Each sensor provides a unique activity to the mobile application, and the data sent by the hardware to the online database, is further gathered and displayed in application form for consumer interest. Based on feedback received by the professor, and our team considerations we had initially planned for the two servo motors to not have anything to do with the mobile application, as the motors would be there for entry/exit purposes of the parking lot. Although through closer inspection, we ultimately decided a admin feature would be a supporting to add into our application and would further enhance the goal and not only through the consumer experience but also through an admin level priority.

The VCNL4010 Proximity sensor is used to manage and detect a vehicles presence on a specific spot on the parking lot and update the status of the parking lot through a visual representation. At first, the VCNL4010 Proximity sensor sends the raw proximity values from its hardware, and these readings are received by the online database through the “ProximityData” structure. The data sent is in the form of a Long object type. Therefore, when the proximity value is isolated to be read, it must be in the form of a long variable type in the application code initially and then converted into a String form to be able to be manipulated and perform the tasks needed. Through the use of the online database, the data is stored each time a vehicle is detected from either far/near distances relative to the VCNL4010 Proximity sensor. In terms of the process taken in

the code itself, the proximity data is checked through three different instances. These scenarios are based on the parking space status available in the lot, and are denoted by individual spots of Slot 1A, Slot 2B, Slot 3C, and Slot 4D. The procedure used to follow this for the application code is in pure resemblance to the firmware code assembled for the hardware in the watech.py python program. In the start, the VCNL4010 Proximity sensor sends various proximity values to the online database. This data is then fetched and read through a DatabaseReference object using the “ProximityData” data structure, and a separate class is created to store this data into called ProxmityData.java. This class holds two primary variables, which are the proximity, and timestamp as well as the status of each individual slot.

Through developing the code for this part of the application, we had planned to have each spot occupy a separate proximity sensor and send a status of 1 or 0 in Boolean form to indicate a true or false status for a parking space. Due to coding issues, we were met with issues in regards to converting and using the long values of 1 or 0 for each individual slot, as each time the APP would display a 0 as a default value. To solve this problem, since we are only using one active spot on the parking lot by Slot 1A, it was evident to us we only had to make use of the proximity values gathered by the VCNL4010 device, and set each other spot to the value of 1, to denote a occupied spot and show this through the APP with a red background color change for the three remaining spots. This value would stay fixed at all times and would not change as we designed it to simulate the parking lot only. For the coding portion the modifications were made to the ParkingLocationAdapter.java file.

The following steps are taken, if the proximity value read is less than the value of 2500, there are no vehicles present or near the VCNL4010 Proximity sensor. Therefore, the only action taken by the sensor is to send the proximity readings to the online database. Then, have this proximity value be read and displayed at the bottom of the screen in a String format. Since the level of proximity is less than the set value, have Slot 1A remain with a light green background color, indicating no change currently. To guide the user, toast messages are added to indicate that the “Gate is Open” and “Entry is Allowed”. If the value is greater than 5000, change the color of Slot 1A to a red representation to indicate, a vehicle is currently occupying the spot. The proximity readings sent by the sensor is updated in the database, which is then read and displayed under the “RT Proximity:” TextView resource and this value is updated each time based on the proximity value. Toast messages are also pushed to notify the user that the “Gate is Closed” and “Entry is not Allowed”. Once all spots are occupied, the overall status of the lot is full and this is then updated as a visual change visible to the user. A message then appears in real-time to change the overall status of the lot, and the number of remaining spots out of the four parking spaces is tracked and displayed.

The “EntryStatus” data structure in Firebase was created to send the status after all four spots are occupied. In this case, the application sends a status of 0 to show all spots are occupied and the lot is full. From the firmware end, this data is read and checked for a 1 or 0 value, if a 0 is to be found then there is no call committed to run the servo motors function for gate entry. Each spot excluding Slot 1A begins with a “occupied” status or 1(true), with Slot 1A being actively listening and starting the data session by a “open” status which is denoted by a 0(false). The third if statement, basically checks the

condition if the value is in between the range of 2500 and 5000(open and occupied) then the status is updated with a “targeted” message on Slot 1A and a blue color change of the spot. The proximity value is also pushed by the sensor to the database as before, and processed by the mobile APP. As initially planned, following this a toast appears alerting the user “Vehicle is approaching the parking space…” before the actual change occurs.

The following screenshots show the open state of Slot 1A and the entry gate, and the occupied state following a successful reservation of a spot. This feature of reserving a spot will be addressed next in this section following these screenshots.

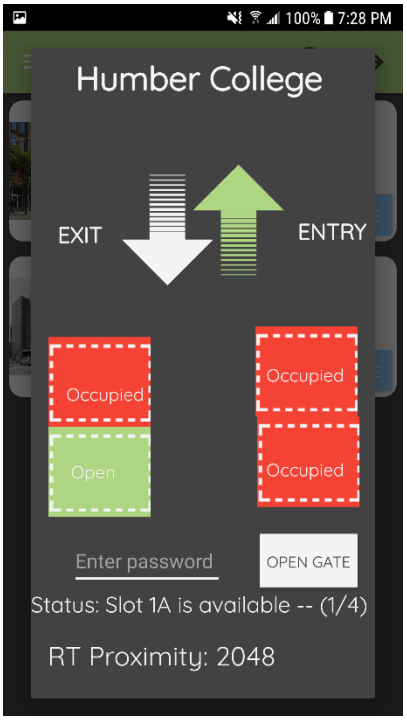


Figure 101 - Data Control Screen (Open Spot)

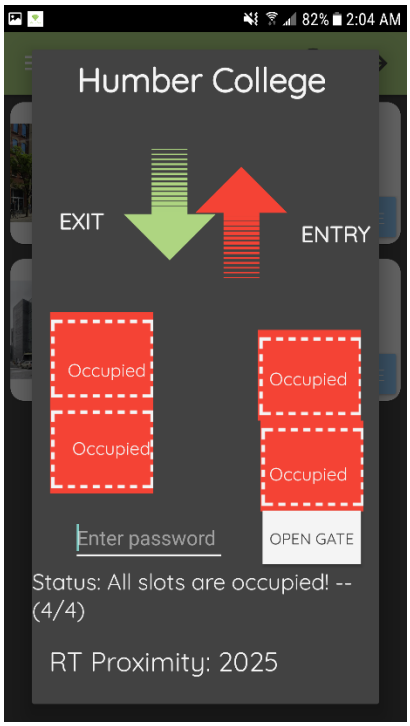


Figure 102 - Data Control Screen (reserving a spot)

Another function of this screen, is to allow the consumer to be accompanied with reservation capabilities. The user can act upon one of the “open” spots at the beginning of viewing the data for the parking lot. If the gate is open and all is clear, then the application allows the user to reserve a spot by tapping on Slot 1A, as it is the only spot available and choose to reserve the spot. Once the spot has been reserved, the data is sent to the Parking Passes screen and a toast pops-up to guide the user to select the pass, and proceed to payment. The reserved spot is also visible to the user as a red color change occurs from the originally green (open) spot, and the overall status is updated immediately. A value of 0(false) is sent to the “AdminControl” data structure to show all spots are taken and entry is not further allowed. Additional details to support the process of sending and storing the data of different sensors/effectors, in the main data structures used is discussed in the Database Configuration section of this report.

On this screen, there are also arrows for entry/exit used by the IR Break Beam sensor which are also updated, with a visual color change from green for ENTRY to red, to show entry is not allowed and the gate is closed. At this time, in-app the EXIT arrow is updated to a green color change, which means exit is the only option available next to the user once all spots are taken. It is important to mention however, for this reservation to take place successfully the proximity value must be less than 2500, or between the range of 2500 and 5000 to book the slot successfully. Any attempt made to reserve an already occupied spot which exceeds the value of 5000, then the attempt will be ignored and met with an error. The following screenshots show the different reservations capabilities and the occurrences of a user choosing Slot 1A on the parking lot.

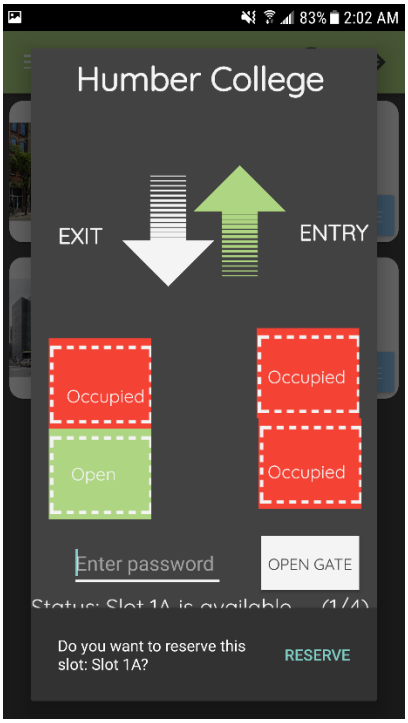


Figure 103 - Data Control Screen (Reserve)

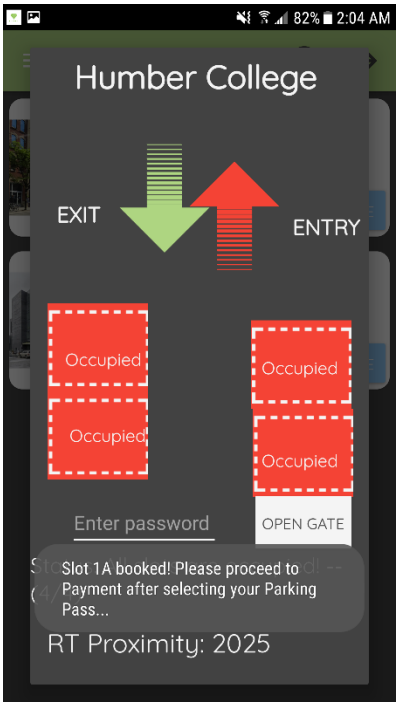


Figure 104 - Data Control Screen(Successful Reservation)

To control gate opening and closing based on entry/exit situations, the IR Break Beam sensor is used to send the current status of the gate. On this activity, this sensor initially sends from its hardware a Boolean value of 1 or 0 to the online database. Once a vehicle is detected and the beam is broken, the license plate number is detected and assessed with the “fixed” string value. The vehicle is validated and a 1 is pushed to the online database to show entry has taken place. The mobile application uses the “GateStatus” data structure in Firebase to locate the status, and read in the value of the gate. The IR Break Beam sensor works alongside of the VCNL4010 Proximity sensor. To explain, in the code the proximity value fetched by the VCNL4010 device is used to gauge whether entry or exit is allowed by a particular vehicle. So, similar to the process of the parking spot feature, the entry/exit arrows are visible on the screen and if the proximity value is less than 2500, then the gate is open. Therefore, the entry arrow

glows with a “light green” color to basically show a visual representation of the action taken and at entrance of the lot. As soon as entry is detected by a vehicle, a toast message appears which displays data fetched from the “GateStatus” data structure alerting the user “Car has ENTERED the lot!” followed with the timestamp retrieved and converted into a String format. This allows the application to also track each specific vehicle action physically occurring on-site of the parking lot.

At this time, the arrow stays green for the duration of the proximity level staying below the set amount. Once the value exceeds 5000, the entry arrows changes to a red color to indicate the gate is closed and ENTRY is not allowed. This is because, since there are four spots, with three fixed values of being “occupied”, once the last spot of Slot 1A is taken then the APP tracks the movement of the vehicles entering the lot and updates the remaining spots as mentioned above. Due to this reason, the arrow turns red to show ENTRY is not allowed until a vehicle exits the lot. During this time, the EXIT arrow switches to a green color to show the only available option to the user. The following screenshot shows the initial entry stage of the parking session, where entry is made by a vehicle and the data is sent to the mobile application.

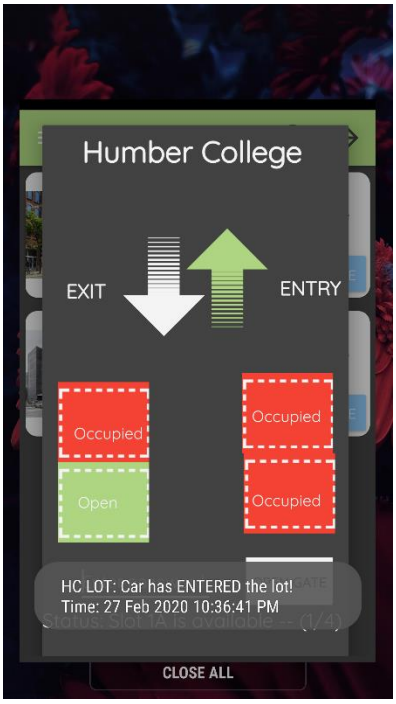


Figure 105 - Data Control Screen (ENTRY)

This sensor also functions with the support of the servo motors which act as the barriers for actual movement and rotation of the gate. Originally, we had planned to not have any sort of preliminary data to be sent or processed by the online database. After successfully communicating the hardware to the app and controlling the APP and its functions through the sensors/effectors, we decided to work on additional features.

The next step, is when an action is taken by the user in-app, to have a feature where the hardware is capable to be controlled by the mobile application. To do this, we introduced a new feature based on revisions for allowing two main types of users to access the APP and its functions. As shown below, we decided to integrate a “OPEN GATE” function which would allow the user to control the hardware present on the parking lot from anywhere, or any place with an accompanying Wi-Fi connection. This

button was added in reference, to a mode we had discussed to implement in early January where we would have the consumer and an additional “admin” user.

This admin user would have access to different features from the consumer side. Such as, having the ability to potentially access parking lot data without being verified, or having to enter the lot. We decided due to time commitments, it would be best to drop this feature but instead workaround this functionality to have an admin feature integrated into the same screen, presenting the parking lot data. We believe this to be the most convenient, and reliable method for the consumer as it would be interesting, quick and effective. Also, working with the support of the VCNL4010 Proximity sensor and its proximity data levels, we had the button be present and added an EditText field for the user. In this field, the consumer would act as the admin, and once the password entered matches the fixed value in the code of “admin” and the button is clicked, then the servo motor function is performed in the firmware code of watech.py. To proceed with this, a value of 1 is sent each time the button is pressed after the consumer is validated as the “admin” for the duration of the session. Once validated, “admin” access is activated and a message appears mentioning the gate is open.

The overall point of this feature is to allow the “admin” user the ability to access the parking lot anytime, anywhere. This means the permissions for this specific user is granted and at any time of entry or exit, or an open/occupied status change where then the admin is allowed to open the gate. Through this feature, this will allow the admin to access the lot during poor weather conditions, such as in the season of Winter where snow may be blocking the entrance or parking spaces inside of the lot. This allows the admin to enter the lot in support of snow removals and other conditional activities. Other

situations, can include providing daily maintenance of the lot, or viewing real-time capacity levels of each parking space and the lot overall. Now, once the user has exited from the data screen and returned to the main menu, the status value of the AdminControl structure is reset to a value of 0 along with the rest of the data. This is to ensure the gate does not always remain open at all times, and is actively listening for changes in the GateStatus and AdminControl data structures. In the occurrence the user enters in invalid credentials as the password for admin access, the system sends an error message and then sends a value of 0(false) to the AdminControl structure in the online database. This makes sure the gate does not open, without the required validation process. The following screenshots below portray how the admin access feature works, in both successful and unsuccessful trials.

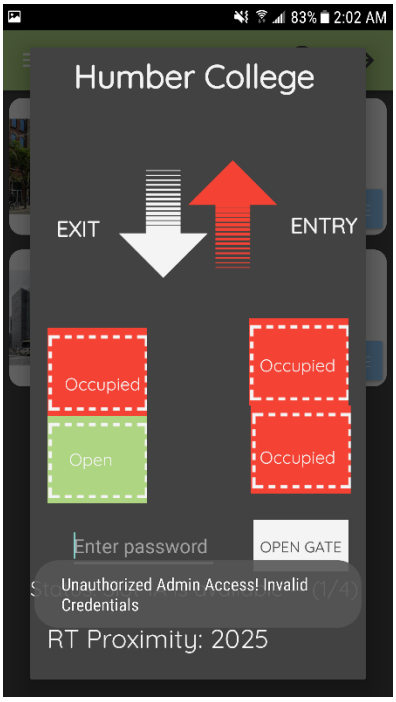


Figure 106 - Data Control Screen (Invalid Access)

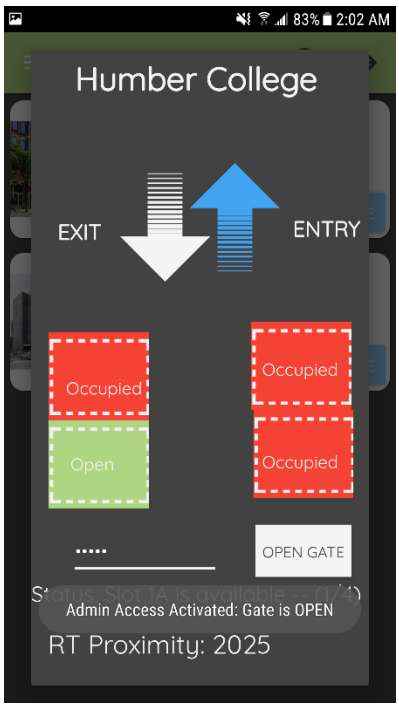


Figure 107 - Data Control Screen (Admin Access)

Overall, for the development of the mobile application the code built made use of only the three main sensors of the VCNL4010 Proximity sensor, IR Break Beam sensor and the two servo motors. These sensors/effectors were used to relay the parking lot data as shown above, through the data visualization feature of the parking lot and the action control features taken by the hardware or the APP itself to control the hardware through an IoT structure to accommodate the final capstone end product.

Link to Complete Application Code in Repository

The following is the link to the main code for the final version of our WatechPark android application. Through the “app” folder are the java classes and main code assembled for the project. This can be found under the “WatechPark(APP)” folder designated on the main “master” branch of our project repository.

https://github.com/VikasCENG/WatechPark/tree/master/WatechPark(APP)/app/src/main/java/com/example/watechpark