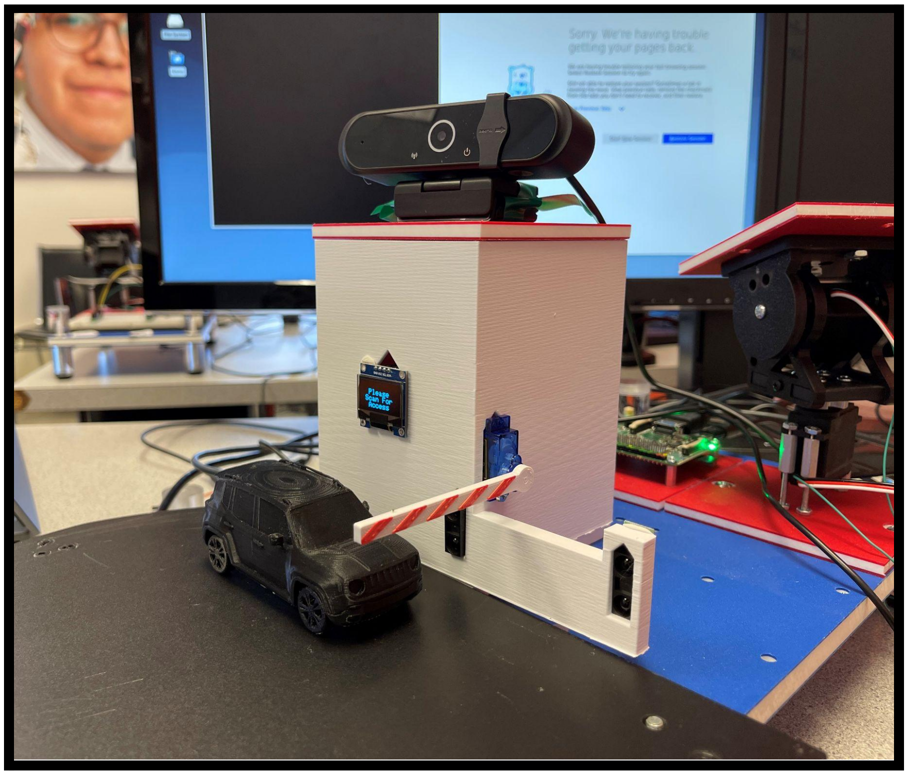
CSE 398
Junior Design Project



Isabel Melo
Alexander Perez
Kyle Maiorana

CSE 398
Duane Marcy
12 May 2022
Syracuse University

Table of Contents

Introduction

According to a paper written by Michael Gips on *swiftlane.com*, The disruption from the COVID-19 pandemic has changed the way we will approach physical security in 2022 and beyond. Following a survey of 473 high ranking security professionals,respondents identified Integration with legacy systems, user convenience and throughput at entrances, and protection against increasing vulnerabilities as top concerns. Our Junior Design Project aims to impact the access management issue in conventional and strategic locations. This project could be applied to enforcing home security, parking garage management, electronic toll collection (such as EZ-pass), restricted area access, college campus security, and more.

To meet the requirements defined by Dr. Marcy, we needed to create an embedded or mobile device and our demonstration must showcase specific hardware. Heavy.ai defines an embedded device as a microprocessor-based computer hardware system with software that is designed to perform a dedicated function, either as an independent system or as a part of a large system. At the core is an integrated circuit designed to carry out computation for real-time operations. Our gate access system satisfies this requirement by handling all of the access and verification logical function by passing information to and from different microcontroller boards, and allowing a user to interact with the circuit concurrently. This communication also takes into account information relayed by sensors, servos, cameras, and display peripherals to make a cohesive decision in real-time and operate as expected, reinforcing that the system operates in an embedded fashion.

In order to undertake this project it was necessary to divide the work into manageable sections that would function independently, coming together for the final demonstration.

Planning Stage

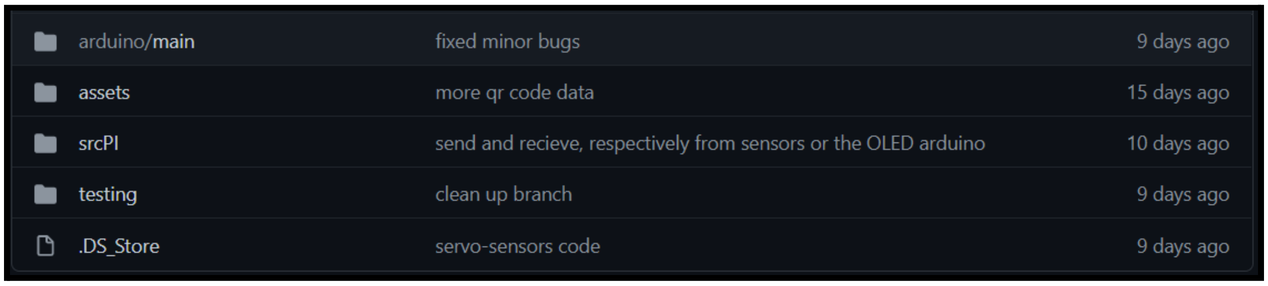
Before starting any work, we needed to assemble a team. Alex, Kyle, and Isabel have all worked together on projects in other classes and know they work well together. In our initial design and proposal talks we quickly came to the conclusion that an Access Management system would be ideal to implement all of the project requirements set forth by the project.

Once we had a clear understanding of the features and functionality we wanted to implement, it was time to research hardware and software components to use in our project.

To implement the hardware side of things we decided to use 2 Arduino Nanos. We made this choice because the two microcontrollers are very small and easy to physically place anywhere on our project. They can also interface with the Arduino IDE to easily flash code to the board.

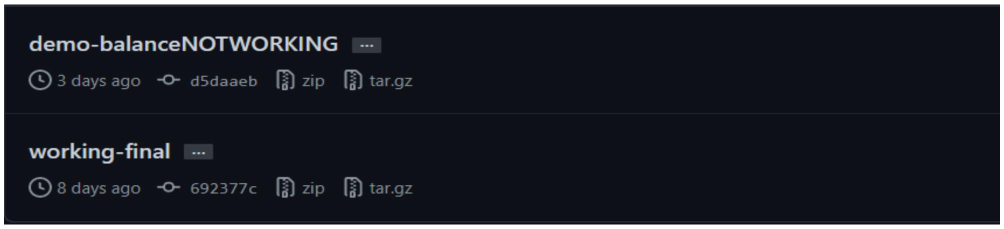
To implement the software side of things we decided to write the main Python program in Python, this code will execute on the Rock Pi 4 computer. Deciding to use python was a decision in itself due to the inefficiency and large overhead of resources required by Python. Other options were C++ and Java. After researching potential libraries to use for the project, Python provided us the best options in terms of documentation of libraries and functionality. Some libraries that we specifically were interested in using include: json, time, OpenCv, pyqrcode, os, and serial.

We decided to use Github for source and version control for this project. Our main development took place on the Master branch, with each team member creating a new branch, developing their code on a personal branch, and making a pull request to the Master branch when ready to merge code. The structure of our code is rather straightforward as shown below.



arduino/main	fixed minor bugs	9 days ago
assets	more qr code data	15 days ago
srcPI	send and recieve, respectively from sensors or the OLED arduino	10 days ago
testing	clean up branch	9 days ago
.DS_Store	servo-sensors code	9 days ago

The files running on the Arduino nanos are located in the arduino/main directory, while Rock Pi source code is located in srcPI/. Within the arduino/main there are independent files for the OLED Display and the Sensors Arduino, located in OLEDArduino and SENSORArduino respectively. In the srcPI/ directory, there is a main.py file that is the main execution point for our program, a User.py class file which defines the attributes of our user, and a data.json file that stores our predefined objects in json format. The python code was developed using Pycharm Professional version 2021.3 and the Arduino code was written in the Arduino IDE. Pycharm Professional provides many useful features when developing code. For example, Alex was able to upload code directly from his laptop and deploy it to the Rock Pi instantly. This was very efficient and helpful when developing, running, and debugging the program. Before completing our project, we Tagged versions of our code to revert back to if we broke the code at all. These tags are shown below:



demo-balanceNOTWORKING ···
3 days ago · d5daaeb · zip · tar.gz

working-final ···
8 days ago · 692377c · zip · tar.gz

Design and Implementation

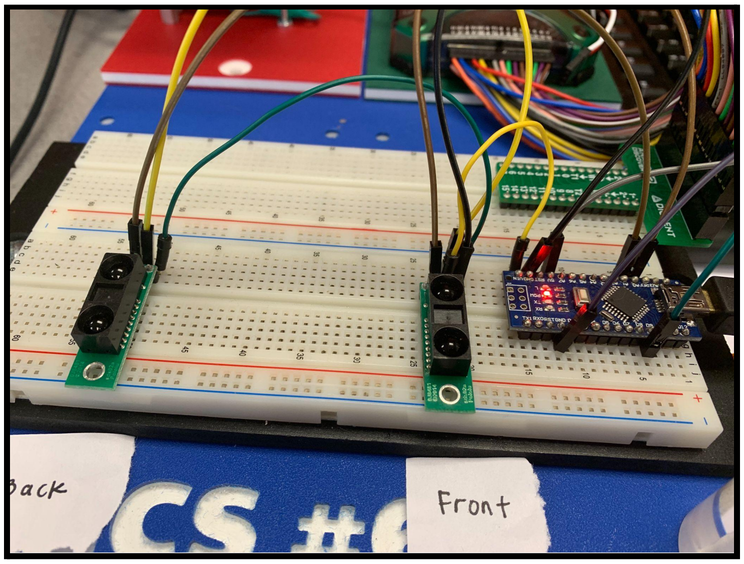
SENSOR ARDUINO SECTION

We used an Arduino Nano device to set up two distance sensors as follows:

Hardware:

The distance sensors both have 4 pins:

SharpIR 1 (front)	**SharpIR 2 (back)**	**Arduino Nano**
Vcc		Vin
Grd		GND
Out		A0
	Vcc	Vin
	Grd	GND
	Out	A1
En --- not in use	En -- not In use	Not in use



For the software, we used the SharpIR arduino library to control both sensors. The goal was to send a ‘frontsensoractive’ message through the serial communication port to the server once the front sensor detects an object in proximity. Likewise, send a ‘rearsensoractive’ message whenever the back sensor detects an object. We accomplished this using the code below:

Using the sharpIR arduino library, we initialized the sensors as follows:

```
SharpIR front( SharpIR::GP2Y0A41SK0F, A0 );
SharpIR back( SharpIR::GP2Y0A41SK0F, A1 );
```

Next, inside the loop() method, we declare two distance variables to constantly get the distance detected by both sensors:

```
int frontDist = front.getDistance();
int backDist = back.getDistance();
```

And finally we set up conditional logic to send a message to the server through serial communication if either sensor is triggered

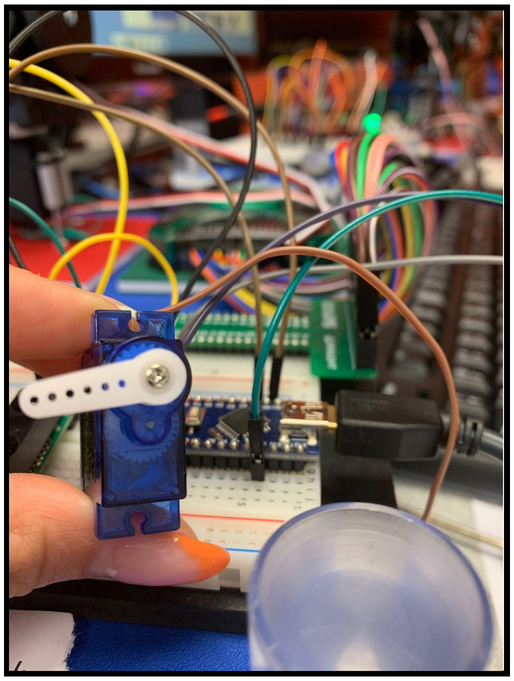
```
if(frontDist > 3 && frontDist < 6){
  Serial.println("frontsensoractive");
      delay(1000);
      }

 if(backDist > 3 && backDist < 6){
  Serial.println("rearsensoractive");
      delay(1000);
  }
```

SERVO MOTOR ARDUINO SECTION

For our gate, we used a small servo motor controlled by the same Arduino Nano that drives both sensors.

For our, hardware, we added a few components to the arduino as follows:

Servo	Arduino
Power	5V
GND	GND
Signal	D9



The software component of this part of our project was facilitated using the Servo arduino library. Our goal is to open the gate whenever the arduino receives an ‘opengate’ message from the server through serial communication, and close the gate whenever the arduino receives a ‘closegate’ message. To achieve this, we used the code below:

Inside the setup() function, we initialize our motor to output a pwm signal to pin D9. We also initialized our motor to be at a "closed state" the first time our code runs:

```
motor.attach(9); //enable pin D9 for pwm signal in Servo Motor
closeGate();
State = 0;
```

Next, inside the loop() function, we create conditional logic to change the state of our motor depending on the instructions given by the server via Serial communication.

```
 String data;
  if (Serial.available() > 0) {
         data = Serial.readStringUntil('\n');
    //Serial.println(data);
  }
  if (data == "opengate" && state != 1){
         state = 1;
         openGate();
    Serial.println("inside open gate if");
  }
  if (data == "closegate" && state == 1){
         state = 0;
         closeGate();
    Serial.println("inside close gate if");
  }
```

The openGate() and closeGate() functions do the following:

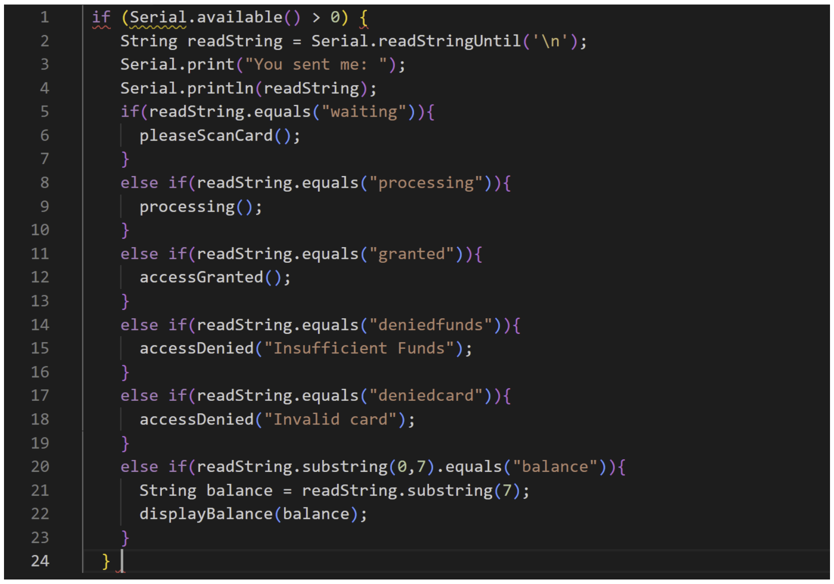
```
void openGate() {
  motor.write(0); //move the gate all the way up
}

void closeGate() {
  motor.write(90);//move the gate all the way down
}
```

OLED ARDUINO SECTION

For this project, it was deemed necessary to have an outward facing display to instruct and inform the gate user. This was accomplished by incorporating an Adafruit 128x64 0.96 OLED display into the gate design. A decision was made to have this display driven by a separate auxiliary Arduino Nano than the sensors and servo so we could divide the work more efficiently amongst group members. The implementation of this OLED display involved connecting the four pins as shown below:

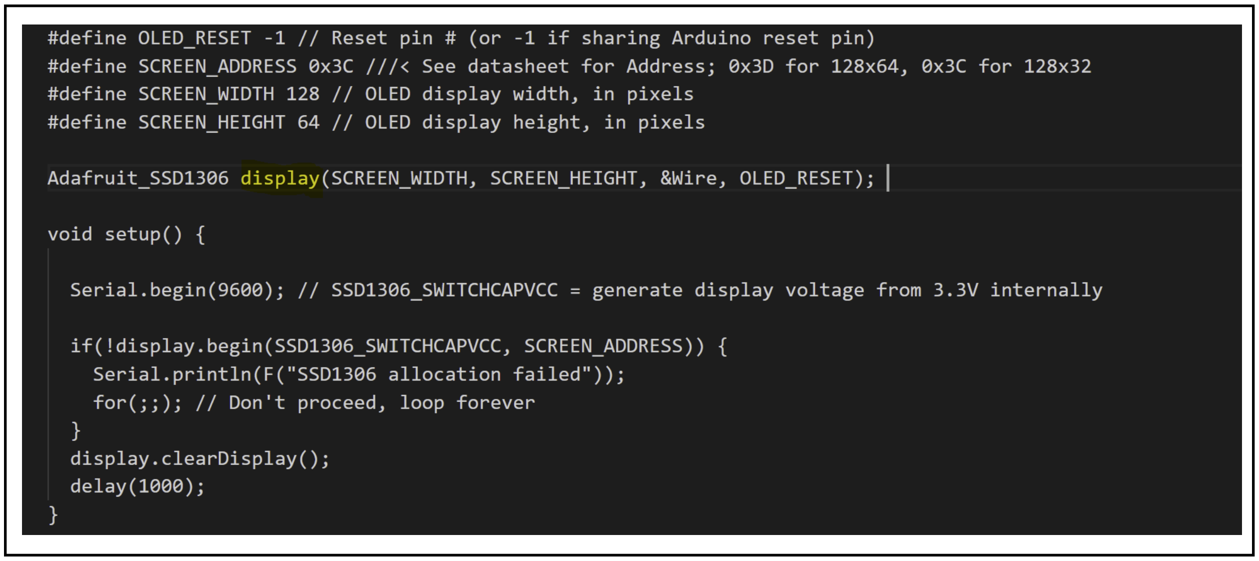
OLED Display	**Arduino Nano**
GND	GND
PWR	5V
SCL	A5
SDA	A4

On the software side, the Adafruit SSD1306 library was used to coordinate the i2c connection that was letting the Arduino communicate with the OLED. The Adafruit GFX library was also used for displaying text, lines, and other graphics. Since the OLED was connected to the Arduino, it made sense to program the screen logic in Arduino code. We knew that we would want to make serial requests from the RockPi to the Arduino that would trigger the appropriate display mode when an event occurred. To accommodate this, we built an if structure that received and recognized different commands and called the method that wrote the correct information to the screen. The serial connection waits for an endline character, and then decides if the previous string was a valid input.



```
if (Serial.available() > 0) {
   String readString = Serial.readStringUntil('\n');
   Serial.print("You sent me: ");
   Serial.println(readString);
   if(readString.equals("waiting")){
     pleaseScanCard();
   }
   else if(readString.equals("processing")){
     processing();
   }
   else if(readString.equals("granted")){
     accessGranted();
   }
   else if(readString.equals("deniedfunds")){
     accessDenied("Insufficient Funds");
   }
   else if(readString.equals("deniedcard")){
     accessDenied("Invalid card");
   }
   else if(readString.substring(0,7).equals("balance")){
     String balance = readString.substring(7);
     displayBalance(balance);
   }
 }
```

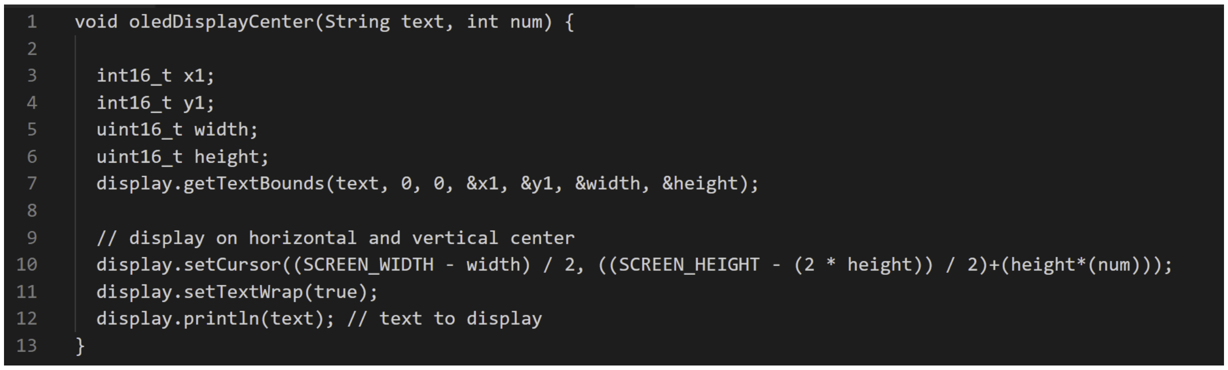
The different modes that were programmed were access granted, display balance, processing, a card scan prompt, and an access denied mode that took in a reason to display. These different states covered every aspect of our user experience, and helped increase the functionality of our design. For the setup, a display object was created, and the proper parameters were defined for screen size and address pin. A serial connection was opened with the baud rate of 9600, allowing the USB port on the Arduino to send and receive packet information.



```
#define OLED_RESET -1 // Reset pin # (or -1 if sharing Arduino reset pin)
#define SCREEN_ADDRESS 0x3C ///< See datasheet for Address; 0x3D for 128x64, 0x3C for 128x32
#define SCREEN_WIDTH 128 // OLED display width, in pixels
#define SCREEN_HEIGHT 64 // OLED display height, in pixels

Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

void setup() {

  Serial.begin(9600); // SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally

  if(!display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
    Serial.println(F("SSD1306 allocation failed"));
    for(;;); // Don't proceed, loop forever
  }
  display.clearDisplay();
  delay(1000);
}
```

To display text to the screen, a function was written to determine where to start the cursor based on the size of the text that was being displayed and centering it in coordination with the display size. A parameter was also added that shifted the y-coordinate additional height values to place consecutive lines above or below each other. When the different display functions were written, they took advantage of this center display method to make sure that text was located in a readable location.

Center Display Function



```
void oledDisplayCenter(String text, int num) {

  int16_t x1;
  int16_t y1;
  uint16_t width;
  uint16_t height;
  display.getTextBounds(text, 0, 0, &x1, &y1, &width, &height);

  // display on horizontal and vertical center
  display.setCursor((SCREEN_WIDTH - width) / 2, ((SCREEN_HEIGHT - (2 * height)) / 2)+(height*(num)));
  display.setTextWrap(true);
  display.println(text); // text to display
}
```

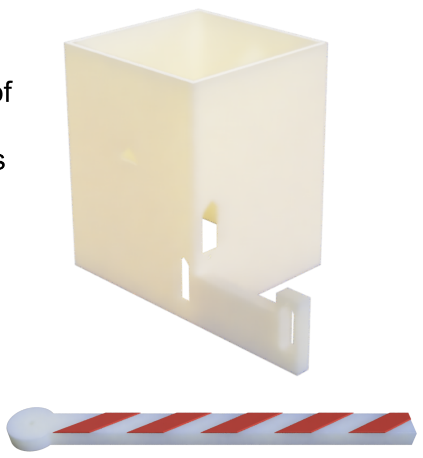
Each of the display functions operated fairly similarly, clearing anything previously on the display, setting up items like the text size and color, and then calling the center display function. Once all of the elements had been written to the screen, the display object's display() method was called to send the information over the SDA wire and write to the display. Depending on what needed to be displayed, the text in these functions was changed. We broke these display functions into their own unique blocks for formatting purposes, as well as to allow the least amount of information possible to be passed from the Pi to the Arduino.

Access Granted Function

```
void accessGranted(void) {
  display.clearDisplay();
  display.setTextSize(2);                // Normal 1:1 pixel scale
  display.setTextColor(SSD1306_WHITE);           // Draw white text
  oledDisplayCenter("Access", 0);
  oledDisplayCenter("Granted", 1);
  display.setTextSize(1);
  oledDisplayCenter("Pull Vehicle Forward", 3);
  display.display();
  delay(2000);
}
```

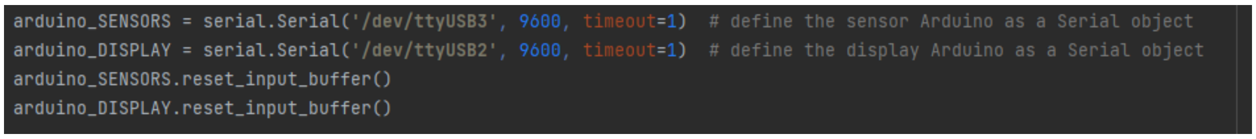
3D DESIGN AND PRINTING

For this project, we wanted a housing for the electronics, sensors, and servo that mimicked real-world parking gates. To accomplish this, a 3D model was designed in Fusion360 based off of measurements taken of each component. The base of this structure was sized to snap onto the top of a square breadboard that bolts to the lab development boards, and has cutouts for wires, the display, sensors, and a servo mount. This rendering shows the design, and the holes where each component screwed into. A cutout was also made in the back for the USB cables that ran from the Arduino to the RockPI. A gate arm was also designed based on measurements taken from the servo, giving an additional sense of realism.



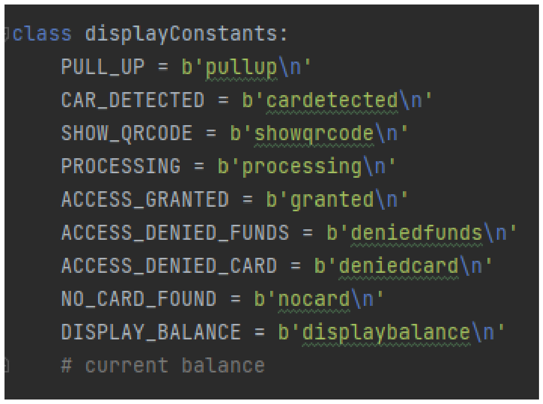
ROCK PI SOURCE CODE

The Rock Pi is the essential piece of the project where the main logic will be handled. The goal of our python program is to listen to events that are defined independently on each Arduino. To do this, we used the Serial library to facilitate Serial communication between the Arduinos. We connected both Arduinos via USB and ran the ls -l /dev/ command on the Pi to recognize which ports on the Pi were being utilized by the Arduinos. Once we had the appropriate paths to the Arduinos, we defined them in our python program.

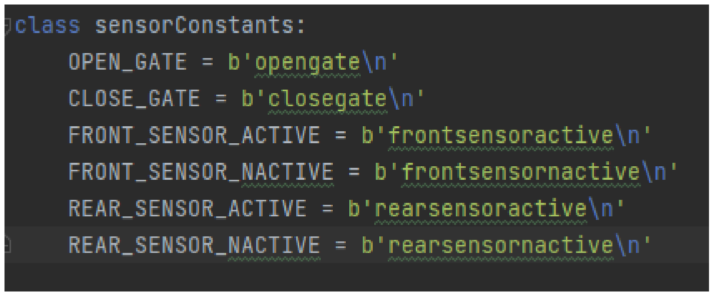


```
arduino_SENSORS = serial.Serial('/dev/ttyUSB3', 9600, timeout=1)  # define the sensor Arduino as a Serial object
arduino_DISPLAY = serial.Serial('/dev/ttyUSB2', 9600, timeout=1)  # define the display Arduino as a Serial object
arduino_SENSORS.reset_input_buffer()
arduino_DISPLAY.reset_input_buffer()
```

Now, we are able to reference and send information to each Arduino by calling their corresponding objects. We needed to define an API, with standard values that can be used to execute commands. We accomplished this by defining constants in our program and calling them whenever a system call needed to be called. Our constants are defined below:

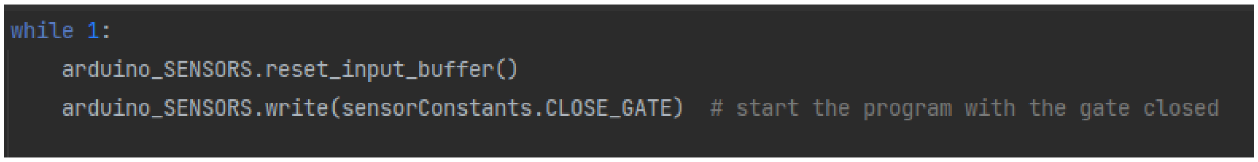


```python
class displayConstants:
    PULL_UP = b'pullup\n'
    CAR_DETECTED = b'cardetected\n'
    SHOW_QRCODE = b'showqrcode\n'
    PROCESSING = b'processing\n'
    ACCESS_GRANTED = b'granted\n'
    ACCESS_DENIED_FUNDS = b'deniedfunds\n'
    ACCESS_DENIED_CARD = b'deniedcard\n'
    NO_CARD_FOUND = b'nocard\n'
    DISPLAY_BALANCE = b'displaybalance\n'
    # current balance
```



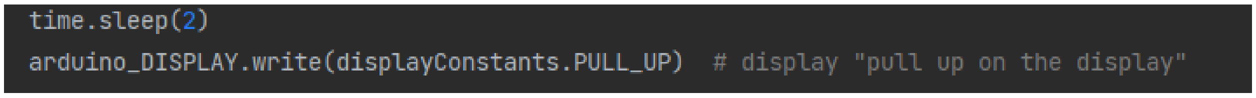
```python
class sensorConstants:
    OPEN_GATE = b'opengate\n'
    CLOSE_GATE = b'closegate\n'
    FRONT_SENSOR_ACTIVE = b'frontsensoractive\n'
    FRONT_SENSOR_NACTIVE = b'frontsensornactive\n'
    REAR_SENSOR_ACTIVE = b'rearsensoractive\n'
    REAR_SENSOR_NACTIVE = b'rearsensornactive\n'
```

When our program is run for the first time, we want to ensure that the gate is closed, so we send the following command, which will be interpreted by the Sensor Arduino and execute the corresponding instruction.



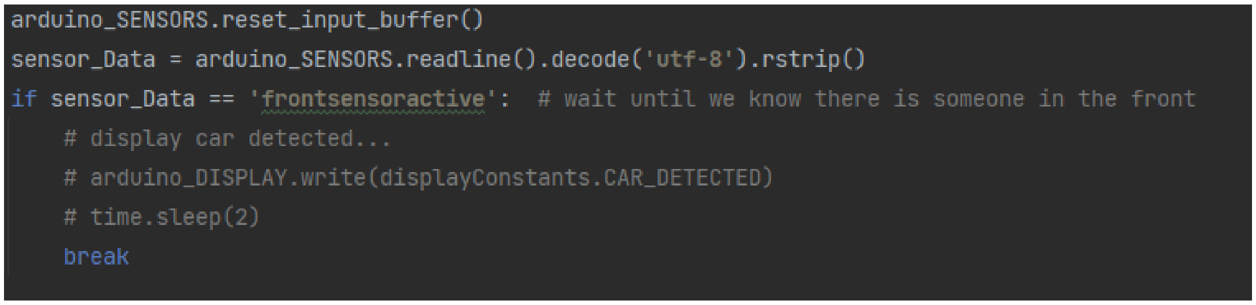
```python
while 1:
    arduino_SENSORS.reset_input_buffer()
    arduino_SENSORS.write(sensorConstants.CLOSE_GATE)  # start the program with the gate closed
```

Then, we want to display on the OLED that we are ready to detect when a car is in the critical area in front of the gate. So to program calls:



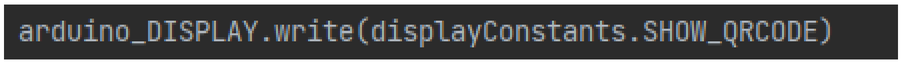
```python
time.sleep(2)
arduino_DISPLAY.write(displayConstants.PULL_UP)  # display "pull up on the display"
```

Our python program then waits for a message from the Sensor Arduino that says “frontsensoractive”, this will verify that there is a car waiting to validate access.



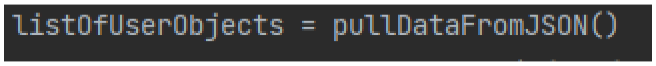
```
arduino_SENSORS.reset_input_buffer()
sensor_Data = arduino_SENSORS.readline().decode('utf-8').rstrip()
if sensor_Data == 'frontsensoractive':  # wait until we know there is someone in the front
    # display car detected...
    # arduino_DISPLAY.write(displayConstants.CAR_DETECTED)
    # time.sleep(2)
    break
```

When the data is equal to ‘frontsensoractive’ we break out of the loop of waiting for the message and then continue to execute the rest of the program. We then send the following command to the OLED Display which will indicate to the user to show a QR Code.

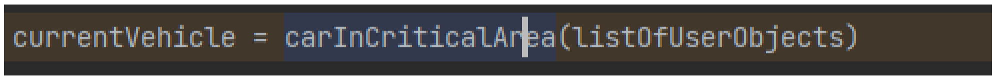


```
arduino_DISPLAY.write(displayConstants.SHOW_QRCODE)
```

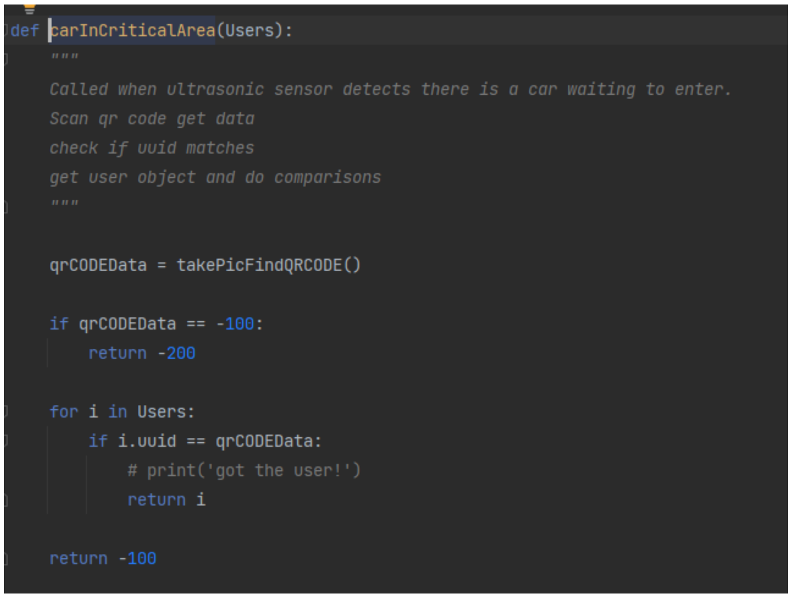
After that we call the carInCriticalArea() method which handles a lot of the logic for validating access. We pass in the listOfUser objects which is the serialized data corresponding to our predefined users. Called below:



```
listOfUserObjects = pullDataFromJSON()
```

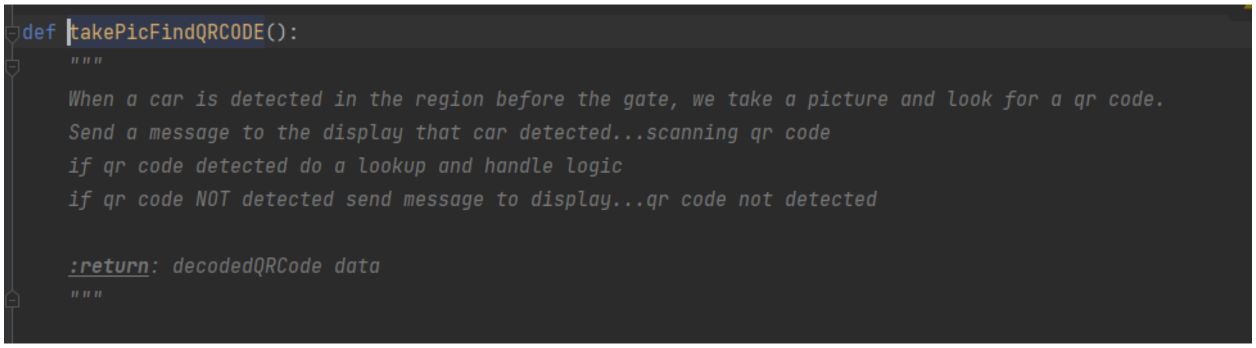


```
currentVehicle = carInCriticalArea(listOfUserObjects)
```



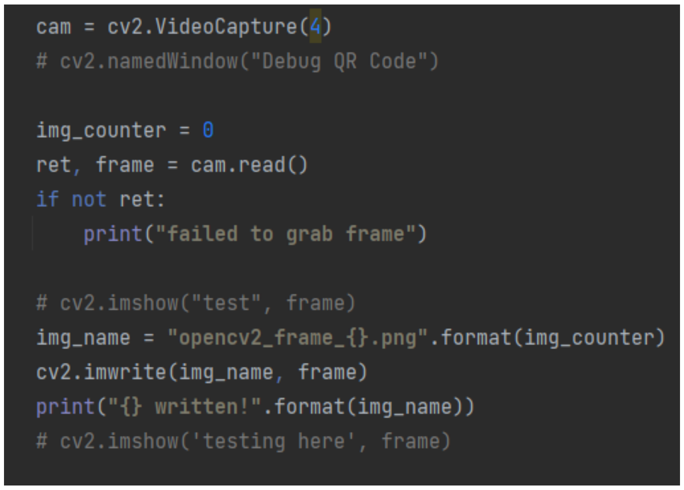
```
def carInCriticalArea(Users):
    """
    Called when ultrasonic sensor detects there is a car waiting to enter.
    Scan qr code get data
    check if uuid matches
    get user object and do comparisons
    """

    qrCODEData = takePicFindQRCODE()

    if qrCODEData == -100:
        return -200

    for i in Users:
        if i.uuid == qrCODEData:
            # print('got the user!')
            return i

    return -100
```

In this method we call takePicFindQRCODE() which does the following:



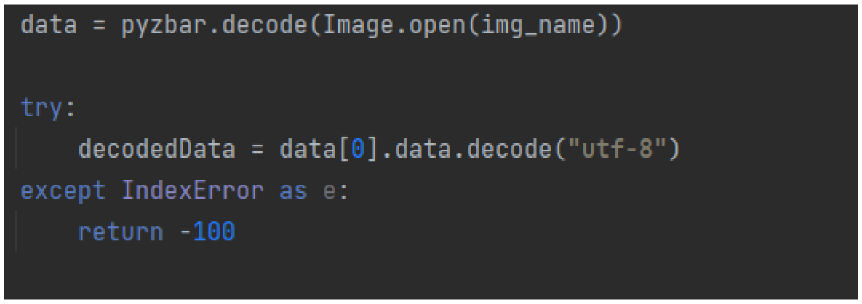
```
def takePicFindQRCODE():
    """
    When a car is detected in the region before the gate, we take a picture and look for a qr code.
    Send a message to the display that car detected...scanning qr code
    if qr code detected do a lookup and handle logic
    if qr code NOT detected send message to display...qr code not detected

    :return: decodedQRCode data
    """
```

Essentially, we take a picture and save it to the disk.



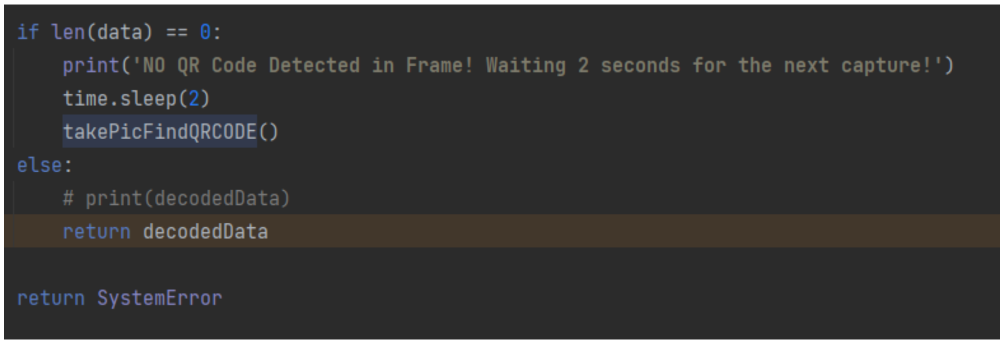
```
cam = cv2.VideoCapture(4)
# cv2.namedWindow("Debug QR Code")

img_counter = 0
ret, frame = cam.read()
if not ret:
    print("failed to grab frame")

# cv2.imshow("test", frame)
img_name = "opencv2_frame_{}.png".format(img_counter)
cv2.imwrite(img_name, frame)
print("{} written!".format(img_name))
# cv2.imshow('testing here', frame)
```

Then we take the same image and use the pyzbar library to decode the image and pull the data from the QR Code shown.



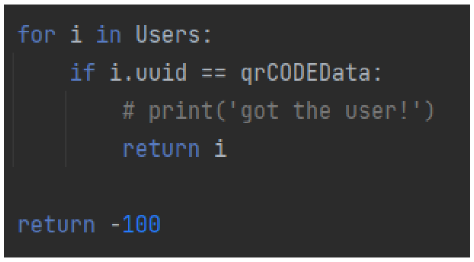
```
data = pyzbar.decode(Image.open(img_name))

try:
    decodedData = data[0].data.decode("utf-8")
except IndexError as e:
    return -100
```

We wrapped the .decode() method in a try except because during testing, if a QR Code did not return a value, our program would crash. To solve this we catch the exception and return a special error code that will be handled in our program in an appropriate manner. If there is data, we return it:



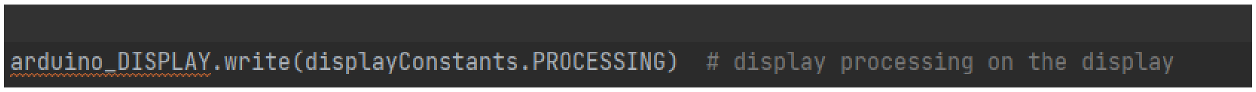
```python
if len(data) == 0:
    print('NO QR Code Detected in Frame! Waiting 2 seconds for the next capture!')
    time.sleep(2)
    takePicFindQRCODE()
else:
    # print(decodedData)
    return decodedData

return SystemError
```

Our data is then returned back to the calling method which is carInCriticalArea(). If there is no QR Code detected a 200 error code is returned back to main, else we want to pull the corresponding user object. We do this by searching the uuid's for a match. \

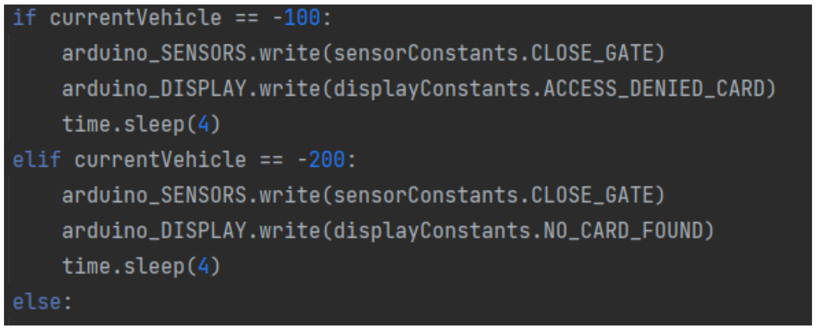


```python
for i in Users:
    if i.uuid == qrCODEData:
        # print('got the user!')
        return i

return -100
```

If there is a user found, we return the object, if not we return -100 to the main method where it will be handled further. While this is executing the OLED Display already received the command to display "Processing" on the display.



```python
arduino_DISPLAY.write(displayConstants.PROCESSING)  # display processing on the display
```

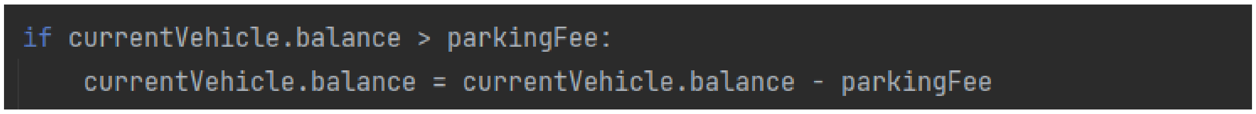


```python
if currentVehicle == -100:
    arduino_SENSORS.write(sensorConstants.CLOSE_GATE)
    arduino_DISPLAY.write(displayConstants.ACCESS_DENIED_CARD)
    time.sleep(4)
elif currentVehicle == -200:
    arduino_SENSORS.write(sensorConstants.CLOSE_GATE)
    arduino_DISPLAY.write(displayConstants.NO_CARD_FOUND)
    time.sleep(4)
else:
```

If a -100 is returned we know that there is not a corresponding user object in our system. So we close the gate and display that Access is Denied due to Card on the OLED.

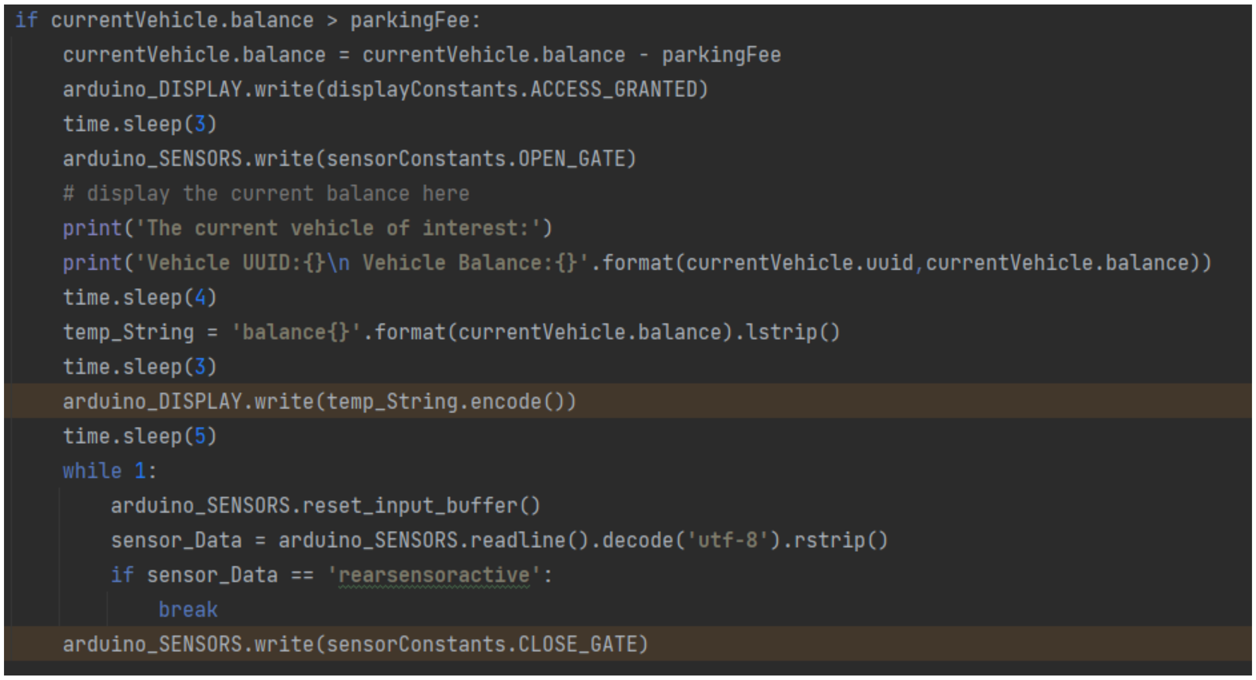
If a -200 is returned, we did not detect a card, so we close the gate and display “No Card Found”.

If a user object was returned from the calling method we can do our validation for entry. Which was only checking that the balance was sufficient for entry.



```
if currentVehicle.balance > parkingFee:
    currentVehicle.balance = currentVehicle.balance - parkingFee
```

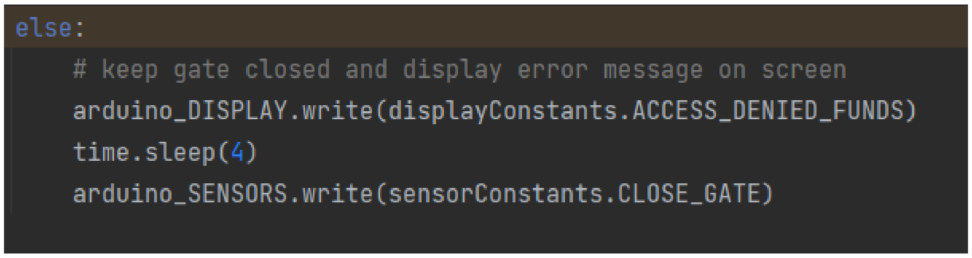
If the balance is good the following code is executed:



```
if currentVehicle.balance > parkingFee:
    currentVehicle.balance = currentVehicle.balance - parkingFee
    arduino_DISPLAY.write(displayConstants.ACCESS_GRANTED)
    time.sleep(3)
    arduino_SENSORS.write(sensorConstants.OPEN_GATE)
    # display the current balance here
    print('The current vehicle of interest:')
    print('Vehicle UUID:{}\n Vehicle Balance:{}'.format(currentVehicle.uuid,currentVehicle.balance))
    time.sleep(4)
    temp_String = 'balance{}'.format(currentVehicle.balance).lstrip()
    time.sleep(3)
    arduino_DISPLAY.write(temp_String.encode())
    time.sleep(5)
    while 1:
        arduino_SENSORS.reset_input_buffer()
        sensor_Data = arduino_SENSORS.readline().decode('utf-8').rstrip()
        if sensor_Data == 'rearsensoractive':
            break
    arduino_SENSORS.write(sensorConstants.CLOSE_GATE)
```

To summarize what happens, Access Granted is displayed on the display, the gate is opened, then we calculate the new balance and save it to the object. Once the new balance is updated, we send it to the OLED display. Finally, once all that is complete, we now want to wait for the event that the car is active at the rear sensor, indicating that the car pulled past the gate, and we want to close it. Just like when we checked if the car was in front, we do the same subroutine but now look for the keyword ‘rearsensoractive’. Once the program knows the rear sensor is active, we can close the gate.

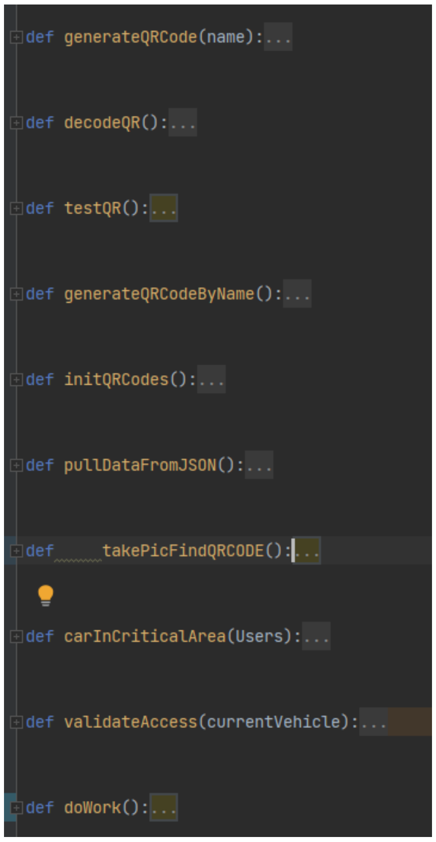
To handle the case that the funds are not sufficient, we display “Access denied insufficient funds” on the display, and close the gate.



```
else:
    # keep gate closed and display error message on screen
    arduino_DISPLAY.write(displayConstants.ACCESS_DENIED_FUNDS)
    time.sleep(4)
    arduino_SENSORS.write(sensorConstants.CLOSE_GATE)
```

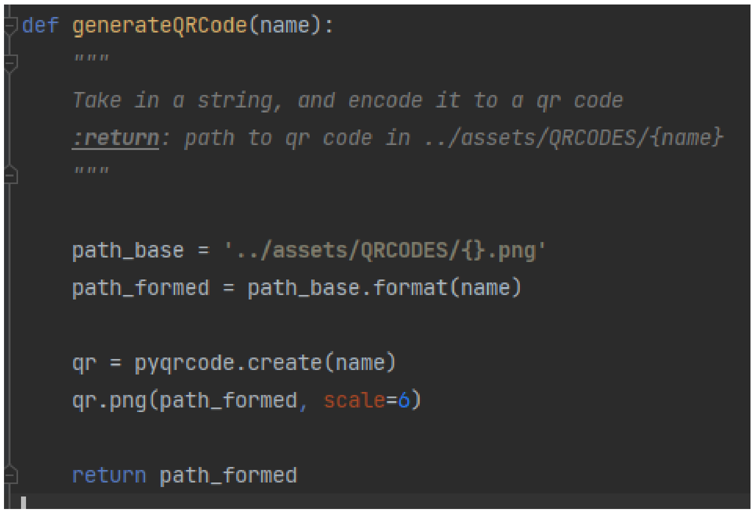
This handles all the logic of the python program, and once execution for any of the previous cases finishes. We return to the top of the method and do it all over again.

To assist in our development and testing Alex wrote many test methods before integrating the code in the main functions. The headers of these functions can be seen below:



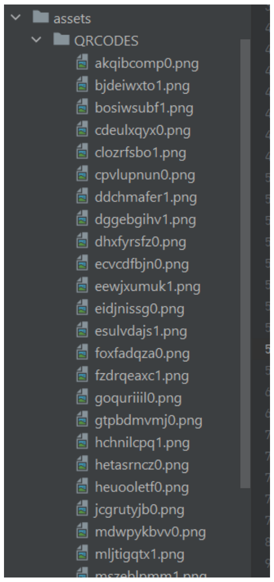
```
def generateQRCode(name):...

def decodeQR():...

def testQR():...

def generateQRCodeByName():...

def initQRCodes():...

def pullDataFromJSON():...

def     takePicFindQRCODE():...

def carInCriticalArea(Users):...

def validateAccess(currentVehicle):...

def doWork():...
```

A lot of these functions were used to generate our initial User objects and create their corresponding QR Codes and instance data. Our QR Code generation and saving was handled in the following function:



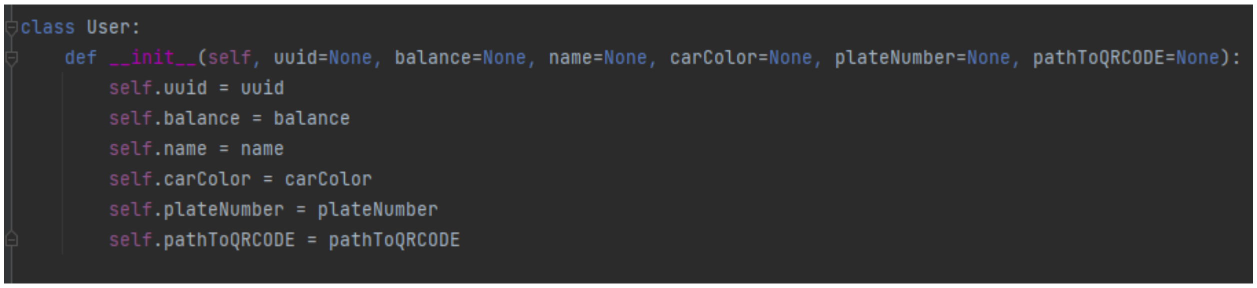
```
def generateQRCode(name):
    """
    Take in a string, and encode it to a qr code
    :return: path to qr code in ../assets/QRCODES/{name}
    """

    path_base = '../assets/QRCODES/{}.png'
    path_formed = path_base.format(name)

    qr = pyqrcode.create(name)
    qr.png(path_formed, scale=6)

    return path_formed
```

To show some of the codes that were generated in the respective directory and an example QR Code:



```
assets
  QRCODES
    akqibcomp0.png
    bjdeiwxto1.png
    bosiwsubf1.png
    cdeulxqyx0.png
    clozrfsbo1.png
    cpvlupnun0.png
    ddchmafer1.png
    dggebgihv1.png
    dhxfyrsfz0.png
    ecvcdfbjn0.png
    eewjxumuk1.png
    eidjnissg0.png
    esulvdajs1.png
    foxfadqza0.png
    fzdrqeaxc1.png
    goquriiil0.png
    gtpbdmvmj0.png
    hchnilcpq1.png
    hetasrncz0.png
    heuooletf0.png
    jcgrutyjb0.png
    mdwpykbvv0.png
    mljtigqtx1.png
    mszeblpmm1.png
```

Each User object that we generated in our program was an instance of the User.py object, following the below schema:



```python
class User:
    def __init__(self, uuid=None, balance=None, name=None, carColor=None, plateNumber=None, pathToQRCODE=None):
        self.uuid = uuid
        self.balance = balance
        self.name = name
        self.carColor = carColor
        self.plateNumber = plateNumber
        self.pathToQRCODE = pathToQRCODE
```

When it came time to run the code and integrate everything we could run the code directly from the terminal using the command $ sudo python main.py. But, more frequently we would remote debug the code to see what was happening at each step of the program.
All the code can be found in the Appendix section as well as on our project Github.

Ethical Justification

We decided to work on this project, as we found that it could be a great way to reduce person-to-person contact, as a measure to increase biosecurity during the COVID-19 pandemic.

Furthermore, we believe that this system will reduce lines at tax-toll booths, as the process to charge a tax and validate entrance will be fully automated. As the process is automated, there will be no space for issues such as conflicts of interest, bribery, and will ensure that all individuals that go through the toll booth are treated equally and fairly, all of which uphold clauses 1 and 2 of the IEEE Code of Ethics. This project, however, does offer some risks and pitfalls. Amongst them, is the elimination of tax toll booth worker jobs, as well as a large opportunity for cybersecurity threats. To mitigate these risks, we must vow to upkeep the data in our system as secure as possible, creating jobs in the maintenance of our systems. Furthermore, These automatic toll booths do not need to fully replace the classic tax-toll booth system, as we have found that a hybrid system, where there are some automatic booths as well as some manual ones, would work best.

Assessment of Experimental Results

Throughout this project, key milestones were met that came together to represent the final functionality of the design. Because we broke the larger project into smaller bite-sized experiments, we had results periodically throughout the process, as well as at the end. As each step was completed and challenges were overcome, a new level of operability was reached. These milestone ‘checkpoints’ included the configuration of the programming environment and printing to the console, demonstrating the working IR sensor, generation of custom QR codes, displaying to the OLED, sending packets, gate logic. Finally we could piece these independently working parts together to demonstrate the finished project, the different modes expressed in the proposal, and the expected actions taking place.

The first results we were able to obtain were seeing that the RockPi was functional following the flashing of the OS on the device, the file structure being configured, and remotely connecting to the devices network using SSH. The results of this step were printing “Hello World” using code written on our personal computer, and pushing using git. This was fairly straightforward and we were able to accomplish this rather quickly. The implications of this result are that we could then use this functionality to print out our color data in future steps, and gain some experience interacting with the new environment we were programming in.

The next result we could obtain was interfacing the IR sensor and outputing to the console when it was triggered. Though this is a very simple interaction, the impact of this specific result was that we could then save the code and wiring configuration for when we needed the sensor to detect a vehicle. We also got an idea of the range and sensitivity of the sensor during this step, which we took into consideration when designing the housing.

The next checkpoint we received feedback and results from was generating and displaying custom QR codes. This step was crucial to our final implementation because it was the only way to let the user interface with the program and hardware. The QR codes needed to be accurate in order for the database to recognize the UUID that was embedded in the image. The implications of this result was giving us the ability to generate these unique codes when we had the database structure completed, and knew which user would have which attributes.

The next result we could obtain was interfacing the OLED display and printing ‘Hello World’ to it. Though this is a very simple interaction, the impact of this specific result was that we could then save the code and wiring configuration for when we needed the screen to display a specific command on demand. We also got an idea of what we would need to continue to develop in terms of functions for unifying formatting.

The next checkpoint we received feedback and results from was sending packets over the serial connection between the Arduino and the RockPi. This step was critical to our final implementation because it was the only way to let the RockPi connect to the sensors, servo, and display. These packets could trigger specific actions in the Arduino code, opening the gate, showing messages on the screen, and telling the RockPi what the sensors were seeing. The implication of this result was giving us the ability to carry out actions that we had defined in our program logic, and give real-world responses to different inputs and conditions.

The final checkpoint before everything came together was working out the different states that were necessary to mimic the expected interaction. This then was demonstrated as the user having the ability to open the gate (or not) depending on their account balance. The implications of this result was that we could verify that we had configured the logic correctly. This was very beneficial because we had encountered trouble at first with the user experience not feeling realistic, so with some modification and testing, it was improved.

The final project implementation yielded the results of allowing the user to ‘drive up’ to the gate, which led to them being detected by the front IR sensor. This would prompt them on the display to present their QR code to the camera. Once a valid card was detected, if the balance was sufficient, the gate would open and remain open for them to pass through. Once the back sensor had determined that the vehicle had cleared the gate sufficiently, it lowered and reset for the next customer. If the balance was insufficient, the card was invalid, or no card was detected, then no access was granted. The user is able to experience the program logic in a way that is familiar to someone who has used a parking or toll system before, which speaks to our results being useful and accurate.

Another significant finding of the final results was that we had finally demonstrated that each piece could communicate successfully in a comprehensive system. Just because the RockPi or Arduino had communication to each part at one point or another didn’t necessarily mean that they were able to talk to each other. We encountered multiple roadblocks on the way and realized that proofreading code, testing components, and verifying the hardware connections is our best bet to figure out the problem. The problem solving process of our projects truly showed us a practical experience of being an engineer and we believe that these skills and understanding would make us much better engineers in the industries. By completing the web of interaction, our mobile communication embedded system was able to operate as a freestanding entity once it had been run, and worked as intended.

Conclusion

For our Junior Design Project, we put together the knowledge we gained from CSE 398 throughout the semester, and came up with a fairly usable product. Our automated tax-toll booth contained Serial Communication between devices, image processing using cv2, and distance detection, servo control, and OLED screen control using arduino libraries. Furthermore, this project required a lot of collaboration and planning amongst our team members. We divided the project evenly amongst the three of us, and worked on small side-projects before bringing them all together to make our final product, which somewhat emulated a system similar to that seen at an engineering workspace. We used tools such as lucidcharts to align on our final project structure, and plan ahead before working on our parts individually. Overall, not only did we learn more about embedded systems while constructing our Junior Design Project, but we also gained project planning and team management skills.

Appendix

Link to source code pseudocode/lucidchart diagram:
https://lucid.app/lucidchart/92d51b25-893f-4943-9498-f3f76931d465/edit?invitationId=inv_43692390-9eff-4bfd-b6f6-cf48e48ea0ed&page=ba.a2y4UDmem#

Link to Github:

https://github.com/iamapez/CSE398-JRDESIGN

Rock Pi source directory:

https://github.com/iamapez/CSE398-JRDESIGN/tree/demoFeatures/srcPI

Rock Pi main python program:

https://github.com/iamapez/CSE398-JRDESIGN/blob/demoFeatures/srcPI/main.py

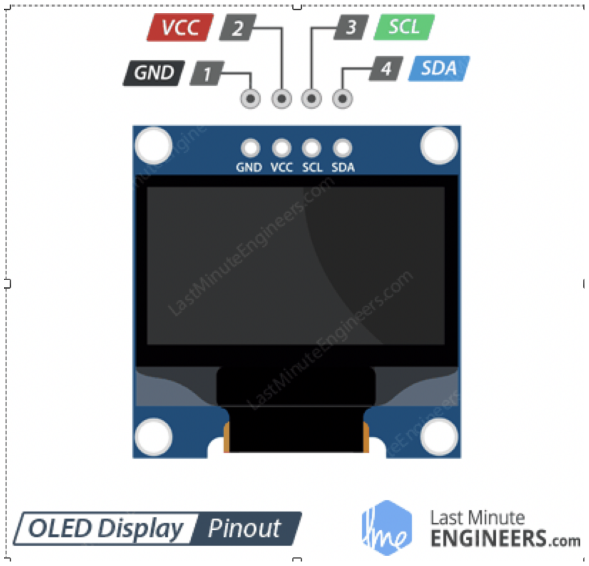
Adafruit SSD1306 Library:

https://github.com/adafruit/Adafruit_SSD1306

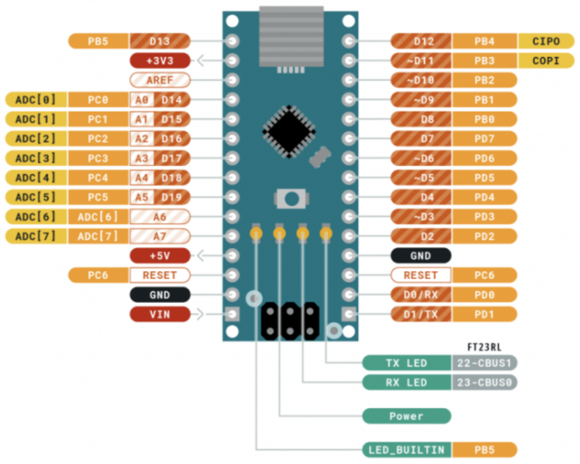
Ethical Justification guide:

https://www.scu.edu/media/school-of-engineering/pdfs/current-student-resources/undergraduate/Senior-Design-Ethics-Guide-1.pdf

OLED screen pinout



Arduino Nano pinout:


PB5
D13
+3V3
AREF
ADC[0] PC0 A0 D14
ADC[1] PC1 A1 D15
ADC[2] PC2 A2 D16
ADC[3] PC3 A3 D17
ADC[4] PC4 A4 D18
ADC[5] PC5 A5 D19
ADC[6] ADC[6] A6
ADC[7] ADC[7] A7
+5V
PC6 RESET
GND
VIN
D12 PB4 CIPO
~D11 PB3 COPI
~D10 PB2
~D9 PB1
D8 PB0
D7 PD7
~D6 PD6
~D5 PD5
D4 PD4
~D3 PD3
D2 PD2
GND
RESET PC6
D0/RX PD0
D1/TX PD1
FT23RL
TX LED 22-CBUS1
RX LED 23-CBUS0
Power
LED_BUILTIN PB5