Richard DeAmicis
64686507

3Feet Term paper

GOAL

The goal of this project is to identify vehicles that come within 3 feet of a bicyclist, which is against the law in California and many other states. A sensor device is connected to the bicycle, which does the distance sensing. 3 feet violations are stored in the cloud. End users can use this data to better plan routes through their city. City planners and consulting firms can use the data to target areas of the city for infrastructure improvements.

MATERIALS LIST

- Raspberry Pi 4B (1)
- TF Mini Plus LiDAR time-of-flight sensor (1)
- PiSugar Plus external battery (1)
- MIT App Inventor app running on Android phone
- Firebase database
- Device container
- Jumper wires (8)
- Breadboard

TF Mini Plus

Sensing accuracy and device robustness is critical in a mobile project like this. In testing, the TF Mini Plus proved capable of meeting the project demands. The sensor comes with a robust outer shell that protects it from dirt and debris. The sensor itself is extremely reliable and fast. Speed tests indicate the device can take successive measurements in less than 10ms per reading. A vehicle moving 40 mph faster than a bicycle would take approximately 250ms - 300ms to pass the sensor. Thus, the device is capable of taking as many as 25 measurements in this time, which is more than enough to get an accurate reading. The average vehicle is 15 feet long (180 inches).

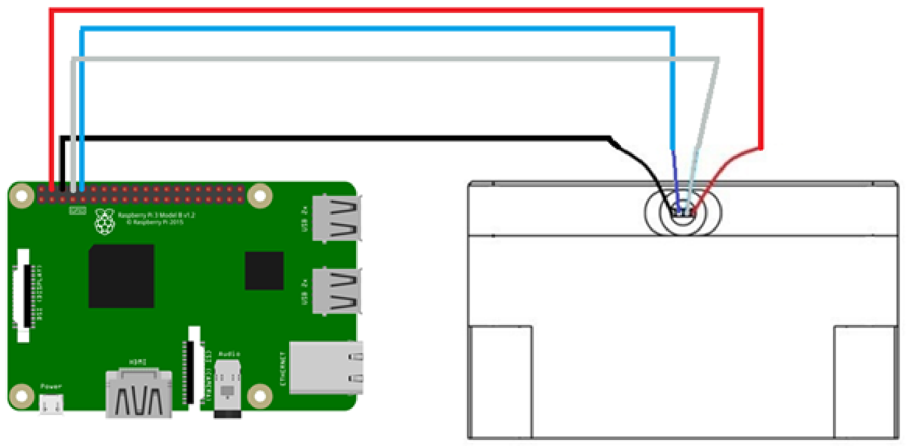
40 M/H = 40 * 63,360 inches / 3,600,000 ms = 0.704 inch per ms
180 inches / 0.704 inch per ms = 255 ms

Even if the vehicle was traveling significantly faster than the bicycle (more than 40mph faster), the device will still be able to grab a reading. As explained later, the device requires 10 successive readings of less than 3 feet to determine a valid violation. Given the above formula, a vehicle could be traveling 100 mph faster than the bicycle, and the sensor could still pick up a 3ft violation.

THE SENSOR DEVICE

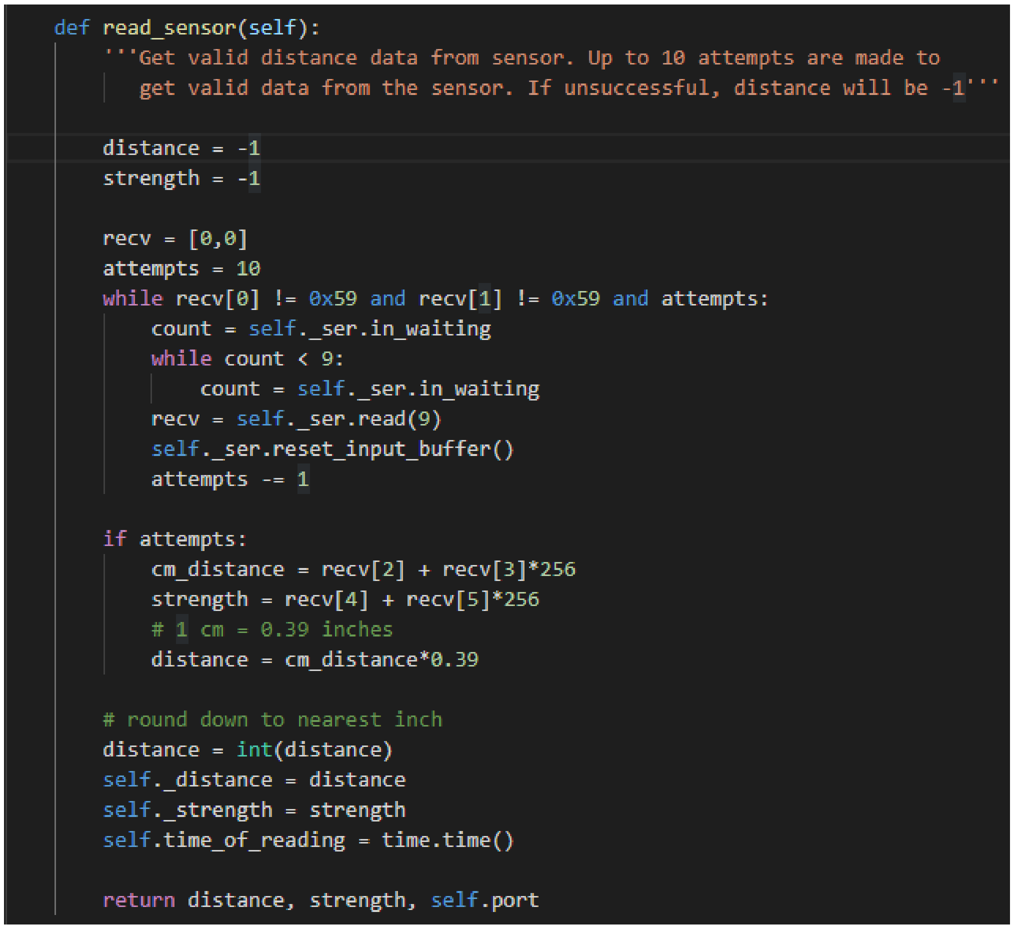
SENSOR TO RPI

The entire device doing the sensing consists of a Raspberry Pi 4B and a TF Mini Plus. The sensor is connected to the Raspberry Pi via UART serial communication. One issue with using this protocol is the Bluetooth module also uses the UART. This is why the Raspberry Pi 4B is so important. The 4B has 5 fully functional PL011 UARTs. Previous models only have (1) PL011 and one mini-UART. This project requires the full functionality of the PL011 for both the bluetooth module and the sensor, so the 4B is vital. More information on this topic can be found in the challenges section.



Unlike the picture above, I chose to use the UART located on pins 8 and 9. These pins are initially assigned to the SPI protocol, so I reassigned it by adding `dtoverlay=uart4` to my `/boot/config.txt` file.

I then created a python file called `tfminiplus.py` that contains a TFMini class. This class is used to create a python object that can read the sensor. The sensor uses the python `Serial` package to read in serial data from the sensor. The main function of interest is the `read_sensor()` function:



```python
def read_sensor(self):
    '''Get valid distance data from sensor. Up to 10 attempts are made to
       get valid data from the sensor. If unsuccessful, distance will be -1'''

    distance = -1
    strength = -1

    recv = [0,0]
    attempts = 10
    while recv[0] != 0x59 and recv[1] != 0x59 and attempts:
        count = self._ser.in_waiting
        while count < 9:
            count = self._ser.in_waiting
        recv = self._ser.read(9)
        self._ser.reset_input_buffer()
        attempts -= 1

    if attempts:
        cm_distance = recv[2] + recv[3]*256
        strength = recv[4] + recv[5]*256
        # 1 cm = 0.39 inches
        distance = cm_distance*0.39

    # round down to nearest inch
    distance = int(distance)
    self._distance = distance
    self._strength = strength
    self.time_of_reading = time.time()

    return distance, strength, self.port
```

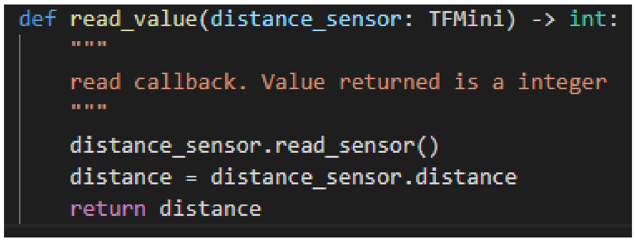
This function obfuscates away a few common errors that occur when reading the sensor. The sensor should return at least 9 bytes, so the program continues to grab data from the sensor until it receives a package of at least 9 bytes.

Errors can still occur after a least 9 bytes are received from the sensor. According to the TF Mini's documentation, If the data is valid, then the first two bytes should be `0x59`. The `while` loop makes 10 attempts to read valid data from the sensor. Once valid data is received, the distance can be calculated.

According to the sensor's specifications, the distance data is located in the 3rd and 4th byte position. This is a 16 bit number, so the value stored in the 4th byte must be shifted to the left by one byte. This is done by multiplying the value by 256. The value stored in the 3rd byte is then added to this number to create the complete distance value. The same is done for the signal strength reading.

Note: the signal strength reading is not used in this project.

Finally, the `distance`, `strength`, and `time_of_reading` are all stored as internal variables. The object can then be used as such:



```
def read_value(distance_sensor: TFMini) -> int:
    """
    read callback. Value returned is a integer
    """
    distance_sensor.read_sensor()
    distance = distance_sensor.distance
    return distance
```

The actual code can be viewed on github @ https://github.com/rdeamici/3Feet-IoT.

RPi TO BLE

The raspberry pi is responsible for using the sensor to detect 3feet violations. It does this with a monitoring service called `threeFtMonitor.py`. This file essentially turns the Raspberry Pi into a BLE peripheral device. The python package `bluezero` is used to do most of the heavy lifting of creating a peripheral device and advertising data. The threeFtMonitor peripheral is instantiated with notify functionality only. The main function of the threeFtMonitor.py script is `update_value()`.

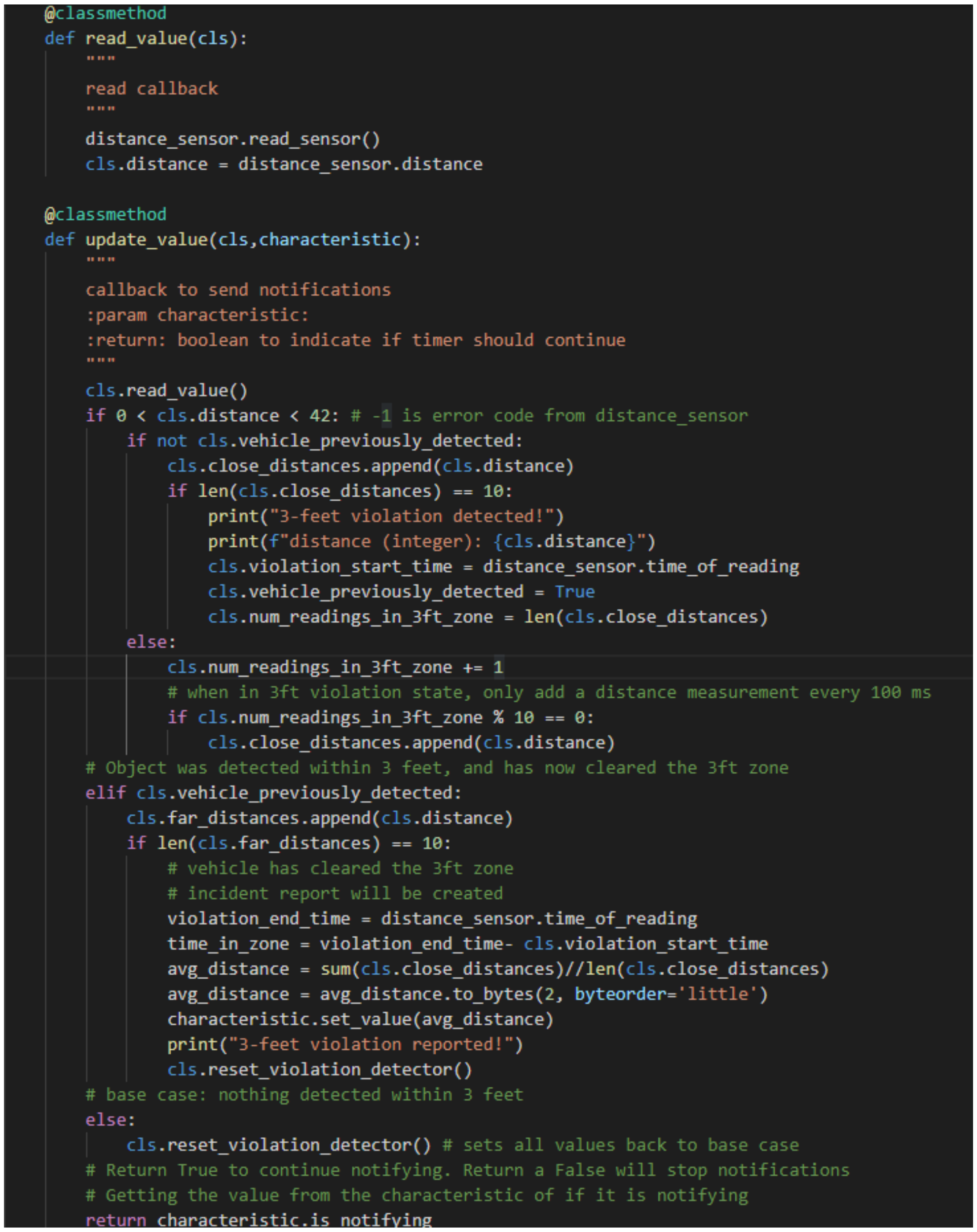
The `update_value()` function uses the `tfminiplus.py` file to read a distance value from the TF- Mini sensor. This function is run every 10 ms, which is the time it takes to read the sensor. The function has 3 states:

1. Base case: no object within 3.5 feet has been detected.

2. Object detected within 3.5 feet: The threeFtMonitor stays in this state until 10 consecutive distance readings come back with a value less than 3.5 feet. If at any time after the first 3.5 foot violation is detected, a reading comes back that is more than 3.5 feet, then the system reverts back to state 1. Once 10 consecutive readings less than 3.5 ft are recorded, the `vehicle_previously_detected` flag is set to `True`. The device stays in state 2 until 10 consecutive readings of more than 3.5 feet are observed.

3. No object is currently detected within 3.5 feet, and `vehicle_previously_detected` flag is `True`. When the device enters this state, it means a vehicle was detected within 3 feet of the bicycle, and has now cleared the 3 feet zone. This is the state where the device sends a BLE notification with the distance reading. While the device is in state 2, it records all the distance readings. In state 3, the average of all the distance readings is taken and broadcast out via BLE to the connected cell phone. This is why the threshold for detecting a 3 feet violation is 3.5 feet and not 3 feet. We don't want to miss

vehicles that hover right at the 3 feet threshold. After the 3 feet violation notification is broadcast, the device resets back to state 1.

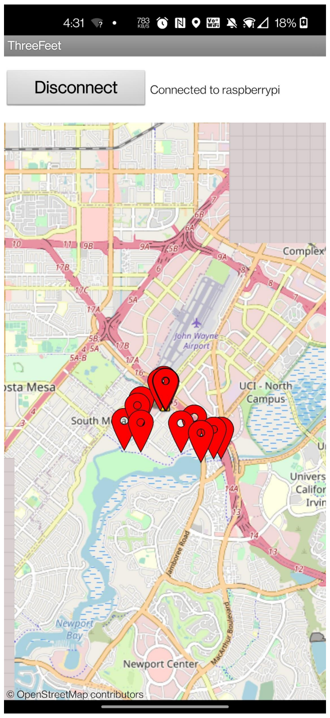


```
@classmethod
def read_value(cls):
    """
    read callback
    """
    distance_sensor.read_sensor()
    cls.distance = distance_sensor.distance

@classmethod
def update_value(cls,characteristic):
    """
    callback to send notifications
    :param characteristic:
    :return: boolean to indicate if timer should continue
    """
    cls.read_value()
    if 0 < cls.distance < 42: # -1 is error code from distance_sensor
        if not cls.vehicle_previously_detected:
            cls.close_distances.append(cls.distance)
            if len(cls.close_distances) == 10:
                print("3-feet violation detected!")
                print(f"distance (integer): {cls.distance}")
                cls.violation_start_time = distance_sensor.time_of_reading
                cls.vehicle_previously_detected = True
                cls.num_readings_in_3ft_zone = len(cls.close_distances)
        else:
            cls.num_readings_in_3ft_zone += 1
            # when in 3ft violation state, only add a distance measurement every 100 ms
            if cls.num_readings_in_3ft_zone % 10 == 0:
                cls.close_distances.append(cls.distance)
    # Object was detected within 3 feet, and has now cleared the 3ft zone
    elif cls.vehicle_previously_detected:
        cls.far_distances.append(cls.distance)
        if len(cls.far_distances) == 10:
            # vehicle has cleared the 3ft zone
            # incident report will be created
            violation_end_time = distance_sensor.time_of_reading
            time_in_zone = violation_end_time- cls.violation_start_time
            avg_distance = sum(cls.close_distances)//len(cls.close_distances)
            avg_distance = avg_distance.to_bytes(2, byteorder='little')
            characteristic.set_value(avg_distance)
            print("3-feet violation reported!")
            cls.reset_violation_detector()
    # base case: nothing detected within 3 feet
    else:
        cls.reset_violation_detector() # sets all values back to base case
    # Return True to continue notifying. Return a False will stop notifications
    # Getting the value from the characteristic of if it is notifying
    return characteristic.is_notifying
```

The `threeFeetMonitor.py` file is run as a `systemctl` service, which means it is always on and always running as long as the raspberry pi machine is on. Once a cell phone app is connected to the raspberry pi, then BLE notifications are turned on and the `update_value()` is called every 10 ms. When the device disconnects, the rpi stops sending BLE notifications.

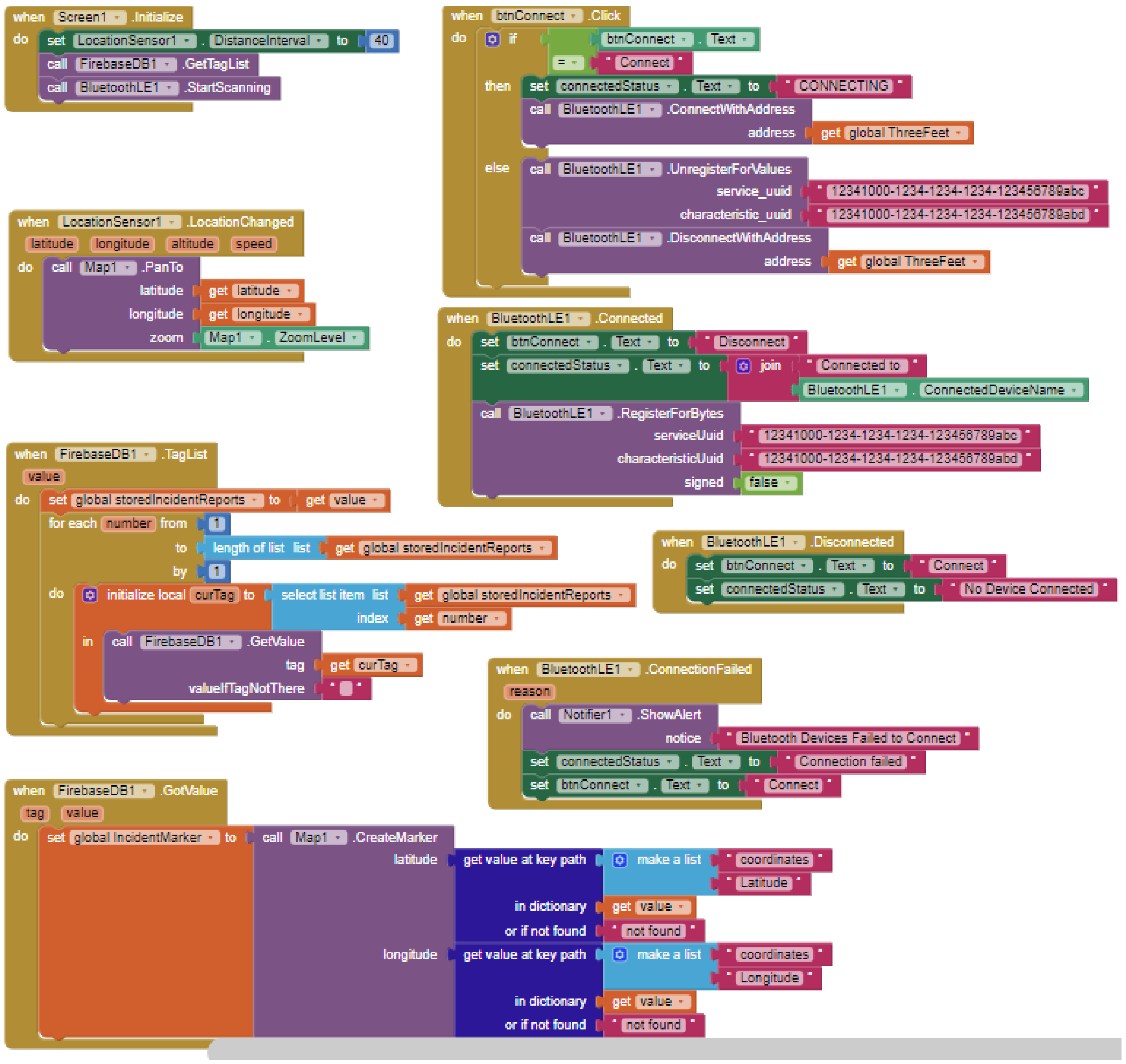
ANDROID APP

I used the MIT App Inventor free android application building software. This software is GUI based and allows for quick prototyping. It is a little finicky, but overall works great. The app is pretty basic. It consists of a button for connecting and disconnecting the cell phone to/from the raspberry pi via BLE. The MAC address of the raspberry-pi is hard coded in the app to avoid having to go through a setup phase. Next to the button is a string that represents the state of the BLE connection. Possible values are: `“No Device Connected”`,`”CONNECTING”`, and `“Connected to <device name>”`. The rest of the screen is used to show a map. The map is populated with markers that indicate locations of 3 feet violations.



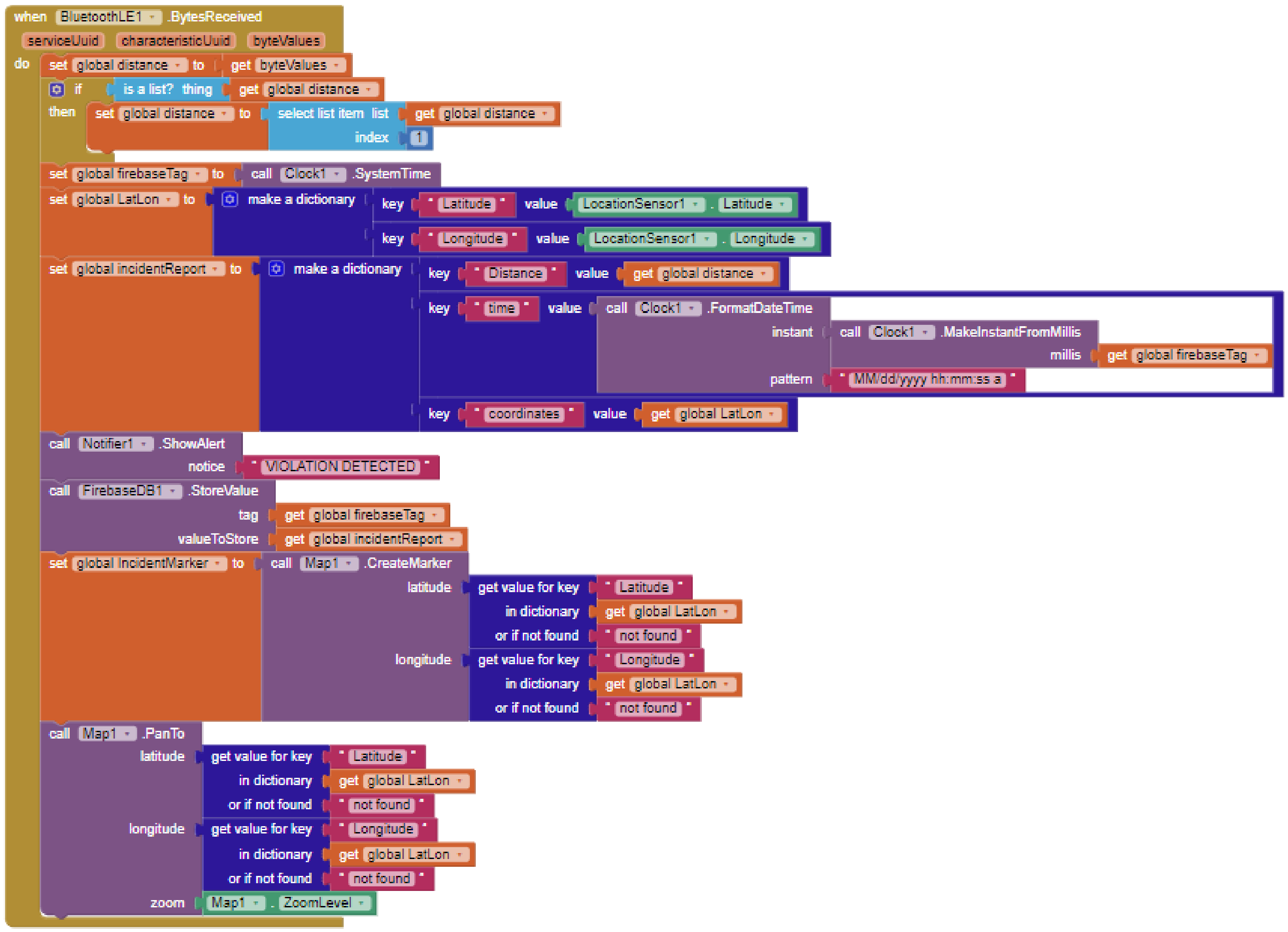
When the user first opens the map, the cell-phone starts scanning for bluetooth devices. It also downloads data from the firebase cloud and creates a marker for each coordinate that exists in the database. Once the user clicks the `Connect` button, the cell phone attempts to connect to the raspberry pi. If the connection is successful, the text on the button changes from `“Connect”` to `“Disconnect”`.

The following screenshot shows all the blocks that trigger actions on the mobile app.



```
when Screen1 .Initialize
do  set LocationSensor1 . DistanceInterval to 40
    call FirebaseDB1 .GetTagList
    call BluetoothLE1 .StartScanning

when LocationSensor1 .LocationChanged
  latitude  longitude  altitude  speed
do  call Map1 .PanTo
        latitude   get latitude
        longitude  get longitude
        zoom       Map1 . ZoomLevel

when FirebaseDB1 .TagList
  value
do  set global storedIncidentReports to get value
    for each number from 1
        to length of list list get global storedIncidentReports
        by 1
    do  initialize local curTag to select list item list get global storedIncidentReports
                                                 index get number
        in  call FirebaseDB1 .GetValue
                tag                get curTag
                valueIfTagNotThere " "

when FirebaseDB1 .GotValue
  tag  value
do  set global IncidentMarker to call Map1 .CreateMarker
        latitude   get value at key path  make a list " coordinates "
                                                      " Latitude "
                   in dictionary  get value
                   or if not found " not found "
        longitude  get value at key path  make a list " coordinates "
                                                      " Longitude "
                   in dictionary  get value
                   or if not found " not found "

when btnConnect .Click
do  if  btnConnect . Text  =  " Connect "
    then  set connectedStatus . Text to " CONNECTING "
          call BluetoothLE1 .ConnectWithAddress
              address  get global ThreeFeet
    else  call BluetoothLE1 .UnregisterForValues
              service_uid         " 12341000-1234-1234-1234-123456789abc "
              characteristic_uid  " 12341000-1234-1234-1234-123456789abd "
          call BluetoothLE1 .DisconnectWithAddress
              address  get global ThreeFeet

when BluetoothLE1 .Connected
do  set btnConnect . Text to " Disconnect "
    set connectedStatus . Text to join " Connected to "
                                       BluetoothLE1 . ConnectedDeviceName
    call BluetoothLE1 .RegisterForBytes
        serviceUuid         " 12341000-1234-1234-1234-123456789abc "
        characteristicUuid  " 12341000-1234-1234-1234-123456789abd "
        signed              false

when BluetoothLE1 .Disconnected
do  set btnConnect . Text to " Connect "
    set connectedStatus . Text to " No Device Connected "

when BluetoothLE1 .ConnectionFailed
  reason
do  call Notifier1 .ShowAlert
        notice  " Bluetooth Devices Failed to Connect "
    set connectedStatus . Text to " Connection failed "
    set btnConnect . Text to " Connect "
```

Nothing too fancy going on here. The action really happened after the block in the middle of the screenshot above that is labeled `when BuetoothLE1.Connected`. This block causes the bluetooth to send a signal to the peripheral device to start notifying. When data is received back the following block is triggered



```
when BluetoothLE1 .BytesReceived
  serviceUuid  characteristicUuid  byteValues
do  set global distance to  get byteValues
    if  is a list? thing  get global distance
    then  set global distance to  select list item  list  get global distance
                                                    index  1
    set global firebaseTag to  call Clock1 .SystemTime
    set global LatLon to  make a dictionary  key " Latitude "  value  LocationSensor1 . Latitude
                                             key " Longitude "  value  LocationSensor1 . Longitude
    set global incidentReport to  make a dictionary  key " Distance "  value  get global distance
                                                     key " time "  value  call Clock1 .FormatDateTime
                                                                             instant  call Clock1 .MakeInstantFromMillis
                                                                                                   millis  get global firebaseTag
                                                                             pattern  " MM/dd/yyyy hh:mm:ss a "
                                                     key " coordinates "  value  get global LatLon
    call Notifier1 .ShowAlert
                    notice  " VIOLATION DETECTED "
    call FirebaseDB1 .StoreValue
                      tag  get global firebaseTag
                      valueToStore  get global incidentReport
    set global IncidentMarker to  call Map1 .CreateMarker
                                            latitude  get value for key  " Latitude "
                                                      in dictionary  get global LatLon
                                                      or if not found  " not found "
                                            longitude  get value for key  " Longitude "
                                                       in dictionary  get global LatLon
                                                       or if not found  " not found "
    call Map1 .PanTo
               latitude  get value for key  " Latitude "
                         in dictionary  get global LatLon
                         or if not found  " not found "
               longitude  get value for key  " Longitude "
                          in dictionary  get global LatLon
                          or if not found  " not found "
               zoom  Map1 . ZoomLevel
```

This block is where the 3 feet incident report is created and uploaded to Firebase. The time is derived from the cell phone itself. This is also used as the unique identifier for entry into the database. The coordinates are also taken from the cell phone. The distance is taken from the BLE notification that triggered this block. These three data points are all stored as a json string and uploaded to firebase. This block is also where the notification is sent to let the user know a 3 feet violation occurred. In addition to a temporary pop up, this block causes a permanent marker to be placed on the map.

CHALLENGES

Bluetooth is notoriously difficult on a linux machine. The `blueZ` software that is used to interface with the bluetooth device is complicated and poorly documented. The `bluezero` python package made it significantly easier, but still not super easy. In hindsight, I wish I had connected the sensor to an arduino BLE board, as there seems to be more tutorials out there for the Arduino.

I chose to use the raspberry pi because I wanted to be able to store the 3 feet violations locally, but I ran out of time to implement this functionality. Furthermore I wanted to connect a second Raspberry Pi/TF Mini sensor combination to the front of the bicycle so I could detect direction and speed of passing objects. However, this too proved to be too complicated to implement in the limited time I had. I only had a Raspberry Pi 3B+, and this device only comes with 2 UARTS, a single PL011, and a mini-UART. The PL011 is used by the bluetooth module, which means all that is left for the sensor is the mini-UART. While I was able to get this to work, I was not confident it would hold up for long, as everything I have read about the mini-UART is it is quite finicky.

Furthermore, I was not able to successfully connect the cell phone to the raspberry pi 3B+ and send sensor data over BLE. I'm not sure why, but I suspect it either has something to do with the UARTs or with the `bluezero` python package I was using. I suspect it was not created to run on two different devices simultaneously. I used a pre-made app by Nordic Semiconductors to test connecting and sending data over BLE. When using this app, I saw the service and characteristic UUIDs were showing up as different values on the app compared to what I had hard coded on the raspberry pi 3B+. They were in fact the same UUIDs that were hard-coded on the Raspberry Pi 4B that was running at the same time..

All the "intelligence" in this project is currently in the raspberry pi's ability to correctly decipher 3 feet violations. My goal was to use the data stored in the cloud to pre-emptively notify users as they ride through a city of upcoming danger zones, but I was not able to implement that in time. I also wanted to create different colored pins depending on a number of different factors such as speed of the vehicle passing and length of time spent within the 3-feet zone, but again, I ran out of time.

I might have been able to get one or two of the above features implemented, but I lost 2 days due to my cat knocking my raspberry pi off my desk, which fried my SD card. This was sadly my only SD card, so I had to rush order some more in 2-day delivery because I couldn't find any locally. Not quite as silly as saying "my dog ate my homework" but pretty silly nonetheless. In all honesty, this mishap didn't really set me back that much, just caused a little more anxiety. Just part of the joys of working with these maker tools.

VIDEOS

An overview of how Raspberry Pi/ TF Mini device can be found here (1 min 23 seconds): https://youtu.be/EtwgQE9hNBY

I had intended to connect the device and ride around town for a week to collect some real data. I barely finished creating the entire system before the deadline, so sadly all the data I collected was fake data where waved my hand in front of the sensor. However, I was able to connect it to my bicycle and ride around my neighborhood, waving my hand in front of the device to create fake data.

A video of this collection of fake data can be found here (4 min 19 seconds):
https://youtu.be/M7xYeRqBaqE

A brief summary of the ride can be found here. I had to stop and get off the bike so I had to stop the video, thus a second video is required for the summary (15 seconds):
https://youtu.be/fSbeY9W6oAI

An overview of the realtime database that stores the incident reports can be found here (1 min):
https://youtu.be/443FG92dJ1k

CODE
The code consists of 4 items. All the files can be found here:
https://github.com/rdeamici/3Feet-IoT

1. tfminiplus.py - python file for interacting with the tf mini plus over UART serial communication.
2. threeFtMonitor.py - program that acts as a BLE server. When a device connects, it starts reading the connected tf mini sensor and sending notifications over BLE when 3ft violations are detected.
3. ThreeFeet.apk - the MIT App Inventor app. One can open this file on their android device, which will create a new application.
4. ThreeFeet.aia - the MIT App Inventor app that can be uploaded to the MIT App Inventor website. Download the file to your local machine, then visit http://appinventor.mit.edu/, create an account, and upload the file by choosing **File | Import Project**