Validation and Full Demo

for

Carbon Monoxide Poisoning Prevention System (COPPS)

Prepared by [redacted]

Texas Tech University

November 16, 2019

Table of Contents

1. Introduction

1.1 Project Description

The goal of the project is to create a system that prevents deaths from carbon monoxide poisoning in vehicles. The final product will contain both hardware and software components, connected via Bluetooth. The hardware component will be implemented using a Raspberry Pi, and a carbon monoxide sensor consisting of an Arduino and MQ3. The carbon monoxide sensor will be set to detect increasing carbon monoxide levels inside the vehicle. Once the increased levels of carbon monoxide are detected the hardware will send an alert to the user's phone via Bluetooth.

The project is also designed to have a software component, which will allow the user to perform some basic functions such as entering and updating their emergency contacts. The app will notify the user with an auditory alert if it receives a signal via Bluetooth that dangerous carbon monoxide levels are detected, as well as notifying their emergency contact with their last recorded location. The application will be implemented to work with the Android operating system.

1.2 Intended Audience and Reading Suggestions

This document is intended both for the designers of the system mentioned herein as well as for users of said system.

1.3 Product Scope

The system to be developed, the Carbon Monoxide Poisoning Prevention System (COPPS), is a system which monitors the carbon monoxide levels within the interior of a motor vehicle, alerting the user a taking potentially life-saving measures if necessary. The system uses a hardware component in order to monitor the carbon monoxide levels in the atmosphere and roll down the windows of the vehicle to vent accumulated carbon monoxide, and uses a software portion in the form of a smartphone application in order to alert the user of the danger in their vehicle and alert the user's emergency contacts if necessary. The purpose and goal of the system is to reduce the number of accidental carbon monoxide poisonings that result from faulty vehicle emission systems, avoiding preventable deaths and life altering injuries.

1.4 References

[1] An introduction to Bluetooth Programming. https://people.csail.mit.edu/albert/bluez-intro/index.html

[2] (2018, May 17). Carbon monoxide poisoning. https://www.mayoclinic.org/diseases-conditions/carbon-monoxide/symptoms-causes/syc-20370642

1.5 Definitions, Acronyms and Abbreviations.

1. **COPPS** : **CO P**oisoning **P**revention **S**ystem
2. **CO**: Carbon monoxide
3. **UI**: **U**ser **I**nterface
4. **MAC** address: **M**edia **A**ccess **C**ontrol address - a unique identifier assigned to a network interface controller.
5. **Bluetooth Socket -** A port which ince made two devices can communicate across.
6. **UUID - U**niversally **U**nique **Id**entifier - 128-bit number to identify information in computer systems
7. **Bluetooth Discovery -** Devices ability to search and discover other Bluetooth devices
8. **Input Stream** - The data coming to the device.
9. **Output Stream -** The data being exported from the device.
10. **Raspberry Pi -** A small computer that runs linux.

2. Design Details

2.1 Class Diagram

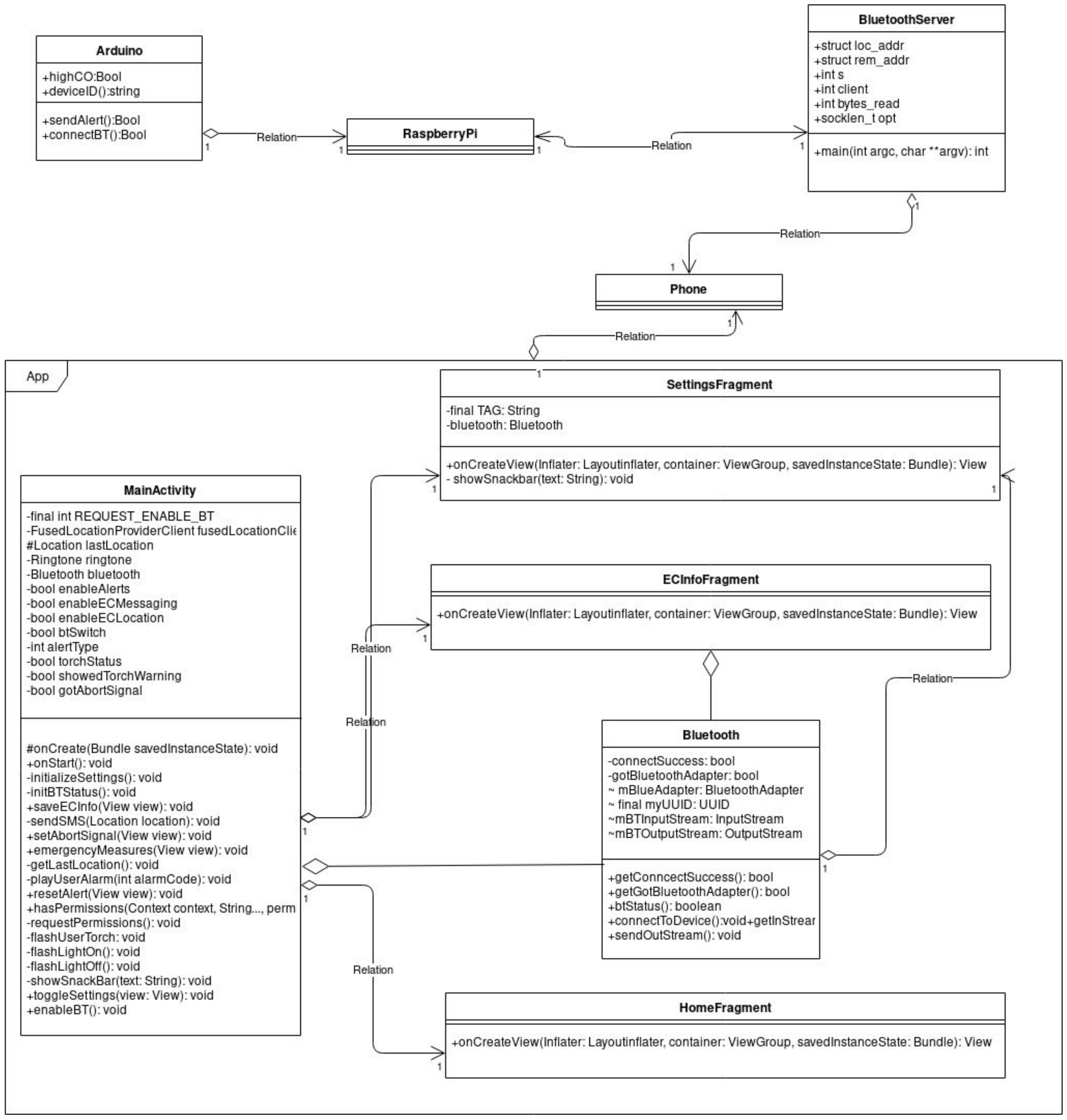

Arduino
+highCO:Bool
+deviceID():string
+sendAlert():Bool
+connectBT():Bool
Relation
RaspberryPi
Relation
BluetoothServer
+struct loc_addr
+struct rem_addr
+int s
+int client
+int bytes_read
+socklen_t opt
+main(int argc, char **argv): int
Relation
Phone
Relation
App
SettingsFragment
-final TAG: String
-bluetooth: Bluetooth
+onCreateView(Inflater: Layoutinflater, container: ViewGroup, savedInstanceState: Bundle): View
- showSnackbar(text: String): void
MainActivity
-final int REQUEST_ENABLE_BT
-FusedLocationProviderClient fusedLocationClie
#Location lastLocation
-Ringtone ringtone
-Bluetooth bluetooth
-bool enableAlerts
-bool enableECMessaging
-bool enableECLocation
-bool btSwitch
-int alertType
-bool torchStatus
-bool showedTorchWarning
-bool gotAbortSignal
#onCreate(Bundle savedInstanceState): void
+onStart(): void
-initializeSettings(): void
-initBTStatus(): void
+saveECInfo(View view): void
-sendSMS(Location location): void
+setAbortSignal(View view): void
+emergencyMeasures(View view): void
-getLastLocation(): void
-playUserAlarm(int alarmCode): void
+resetAlert(View view): void
+hasPermissions(Context context, String..., perm
-requestPermissions(): void
-flashUserTorch: void
-flashLightOn(): void
-flashLightOff(): void
-showSnackBar(text: String): void
+toggleSettings(view: View): void
+enableBT(): void
ECInfoFragment
+onCreateView(Inflater: Layoutinflater, container: ViewGroup, savedInstanceState: Bundle): View
Relation
Relation
Relation
Bluetooth
-connectSuccess: bool
-gotBluetoothAdapter: bool
~ mBlueAdapter: BluetoothAdapter
~ final myUUID: UUID
~mBTInputStream: InputStream
~mBTOutputStream: OutputStream
+getConncectSuccess(): bool
+getGotBluetoothAdapter(): bool
+btStatus(): boolean
+connectToDevice():void+getInStrear
+sendOutStream(): void
Relation
HomeFragment
+onCreateView(Inflater: Layoutinflater, container: ViewGroup, savedInstanceState: Bundle): View

Figure 2.1

The class diagram for the COPPS system has changed significantly. In the beginning the class diagram was created to create a general idea of what the app may look like. As the app is now completely developed it is much more detailed. The class diagram is separated into two distinct parts one is the software or app portion and the other is the hardware portion of the project. The app itself consists of five distinct classes:

- MainActivity
- settingsFragment

- ECinfoFragment
- HomeFragment
- Bluetooth

The hardware portion now consists of four classes which include:

- Phone
- Bluetooth server
- Raspberry Pi
- Arduino

All classes within the app interact with the MainActivity class with all of them having a one to one relation. The Bluetooth module could have 0..* relations as a Bluetooth client could be associated with many Bluetooth servers, but to reduce complexity and improve reliability the MAC address for the carbon monoxide device has been hardcoded into the app. All of the fragments have a one to one relation with the main activity because each fragment is only used once by the main activity.

2.2 Sequence Diagram

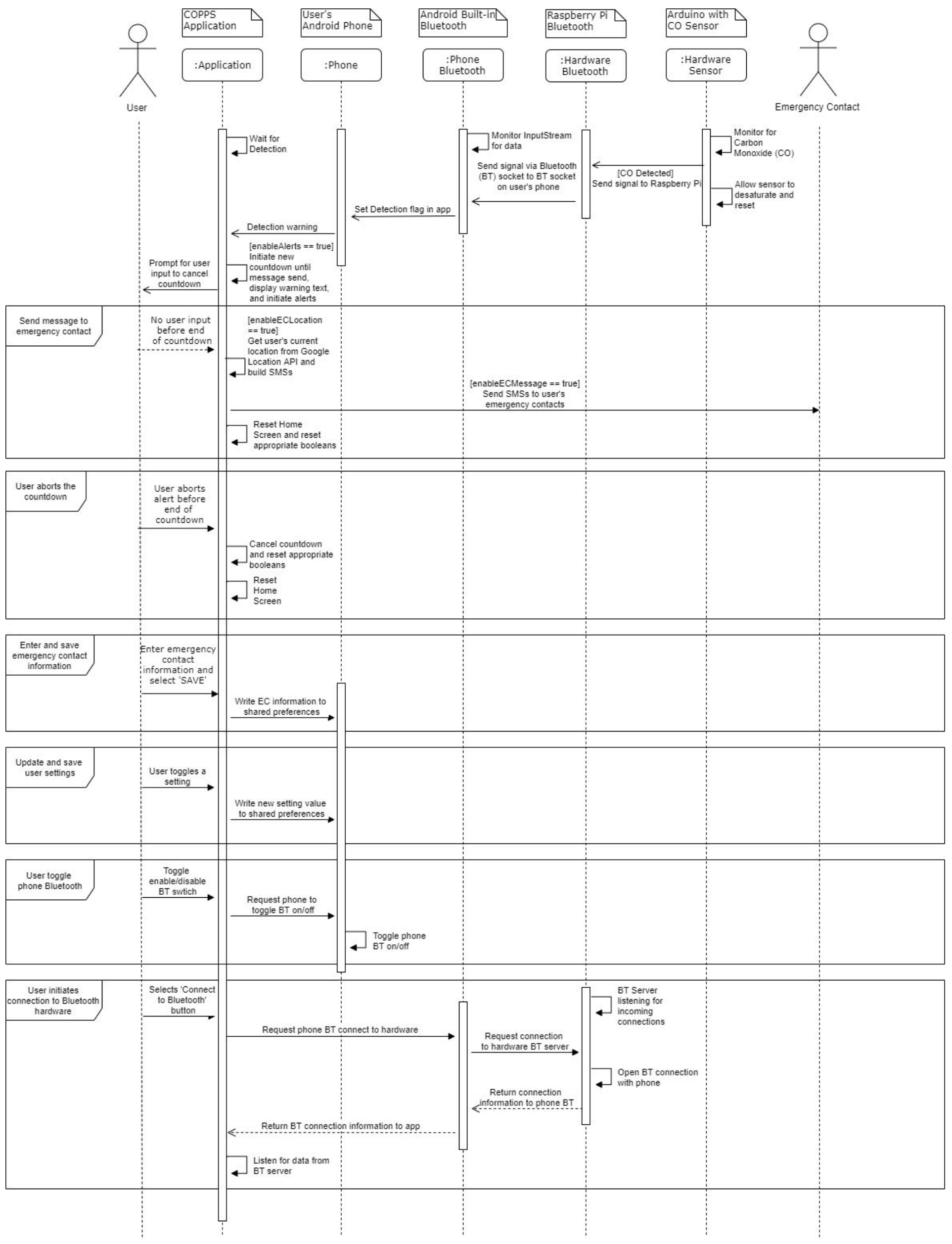

COPPS Application
User's Android Phone
Android Built-in Bluetooth
Raspberry Pi Bluetooth
Arduino with CO Sensor
:Application
:Phone
:Phone Bluetooth
:Hardware Bluetooth
:Hardware Sensor
User
Emergency Contact
Wait for Detection
Monitor InputStream for data
Monitor for Carbon Monoxide (CO)
Send signal via Bluetooth (BT) socket to BT socket on user's phone
[CO Detected] Send signal to Raspberry Pi
Allow sensor to desaturate and reset
Set Detection flag in app
Detection warning
[enableAlerts == true] Initiate new countdown until message send, display warning text, and initiate alerts
Prompt for user input to cancel countdown
Send message to emergency contact
No user input before end of countdown
[enableECLocation == true] Get user's current location from Google Location API and build SMSs
[enableECMessage == true] Send SMSs to user's emergency contacts
Reset Home Screen and reset appropriate booleans
User aborts the countdown
User aborts alert before end of countdown
Cancel countdown and reset appropriate booleans
Reset Home Screen
Enter and save emergency contact information
Enter emergency contact information and select 'SAVE'
Write EC information to shared preferences
Update and save user settings
User toggles a setting
Write new setting value to shared preferences
User toggle phone Bluetooth
Toggle enable/disable BT swtich
Request phone to toggle BT on/off
Toggle phone BT on/off
User initiates connection to Bluetooth hardware
Selects 'Connect to Bluetooth' button
Request phone BT connect to hardware
BT Server listening for incoming connections
Request connection to hardware BT server
Open BT connection with phone
Return connection information to phone BT
Return BT connection information to app
Listen for data from BT server

Figure 2.2

The Sequence diagram has also been updated to reflect the changes to the application that have been made since the previous report. The diagram shows the sequence of events involved in sending messages to the user's emergency contacts, updating the user's settings and emergency contact information, as well as the sequence of events involved in the application connecting to the hardware via Bluetooth. The application waits for a signal from the hardware via Bluetooth indicating that the CO levels have crossed the set threshold, then activates the alert when the signal is received, at which point it triggers the countdown. If no input is received from the user before the end of the countdown, the application will send an SMS to the user's emergency contact. In doing so, it will first acquire the user's last reported location from the Google FusedLocationProvider API, then use this location to send a message to the user's emergency contact alerting them that the user may be in need of help and providing them with the user's location. However, if the user does abort the countdown before it ends; the application will cancel the countdown and reset the Home Screen to its original state. As mentioned above, the diagram further shows the sequence of events involved in the user setting and saving their settings, as well as their input emergency contact information. Finally, the diagram outlines the sequence of events involved in the application connecting to the hardware component via Bluetooth.

2.3 Use Case Diagram

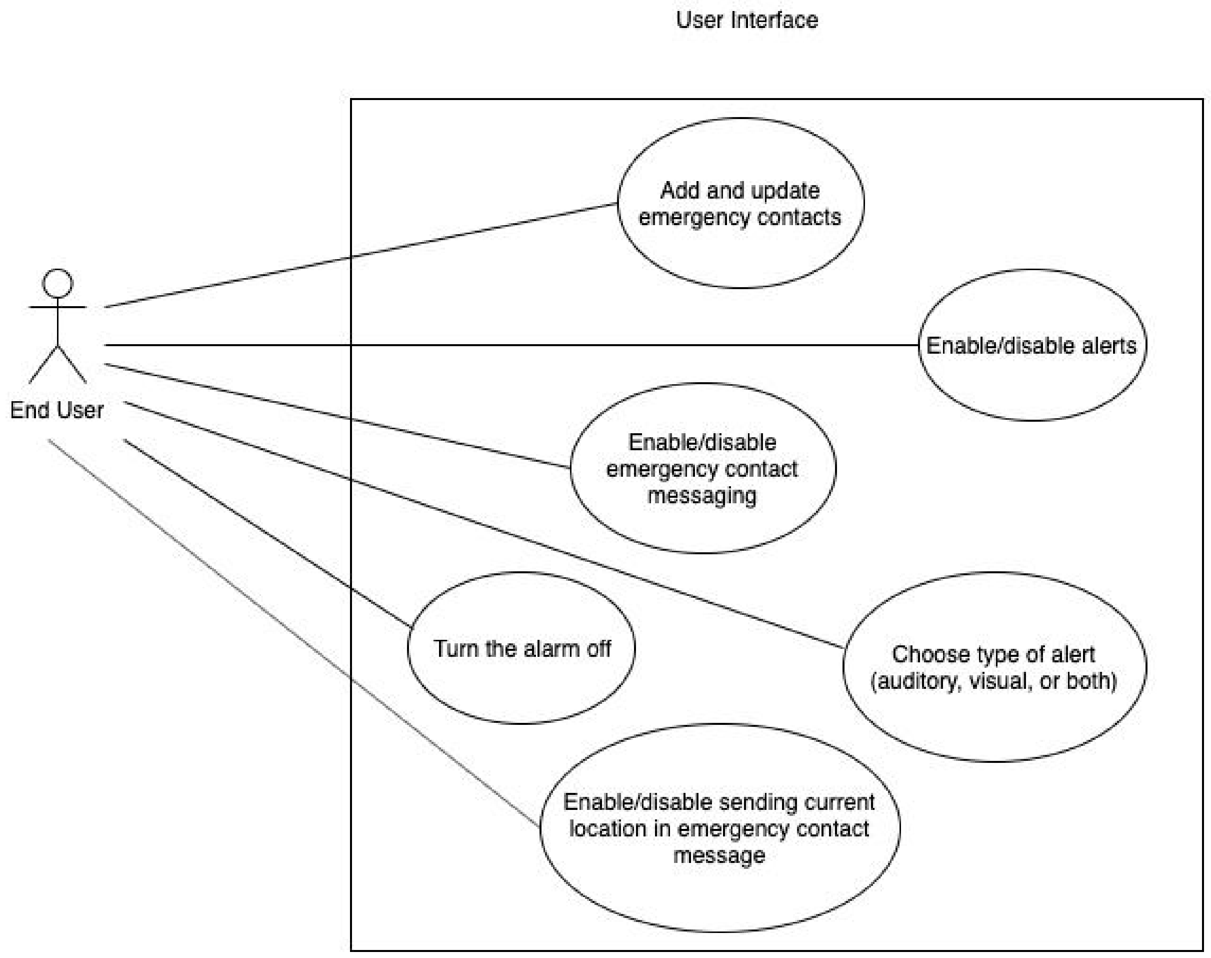

User Interface
End User
Add and update emergency contacts
Enable/disable alerts
Enable/disable emergency contact messaging
Turn the alarm off
Choose type of alert (auditory, visual, or both)
Enable/disable sending current location in emergency contact message

Figure 2.3
The use case diagram in Figure 2.3 shows the interactions between the end user and UI. First of all, the user is able to add emergency contacts and update their information. Secondly, the user can enable/disable alert and choose the type of alert to be played. In addition, it is possible to turn off emergency contact messaging at dangerous, but not deadly, levels of carbon monoxide concentration. The user can also enable or disable sending their current location to the emergency contacts. Finally, the user is always able to turn off the alarm by clicking "STOP ALARM" on Home Screen.

3. Implementation Details

3.1 Implementation Overview

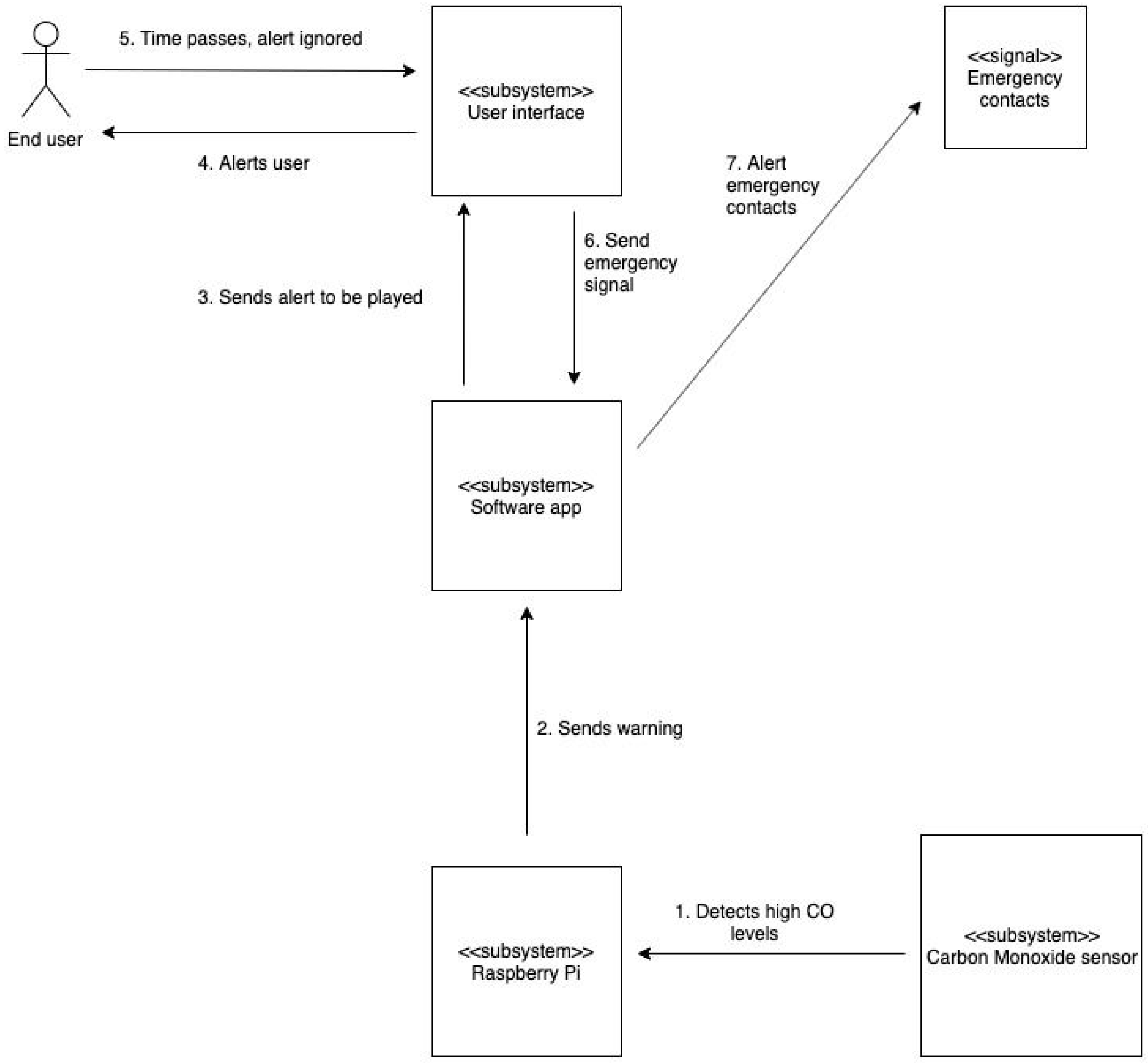

5. Time passes, alert ignored
End user
4. Alerts user
<<subsystem>> User interface
<<signal>> Emergency contacts
7. Alert emergency contacts
6. Send emergency signal
3. Sends alert to be played
<<subsystem>> Software app
2. Sends warning
<<subsystem>> Raspberry Pi
1. Detects high CO levels
<<subsystem>> Carbon Monoxide sensor

Figure 3.1
The diagram above details the structure of COPPS and shows what happens when the user ignores the alarm and fails to respond within a timeframe of 10 seconds to indicate that they are not in danger. For this use case we assume that the user has enabled alerts and emergency contact messaging. Initially, the carbon monoxide sensor detects the high levels of Carbon Monoxide in the interior of a vehicle. At that point Raspberry Pi

communicates with a smartphone via Bluetooth and the software application will send an alert to display to a user through the user interface. When an alert is ignored the application receives an emergency signal and alerts emergency contacts and the authorities.

3.2 Software Implementation

3.2.1 UI implementation

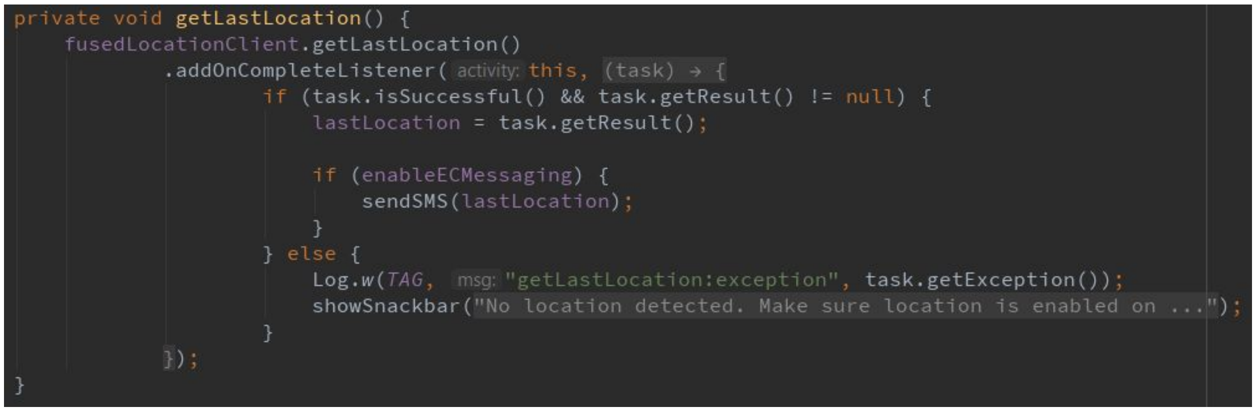
The user interface (UI) was implemented in Android Studio using a bottom navigation bar layout, the Main Activity calls 3 separate Fragments, Home, Settings, and ECInfo depending the the page requested by the user:

- The Home Fragment (Figure 1) displays the main screen of the application, which shows the name of the application and has the navigation bar at the bottom. This navigation bar allows the user to select their desired page, allowing them to update their settings and emergency contacts.
- The Settings Fragment (Figure 2) allows the user to enable/disable alerts entirely, or to select which type of alert they would like to receive: auditory, visual, or both. Additionally, the user will be able to disable sending messages to their emergency contacts, as well as disable the inclusion of their location in said message. The user's settings will persist between sessions.
- The ECInfo Fragment (Figure 3) displays a screen allowing the user to enter their name, as well as the name and phone numbers of up to 3 emergency contacts. The user can then select the 'Save' button to confirm their entered information. The user's emergency contact information will then persist between sessions.

3.2.2 Feature Implementation

- Getting User Location:

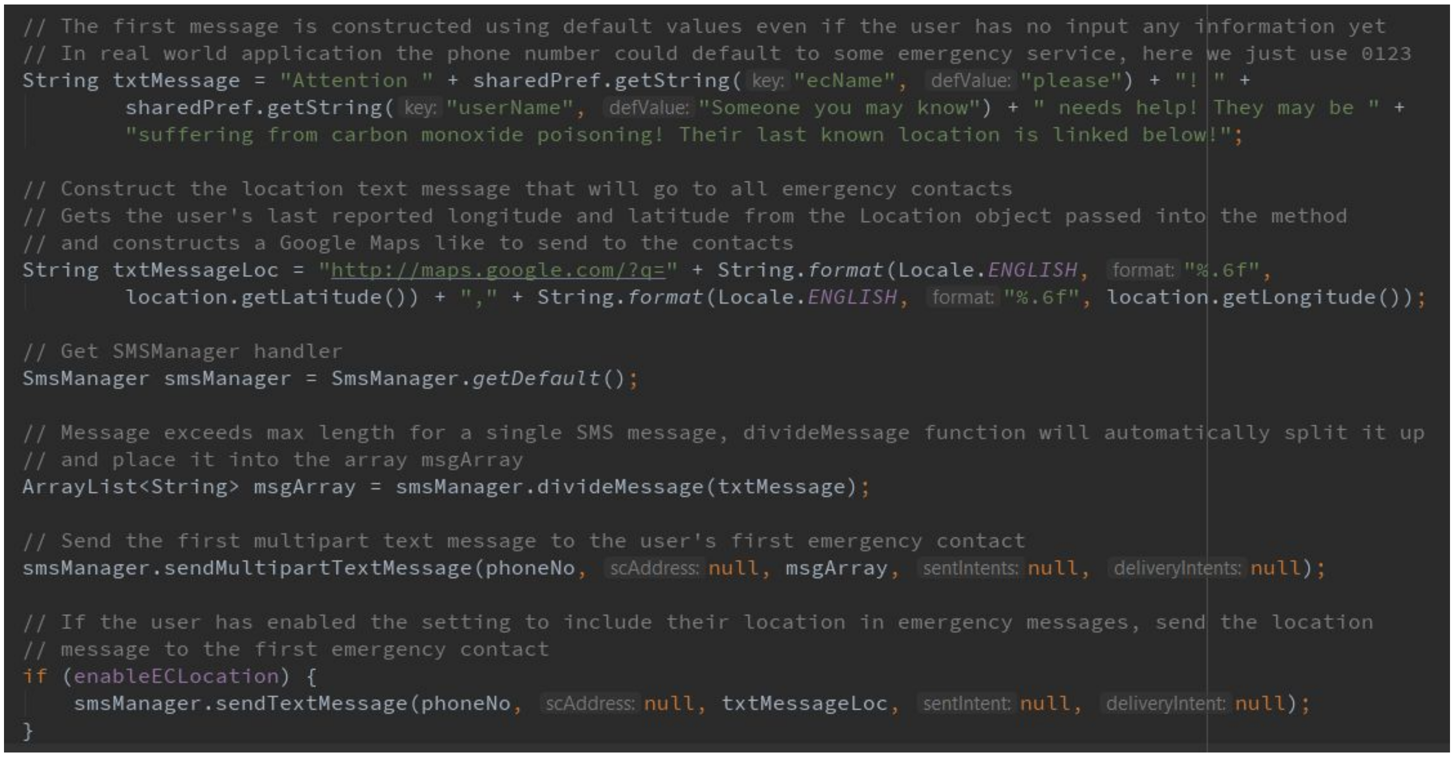
  The application uses the Google Play services FusedLocationProvider API in order to get the user's most recently reported location from the phone



```
private void getLastLocation() {
    fusedLocationClient.getLastLocation()
            .addOnCompleteListener( activity: this, (task) -> {
                if (task.isSuccessful() && task.getResult() != null) {
                    lastLocation = task.getResult();

                    if (enableECMessaging) {
                        sendSMS(lastLocation);
                    }
                } else {
                    Log.w(TAG,  msg: "getLastLocation:exception", task.getException());
                    showSnackbar("No location detected. Make sure location is enabled on ...");
                }
            });
}
```

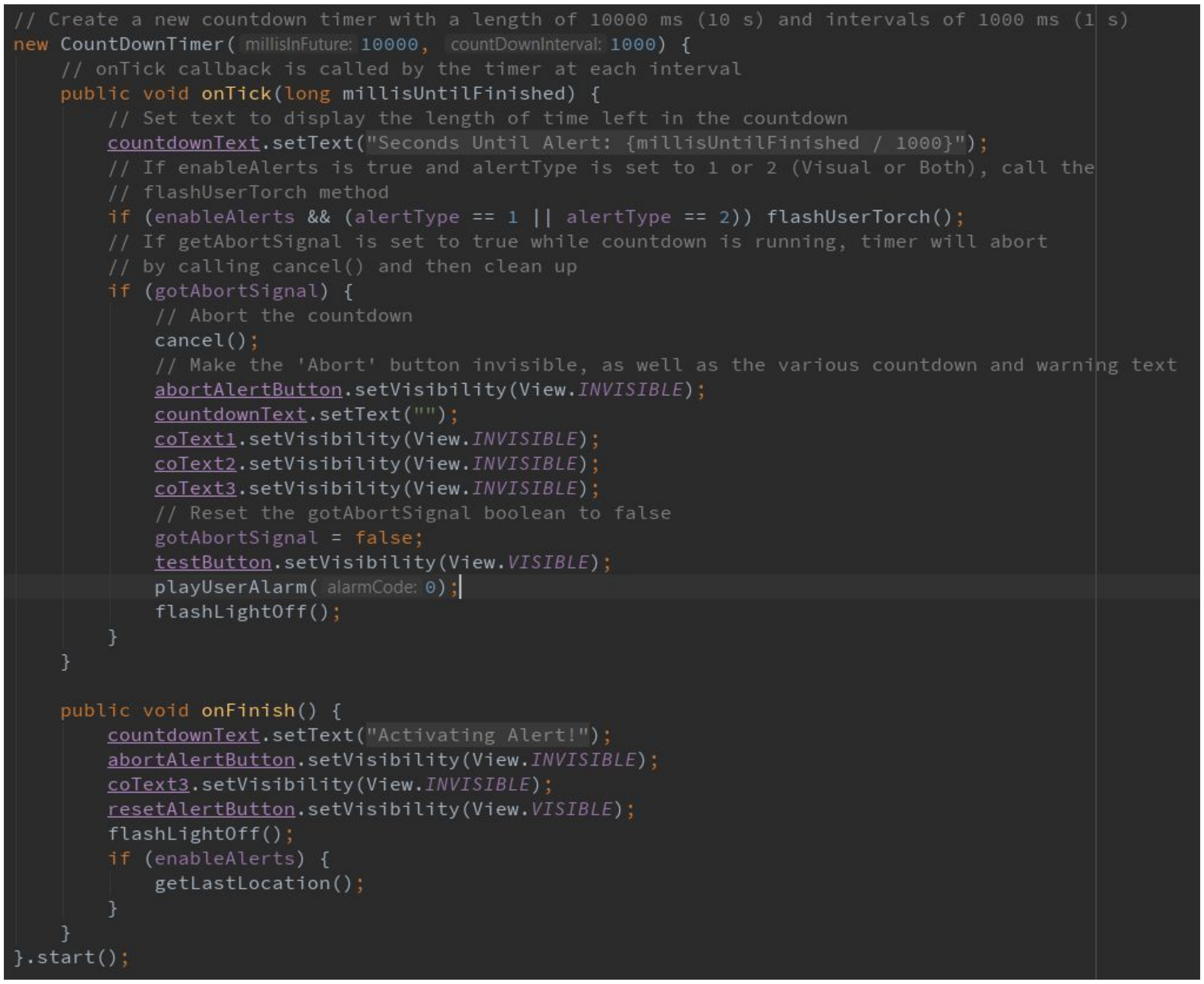
If the user has the setting enabled, this method will also call the method to send this location to their emergency contact.

- Sending SMS to the user’s emergency contacts:
  The method to send the user’s location to their emergency contacts takes a Location object as an argument, and uses this to create a Google Maps link with the user’s current longitude and latitude . This link, along with a message altering the emergency contact that the user may need help is then sent to the saved phone number(s) of the user’s emergency contact(s).
- Alert Countdown and Timeout:



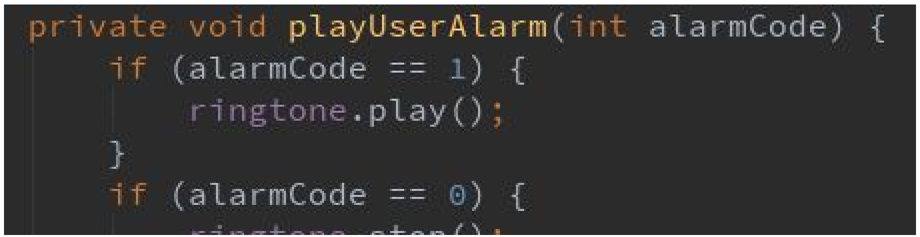
```
// The first message is constructed using default values even if the user has no input any information yet
// In real world application the phone number could default to some emergency service, here we just use 0123
String txtMessage = "Attention " + sharedPref.getString( key: "ecName",  defValue: "please") + "! " +
        sharedPref.getString( key: "userName",  defValue: "Someone you may know") + " needs help! They may be " +
        "suffering from carbon monoxide poisoning! Their last known location is linked below!";

// Construct the location text message that will go to all emergency contacts
// Gets the user's last reported longitude and latitude from the Location object passed into the method
// and constructs a Google Maps like to send to the contacts
String txtMessageLoc = "http://maps.google.com/?q=" + String.format(Locale.ENGLISH,  format: "%.6f",
        location.getLatitude()) + "," + String.format(Locale.ENGLISH,  format: "%.6f", location.getLongitude());

// Get SMSManager handler
SmsManager smsManager = SmsManager.getDefault();

// Message exceeds max length for a single SMS message, divideMessage function will automatically split it up
// and place it into the array msgArray
ArrayList<String> msgArray = smsManager.divideMessage(txtMessage);

// Send the first multipart text message to the user's first emergency contact
smsManager.sendMultipartTextMessage(phoneNo,  scAddress: null, msgArray,  sentIntents: null,  deliveryIntents: null);

// If the user has enabled the setting to include their location in emergency messages, send the location
// message to the first emergency contact
if (enableECLocation) {
    smsManager.sendTextMessage(phoneNo,  scAddress: null, txtMessageLoc,  sentIntent: null,  deliveryIntent: null);
}
```

When an alert signal is received from the hardware, the application will initiate a countdown and display it to the user, giving them the chance to abort the alert before the alert is sent to their emergency contacts. This allows the user to abort the alert in the case that they are able to get themselves to safety and do not require further assistance, or a case where there was a false alarm.



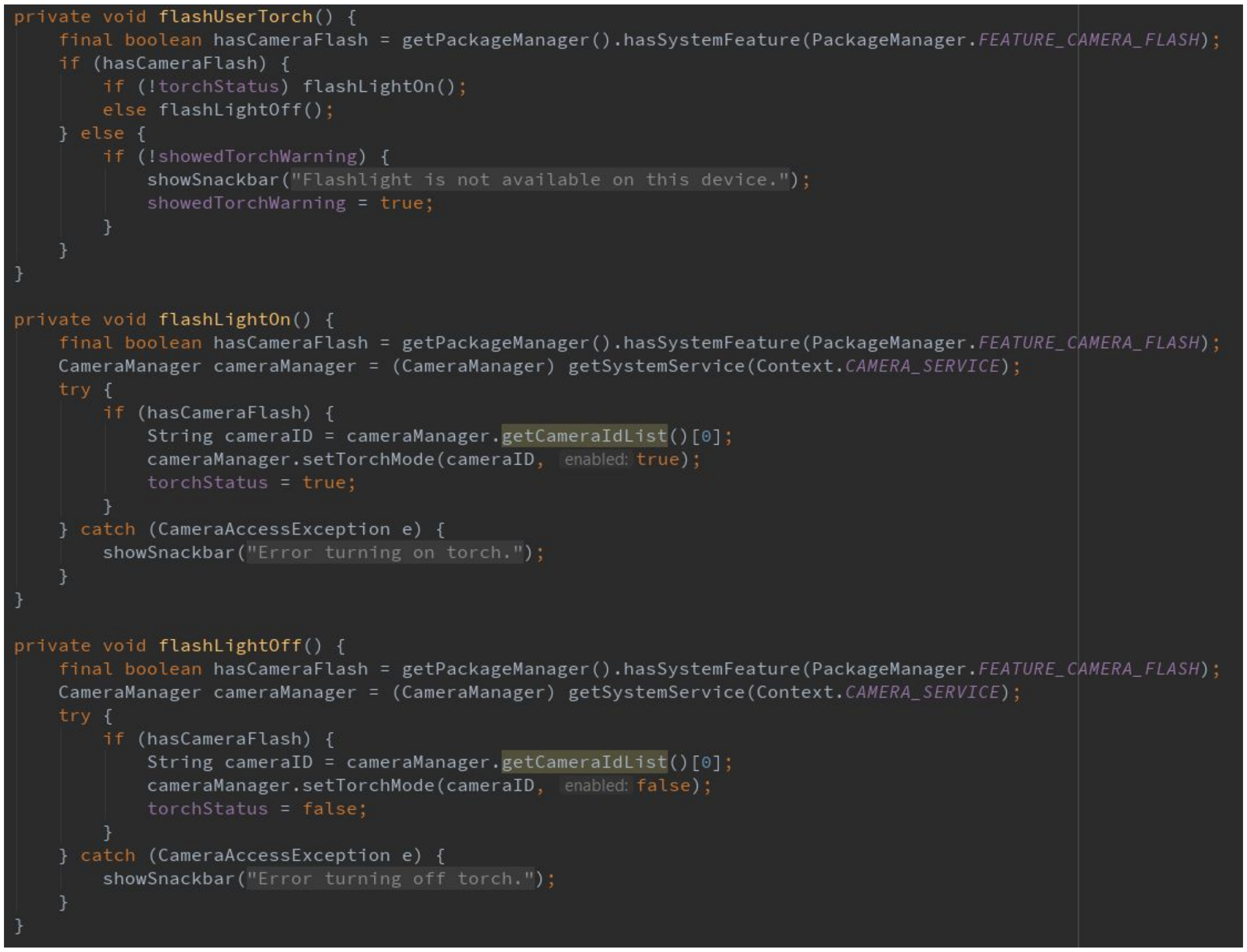
```
// Create a new countdown timer with a length of 10000 ms (10 s) and intervals of 1000 ms (1 s)
new CountDownTimer( millisInFuture: 10000,  countDownInterval: 1000) {
    // onTick callback is called by the timer at each interval
    public void onTick(long millisUntilFinished) {
        // Set text to display the length of time left in the countdown
        countdownText.setText("Seconds Until Alert: {millisUntilFinished / 1000}");
        // If enableAlerts is true and alertType is set to 1 or 2 (Visual or Both), call the
        // flashUserTorch method
        if (enableAlerts && (alertType == 1 || alertType == 2)) flashUserTorch();
        // If getAbortSignal is set to true while countdown is running, timer will abort
        // by calling cancel() and then clean up
        if (gotAbortSignal) {
            // Abort the countdown
            cancel();
            // Make the 'Abort' button invisible, as well as the various countdown and warning text
            abortAlertButton.setVisibility(View.INVISIBLE);
            countdownText.setText("");
            coText1.setVisibility(View.INVISIBLE);
            coText2.setVisibility(View.INVISIBLE);
            coText3.setVisibility(View.INVISIBLE);
            // Reset the gotAbortSignal boolean to false
            gotAbortSignal = false;
            testButton.setVisibility(View.VISIBLE);
            playUserAlarm( alarmCode: 0);
            flashLightOff();
        }
    }

    public void onFinish() {
        countdownText.setText("Activating Alert!");
        abortAlertButton.setVisibility(View.INVISIBLE);
        coText3.setVisibility(View.INVISIBLE);
        resetAlertButton.setVisibility(View.VISIBLE);
        flashLightOff();
        if (enableAlerts) {
            getLastLocation();
        }
    }
}.start();
```

- Auditory and Visual Alerts:
  The application uses the alarm and flashlight features on the user's phone in order to alert the user in the case of a carbon monoxide detection. For the auditory alarm, the application gets a handler for the phone's alarm and then plays and stops the alarm depending on need.



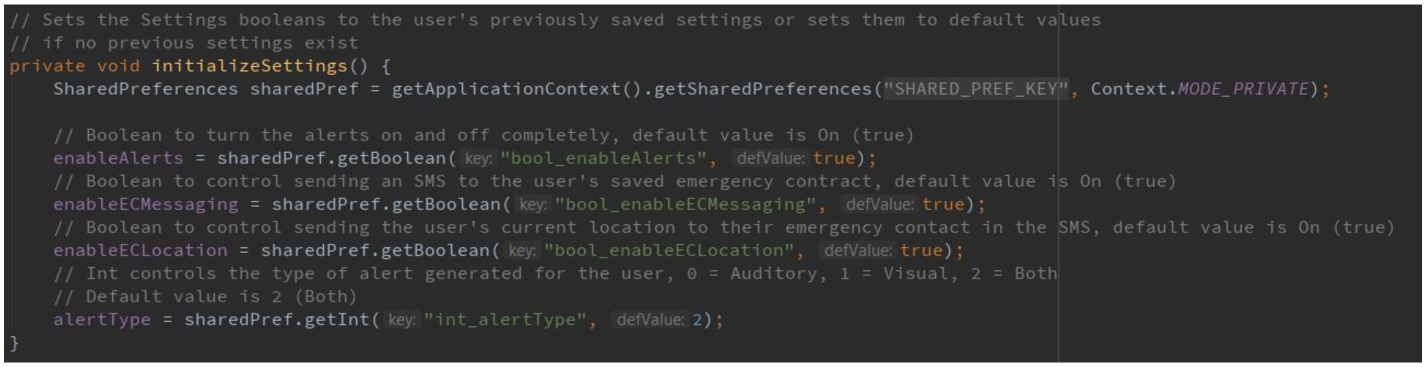
```
private void playUserAlarm(int alarmCode) {
    if (alarmCode == 1) {
        ringtone.play();
    }
    if (alarmCode == 0) {
```

For the visual alarm, the same premise is used, with the application getting a handler for the user's Camera and then activating and deactivating the flashlight for the duration of the alert.



```
private void flashUserTorch() {
    final boolean hasCameraFlash = getPackageManager().hasSystemFeature(PackageManager.FEATURE_CAMERA_FLASH);
    if (hasCameraFlash) {
        if (!torchStatus) flashLightOn();
        else flashLightOff();
    } else {
        if (!showedTorchWarning) {
            showSnackbar("Flashlight is not available on this device.");
            showedTorchWarning = true;
        }
    }
}

private void flashLightOn() {
    final boolean hasCameraFlash = getPackageManager().hasSystemFeature(PackageManager.FEATURE_CAMERA_FLASH);
    CameraManager cameraManager = (CameraManager) getSystemService(Context.CAMERA_SERVICE);
    try {
        if (hasCameraFlash) {
            String cameraID = cameraManager.getCameraIdList()[0];
            cameraManager.setTorchMode(cameraID,  enabled: true);
            torchStatus = true;
        }
    } catch (CameraAccessException e) {
        showSnackbar("Error turning on torch.");
    }
}

private void flashLightOff() {
    final boolean hasCameraFlash = getPackageManager().hasSystemFeature(PackageManager.FEATURE_CAMERA_FLASH);
    CameraManager cameraManager = (CameraManager) getSystemService(Context.CAMERA_SERVICE);
    try {
        if (hasCameraFlash) {
            String cameraID = cameraManager.getCameraIdList()[0];
            cameraManager.setTorchMode(cameraID,  enabled: false);
            torchStatus = false;
        }
    } catch (CameraAccessException e) {
        showSnackbar("Error turning off torch.");
    }
}
```

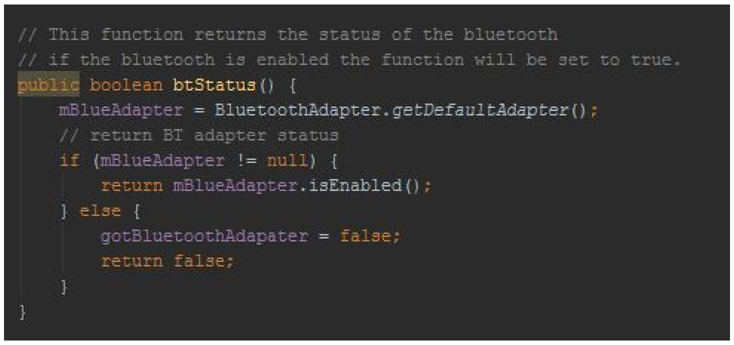
- Data Persistence:
  The application makes use of shared preferences in order to provide data persistence between sessions for the user's settings and emergency contact information. Whenever the user makes changes to their settings or saves their emergency contacts, the application writes the new data to the shared preferences file for the application on the user's phone. When the user reopens the app, the application will load the data from the shared preferences and restore the previous settings and emergency contact information for the user.



```
// Sets the Settings booleans to the user's previously saved settings or sets them to default values
// if no previous settings exist
private void initializeSettings() {
    SharedPreferences sharedPref = getApplicationContext().getSharedPreferences("SHARED_PREF_KEY", Context.MODE_PRIVATE);

    // Boolean to turn the alerts on and off completely, default value is On (true)
    enableAlerts = sharedPref.getBoolean( key: "bool_enableAlerts",  defValue: true);
    // Boolean to control sending an SMS to the user's saved emergency contract, default value is On (true)
    enableECMessaging = sharedPref.getBoolean( key: "bool_enableECMessaging",  defValue: true);
    // Boolean to control sending the user's current location to their emergency contact in the SMS, default value is On (true)
    enableECLocation = sharedPref.getBoolean( key: "bool_enableECLocation",  defValue: true);
    // Int controls the type of alert generated for the user, 0 = Auditory, 1 = Visual, 2 = Both
    // Default value is 2 (Both)
    alertType = sharedPref.getInt( key: "int_alertType",  defValue: 2);
}
```

3.2.3 Bluetooth Implementation

The Bluetooth connection is implemented using the Raspberry Pi, where it runs a Bluetooth server listening for incoming messages. The process will send a message to the phone when an alert signal is triggered by the attached carbon monoxide sensor. The Bluetooth connection between the Raspberry Pi and the phones Bluetooth is implemented using the Android OS compatible BlueZ libraries.

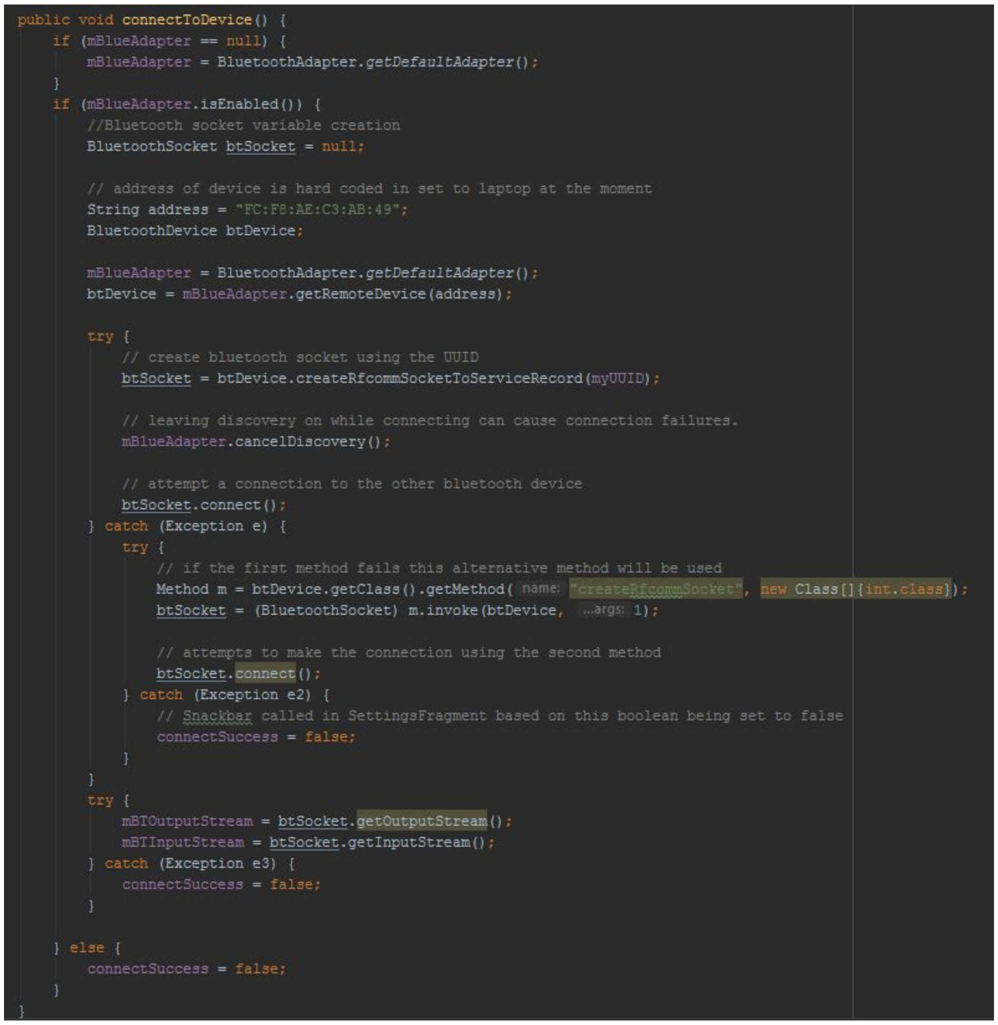


```
// This function returns the status of the bluetooth
// if the bluetooth is enabled the function will be set to true.
public boolean btStatus() {
    mBlueAdapter = BluetoothAdapter.getDefaultAdapter();
    // return BT adapter status
    if (mBlueAdapter != null) {
        return mBlueAdapter.isEnabled();
    } else {
        gotBluetoothAdapater = false;
        return false;
    }
}
```

Getting Bluetooth Status - Figure 3.2.2.1

To establish a Bluetooth connection we must know if the Bluetooth on the device is enabled or disabled. The function in figure 3.2.2.1 uses the Bluetooth adapter class to check if the Bluetooth is enabled. If it is enabled the value true is returned to the function and vice versa for if it is disabled. This function allows us to have a switch in the app that will tell the user whether or not their Bluetooth is enabled or disabled. This switch of course allows us to also turn the Bluetooth on or off within the app.

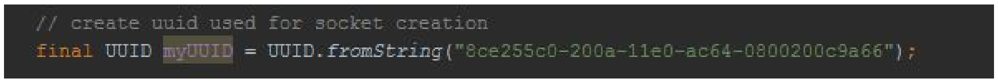
Now that the app has established that the Bluetooth is enabled, the process of connecting to the Raspberry Pi can be carried out.



```java
public void connectToDevice() {
    if (mBlueAdapter == null) {
        mBlueAdapter = BluetoothAdapter.getDefaultAdapter();
    }
    if (mBlueAdapter.isEnabled()) {
        //Bluetooth socket variable creation
        BluetoothSocket btSocket = null;

        // address of device is hard coded in set to laptop at the moment
        String address = "FC:F8:AE:C3:AB:49";
        BluetoothDevice btDevice;

        mBlueAdapter = BluetoothAdapter.getDefaultAdapter();
        btDevice = mBlueAdapter.getRemoteDevice(address);

        try {
            // create bluetooth socket using the UUID
            btSocket = btDevice.createRfcommSocketToServiceRecord(myUUID);

            // leaving discovery on while connecting can cause connection failures.
            mBlueAdapter.cancelDiscovery();

            // attempt a connection to the other bluetooth device
            btSocket.connect();
        } catch (Exception e) {
            try {
                // if the first method fails this alternative method will be used
                Method m = btDevice.getClass().getMethod( name: "createRfcommSocket", new Class[]{int.class});
                btSocket = (BluetoothSocket) m.invoke(btDevice,  ...args: 1);

                // attempts to make the connection using the second method
                btSocket.connect();
            } catch (Exception e2) {
                // Snackbar called in SettingsFragment based on this boolean being set to false
                connectSuccess = false;
            }
        }
        try {
            mBTOutputStream = btSocket.getOutputStream();
            mBTInputStream = btSocket.getInputStream();
        } catch (Exception e3) {
            connectSuccess = false;
        }

    } else {
        connectSuccess = false;
    }
}
```

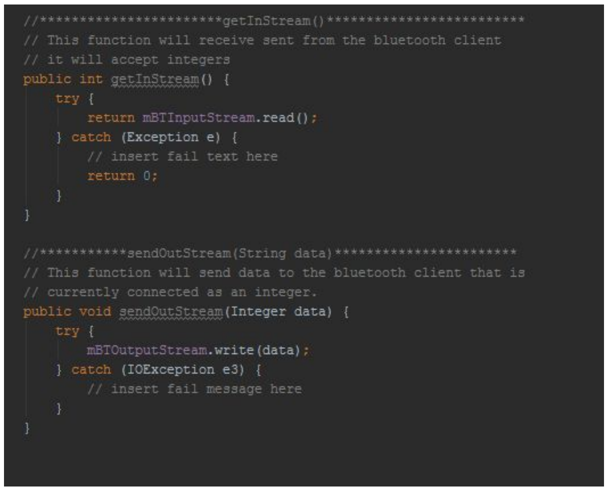
Connect To Bluetooth Device - Figure 3.2.2.2

Figure 3.2.2.2 shows our implementation of the client side Bluetooth connection. This function is called when the user presses the "Connect" button on the settings page of the app. The code is simpler than it looks, when the connection is made we must have three values: the remote devices address (address), the Bluetooth adapter (mBlueAdapter), and the remote Bluetooth device (btDevice). As you can see the remote devices address (MAC address) in our program is hardcoded. This decision was made to reduce the complexity of the program. Once all three of these pieces of information are collected the Bluetooth socket can be created using a UUID like the one in figure 3.2.2.3. Once this is done Bluetooth discovery must be cancelled to prevent connection problems. Then the function is able to attempt a connection with the Bluetooth server. If that connection fails there is an alternate method that is called, but I won't go into detail about the secondary method. Finally, once the client has made a successful connection the input and output streams can be initialized.



```
// create uuid used for socket creation
final UUID myUUID = UUID.fromString("8ce255c0-200a-11e0-ac64-0800200c9a66");
```

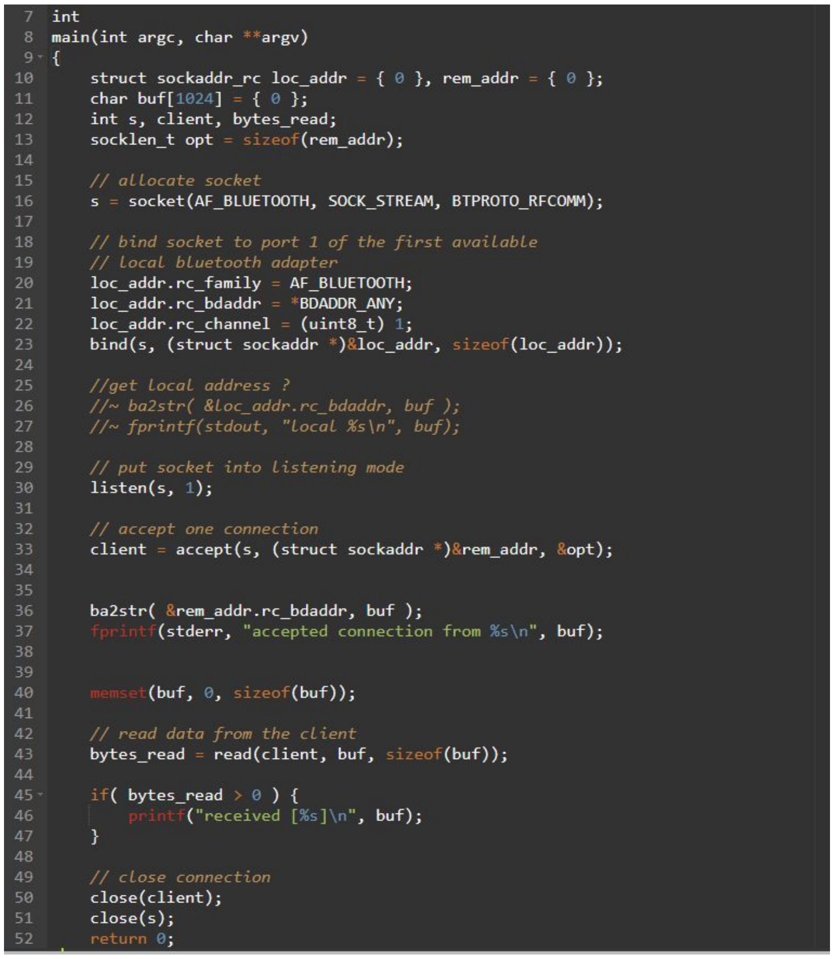
UUID - Figure 3.2.2.3



```
//***********************getInStream()*************************
// This function will receive sent from the bluetooth client
// it will accept integers
public int getInStream() {
    try {
        return mBTInputStream.read();
    } catch (Exception e) {
        // insert fail text here
        return 0;
    }
}

//***********sendOutStream(String data)***********************
// This function will send data to the bluetooth client that is
// currently connected as an integer.
public void sendOutStream(Integer data) {
    try {
        mBTOutputStream.write(data);
    } catch (IOException e3) {
        // insert fail message here
    }
}
```

Grabbing the input/output stream - Figure 3.2.2.4

These functions are responsible for sending out data (sendOutStream) or receiving data using the previously declared input/output stream variables. The data sent and received is received as an integer and sent as an integer. For example, an alert is sent to the phone as a 1 and no alert is sent as a 0 and so on.



```
int
main(int argc, char **argv)
{
    struct sockaddr_rc loc_addr = { 0 }, rem_addr = { 0 };
    char buf[1024] = { 0 };
    int s, client, bytes_read;
    socklen_t opt = sizeof(rem_addr);

    // allocate socket
    s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);

    // bind socket to port 1 of the first available
    // local bluetooth adapter
    loc_addr.rc_family = AF_BLUETOOTH;
    loc_addr.rc_bdaddr = *BDADDR_ANY;
    loc_addr.rc_channel = (uint8_t) 1;
    bind(s, (struct sockaddr *)&loc_addr, sizeof(loc_addr));

    //get local address ?
    //~ ba2str( &loc_addr.rc_bdaddr, buf );
    //~ fprintf(stdout, "local %s\n", buf);

    // put socket into listening mode
    listen(s, 1);

    // accept one connection
    client = accept(s, (struct sockaddr *)&rem_addr, &opt);


    ba2str( &rem_addr.rc_bdaddr, buf );
    fprintf(stderr, "accepted connection from %s\n", buf);


    memset(buf, 0, sizeof(buf));

    // read data from the client
    bytes_read = read(client, buf, sizeof(buf));

    if( bytes_read > 0 ) {
        printf("received [%s]\n", buf);
    }

    // close connection
    close(client);
    close(s);
    return 0;
```

Bluetooth Server - Figure 3.2.2.5

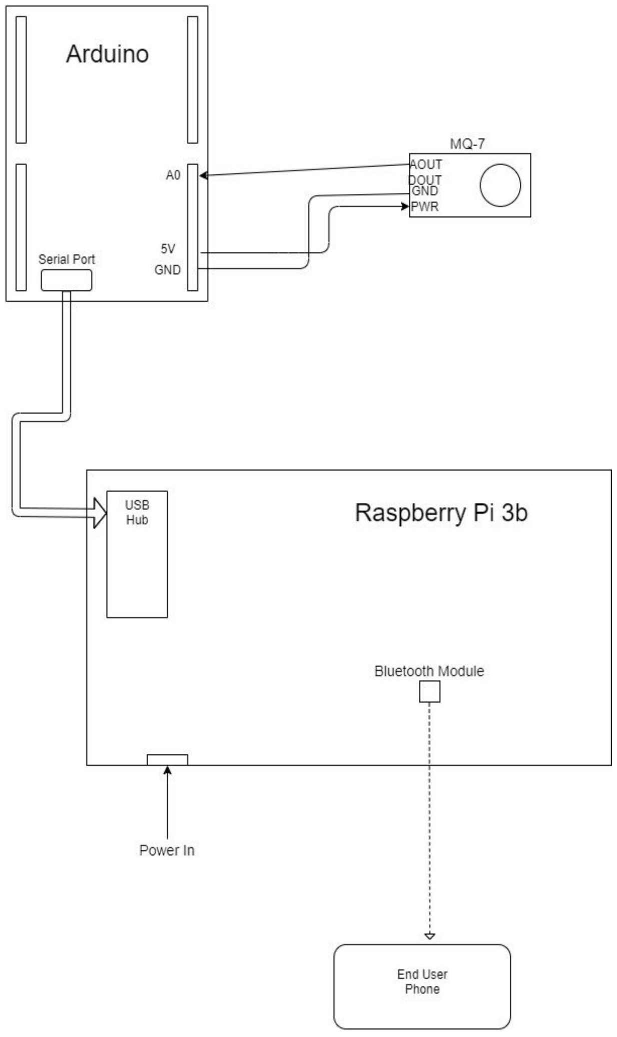
In figure 3.2.2.5 the Bluetooth server script that will be running on the Raspberry Pi is shown. This is a template from Albert Huang's Bluetooth programming tutorial [1]. This simple script can be easily modified to have a function that writes to the out stream instead of reading. The script also accepts the Bluetooth connection of the phones requested Bluetooth socket. Once the connection is made communication between the two devices can be done using the read and write functions. As it stands this script only accepts the connection, reads the out stream, and then closes the connection.

3.3 Hardware Implementation

Overview

The project’s hardware consists of five different devices and two separate subsystems. In the order of which they will be introduced, there is the MQ-7 sensor, the Arduino Uno, the Raspberry Pi 3b, the end user’s cellular phone, and the mobile power for the Raspberry Pi. The two subsystems will be the Arduino Subsystem, and the Pi Subsystem.

Beginning with the Arduino Subsystem, the subsystem contains the MQ-7 sensor and the Arduino Uno board. The MQ-7 device is a sensor used to directly measure the trends in the amount of carbon monoxide in the air. The Arduino Uno is a microcontroller programmed through the Arduino IDE in C. The MQ-7 sensor outputs an analog signal that is read by the Arduino through an analog input pin, and the MQ-7 is powered directly from the Arduino board.



The Raspberry Pi subsystem contains the Raspberry Pi 3b, the end user’s phone, and the power source for the Raspberry Pi. Within these devices, there are several assumptions that the project was designed with.

1. The end user’s phone will be capable of Bluetooth connectivity.
2. The hardware of the end user’s phone will not need to be modified in any way.
3. The end user’s phone will be running a version of Android OS.
4. The power source for the Raspberry Pi 3b will be capable of powering both subsystems.

The Raspberry Pi 3b is a computer running Raspbian OS. It will contain the needed software for the Bluetooth Server (written in C), as well as the software required to read the serial output of the Arduino Subsystem (written in Python).
For the end user’s phone it is assumed it will have full compatibility with the Raspberry Pi 3b and that it will not need to be modified, as such it will not be discussed in detail here.

To the right is a diagram of the entire system as it is connected.

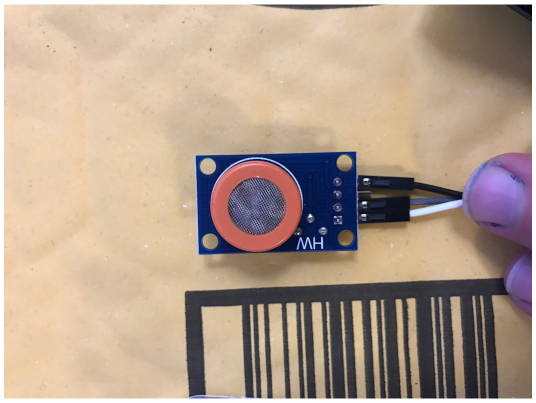
3.3.1 Arduino Subsystem

3.3.1.1 MQ-7 Sensor

The MQ-7 sensor has a high sensitivity to carbon monoxide and is commonly used in consumer products to measure trends in carbon monoxide concentration. It can be used to measure the exact amount of carbon monoxide in the atmosphere, however it is highly recommended that for project other than toy applications a more precise instrument be used.

The MQ-7 is connected to the Arduino Uno by the 5V, GND, and A0 pins.

1. The A0 pin of the Arduino Uno is connected to the digital out (AOUT) pin of the MQ-7.
2. The 5V pin of the Arduino Uno is connected to the voltage in (VIN) pin of the MQ-7.
3. The GND pin of the Arduino Uno is connected to the ground (GND) pin of the MQ-7.

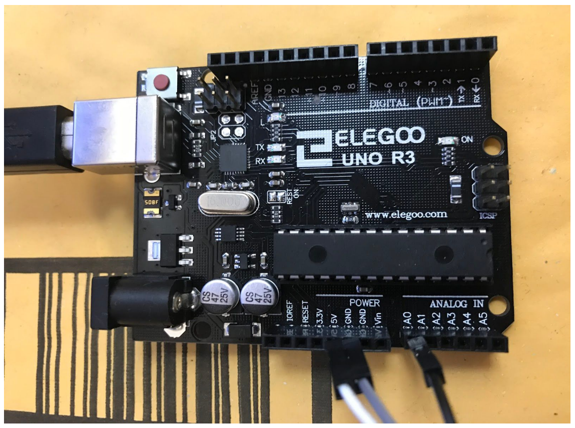


Note: In this picture, the white wire is VIN, grey is GND, black is AOUT

3.3.1.2 Arduino Uno

The Arduino Uno is a microcontroller board commonly used in consumer electronic projects. It has an operating voltage of 5V, with a recommended input voltage of 6-20V. There are many different manufacturers of the Arduino Uno, the project specifically uses the Elegoo, however any of the variations should be compatible with the subsystem.

The Arduino Uno was programmed in C using the Arduino IDE. It receives input from the MQ-7 sensor through the Analog Input Pin 0 (A0), and outputs the trend data through the Serial Port to the Raspberry Pi 3b.



Source Code: Arduino Uno

```
void setup()
{
 Serial.begin(9600);
}
void loop()
{
 float sensorValue = analogRead(A0);

 //Getting the average to mitigate spikes
 for (int i = 0; i < 100; i++)
 {
  delay(50);
  sensorValue = sensorValue + analogRead(A0);
 }
 sensorValue = sensorValue / 100;
 sensorValue = (int)sensorValue;
 //Printing to the console, read by Arduino
 Serial.println(sensorValue);

 //Delay to make sure that the sensor doesn't trip over itself.
 delay(250);
}
```

3.3.2 Raspberry Pi 3b Subsystem

3.3.2.1 End User's Phone

It is an assumption of our project that the phone or connecting device of the end user (Referred to as the device) will not need to be modified in any way. As such, below are the specifications of the device that will need to be connected to the system.

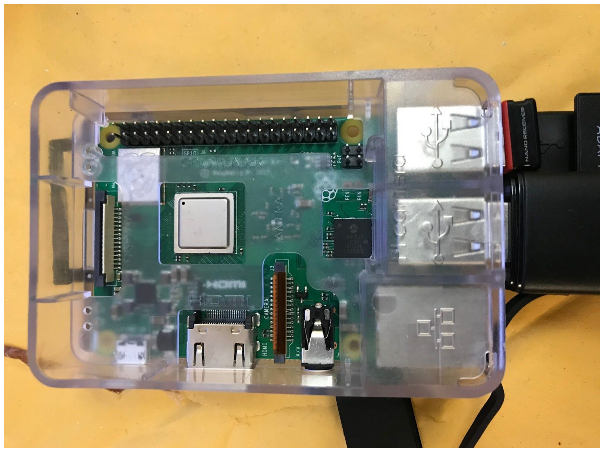
1. The device will have Bluetooth and GPS connectivity.
2. The device will be running some version of Android OS.
3. The device will be capable of successfully and consistently running the compatible software application.

3.3.2.2 Power Source

It is an assumption of our project that we will not have to design the power source The source will be capable of powering the device for long periods of time, potentially due to drawing power directly from the vehicle though this has not been tested. As such, we were capable of simulating this will power from a standard electrical outlet.

3.3.2.3 Raspberry Pi 3b

The Raspberry Pi 3b is a computer running Raspbian OS commonly used in consumer electronics projects. It has a recommended operating voltage of 5V and can be purchased from many different retailers without any difference in operation.
The Raspberry Pi will in the final product be running the code for the Bluetooth server. As part of this it will also be directly reading the data from the Arduino, which it can do in its current iteration.



Source Code: Raspberry Pi

```
#Connecting the Raspberry Pi to the Arduino
ser = serial.Serial("/dev/ttyACM0", 9600)
ser.baudrate = 9600

#Control Booleans
all_clear = True
alert = False
is_digit = False
done = False

#While loop that controls when the program runs.
#Note that done is not currently connected to anything, so this statement is equivalent
#to while True.
#done is in the program in case an emergency exit for the program is needed in the future.
while done == False:
  #Try/Except block to ensure that the data being read is legible
  try:
    b = ser.readline()
  except:
    break
  #If there is no alert currently being activated, read the next variable
  if alert == False:
    #Formatting
    b_decoded = b.decode()
    string = b_decoded.rstrip()
    try:
      float(string)
      is_digit = True
    except:
      is_digit = False
    if(is_digit):
      value = float(string)
      #Checking to ensure that we print values in the operating range. This could be not the
      #case if the sensor isn't properly warmed up.
      if (value >= 80.00 and value <= 1000.00):
        if (value >= 300):
          print("Detecting Moderate to High Levels of CO: " + string)
          alert = True
        elif (value >= 150):
          print("Detecting Low Levels of CO: " + string)
        else:
          print("Non-Concern: " + string)
          #This would be where we would need to send the signal to the Raspberry Pi
  else:
    #If we receive the signal that everything's okay, proceed with the following code.
    #The value selected for sleep is based on the 2 minute time it takes for the MQ-7 to desaturate.
    if (all_clear):
      time.sleep(60 * 2)
      print("We're all clear")
      alert = False
```

3.4 Implementation Issues

- Our Raspberry Pi used for testing implementation had an issue where, due to several malfunctioning parts and the power brick being inadequate, the sensor would fail at random after a few minutes without any regard to other factors. In addition, the sensor would sometimes send data that could not be decoded when these power fluctuations occurred.
    - The issue was solved by implementing several error catching and data parsing methods within the code for the Raspberry Pi, however it did lead to a drop in the amount of data collected.
- With the Bluetooth implementation we are currently facing an issue that does not allow us to connect to another Bluetooth device. The connection will be made with the other device, but then immediately times out. We are working on resolving this issue.
    - This was resolved by running a Bluetooth server on the other device which listens for a Bluetooth socket and then accepts a Bluetooth connection, allowing communication between the two devices.

4. Functionalities

4.1 Hardware Functionalities

The hardware has the majority of the functionality for the project that can be accomplished without a stable Bluetooth connection. The hardware functionality can be explained in terms of the two separate subsystems, the Arduino Uno subsystem and the Raspberry Pi 3b subsystem.

1. Currently, the Arduino Uno subsystem is capable of reading and tracking trends of carbon monoxide in the air from the MQ-7 sensor, and sending a signal to the Raspberry Pi .
2. The Raspberry Pi 3b subsystem is currently capable of connecting via Bluetooth to a mobile phone, or other appropriate mobile device, running the associated software application. In addition, it can receive a signal from the Arduino Uno subsystem via the Serial port.

4.2 App Functionalities Overview

The app has a majority of the functionality we intended to develop. This being an alert system, emergency contacts and Bluetooth connectivity. Where the alert system warns you when CO levels are high visually and or auditorily. The emergency contacts feature correctly stores and sends a message to the contact when an alert is triggered. Finally, the ability to enable and disable Bluetooth as well as connect to the remote Bluetooth device has been implemented into the app. For the time being the app is still in prototype mode so the alert has to be triggered manually and is not yet triggered from a Bluetooth message, that functionality is still in progress.

4.3 App Functionalities

4.3.1 Completed Functionalities

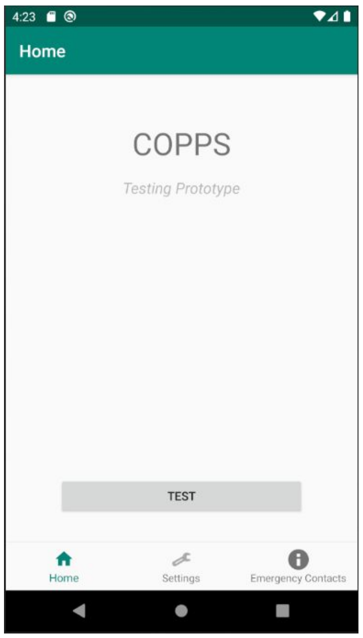
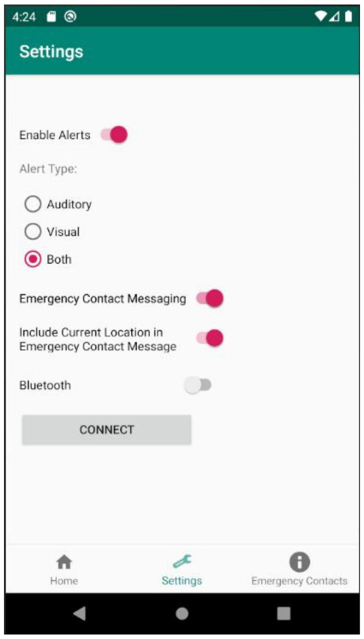


Figure 1: Home Screen



Enable Alerts [✓ on]

Alert Type:

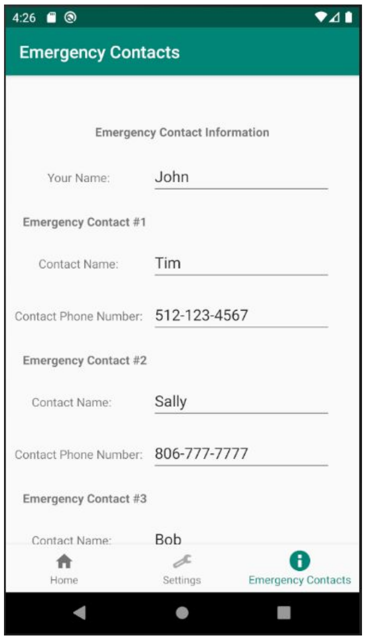
Auditory Visual [✓ Both]

Emergency Contact Messaging [✓ on]

Include Current Location in Emergency Contact Message [✓ on]

Bluetooth [off]

Figure 2: Settings



Emergency Contact Information

Your Name: John

Emergency Contact #1

Contact Name: Tim

Contact Phone Number: 512-123-4567

Emergency Contact #2

Contact Name: Sally

Contact Phone Number: 806-777-7777

Emergency Contact #3

Contact Name: Bob

Figure 3: Emergency Contacts

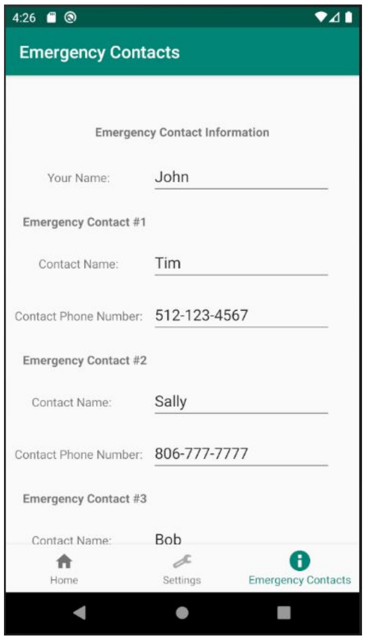
The software application has been fully implemented. Below is a detailed list of its functionalities.

Functionality 1:

- From the Settings screen (Figure 2), users are able to choose the type of alerts they prefer. In addition to that, users also can turn on or off SMS message functionality and the location sharing feature.

Functionality 2:

- After setting an emergency contact and enabling text messaging, a message will be sent to the emergency contact if the user ignores the displayed alarm. The message will inform emergency contact that the user may be in danger. In addition, the message will share user's location (if this feature has been enabled by the user).



Emergency Contacts

Emergency Contact Information

Your Name: John

Emergency Contact #1

Contact Name: Tim

Contact Phone Number: 512-123-4567

Emergency Contact #2

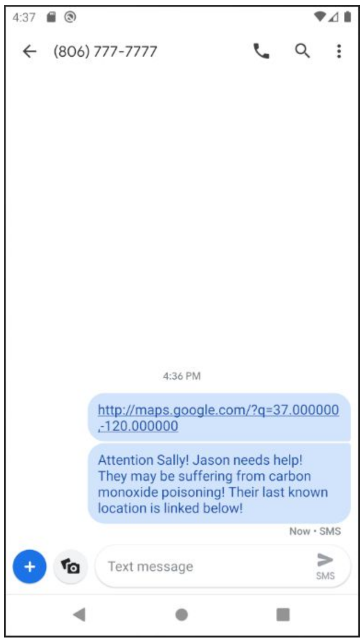
Contact Name: Sally

Contact Phone Number: 806-777-7777

Emergency Contact #3

Contact Name: Bob

Emergency Contacts



(806) 777-7777

4:36 PM

http://maps.google.com/?q=37.000000,-120.000000

Attention Sally! Jason needs help! They may be suffering from carbon monoxide poisoning! Their last known location is linked below!

Now • SMS

Functionality 3:

- After user set alert type to either auditory or both on the Settings screen, clicking 'test' button on the Home screen will ring alert. If user disables alert or sets the alert type to visual, then the device will not ring the auditory alarm.
- User is always able to turn off alarm clicking “STOP ALARM” on Home screen.

Functionality 4:

- The hardware is currently able to sense impurities in the atmosphere and can respond to changing levels. This will be easily adapted to sending signals once the Bluetooth libraries are implemented.

4.3.2 Remaining Functionalities

1. Complete Bluetooth implementation
   Bluetooth component of the project still requires the server side to be fully implemented.
2. Auto window roll down feature
   This feature has been omitted for now as it is more important to focus on the hardware and software parts of the project. However, in the future we might come up with ideas on how to implement this functionality.

4.3.3 Removed Functionalities

- Push notifications
- App running in background

5. Team Contributions

Work was evenly distributed amongst team members as best as we could, below is a description in detail of each members contribution to the project.

[redacted]- Responsible for initial research into Bluetooth, as well as the Bluetooth java class which was implemented into the main app. This class allows the App to enable/disable bluetooth, establish a Bluetooth connection, and send/receive data from the device it is connected to.

[redacted]- Responsible for the development of the application. In particular, worked on location service functionalities, SMS functionalities, auditory and visual alarm, user settings and emergency contact info, as well as data persistence and UI development.

[redacted]- Responsible for locating and connecting the different elements of the hardware, programming all files downloaded to the hardware at the time of the final report, and writing up the report sections on hardware.

[redacted]- Responsible for development of the server side of Bluetooth implementation and research into server Bluetooth implementation.

[redacted]- Responsible for server implementation of the Bluetooth functionality, initial research, and testing.