Final Project: *Eye Blinking Detection and Remote Access to IoT Sensors and Devices Design Document*

Team Hard Coders
Raymond Fey, Javier Garcia, Ifeanyi Anyika
1612920, 1266111, 2051191

Introduction

This project will allow a single client to use the eye-blinking detection application on the Raspberry Pi to control certain aspects of Internet of Things (IoT) devices connected to them, along with requesting help from a remote user. The remote user will be able to view certain sensor data, send messages, and remotely control the IoT devices from a web application on an Android smartphone.

The motivation for this project stems from the need to establish efficient communication between medical personnel and patients that have trouble speaking or moving their hands to use communication devices, and can also be extended to the deaf and people with other mental disorders. Being able to bridge the gap in fluent communication especially with technology and IoT will help medical personnel assist their patients more efficiently by enabling prompt responses.

System Architecture Design

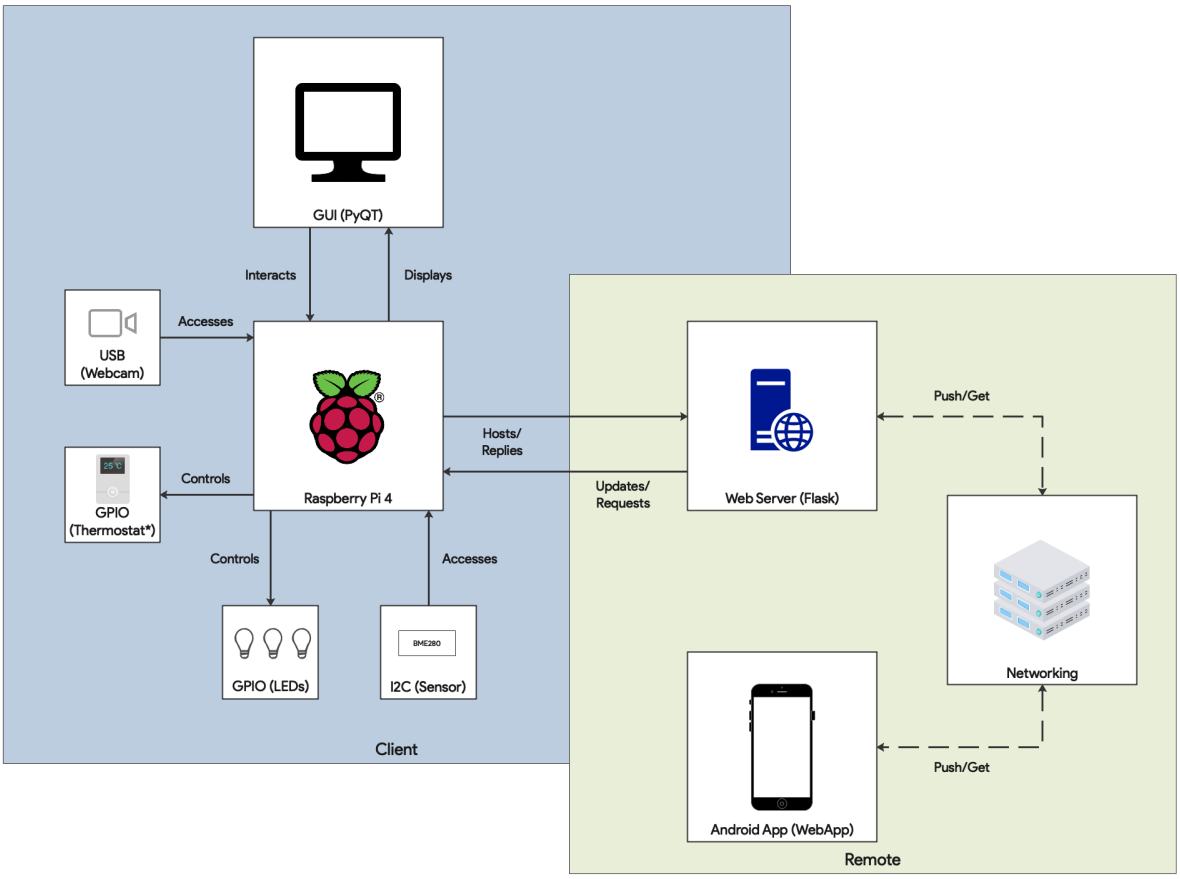

Figure 1. Top-Level System Architecture Schematic

In the blue box within **Figure 1**, the centerpiece for this project is the Raspberry Pi 4. The Raspberry Pi 4 does most of the heavy work as it runs the local application and the webserver. In the script, the Raspberry Pi connects to several peripherals to collect and configure. For the eye blinking detection part, a webcam is connected to the Raspberry Pi over USB protocols.

The IoT devices used in this project are a couple of LEDs to emulate the behavior of an actual lighting system and the thermostat isn't connected to the Raspberry Pi; however, the functionality is still present in the code for future additions. These IoT devices use the GPIO pins 35, 37, and 39 on the Raspberry Pi to communicate.

The BME280 sensor is an environmental sensor, collecting information such as temperature, pressure, and humidity and sends the information over the I2C protocol with the SCL and SCA pins on the Raspberry Pi's GPIO pins. Also, the SDO needs to be connected to the 3.3V on the Raspberry Pi to set the addressing mode of the BME280 to 0x77.

Finally, PyQt5 is used on the Raspberry Pi to run the Graphical User Interface (GUI) on the connected display, provide status updates, and allow for interaction with the other components through the user interface.

In the green box within **Figure 1**, the Raspberry Pi hosts a web server using Flask to run a local socket that can then be accessed by any device within the Local Area Network or any routable network. The website can be accessed on any device, including the Android Application. The website is used to post and get requests from the webserver and can interact with the Raspberry Pi on a bi-directional basis by sending requests and getting replies or vice versa.

The Android mobile application is a simple app that simply directs the web view of the app to the predefined socket of the webserver.

Libraries, Dependencies, and Other Software

This project is built on top of the Raspberry Pi 4 using several software packages and library dependencies to drive the application. These software packages and library dependencies are detailed as follows:

- Raspberry Pi Desktop OS Debian Buster
- Python 3.7.3 32-bit: Although an attempt to install and use Python version 3.10 was performed, some other software packages would not behave appropriately.
- OpenCV-Contrib-Python `<pip3 install opencv-contrib-python>`: This is a software library containing an open-source computer version package with pre-trained models and APIs to assist with face and eye detection in this project.
- PyQt5 `<pip3 install PyQt5>`: This is a software library that wraps the C++ version of Qt into a Python-compatible library to design Graphical User Interfaces on the platform it is on. It is important to note that qmake is a prerequisite before installing PyQt5. To download qmake, use `<sudo apt install qt5-qmake>` to allow for the source code to build and compile
- Netifaces `<pip3 install netifaces>`: This is a software package that can easily grab the network IP addresses of the machine within Python.
- Adafruit_BME280 `<pip3 install adafruit-circuitpython-bme>`: This is a Python driver file that communicates to the BME280 environmental sensor.
- Flask `<pip3 install Flask>`: This is a Python web application framework to assist in easily creating an Apache-like web server on a defined socket.

Also, for the Web Application on the Android Smartphone, Android Studio was used to develop a build of the app. The language used in the application is Kotlin.

Graphical User Interface Design

Website

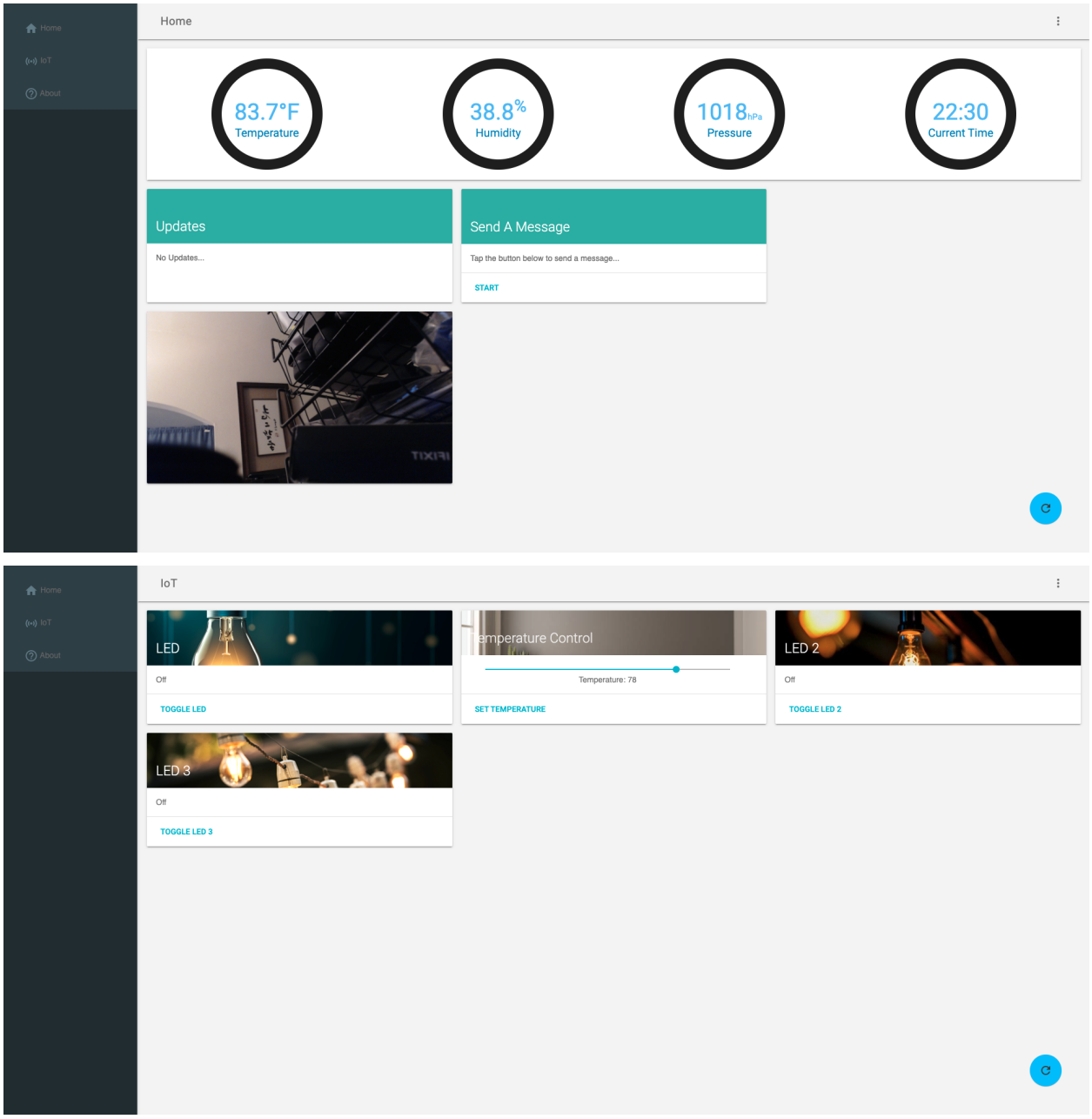

Figure 2. Screenshots of the website

In **Figure 2**, the layout of the website can be observed. The general design language used in this project is Material Design by Google for simplicity, but a sophisticated look for the user to easily understand what is presented to them. The cards, or the individual box, are used to easily

distinguish each different functionality or information from each other. Furthermore, the color choices on the website are kept to a minimum to reduce distractions.

On the top picture in **Figure 2**, the index or home page is loaded. On the home page, several important elements are displayed. The top four circles on the home page display the current temperature, humidity, and pressure values within the client's location. Also, the current time is displayed in 24hour format. Below these four circles, there are two cards for any updates that the client has sent or an option to send a message back to the client. Below those two cards, an image is shown. This image is intended to be a live stream of the client; however, in this project implementation, only still image frames can be loaded.

On the bottom right of the page, there is a floating refresh button. This refresh button should be used whenever the remote user needs to refresh the page. The remote user should never use the refresh button that is built into the browser as it may cause form resubmission and can cause duplication or corruption of data.

On the bottom picture in **Figure 2**, the IoT page is loaded. On the IoT page, a couple of cards are shown. These cards comprise three led toggling and one temperature set. Although the page is relatively empty, it is quite simple to expand and add more cards to the page by using similar card templates in the IoT.html page.

Android App

Figure 3. The Base Layout of the Android Application

The layout in **Figure 3** shows the layout of the Android application used in this project. As seen in the figure, there are very few elements comprising the interface. There is the title bar that simply mentions the App name and the Web View, which directs the web server's IP address.

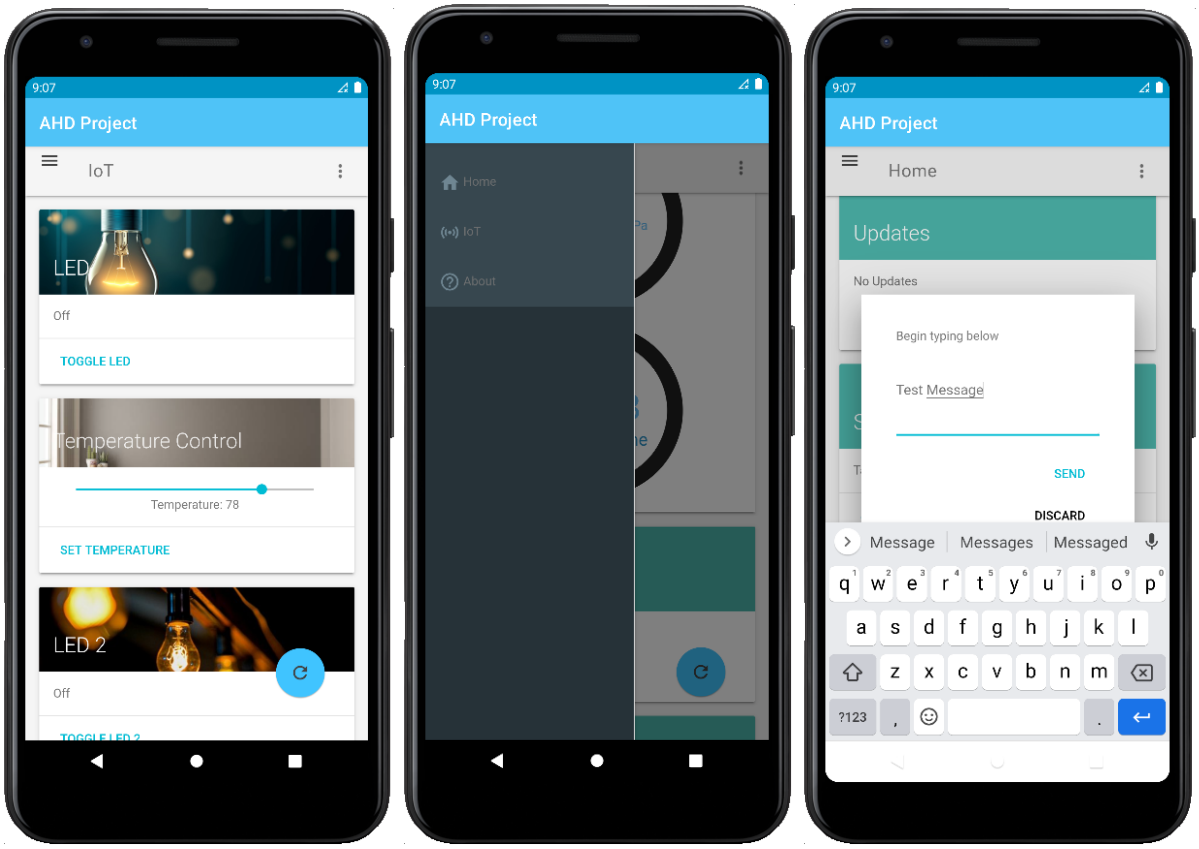

Figure 4. Screenshots of the website running on the Android application

The Android app in this project as seen in **Figure 4**, is simply a web view app. In other words, it displays the same webpage as what will be seen on other browsers through a desktop; however, the mobile version of the website will be loaded for a better user experience.

Raspberry Pi Application

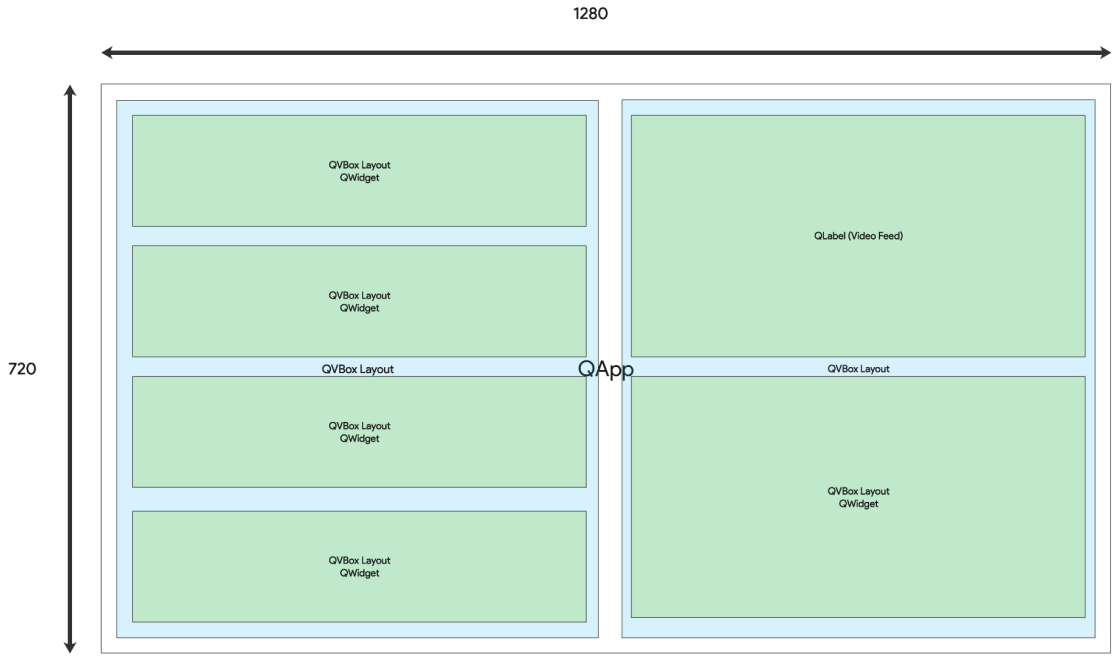

Figure 5. The simplified layout of the local Raspberry Pi Application

The local application on the Raspberry Pi uses the PyQt5 framework to build the graphical user interface. The simplified layout of the application built for this project is seen in **Figure 5**. The application window in this project is restricted to 1280 x 720 pixels and cannot be resized; therefore the display must have at least a 720p display to show properly. The rationale for choosing such a resolution is to provide enough real estate for the widgets to be drawn, but not be too demanding on the Raspberry Pi's resources.

The actual layout of the PyQt5 application contains many more layers than what can be drawn here. The QApp is the base layer of the entire application, which contains two QVBoxLayout, which are premade layouts that can easily arrange multiple child layouts or widgets within. The left side parent QVBoxLayout contains child QVBoxLayouts, which then contains QWidgets with different elements, such as QPushButtons, QSliders, QLabels, etc for manual input to the interface. On the right parent QVBoxLayout, the top contains a QLabel that displays the video feed streamed in from the attached camera using OpenCV. The child QVBoxLayout on the right contains QLabels that are linked to certain functions shared by the left QVBoxLayout; however, it is controlled through eye blinking.

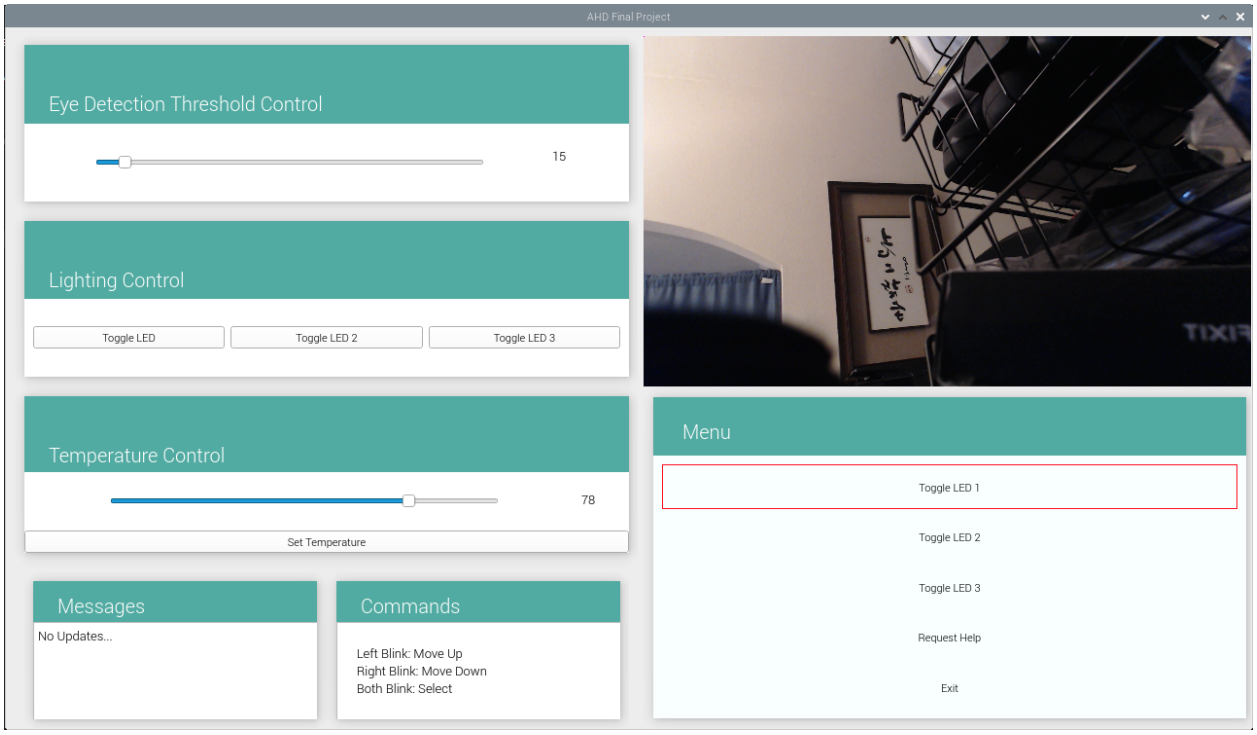

Figure 6. The actual layout of the local Raspberry Pi Application

In **Figure 6,** the local Raspberry Pi Application is running. It can be observed that a similar Material Design Card UI design is used here, albeit, a little different. The layout of the program has been subdivided into two main areas. The left side is generally composed of elements that can be interacted with a mouse, and the right side and bottom left are elements that are used by the client with their eyes.

Modules

In this project, there are two functional modules and two core modules that drive the system. The functional modules are Mailbox.py and GPIO_Test.py; whereas, the two core modules are GUI.py and test.py. Despite calling them modules, these modules are scripts with very few interfaces, especially on the core modules. The core modules are mainly self-contained software with functional modules that help communicate between the two core modules.

Mailbox.py

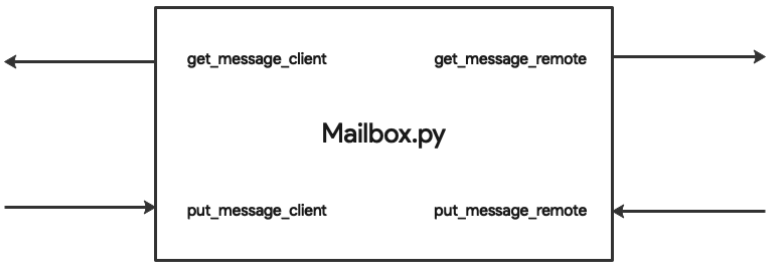

Figure 7. Mailbox.py Module Interfaces

Figure 7 shows Mailbox.py, which has four interface ports, two of which are inputs, put_message_client, and put_message_remote and the other two are outputs. This module is used for carrying and communicating messages between the GUI.py's interface and the test.py's web server message center.

To ensure clarity of each interface in this module, each interface will be detailed as follows:

- **get_message_client**: This interface is used by the GUI.py module, as a client, to return *client_msg*, which is a string that was put in by the remote user.
- **put_message_client**: This interface is used by the GUI.py module, as a client, to put in a message for the remote user and store this string into *remote_msg*.
- **get_message_remote**: This interface is used by the test.py module, as the remote user, to return *remote_msg*, which is a string that was put in by the client.
- **put_message_remote**: This interface is used by the test.py module, as the remote user, to put i,n a message for the client user and store this string into *client_msg*.

GPIO_Test.py

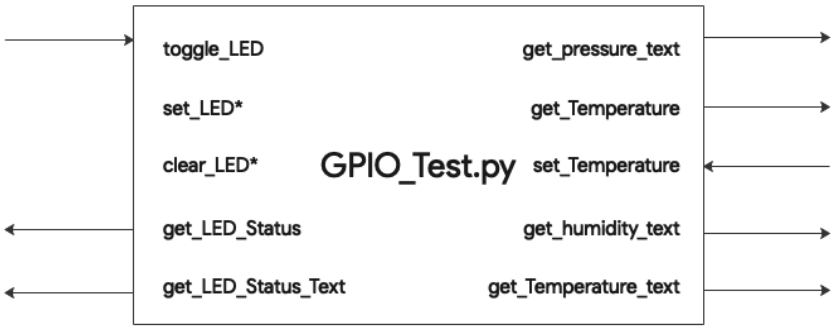

Figure 8. GPIO_Test.py Module Interfaces

GPIO_Test.py, in **Figure 8** above, has eight external interface ports, two of which are inputs, toggle_LED and set_Temperature and the remaining six are outputs. This module is used for controlling the GPIO pins on the Raspberry Pi from either the Raspberry Pi application or through the webserver. Two interfaces are available to use for other modules; however, in this project, the set_LED and clear_LED are solely used for internal purposes.

To ensure clarity of each interface in this module, each interface will be detailed as follows:

- **toggle_LED**: This interface is used by both the GUI.py and test.py to request a specific LED to be toggled either on or off depending on the current state. An argument is passed to define which LED.
- **get_LED_Status**: This interface is used to request the binary status of a specific LED with an argument passed in defining which LED.
- **get_LED_Status_Text**: This interface is used to request the status of a specific LED as a string with an argument passed in defining which LED.
- **get_pressure_text**: This interface is used to request the pressure value from the BME280 sensor as a string.
- **get_humidity_text**: This interface is used to request the humidity value from the BME280 sensor as a string.
- **get_Temperature_text**: This interface is used to request the temperature value from the BME280 sensor as a string.
- **get_temperature**: This interface is used to request the set temperature of the thermostat as an integer.
- **set_Temperature**: This interface is used to update the set temperature of the thermostat as an integer.
- **set_LED**: This is an internal interface that is used by **toggle_LED** to turn on a specific LED.
- **clear_LED**: This is an internal interface that is used by **toggle_LED** to turn off a specific LED.

test.py

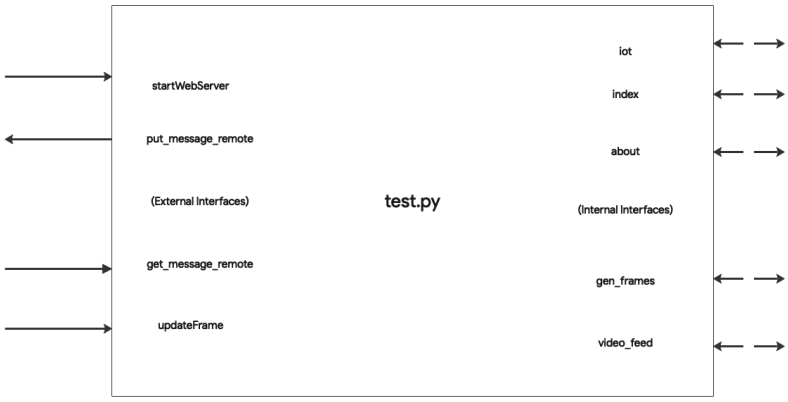

Figure 9. test.py Module Interface

In test.py, shown in **Figure 9**, there are four external interface ports, three of which are inputs, startWebServer, get_message_remote, and update_Frame, and the other port is an output. The other five interface ports are for internal uses only and not accessible by other modules. These five ports are used to run the Flask Webserver and respond to GET and PUSH requests based on the user's interaction with the web application. Module test.py is one of the two core modules that run this project. Despite the naming of this module being *test*, the actual purpose of this module is to host and control the webserver on the Raspberry Pi as well as update relevant information to the end-user or the client.

To ensure clarity of each interface in this module, each interface will be detailed as follows:

- **startWebServer**: This interface is used by the GUI.py to tell this module to initialize and start the webserver on the IP address of the Raspberry Pi's WLAN0 and port 8080.
- **put_message_remote**: This interface is used by this module to put a message to the mailbox.py module.
- **get_message_remote**: This interface is used by this module to retrieve a message from the mailbox.py module.
- **updateFrame**: This interface is used by the GUI.py to pass in the video capture frames and allow for the webserver to display these frames.
- **iot**: This is an internal interface to load the iot.html file when the remote user's web browser requests for it.
- **index**: This is an internal interface to load the index.html file when the remote user's web browser requests for it.
- **about**: This is an internal interface to load the about.html file when the remote user's web browser requests for it.
- **gen_frames**: This is an internal interface that grabs the global variable frames and generates HTML-friendly frames as JPEG images.

- **video_feed**: This is an internal interface that returns the HTML-friendly frames that can be shown on the webserver.

GUI.py

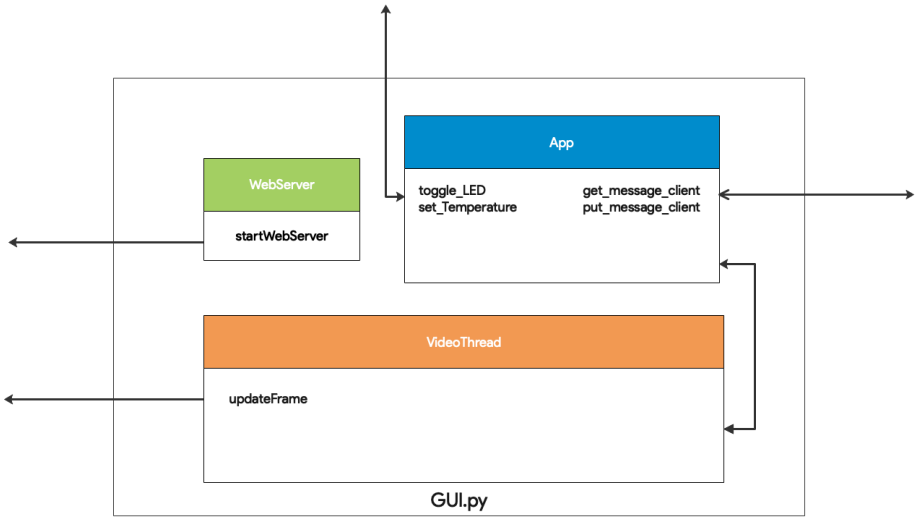

Figure 10. GUI.py Module Interface

GUI.py in **Figure 10** is the predominant core module in this project, as it uses OpenCV to capture image frames from the webcam and PyQt5 to drive the Graphical User Interface (GUI) of the program on the Raspberry Pi. GUI.py consists of three lower-level components or classes: App, VideoThread, and WebServer. The App class essentially draws the user interface elements onto the window and allows for interactions with the user. The VideoThread class is responsible for capturing frames of the user and performing face and eye detection with pre-trained models. Furthermore, the VideoThread class deals with updating the images to the Flask webserver and the video label on the Raspberry Pi application.Within the VideoThread class, there is a state machine to help determine the type of eye blinking that has occurred. The finite state machine (FSM) is provided in **Figure 11**. Finally, the WebServer class is strictly responsible for starting the webserver whenever the GUI.py is started.

Most of the interfaces in this module are self-contained, meaning that there are little to no external interfaces to other modules except for helper modules such as mailbox.py and GPIO_Test.py.

A detailed explanation of each function within the GUI.py module will be explained here.

- Class: WebServer
    - Function 1: *__init__(self)*

        - This function is always run when a WebServer object is created and accepts a parameter of "self". The function will call the startWebServer within the test.py module to start the webserver.
- Class: VideoThread
    - Function 1: *__init__(self)*
        - This function is always run when a VideoThread object is called and accepts a parameter of "self". This function imports the important pre-trained facial and eye recognition models that are needed for the OpenCV algorithm. Also, other parameters for the recognition models are pre-defined here. Lastly, the state machine and the initial states are pre-defined here for later usage.
    - Function 2: *detect_faces(self, img, classifier)*
        - This function takes in the arguments self, img, and classifier. Img is the image that was captured from the camera, and classifier is the facial recognition model that was imported in the *__init__* function. The purpose of this function is to detect and return the location and dimensions of the face. This function converts the image passed into grayscale and uses the algorithm, K-Nearest Neighbors (KNN) to find the face.
    - Function 3: *detect_eyes(self, img, classifier)*
        - This function takes in the arguments self, img, and classifier. Img is a cropped image from the camera that focuses solely on the location of the face that was detected by detect_faces. Similar to detect_faces, detect_eyes also converts the face image into grayscale and uses the algorithm K-Nearest Neighbors (KNN) to find the eyes. The function returns two arrays, the first being the left and right location based on the face and the second being the dimensions of each eye.
    - Function 4: *cut_eyebrows(self,img)*
        - This function takes in the arguments self and img. Img is the cropped image from the image that focuses solely on the location of an individual eye. The method of removing the eyebrow is based on the assumption that it is located about the top one-fourth of each eye image. This then returns the updated version of the eye image without the eyebrows.
    - Function 5: *blob_process(self, img, threshold, detector)*
        - This function takes in the arguments self, img, threshold, and detector. The argument img takes in an eye image, threshold is an integer value that controls the image brightness or darkness (can be thought of as contrast), and detector is the type of detection method. The function then calculates and attempts to find blobs or clusters of pixels with similar properties and returns the location of the blobs, or eyes in this project.
    - Function 6: *run(self)*

- This is the driver function of the VideoThread class. The image from the camera is captured here and sent to various other functions within this class. The main functionality of this function is to run through a finite state machine that is used to detect how the eye is blinking. The finite state machine is shown below in **Figure 11**.

○ Function 7: *stop(self)*

- This function call simply stops the while loop in the *run* function.

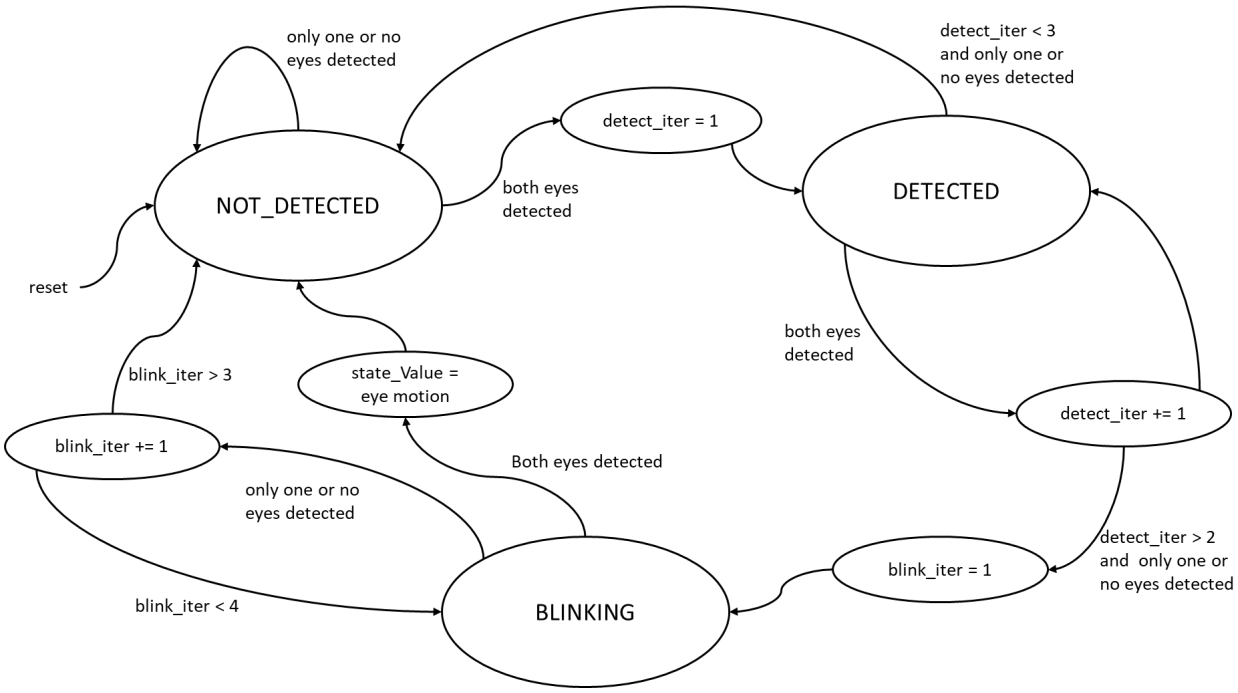

Figure 11. The FSM for Eye Blinking Detection

The FSM that determines the type of eye motion consists of three main states. If both of the user's eyes haven't been detected by the program, the state is set to NOT_DETECTED. The machine also returns to this state once an eye movement is detected or the program believes it lost track of the eyes, or if the face is no longer detected. Once both eyes are detected, the machine switches to the DETECTED state. In this state, both eyes must remain detected for a predetermined number of iterations, before being able to pass to the last state. Otherwise, it also returns to NOT_DETECTED. This helps with eliminating false positives, as it is less likely for unexpected motions to send commands if both eyes have remained detected for longer. The last state is BLINKING, where the type of motion is determined. Here, the state counts up while only one or no eyes are detected, and uses that count to know if the winks or blinks were quick or long. Whether the left, right, or both eyes were closed is saved from DETECTED, so BLINKING can determine the eye motion once both eyes are detected again. For robustness, if

both eyes aren't detected again after a certain amount of iterations pass, the machine assumes something went wrong with the eye-tracking and returns to NOT_DETECTED.

In our current application, the length of the winks and blinks doesn't yield different commands, but it could easily be extended to increase the number of things a user can do. One trade-off we had to choose is the inability to detect quick successive winks and blinks since both eyes have to be detected for a while before the next motion can be determined.

- Class: App
    - Function 1: *update_image(self, cv_img)*
        - This function takes in the arguments self and cv_img. The argument cv_img is an image that is passed in by the video capture from the camera with any modifications that were performed in the VideoThread class. This function then converts this image frame into a PyQt compatible data type by calling the *convert_cv_img* function that can be set into the QLabel inside a QWidget. This function is called every time a new cv_img is captured and updates the QLabel.
    - Function 2: *convert_cv_qt(self, cv_img)*
        - This function takes in the arguments self and cv_img. The argument cv_img is an image that is passed in by the video capture from the camera with any modifications that were performed in the VideoThread class. This function does the conversion from an OpenCV image to a QPixmap with any necessary scaling and formatting. This function returns the QPixmap for the function *update_image* to use.
    - Function 3: *changeThreshold(self, value)*
        - This function takes in the arguments self and value. The argument value is the threshold value that is passed in by reference. This value is then stored into a global variable blob_threshold, which is used by the VideoThread class.
    - Function 4: *get_Temp_Slider(self)*
        - This function simply takes in the value of the temperature slider from the PyQt application and displays the value as an integer next to the slider.
    - Function 5: *get_Threshold_Slider(self)*
        - This function simply takes in the value of the threshold slider from the PyQt application and displays the value as an integer next to the slider.
    - Function 6: *updateMsg(self)*
        - This function taps into the mailbox.py module and attempts to get the message that the remote user's message was put in. It calls the mailbox.get_message_client() and sets the plain text to that string.
    - Function 7: *updateTemp(self)*

        - This function taps into the GPIO_test.py module and retrieves the set temperature of the room and updates the slider on the PyQt application.
    - Function 8: *move_up(self)*
        - This function is used to move the red box up by clearing the style sheet of the current QLabel and decrementing the index and then sets the style sheet of the new QLabel on the PyQt application. This function is called by the *blinkControl* function.
    - Function 9: *move_down(self)*
        - This function is used to move the red box down by clearing the style sheet of the current QLabel and incrementing the index and then sets the style sheet of the new QLabel on the PyQt application. This function is called by the *blinkControl* function.
    - Function 10: *select(self)*
        - This function is used to select the command highlighted by the red box. This function checks the index value and calls other functions based on the value. This function is called by the *blinkControl* function.
    - Function 11: *blinkControl(self)*
        - This function is used to determine what the user intends to perform with their eye blinking, such as moving up/down or selecting the command within the PyQt application.
    - Function 12: *__init__(self)*
        - This is the driver function of the App class. All the graphical user interface elements are initialized, set, and stylized. Furthermore, there are a few QTimers that schedule function calls to update certain elements or perform certain duties at a certain time interval.

Design Compromises and Complications

- Video Live Stream
    - The Feature:
        - It was a requested feature to also stream the live stream recording from the camera onto the Flask Web Server, similar to the local Raspberry Pi's App Video Feed.
    - The Issue:
        - To enable live streaming on the Flask Web Server, the argument <use_reloader = True> must be set to enable streaming without the need to manually GET the webpage by the client's refresh request. The catch is that to do so, the Flask Web Server must be the main thread. This is further complicated by the fact that the local Raspberry Pi application using PyQt5 also needs to be the main thread; however, this isn't possible because we needed both to start at the same time. Therefore, a decision to choose which one to prioritize was required.
    - The Options:
        - Option 1: Create a second Flask Web Server on Port 8081 that is asynchronous from the local Raspberry Pi's application and embed this HTML page into the main Web Server on Port 8080. However, an issue arose quickly that the Raspberry Pi could not handle both cameras at the same time without running out of resources.
        - Option 2: Use JQuery to refresh certain parts of the page without requiring <use_reloader = True> and pass in values with JSONify. The issue with this is that JPEG images cannot be easily JSONified without a lot of manipulation to the data. In some cases, it may corrupt the data.
        - Option 3 (Worst case scenario): Give up on live streaming on the Web Server and only allow for single frame rendering on the web page per GET request.
    - The Decision:
        - Unfortunately, Option 3 was chosen due to Options 1 and 2 not working properly. Although we wanted to implement live streaming on the webserver, it is a feature that may only be implemented with more time and a code rewrite.

- Native Android App
    - The Feature:
        - It was a requested feature to have the OpenCV and the eye blinking software run on a phone and send information over Bluetooth or Wi-Fi to a Raspberry Pi to control the GPIO. This is to offload the processing

power from the Raspberry Pi, which typically has less computing resources than a typical modern smartphone.

- The Issue:
  - An attempt was made to run OpenCV on both iOS and Android; however, a lack of experience in app development in both Swift and Kotlin made it difficult to implement.
- The Options:
  - Option 1: Use Unity game development framework and C# to run the OpenCV software on an Android smartphone. There was a YouTube video online explaining how to do so; however, it fails to implement the OpenCV live streaming or provide source code on how to fully and correctly implement the feature. The video wanted the user to pay hundreds of dollars to unlock the full version of the video and gain access to the source code.
  - Option 2: Make a simplified smartphone web view application that simply shows the website.
- The Decision:
  - Option 2 was chosen due to sufficient knowledge in HTML and CSS programming to make a webpage look decent. Also, the website was written for two types of devices, a small display, such as smartphones, or large displays, such as tablets or desktops. This made it easier to develop the smartphone application, as it is simply a hard-coded browser to always direct to a static IP address of the webserver.
  - This decision effectively meant that the Raspberry Pi had to handle everything, which can be very stressful on the limited hardware resources and may cause some freezing; however, since time was a constraint, this approach was chosen to deliver on a similar experience in a short amount of time.

- Menu Options within PyQt Application
  - The Feature:
    - It was requested to have a well-built-out menu structure that can be navigated and functions to be controlled with eye blinking.
  - The Issue:
    - Time was a constraint and needed something to work. Furthermore, the eye blinking detection software sometimes has trouble detecting our eyes, which makes navigating through an entire menu difficult, and making a full menu will mean much more time spent layering widgets. This will result in hundreds of lines or more code that needs to be written if pursued.
  - The Decision:

- Our group decided to restrict our menu options to 5 options to demonstrate that eye blinking detection works as intended. If time was not a constraint, then a more sophisticated navigation menu can be created with more options.

Code Listing

Due to the large size and length of the code and quantity of files, please access the GitHub repository using the following link or access the Code Files in the downloaded zip file.

https://github.com/fey432/AHD-FInal-Project