BRO-NS: Bluetooth Remote Operated Nerf Sentry

Juan Landron, Daanish Habibi, Ross Ellis, and Kyle Quarti

Dept. of Electrical and Computer Engineering, University of Central Florida, Orlando, Florida, 32816-2450

***Abstract* — Many forms of autonomous control rely on complex systems of cameras and motors that drive up the cost of a product. By implementing inexpensive and simple components we created a turret that will allow the user to autonomously track and engage a target while remaining inexpensive to build and easy to assemble.**

***Index Terms* — Autonomous systems, Bluetooth, Face Detection, Computer vision, Ultrasonic detection.**

I. Introduction

The systems that make up an autonomous machine are directly dependent on the task that the machine is trying to accomplish. For our project we wanted to design a Bluetooth controlled Nerf gun that would also allow it's user to enable a "sentry" mode, where-in the gun would be able to aim and fire itself at a desired target. To accomplish this task, our project is made up of four main subsystems; computer vision to enable facial recognition, a color imaging sensor to detect a target at longer ranges, an ultrasonic sensor to detect distance in a close proximity, and the Bluetooth interface to allow the user to easily control the turret from an external device. All of these subsystems allow our turret to identify a target at a variety of ranges so that the turret can operate in a wider variety of locations.

The primary design philosophy for this project was to design the turret to be as compact and as lightweight as possible to aid in portability and maneuverability. We also designed the turret to run on low power so that it could effectively be powered by small battery packs for an extended amount of time.

II. System overview

Outside of the four main subsystems that will be discussed in sections III. – VI. this section will briefly discuss the other vital components used in our project.

A. Microcontroller

Our project is run off of two microcontrollers; a custom Arduino based PCB that runs data and power to our motors and sensors, and a Raspberry Pi 4 that is used solely for computing the vectors and coordinated needed for the facial recognition. At the heart of the Pi 4 is a 64-bit 1.4GHz quad-core processor and 2GB of RAM, all of which is needed for the algorithms to run the computer vision software.

For our custom designed board, we decided to use the Atmega328p. The Atmega328p is an 8-bit RISC based chip, and is much less powerful that the Pi 4, but we only need it control two servos, an ultrasonic sensor, and be capable of communicating with the Pi 4 over UART or SPI to relay the coordinates of the computer vision to the servos.

B. Servo Motors

The Towerpro MG996R is used to control the x-axis of our turret. It has a stall-torque of approximately 11.2Kg/cm and a relatively low stall current of 1.25A. For the y-axis servo, we needed a motor with a higher stall-torque as the Nerf gun is not balanced very well and a lot of the weight sits over the rear of the gun. We went with the LJWRC metal geared servo as it has a stall-torque of 25Kg/cm and its stall current of 1.5A is still within our power range.

C. Nerf gun

We chose to use a Nerf gun for our project as it was small, lightweight, and a safe way to fire a projectile. We established the effective range of our turret by testing the "bullet" drop of the foam Nerf projectiles over a range of 5m and 10m. Using the results from the figure 1 below, we were able to establish the drop compensation for targets at various ranges to ensure our turret was as accurate as possible.

Barrel Height	Dart 1	Dart 2	Dart 3	Dart 4	Dart 5	Average	Bullet Drop
25"	12"	17"	11"	17"	9"	13.2"	11.8"
42"	7.5"	14"	9.5"	26"	14"	14.2"	27.8"

Figure 1: Nerf Gun Bullet drop test

D. Batteries

We chose to use two separate battery packs to power our turret due to the large difference in power demands of our two microcontrollers. Our Raspberry Pi 4, webcam, and Pixycam are all powered by a 10000mAh portable battery bank. The battery bank is able to supply 5V at up to 2.5A. The total net power draw of all of the electronics in this part of the system is approximately 600mAh, and is well within the capabilities of this battery pack.

The second battery pack is being used to power the Atmega328p, the Bluetooth module, our servos, and the ultrasonic sensor. Most of the components in this part of the system only draw a couple hundred milliamps combined, but the servos are each capable of drawing over an amp at peak stall torque. In order to meet these power demands, we chose to use a high performance lithium-polymer battery that is rated for a continuous output of 7.4V at 10A.

III. Computer Vision Servo Control

The computer vision will be handled by the Raspberry Pi 4 which will be communicating whether a face has been detected and where it found the face in relation to the image. In order to process the images, decided to use the open-source library called OpenCV. This library helps us with the facial tracking feature which we are coding in python as it was a familiar and widely used language for artificial intelligence programming often. The custom board will get the information from the Pi 4 along with the Pixycam information regarding the face detected and whether it is a potential threat.

The first aspect of the computer vision will be the facial monitoring which will try to differentiate a face from the live feed that it receives. This is where the OpenCV library comes into play as it supports the execution of machine learning which is a beneficial and important part of facial recognition. To even achieve facial recognition and try to interpret what defines a face, machine learning is using the software to match content for similarity. We perceive pictures as colors which include blue, red, white, and many others whereas technology perceives them as numbers. Therefore, each color will have a predefined number assigned to it, which will be used in an array to form pictures. These numbers will be defined and referenced using the Haar Cascade as it will be the fastest to use to provide live feed back to the servos to help track faces.

Faces have a unique outline to them that are defined as the borders when referenced to the surrounding area due to the colors and color differences. These are referred to as edges in face tracking as an edge will represent when a face starts and when it ends.

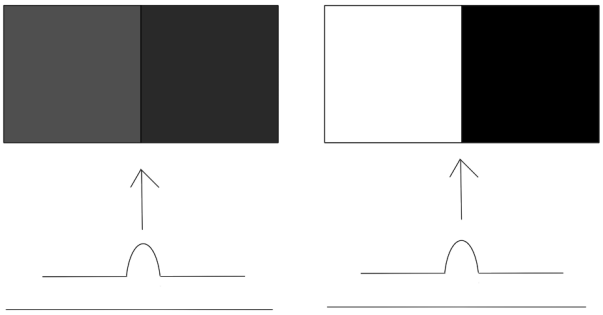


Fig. 1 Edge Identification

As shown in figure **BLANK**, these edges are defined by the color differences between a section is what is identified as a form of an edge as it marks the start of a new section. Using this there are specific patterns that the code will be looking for which helps to identify if a face is present or not.

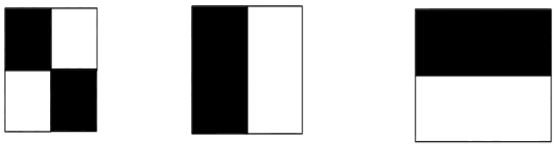


Fig. 1 Defining an Edge

Some of the edges are shown in Figure **BLANK** which shows how an edge can be shown, the black area representing a 1 and the white area representing a 0. The figure to the far left references a diagonal line or a form of a curve, the middle references to a defined vertical line, the image to the right is a horizontal separation of colors. For the turret to even determine a face it needs to run calculation with preset definitions from a database which are usually in the form of pictures. It is a three-step process for setting up the determining factor on whether a face is present and where it is. The first step is taking an input of pictures and letting the program or algorithm determine whether there is a face present in which case it will focus on it. the second is taking the results to see the accuracy of it and letting the algorithm redetermine its parameters for what is a face and what is not. The last step is using the final calculations to determine the pictures or live feed coming from the camera as a face or not.

The last step for facial monitoring is keeping the view of the face if it has been detected. To accomplish this the algorithm will constantly scan the whole range of the cameras viewing angle and get the location of the face regarding the cameras view.

When using OpenCV it requires an image from the main camera to so once it receives an image, it will decipher it into where the face is in the picture. It then returns a pair of coordinates x, y, w, h. The x and y coordinates reference the location of the face while the w and h represent the size of the frame. In order to get the center of the face we would need to do some calculations as the coordinates received are the references to a cutout of the face that OpenCV returns.

$$(x + x + w) / 2$$

Fig. 1 Center of Object Formula

As shown in figure **BLANK** if we take x and add itself and add it to the width which will all be divided by two. This will return the relative center of the face with reference to the pixel that it is sitting on. In this case the center of the face will range from 0 – 500 because the image that we are using is 500 x 500 once it has been rescaled. The rescaling is done to keep the speed of the image processing to the point where it is working smoothly and not lagging the movement of the target that is supposedly in front of the turret.

Each target will need to be scanned to make sure they are a valid target rather than a random individual walking by. This is where the color image processing from the Pixycam will track the person with specific colors on them. Only to initiate the firing sequence when the specified color along with the confirmation from the raspberry pi that the object is a person will it light the laser at the target.

Another aspect of utilizing OpenCV allows the modularity of how the image processing works or even to use different libraries to track different objects. This is done through the xml files that OpenCV has that define an object that you are trying to reference. For example, for this project we are using a facial recognition for a person but there is also recognition for the overall human figure or even animals. Therefore, we can confirm a target with the specified color on the object along with confirmation from the facial recognition program that it is indeed the object that we are searching for.

IV. Color image processing using Pixycam

The Pixycam 2.0 is able detect colors and recognize them through programming such colors to be interpreted as “friendly” or “hostile”. It can do this through a framerate of 60fps while at a resolution of 640x400 which requires that the Pixycam 2 be able to process and send 1Mbits/sec through its serial interface. When using a slower serial data link like UART, however, the Pixycam may not have enough time to properly analyze and upload the photographs before the next frame is identified and processed. Only the most recent data is processed and delivered to enhance data processing efficiency, resulting in earlier, unprocessed frames being discarded. Smaller things may be missed or unnoticed if the serial data interface cannot analyze the photos quickly enough, as the Pixycam is built to prioritize larger objects.

Using the supplied "Pixymon" software, the serial data interface may be manually selected. The Pixymon software is a Windows and Linux executable program that lets you adjust the Pixycam's basic characteristics without having to update the data interface using coding functions. As we are connecting the Pixycam 2 to an Arduino, this will be easily achieved with SPI since Arduinos have a 1MHz SPI clock rate that is adjustable. It also contains a light source to help with detecting targets in lower light conditions which helps save on more complex designs that include a higher power draw and integrating additional circuitry of an external light source. We will have each target wearing a certain color combination to be recognized and tracked by the Pixycam. The Pixycam 2 has quite a low power usage of 140mA(5V) for a hardware and software integrated camera system. This will create a very small impact on battery life for our sentry whilst incorporating many features that otherwise might not have been included.

The Pixycam 2 connects via I/O connectors and uses micro-USB interface to exchange data. The Raspberry Pi 4 will be powered by the 5-volt input provided by the Anker power supply. With the Raspberry Pi 4 consuming just 0.6 amps and the Pixycam drawing only 0.14 amps, the Anker battery bank can offer up to 2.4 amps of electricity. The Raspberry Pi 4 has built-in voltage control, so it will be able to supply the Pixycam with the necessary quantity of power.

The data between the Pixycam and the MCU is transferred over a wired link via the micro-USB interface. The Pi has a total of twenty-six GPIO pins, while the Pixycam draws power via a six-pin connection. Twenty-six of the forty output pins of the Raspberry Pi are GPIO, two are 5-volt supply, two are 3.3-volt supply, and seven are ground. A micro-USB cable will also be used to

connect the Raspberry Pi 4 to the master MCU. A certain hue will be set into the Pixycam to be interpreted as "hostile." The Pixycam will communicate the required data to the Raspberry Pi 4 when it detects a hostile color.

After that, the Pi 4 is utilized to communicate with the master MCU. By allowing the secondary MCU to handle all the visual data, the master device will be relieved of some of the effort. We considered having the secondary MCU control the target's range detection, but we wanted to keep the amount of time spent sending data between the two MCUs to a minimum because we want our master MCU to control all our motors and movement. If a single MCU is overloaded, the registers in the MCU may get overwhelmed, and the microcontroller will be unable to keep up with the orders required to conduct the movement and tracking accurately.

The Pixycam is programmed with the help of a supplied utility and solely communicates with the MCU to deliver data. The Pixycam will be powered by the Raspberry Pi's GPIO pins, which will provide a regulated 5-volt micro-USB input. The Pixycam will require six total pins of the Raspberry Pi's GPIO output, as determined by the GPIO cable provided with the Pixycam.

Because the Raspberry Pi has greater capacity and can manage the bigger memory footprint required by the USB protocol, the Pixycam utilizes the USB interface to send data to it. If we use an MCU other than the Raspberry Pi 4, the Pixycam will have to connect with the MCU using a serial data interface rather than the USB protocol.

The Pixycam communicates with the Arduino family of MCUs utilizing SPI using a supplied cable. Because of the amount of data that can be shared through the SPI interface, it is the preferred serial data interfacing method. The Pixycam can recognize and analyze images at a rate of 60 frames per second and can detect 155 objects per frame. The Pixycam captures photos at 640x400 resolution while the image sensor is running at fifty hertz. Although the Pixycam's maximum resolution can be set to 1280x800, it can only record images at a rate of 25 frames per second.

The Pixycam will be able to detect smaller items or objects that are further away with a higher resolution, but this will demand more CPU and memory resources to encode and convey the data. The Pixycam must be able to process and transfer around 1Mbits/sec of data across the serial interface while processing photos at the 640x400 resolution. This is simple to achieve with SPI, as most Arduinos have a 1MHz SPI clock rate that may be increased if necessary.

However, when using a slower serial data interface like UART, the Pixycam may not have enough time to properly process and send the images through the UART interface before the next frame is recognized and processed. Only the most recent data is processed and delivered to enhance data processing efficiency, resulting in earlier, unprocessed frames being discarded. Smaller things may be missed or unnoticed if the serial data interface cannot analyze the photos quickly enough, as the Pixycam is built to prioritize larger objects.

To begin detecting and recognizing objects, we use a hue-based color filtering algorithm to learn an object with a distinct hue which means anything with grey, black, or white cannot be used. After powering on the Pixy2 you can hold down its integrated button to proceed with a series of LED's which denote each signature. When you release the button, it will enter what is known as "light pipe" mode where the Pixy2's LED attempts to match the color of the object placed in front of it. Pixels that are part of the supplied object and that of what makes up the background are distinguished and determined by the region growing algorithm. A model is created to allow the Pixy2 to detect the supplied object under variable lighting conditions.

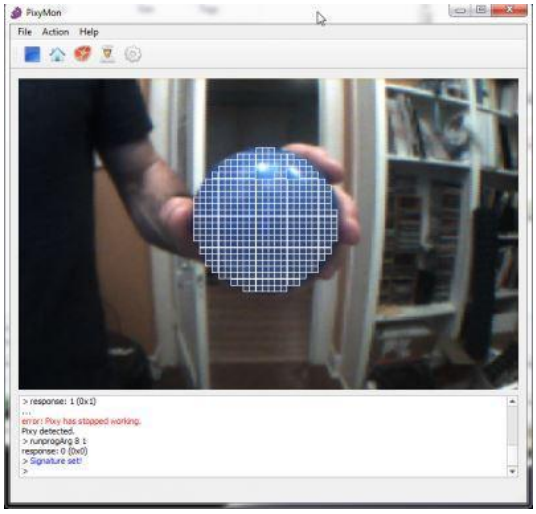


Figure x: Pixycam Object Detection

Using the supplied "Pixymon" software, the serial data interface can be manually selected. The Pixymon software is a Windows and Linux executable program that lets you adjust the Pixycam's basic characteristics without having to update the data interface using coding functions.

We programmed the servos on the turret to pan and tilt in accordance with the detection of the "learned object" through the Pixy2. We find the x and y coordinates of the object within the pixels of what the Pixy2 can see and check to see if it falls in the bounds of camera to thus write to the servos for the pan and tilt instructions. We also program the servos to move at the speeds in which we

desire to acquire the target as rapidly or slowly as needed before the turret fires upon the target.

We chose color image processing as a secondary to the facial recognition through a more basic methodology towards target acquisition as on its own it cannot accomplish facial recognition.

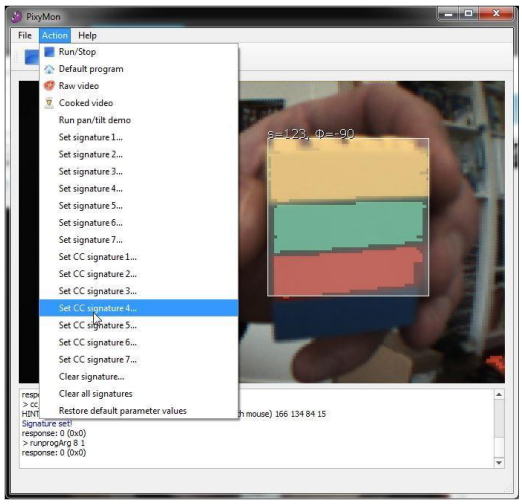


Figure x: Pixycam Color Signature Detection

V. Ultrasonic range indication for proximity targeting

Ultrasonic range finders use timed sound pulses to detect whether an object is in front of them, similar to how a bat would detect it's prey using echo-location. By emitting ultrasonic frequency pulses, and then waiting for the emitted waves to bounce off of an object and then return, the sensor can use simple mathematic equations to determine how far away the object is. The following equation shows the distance calculation of the ultrasonic sensor.

$$Distance = (0.5)\ T\ x\ c_s$$

Where T is the time delta from the signal being emitted, until it is detected, and c_s is the speed of sound.

Because ultrasonic sensors use sound waves to detect objects, it is harder to focus the sensor on one specific object at longer distances. The further away an object is from the sensor, the more likely other objects within the immediate environment will reflect those sound waves and interfere with the signal. The sensor will also not work properly if there are any larger objects between the sensor and the target. Due to the nature of how the sound waves reflect, ultrasonic sensors work best if their target is large and has a flat surface. Smaller objects will not deflect enough of the signal back to the sensor, and softer textured objects will absorb some of the signal causing it to work less effectively. In order to achieve the most consistent results using our ultrasonic sensor, we used a human-sized cardboard cut out as our target. It's flat surface and relatively large dimensions gave the sound waves adequate surface area to reflect off of.

Typical implementations of ultrasonic sensors are for anticollision systems on small robots as they have very minimal range, typically only a few meters. We chose to implement the ultrasonic sensor into our project as a means of proximity detection. With our main imaging sensor relying on detecting the features of a human face, and our color imaging sensor only picking up on color patterns that we have taught it, we wanted to have a means of detecting a target that may have potentially eluded our other means of target detection. We have set up our ultrasonic sensor to detect objects within four meters of the front of the turret. If no target is detected, the sensor returns a null value to the serial monitor. If an object is detected within four meters of the turret, a digital signal is written to the laser diode to turn it on.

VI. Testing Procedures and Compatibility Validation

Each of our project's primary systems; facial recognition, Bluetooth operation, color and image recognition, and the electrical systems, were each developed independently by one of our group members. This was done to minimize the amount of travel required for us to develop our turret and to increase productivity as each member was in charge of a specific task. To ensure the best possible chance of all of our parts working together, constant communication about standards and specifications was needed so that all factors regarding power requirements or coding protocols would be accounted for.

We ran stress tests for each of our systems to ensure there were no underlying bugs or errors that would cause our project to malfunction over prolonged use.

For the power supplies we used in our project, we ran looping code to run the servos indefinitely in order to completely drain the batteries to see what the effective lifespan of the turret would be on a single charge. With our y-axis servo drawing a peak stall current of 1.35A, our x-axis servo drawing up to 1A at full load, and the Atmega328p drawing somewhere in the range of 100mA to 200mA, we were expecting our 2000mAh battery to last just under an hour. Upon testing we were able to achieve a battery life of fifty-three minutes. We expect our real-world scenario battery life to be slightly over an hour as the turret would not be continuously moving unless a target was detected. For the second battery that is used to power the Raspberry Pi and the two camera sensors, we did not do an extensive battery test as the capacity of the battery is 10000mAh and the combined current draw of all

components powered by it is approximately 600mAh. This would give us an effective battery life of over ten hours.

To test the Pixycam, we experimented with several different colors and shaped targets in order to determine which would be easiest for the Pixycam to pick up. We found that using bright, high visibility colors such as oranges and greens allowed the Pixycam to easily distinguish the object from its surroundings.

VII. Android Application: Bluetooth operation and connectivity

For the Bluetooth connectivity, we decided to create an android app due to the relative ease of creating and downloading the app onto the device while having many resources that help in understanding and assisting in the development of the app such as developer guides that explain the methodology of how to solve specific problems. We decided to use Kotlin as our programming language with the Android Studio IDE to help with the construction of the UI with a built in GUI that has the many UI parts that can be used to just drag and drop into a model of the app and would perform the necessary XML programming without the needed to type it out. Android works by having a layout, the UI component, and an Activity, which is the functionality of the app. The layout is tied to an activity to facilitate the construction of the operations of the app so that users can interact with the app using the UI. Each layout can only be tied to one Activity and for every new UI, there is an Activity class that gives it functionality. The Activity class is where the Kotlin program will be located and they are also used to call another activity which will shift the app to the other Activity and then will call the layout it is associated with and inflate it, meaning it will construct the UI and display it.

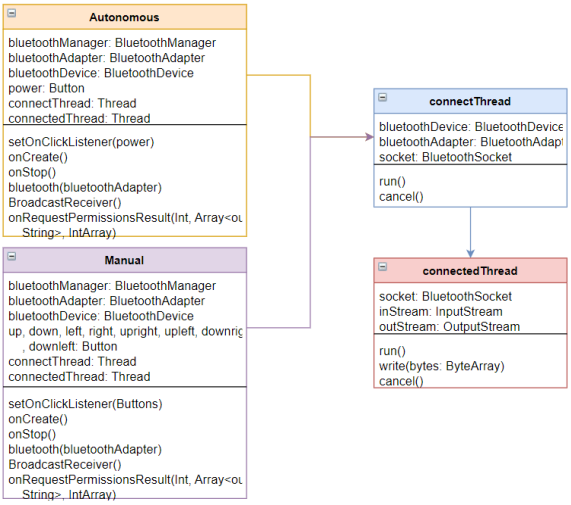


Fig. 2 Bluetooth mode select protocol

A. User Authentication

For the android application, there were some requirements that were imposed in order to make an app that has useful features. The first thing is a sign in with an email and password authentication. The authentication for the app uses Google's Firebase as there are built in features in the Android Studio IDE with libraries and functions that make it seamless to use for registration and verification of email and passwords. The steps that were performed to integrate Firebase with the android app was to first register the app to the Firebase project, which requires the android package name to identify the app that the authentication server must be able to be called upon. Then, add the Firebase configuration file, a JSON file, to the app's directory, giving it the associations and resources of the server needed to call the API's for operating the functionality of the authentication. Finally, the last step of the setup is adding the dependencies and plugins required in the Gradle files of the app which will grant access to all the libraries that can call the functions of the authentication.

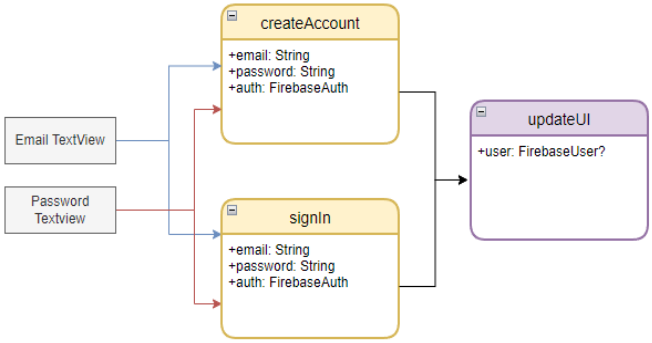


Fig. 2 Entity State Diagram of User Authentication

The user authentication layout will have two input textviews, widget objects that obtain text from the user, and will obtain those texts as strings to pass to the Firebase server. This is shown in figure 2, where there is an email and a password textview and will pass the information onto one of two functions, the createAccount or the signIn. The auth object is a FirebaseAuth value that is used to call the functions for signing in or registering. Once the user decides which function to use, they both call the function updateUI that takes a FirebaseUser? and makes a conditional check of whether the user has successfully signed in or registered based on if that value is null or not. If successful, the function will call the next Activity, which will inflate and populate the UI to the app.

Otherwise, it will maintain the current UI signaling a failure.

B. *Bluetooth Communication*

The Bluetooth communication has the hardware component and the software component. The hardware component is the specific Bluetooth module we are using, which is the HC-05. The default baud rate and the one that we will be using is a 9600 baud rate. HC-05 is used to communicate the app to the BRO-NS nerf turret. It is supplied with a DC voltage of 3.3V and is connected to the RX and TX pins of the Atmega328p to receive the commands from the android application. The software component starts with adding a uses permissions call in the android manifest file, which determines what the android application is able to use and specify Bluetooth connect and Bluetooth scan as two different uses permissions to allow the app to have access to calling system applications that will handle the action of scanning Bluetooth devices and connecting to them. Once the manifest gives the app access to these commands, we first then check whether the app is allowed to use Bluetooth by asking permission from the user. This is due to androids' policy on Bluetooth which dictates that these permissions are runtime permissions, therefore the user must agree to allow the app to use Bluetooth on runtime. After the check on Bluetooth user permission, the app checks if the Bluetooth state is on or off. If the state is off, we call the android system to turn it on after we make a check on whether we have the permission to call to the android system. Once on, the app will then get a BluetoothManager object from the android system that will be used to acquire a BluetoothAdapter. The adapter is needed because it is the object that will be able to call the API functions to determine the state, determine and control connectivity, and find paired devices. The app will then first call cancelDiscovery, which will end the discovery of Bluetooth first in case there is already an ongoing discovery at which point then we call the bondedDevices which will give us the paired devices. If we do not have the Bluetooth paired, then we use an Intent which calls a BroadcastReceiver object that gets the Bluetooth devices. Once the BroadcastReceiver finds a device that is the Bluetooth we are using, it will then call a thread class called connectThread that will open an RFCOMM Socket that takes in the UUID of the app, which will need to be the same as the UUID of the Bluetooth in order to connect, and then obtain the socket of the Bluetooth, this is a blocking function, which means it requires a separate class, an inner class, that is threaded because it takes up the resources of a class. The connectThread then calls a function that will call another thread called connectedThread. This thread will retrieve the input and output streams of the socket, then has two functions, run() and write(). The run() function holds the input stream and calls the read function that will obtain the bytes sent from the turret to the app. To obtain the whole message, the function has to send the bytes to a Handler object that will obtain each byte and construct the message. The write() function holds the output stream which will call output stream's write and give it the parameter of a string converted into a byte array to send to the Bluetooth. The read and write functions of the streams are blocking functions. Due to this, the app needed to have this threaded class. The fun() function that has the read will continue to operate until there is nothing being sent and the read returns an error that is caught with a try and catch. Once this error is caught it exits, freeing the resources and allowing any writes the app calls. These functionalities are essential for the operation of the Autonomous and Manual modes of the turret as both of these modes have to communicate to the turret by connecting through the Bluetooth and sending commands to the turret.

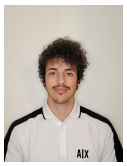
C. *Autonomous Mode Operation*

The autonomous mode has a simple UI that has a button to connect to the Bluetooth and another button that will toggle the turret on and off. The button will first start by implementing a toggle on that will be sent to the turret which will turn on the motor and firing mechanisms and allow the turret to operate independently of the user, except for the toggle operation. If pressed again, the app will send an off command and the turret will be stationary but ready to receive an on command.

D. *Manual Mode Operation*

The manual mode has a layout UI with a button to connect to Bluetooth, similar to the autonomous mode. Both modes will have the ability to connect to the Bluetooth because it gives the user the ability to connect to Bluetooth in whichever mode they desire. The other buttons are the directional buttons and the fire button. The fire button is used to send a fire command that will tell the turret that the user wants to fire. The other buttons, the directional buttons, are eight buttons that hold the traditional directions of up, down, left and right as well as the directions in between up right, up left, down right, and down left.

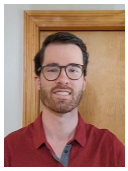
ENGINEERS



Juan Landron (C.P.E): Juan is pursuing a career in computing components fabrication and research and would like to be employed by AMD.

Daanish Habibi (C.P.E): Daanish is currently employed as an associate server engineer for SightPlan, and wishes to continue pursuing jobs in the field of backend server maintenance.

Ross Ellis (C.P.E): Ross is currently employed as an I.T systems administrator for NextEra Energy.



Kyle Quarti (E.E): Kyle is pursuing a career in the field of utilities engineering, and wants to work for local municipalities as a power grid engineer or as a transformer and relay engineer.

References

[1] Cook, Jeremy S. "Ultrasonic Sensors: How They Work." *Arrow.com*,4Apr.2018, https://www.arrow.com/en/research-and-events/articles/ultrasonic-sensors-how-they-work-and-how-to-use-them-with-arduino.

[2] Kamdar, A. (2021, May 6). *Python: Haar Cascades for object detection*. GeeksforGeeks. Retrieved April 15, 2022, from https://www.geeksforgeeks.org/python-haar-cascades-for-object-detection/

[3] Behera, Girija Shankar. "Face Detection with Haar Cascade." *Medium*, 29 Dec. 2020, towardsdatascience.com/face-detection-with-haar-cascade-727f68dafd08#:~:text=So%20what%20is%20Haar%20Cascade.

[4] French, j. "Teaching Pixycam an Object." *Wiki:v1:teach_pixy_an_object_2 [Documentation]*, 24 Jan. 2018,