P.I.X.E.L.

Precise Image eXtraction and Enhancement Lab

Jack Allardyce, Lindsey Canessa, Josiah Owens, Victoria Ryan, Delaney Smith

2025 Notre Dame Electrical Engineering Senior Design

Final Report

Table of Contents

1 Introduction

Problem Statement

Digital cameras are making a comeback, valued for their image quality and nostalgic charm, but they still face significant limitations that hinder their usability in today's fast-paced, interconnected world. One of the main challenges is the lack of accessibility of the images once they are captured. Unlike modern smartphones, which seamlessly integrate with cloud services for instant photo backup and sharing, digital cameras often require manual intervention to transfer images. It can be a burden to upload all of your photos after an event or trip and add them to a massive shared album, individually inviting each person you have taken a picture of. Additionally, the reliance on proprietary or outdated connectors and cables adds to the inconvenience of downloading and sharing images from a digital camera. Many users lose the original cords needed to connect their cameras to laptops or find that these cords are incompatible with newer devices. This forces users to either buy new accessories or rely on external card readers, which adds cost and complexity. For casual users, this technical barrier can lead to frustration, while for professionals, it can slow down workflows and delay the delivery of images to clients. Similarly, digital cameras often have a very limited memory, so only a fixed number of images can be stored on the device. This limitation means that to take new pictures, old ones must be deleted, or pictures must be downloaded off the device and stored on external memory every time it is used. These difficulties of uploading and saving images create a significant usability gap that limits the potential of digital cameras in a world in which immediate, cloud-based solutions exist.

It is also difficult to set up a self-timer photo with just a digital camera and also know that everyone is in frame, especially given that no one is actually holding the camera. When you set up a camera on a bookshelf or ledge so that everyone can be in the photo, you often have to go through several iterations of starting the timer, getting into position, waiting, and then checking to see if the photo turned out well. The taker of the photo also has to rush to push down the timer button and then get into frame, which can be chaotic, resulting in a lower quality photo. Even if the correct framing is obtained, the images are often still low quality because of too bright or dull lighting.

Solution

To address the limitations of traditional digital cameras, we propose developing a smart, WiFi-enabled digital camera that not only takes high-quality images but also integrates seamlessly with cloud services and provides enhanced usability features. Our solution is designed to eliminate the need for extensive manual intervention in photo transfers, simplify group photography, and solve lighting challenges.

Instant Image Upload and Sharing - This solution employs the ESP32 camera module and WiFi to take images and automatically upload them to a cloud integration platform, such as Google Drive or a custom website. As part of this uploading process, an AI facial recognition algorithm developed using OpenCV will be used to recognize who is in each picture and create individual pages of the website or albums for each person. Then, everyone will not have to sort through all the pictures in the entire collection and can easily find the ones relevant to their use. In the event that the camera is not connected to WiFi, there will also be a microSD card onto which the images can be saved so that the internal ESP32 memory is not instantly filled and the images are not lost forever if a connection is not available. This solution eliminates the need for individually downloading and sharing each image, external cords or accessories, and the limited memories of current digital cameras.

Self-Timer and Framing - The proposed solution for the difficulties of taking a self-timer photo and correctly framing the group in a shot will be addressed with a separate button clicker and laser levels. A small, handheld, wireless device that is separate from the camera will have a button that is connected to the camera ESP32 over bluetooth that someone can click and the photo will be taken. They will be able to be in the camera shot without having to directly press the button on the camera and then run to be in the picture. Additionally, four laser line levels will be used to project a rectangular outline from the camera. This box will represent the frame that is actually being captured by the camera. This way, even though there is not someone holding the camera and looking through the viewfinder, the group can ensure that they are properly positioned in the shot.

Adjustable Flash Brightness - To address the fact that many digital camera images are overexposed due to the flash being too bright or too dark due to the lack of a flash, an adjustable flash system is proposed. A potentiometer will be used to adjust the brightness for a camera flash LED driver. The user can turn the dial on the potentiometer based on if they want the full flash brightness, no flash, or variable levels in between. This solution will allow bright images or objects to not become washed out and dark images and spaces to clearly, effectively be photographed.

Evaluation Against Expectations

The final implementation of the camera system met the initial design expectations in terms of core functionality and usability. The system is capable of capturing images and uploading photos to a web server when WiFi is available. Regardless of whether WiFi is available, the photo will be automatically saved onto an SD card. It is capable of triggering the shutter remotely via a Bluetooth-connected device and then switching back to WiFi mode. In Bluetooth mode, the user has the option to turn on a laser framing feature, successfully implemented to tackle the difficulties of taking a self-timer photo and correctly framing the group in a shot. The real-time image preview and laser-based framing features work as intended, significantly improving the user's ability to compose group photos without needing to be behind the camera. This validated one of the goals of the project: making self-timer photography more intuitive and accurate with less hassle.

The adjustable flash brightness using a potentiometer also proved to be a successful enhancement, allowing the user to take well-lit photos across a range of ambient lighting conditions without over or underexposing the subjects in frame.

The camera captures images with a resolution and clarity that align with what was anticipated given the limitations of the ESP32 (storage-wise) and the camera module selected.

Some areas evolved during the design process. For example, the original plan was to include GPS tagging. However, this was abandoned due to poor performance indoors and added unnecessary hardware. The time-stamp on photos is successfully added within the Python script pulling photos from the web server. Additionally, minor trade-offs were made in terms of size and component layout to accommodate batteries, circuitry and user-buttons.

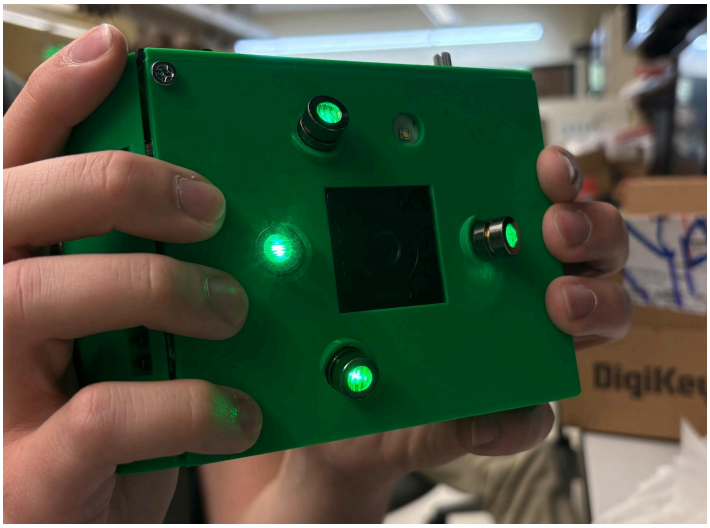


Figure 1. Picture of final product.

Overall, this project is successful in translating the initial concept to the final implementation, demonstrating and validating the proposed technical solutions that could solve the real limitations of traditional digital cameras. That is, the prototype aligns closely with the initial vision.

2 Detailed System Requirements

Satisfied Requirements:

Camera and Imaging Requirements

1. The camera must be able to take a picture when a certain button on the camera is pressed.
2. The camera must be able to display the correct image after capture on the screen.
3. The pictures taken on the camera must be an accurate representation of the field of view and color that the user sees through the viewfinder.
4. The camera must be able to correctly outline the field of view of the camera with lasers for remote image taking.
5. The camera must be able to take a picture with reasonable clarity (5 MP, 2560 x 1920 pixels) while still being within the processing capabilities of an ESP32 [1].

Connectivity and Data Management

6. The camera must be able to connect to WiFi.
7. The camera must be able to connect to Bluetooth when a user tries to search for Bluetooth devices.
8. The camera must be able to send a captured image over WiFi to a cloud-based platform (website or app).
9. The pictures must accurately be received over the WiFi (correct color, field of view, image type).
10. Users must be able to access the website or app.
11. If a website is used, it should be easily accessible. If an app is used, the user should be able to download it to their own personal device, such as an iPhone. Either interface should be user friendly.
12. The website must be able to use an AI facial recognition algorithm to group together images of the same person.
13. The website must display separate albums for the people who appear in multiple photos.
14. The AI facial recognition algorithm must be adequately trained so that it avoids as much bias in identification as possible.
15. The website should also have an option that allows the specified users to view all of the images shared with them, even if they are not in that exact photo or the photo is not of people.

Power and Indicator Requirements

16. A battery should be able to power the microcontroller, lasers, LEDs, and flash. 3.7 volt lithium ion batteries in series should be utilized.
17. The camera battery must be able to be safely recharged.
18. There must be an LED to indicate when the camera battery is low.
19. The camera electronics must not overheat or discharge too much heat that it is uncomfortable to the user.
20. The charging LED should turn off when the device is removed from the charger (the indicator on the battery charger turns off when the battery is removed).
21. The battery life of the camera must be on par with other cameras, lasting at least 3 hours.

Safety and Accessibility

22. The laser outline must be visible to the people in the self-taken image.
23. The lasers must be Class IIIA (between 1 mw and 5 mw) for eye safety [2].
24. Lithium ion batteries should be used for charging and user safety.
25. The buttons on the camera should be labeled so that the user knows which one corresponds to which functionality.

Practicality of Device

26. The buttons must be easily pressed by the user.
27. The camera and cloud client communications must not interfere with other WiFi transmissions.
28. The camera must be able to stand up on its own so that the user can walk away and take the image with the remote control.
29. The batteries can be recharged with a pre-existing charging cord or device.
30. The camera should be comfortable to hold and use. The buttons must be in convenient locations so the user can press them while also looking at the screen or viewfinder.
31. The camera must have an adjustable flash that can be controlled with a potentiometer by the user.
32. The flash must go off right after the user presses the button to take the image.
33. The flash must be able to be turned completely off.
34. The flash brightness must correctly correspond to the dial turn on the potentiometer.
35. The camera must be able to save captured images to an SD card if the user chooses to do so.
36. The camera must be able to take images and save them to an SD card even when it is not connected to WiFi.

37. The user must be able to press buttons to decide to upload an image to the website or to delete it, and the camera should either begin sending the image data or delete the image.
38. The remote control must be able to interface with the camera from up to 25 feet away [7].

Additional Camera Features

39. The camera must be durable enough to withstand small water droplets and be placed in a bag.
40. The camera must be able to correctly outline the field of view of the camera with lasers.

Satisfied Requirements with Edits:

Camera and Imaging Requirements

41. An ~~OLED~~ LCD screen should preview the image before it is taken.
42. The user must be able to ~~press a button~~ use a switch to enable the laser frame.
43. The camera must be able to take a picture when the button on ~~a remote control~~ an app is pressed.
44. The ~~remote control~~ app must be able to send a signal over Bluetooth in real-time when its button is pressed.

Power and Indicator Requirements

45. ~~There must be an LED to indicate when the camera is fully charged~~. There is not an LED to indicate that the camera is fully charged. There is an LED that indicates that the camera has low power, and there are indicators on the actual charger for the batteries that show when the batteries are fully charged. Because of these indicators, an LED on the actual camera showing that it is charged was not necessary (the actual batteries are recharged, not the whole camera itself).

Additional Camera Features

46. ~~There must be an LED that indicates when an image is taken.~~ There is not an LED, but the flash goes off and the screen stops the live view, making it clear that the image has been taken.

Connectivity and Data Management

47. ~~A GPS device must be able to record the location when an image is taken.~~ A GPS device is no longer being used because it would not record the location indoors, which is where the camera will often be used.
48. The ~~location,~~ time and date of each image must be able to be sent over WiFi and received by the client website.
49. The ~~GPS,~~ time and date information must correspond to the correct image.

Practicality of Device

50. ~~The camera should be the same or smaller in size and weight than typical digital cameras (height - 2.4 inches, width - 4.16 inches, depth - 1.6 inches, weight - 2 lbs) [3] [4].~~ The dimensions were originally based on the small cameras found at [3] and [4]. While going through the design process and incorporating all of the desired features, it made more sense to have a bigger frame, which is still on par with some existing digital cameras.

3 Detailed Project Description

3.1 System theory of operation

The PIXEL camera system integrates four distinct and also interdependent subsystems to function as a smart digital camera capable of capturing, storing, displaying, and uploading images. The ESP32-S3 is at the heart of the system and manages communication between the various hardware and software components as well as wireless connectivity.

When the user powers on the system using the main toggle switch, the ESP32 initializes all key peripherals including the camera, LCD display, SD card, flash, and user input buttons. The LCD immediately begins displaying a live preview from the camera, acting as a viewfinder so the user can frame their shot in real time.

The user can choose to take a picture by pressing a physical shutter button or by sending a Bluetooth Low Energy signal using the LightBlue iOS app. If the shutter is triggered, the ESP32 captures an image from the ArduCam Mega 5MP module, activates the flash if enabled at a user-specified intensity, and displays the captured photo on the LCD screen. The user then has 5 seconds as a review period to delete the image using a button. If the delete button is not pressed, the image is saved to a microSD card. In WiFi mode, which the user controls with a switch, the system creates and serves the image to a web server. The server allows images to be accessed by typing in the ESP32's IP address. A python script running on an external laptop continuously polls the web server for new images and uploads the new image to a Google Drive. Then a script uses OpenCV-based facial recognition to sort the images into user-specific albums.

There is a laser outline framing feature controlled separately from the ESP32 toggled via a switch, enabling users to visualize the camera's field of view when in Bluetooth mode, taking a picture remotely.

Together, the system's hardware and software ensure stable power distribution to all peripherals, fast image capture, clear display, and innovative integration with cloud-based storage and sorting tools.

3.2 System block diagram

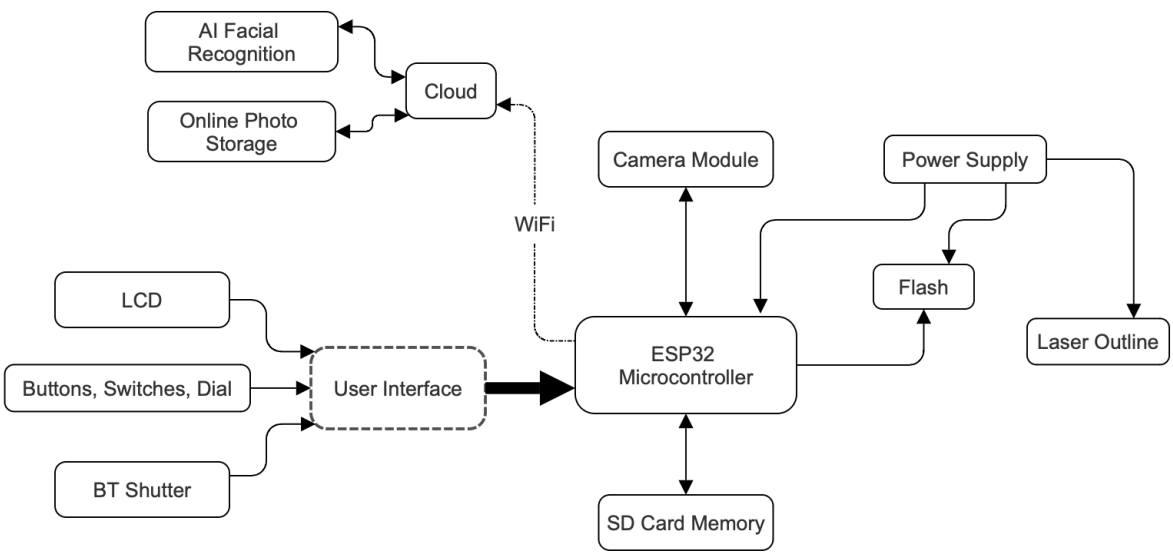


Figure 2. Block diagram of overall system.

3.3 Detailed Design/Operation of Subsystem 1: Camera Control

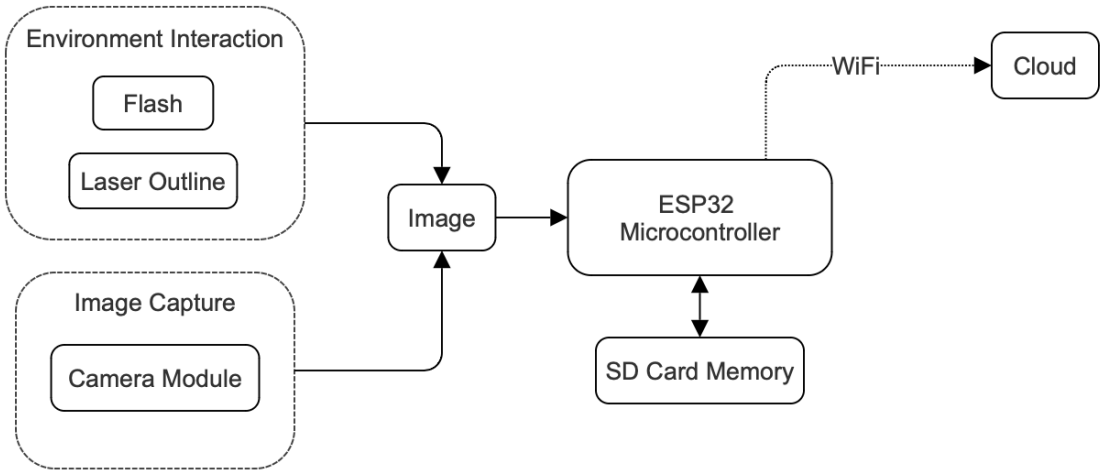


Figure 3. Subsystem 1 block diagram.

Engineering Decisions

The ESP32-S3-WROOM1-N16R8 microcontroller was used as the center of the subsystem design. This particular microcontroller was chosen for its ability to communicate with SPI devices, its PCB antenna that allows for use of either Bluetooth Low Energy (BLE) or WiFi, and the accessible programming options. The ESP32 was able to be programmed in C++, which allowed for flexible programming and access to many free libraries to facilitate the use of each added component. Another reason this microcontroller was chosen was because of the developer boards readily available in the lab for use in prototyping. Other than having a higher PSRAM (8 MB versus 2 MB), the chosen microcontroller is the same as what is in the lab. This allowed for early testing of the chosen peripherals and quick adaptation from the prototype board to the final PCB. Note: not included in the below screenshots of the individual parts of the schematic is a 10uF and a 0.1uF capacitor wired in parallel, acting as the decoupling capacitors for the 3.3V pin on the ESP32; also, there is a boot and an enable button that are inaccessible to the user as they are for use in programming the board.

Reliable image capture was pivotal to the project. This led to the choice of the *Mega 5MP SPI Camera Module with Autofocus Lens* as the camera module. A SPI communication protocol was desired for its speed, and the chosen module provided the best image quality at that price range for a component that was compatible with the ESP32. The camera was connected to the ESP32 as seen in Figure 4. The camera module was programmed using the ArduCam_Mega library.

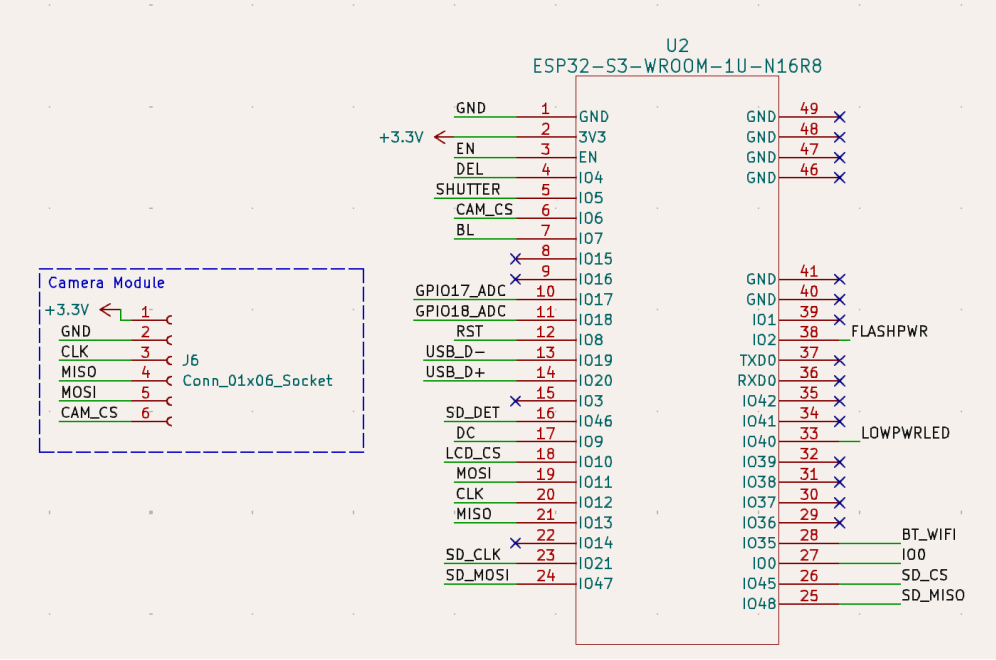


Figure 4. Schematic of 5MP ArduCam camera module SPI connections to the ESP32.

The flash LED used was the JH-3535WW12L48-T8B. It was chosen for its brightness, price, its white color temperature, and its forward voltage range including 3.3V. The 3W LED has a brightness not much different from the average smartphone flash brightness, and it was fairly priced at $10 for 5. This flash LED was to be dimmable, so a 10k ohm potentiometer was wired in series. A n-channel MOSFET was used to control when the current would flow through the LED, with the gate of the transistor connected to a GPIO pin of the ESP32. The schematic is seen in Figure 5.

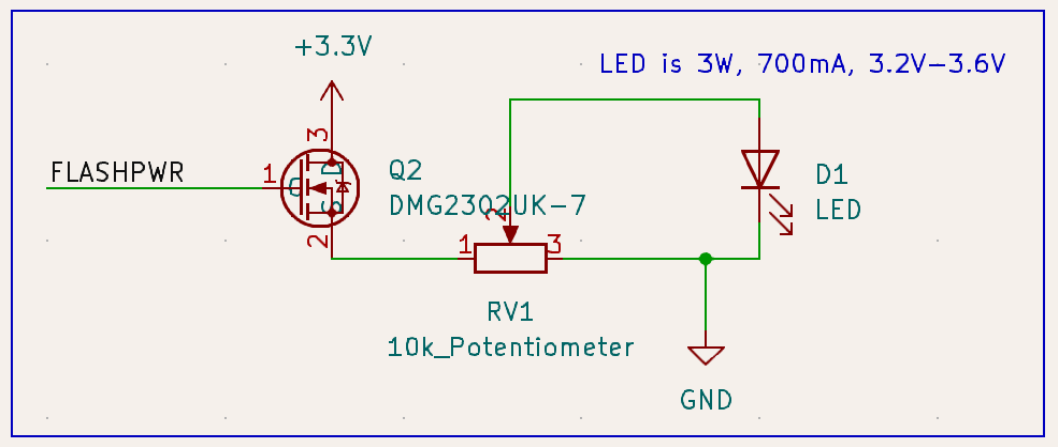


Figure 5. Flash LED schematic.

The FLASHPWR pin was programmed to be 0V by default and is set to 3.3V when turning the flash on. The flash is programmed to be on for 0.2 seconds while the image is being taken. Because the “on” duration is short, the 0.7A current draw does not greatly impact battery life.

There were four lasers used to create the image framing for taking a photo. The laser chosen was the 520-L12 laser line: a 3V-3.7V green 520nm laser that operates at 5mW. This intensity of laser falls within the Class IIIa rating and is safe for general use. In addition, since the laser uses a beam divergence lens to create a line, the light is not focused on a single point and is thus safer than a single point laser. The lasers are switched on and off through a physical switch in order to reduce the current draw from the ESP32. The schematic is seen in Figure 6. The lasers are located external to the board, and since they are wired in parallel, they are represented in the schematic as a single laser diode.

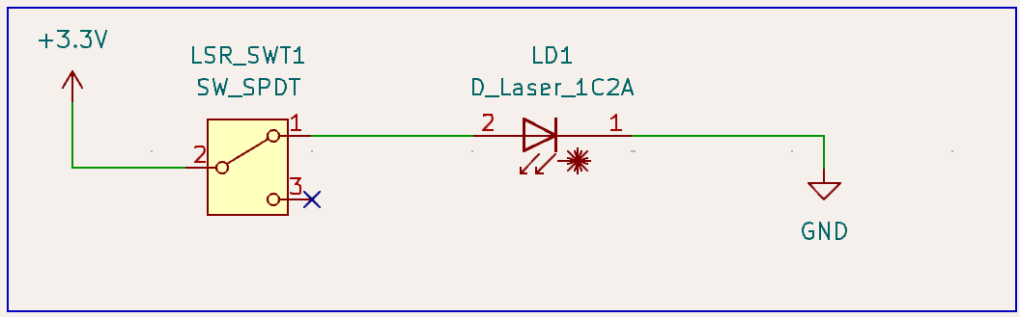


Figure 6. Laser line schematic.

The mounting of the lasers in the camera case was vital to their accuracy as a representative image frame. The case was made of 3D printed ABS-M30 material, and was designed in SolidWorks to position the lasers into a box pattern. Because the ArduCam camera module had a field of view that was 68.75˚ from the center of the camera, the lasers were positioned at the same angle but 3.25cm away from the center. This was so that the laser lines would always match the camera view but still fall just outside of the frame. The camera case design for the lasers is shown in Figure 7.

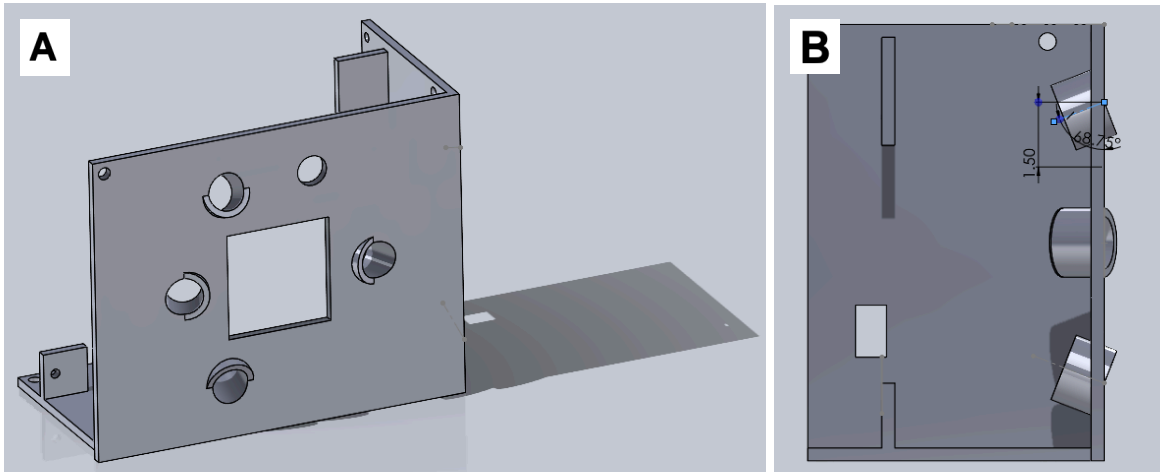


Figure 7. (A) Camera case front with holes for lasers and camera module. (B) Side view of camera case with lasers angled 68.75˚ from camera center.

The original design and set of requirements included a GPS module to be used to send location, time, and date information with each picture. However, the other necessary peripherals (namely the display and camera module) occupied many of the ESP32's pins, leaving few for the GPS. The camera module itself records the time and date of each photo, and location information can be obtained from just WiFi. That being said, it was decided to remove the GPS module from the final design.

A micro SD card was used to store images in case there is no WiFi connection available. The specific holder used was the DM3D-SF Hirose Connector. It was chosen

because it is a push-pull SD card connector instead of a spring-loading option. The spring-loading mechanism could serve as an unnecessary point of failure and so the push-pull option was preferred. The micro SD card is inserted/removed by the user by opening the door on the case. The schematic detailing the connector's wiring to the ESP32 is seen in Figure 8.

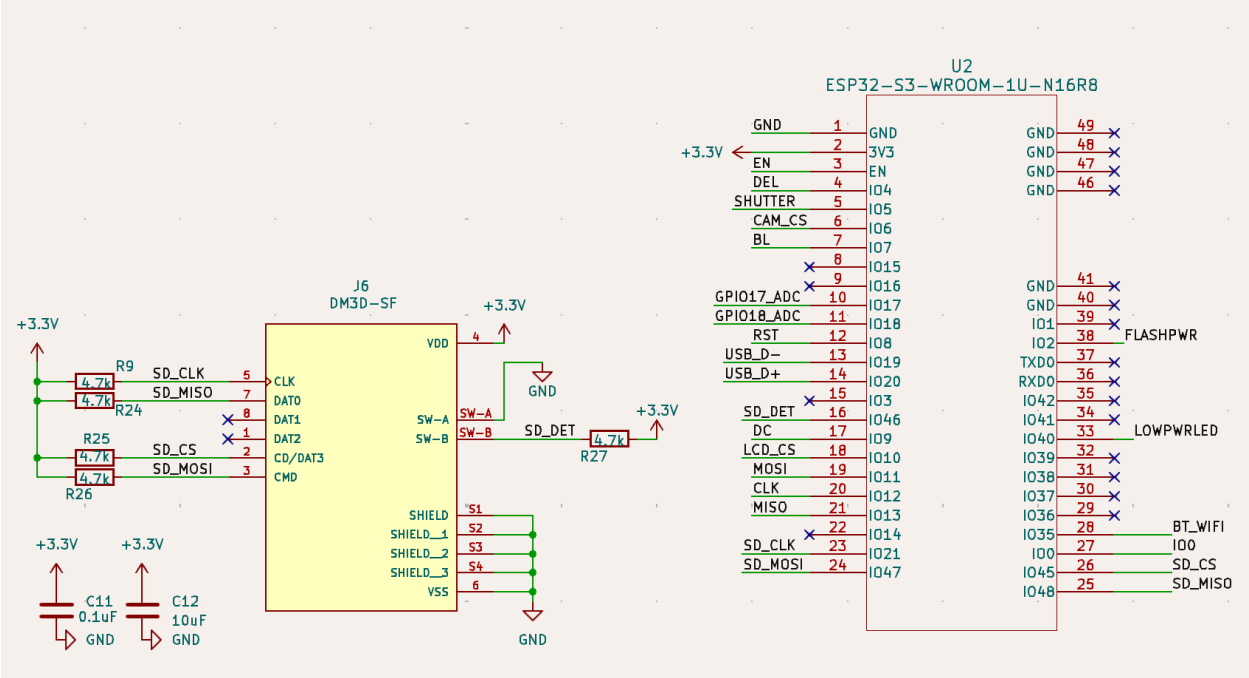


Figure 8. Schematic of the micro SD card connector, detailing its ground connections, pull-up resistors, decoupling capacitors, and its wiring to the ESP32.

The subsystem programming involved initialization of the SD card, ArduCam camera module, the LCD screen, and setting up the button interrupts. The flow chart describing the main logic is seen in Figure 9.

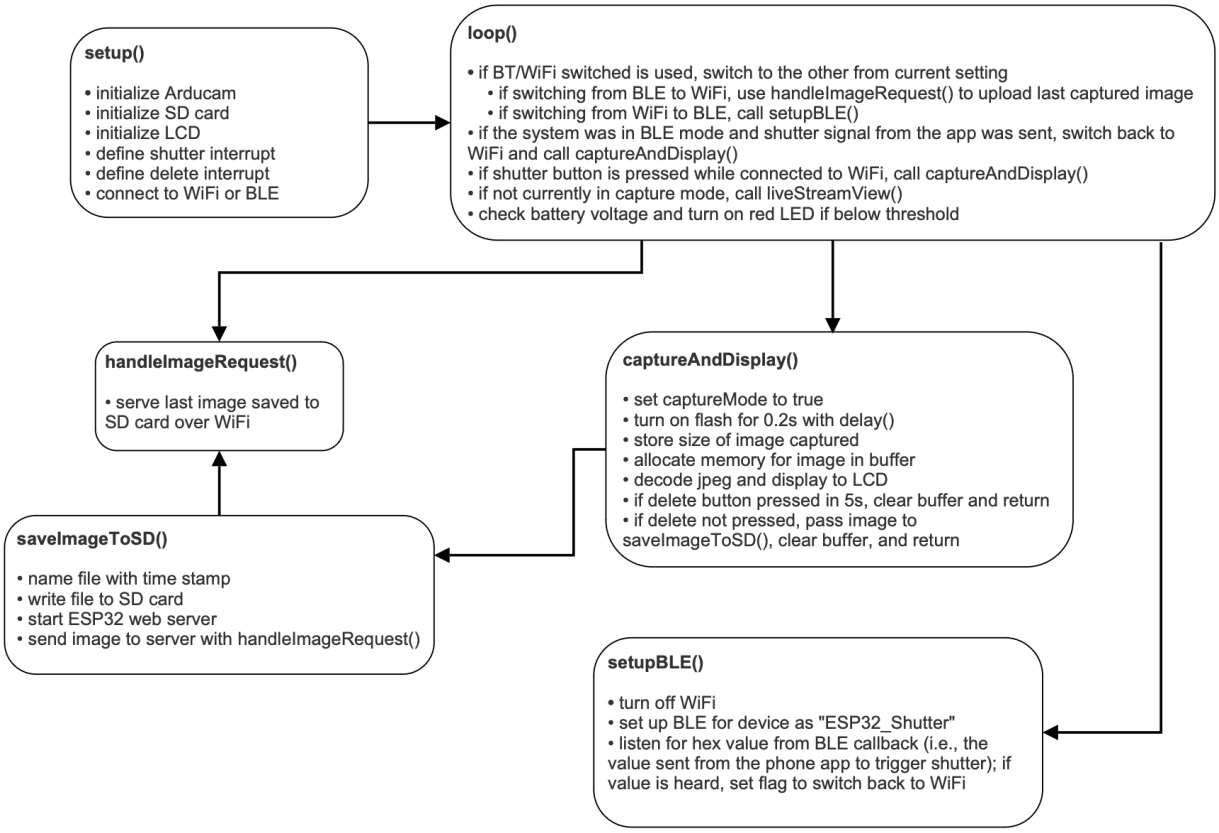


Figure 9. Flow chart of main code logic.

Testing Subsystem 1: Camera Control

Image Capture

We began testing with a commercial ESP32 development board and breadboard to make connections. According to the documentation for the ArduCam Mega 5MP SPI module, communication would be established over SPI, so we wired the camera module to the ESP32 using the SPI interface (MOSI, MISO, SCK, and CS pins), along with power and ground. To verify camera functionality, we used the ArduCam_Mega library and example sketches to capture still images and display debugging information via serial output. However, while we received confirmation through the serial monitor, without an LCD screen displaying the image, we did not yet have a way to view the image and verify its quality.

Image to Cloud Over WiFi

We shifted focus to transmitting that image to an external web server. Since the ESP32-S3 supports HTTP and WiFi natively, we configured the board to connect to a local WiFi network (NDguest) and initiated a POST request containing binary image data to a Flask-based server hosted on a laptop. This web server was set up to receive the image, save it locally, and associate it with metadata such as timestamp.

Image to SD Card

Once an image could be reliably captured and received on the web server, the saving of the photo to external storage was tested. First, a microSD card and SD card module for an ESP32 were purchased. Once these components arrived, they were initially plugged directly into the ESP32 based on the different pins of the SD card module and the SPI pins of the ESP32. However, various initialization errors were detected, so the SD card was configured to the FAT32 format by plugging it into a MacBook laptop and using the Disk Utility feature in the iOS software. Then, the SD card was placed back into the SD card module and wired again, which can be seen in Figure 10, according to a sample project with the specific SD card module being used. At this point, when an image was captured and uploaded, it was also able to be saved to the SD card using the SD library and a Jpeg buffer. The SD card was then removed from the module and placed back into the laptop so that the image files could be inspected. Because the files on the SD card matched the expected size, field of view, name, and coloring as the camera images uploaded to the web server, the SD card was successfully tested for image uploading directly from the camera module.

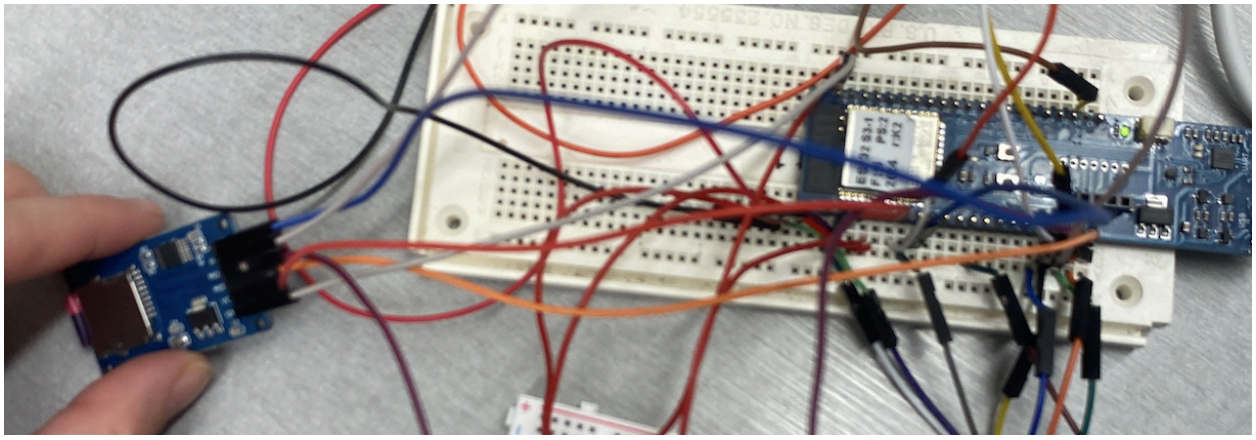


Figure 10. MicroSD Card Module to ESP32 Connections for Testing

Environment Interaction – Flash and Laser Outline

The flash was tested by setting the brightness potentiometer to a fully “ON” position, a middle brightness position, and a fully “OFF” position. Multiple photos taken at each setting had the flash at constant brightness according to the position of the dial.

The lasers were tested by turning them on while the camera view was being displayed live to the LCD screen. During testing, some glow could be seen along the edge of the frame (see Figure 11). The laser angles were altered by using plastic shims inserted into the holes where the lasers sat. The final result of the laser frame resolves this issue and the laser lines successfully sit just outside the image borders. This was confirmed by pointing the camera at different angles and walls and taking various photos.

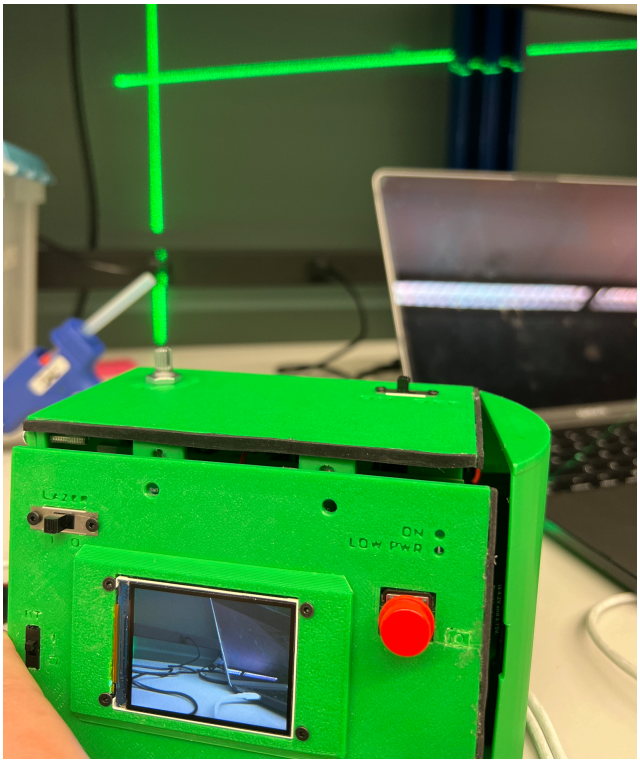


Figure 11. Laser outline during testing and adjustment phase

3.4 Detailed Design/Operation of Subsystem 2: Power

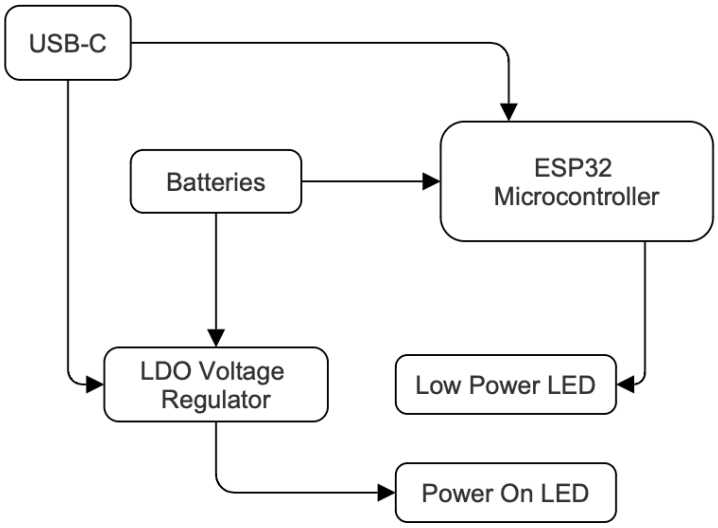


Figure 12. Subsystem diagram for power

Engineering Decisions

The batteries chosen were BENKIA 3.7V lithium-ion rechargeable 18650 batteries, rated at 9900mAh. The voltage and charge capacity were sufficient to run the system for more than a few hours and they had built in overcharge, over-discharge, and short circuit protection. They were purchased along with a compatible 18650 battery charger by Skywolfeye, which also incorporated overcharge, overdischarge, and over-current protection. The subsystem includes wiring two of these batteries in series (which is stepped down to 3.3V using a voltage regulator). Each battery sits in its own holder, attached to the board with wires. The batteries are accessed through the side door of the camera case. The current charge capacity of each battery is checked using voltage dividers and the analog pins of the ESP32. The schematic is found in Figure 13.

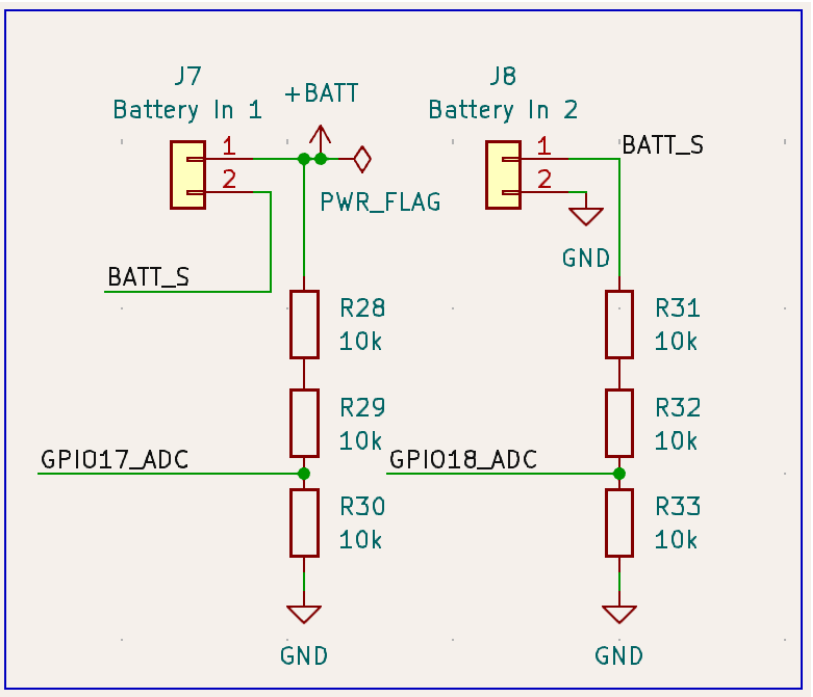


Figure 13. Schematic of battery input and voltage dividers.

To tell if the board was receiving power, a green LED would light up as long as there was power being supplied through either the batteries or the USB-C connection. Then, if the battery voltage fell below a certain threshold, a red LED would turn on to signify this. The schematic for the indicator LEDs is seen in Figure 14.

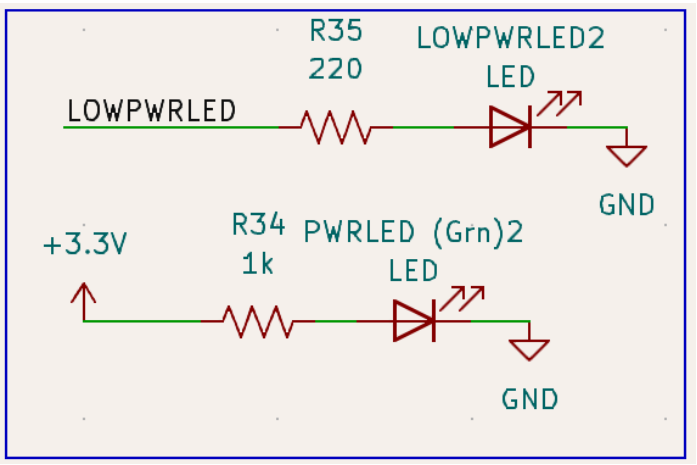


Figure 14. Schematic of the battery power LEDs.

In the above figure, LOWPWRLED is a pin connected to the ESP32 which goes high when the combined battery voltages fall below 4.2V. The code for this is seen in Figure 15. This code runs in the main loop of the ESP32 logic.

```
float voltage1 = analogReadMilliVolts(analogPin1) / 1000.0; // Get voltage directly
float batteryVoltage1 = voltage1 * (1 + (R1 / (R1+R2_3))); // Adjust for voltage divider

float voltage2 = analogReadMilliVolts(analogPin2) / 1000.0; // Get voltage directly
float batteryVoltage2 = voltage2 * (1 + (R1 / (R1+R2_3))); // Adjust for voltage divider

  // note: battery voltage is measured as .12V higher than multimeter value

if (batteryVoltage1 + batteryVoltage2 < threshold) {
    digitalWrite(LED, HIGH);
} else { // blink if low battery
    digitalWrite(LED, LOW);
}

Serial.print("Battery Voltage1: ");
Serial.println(batteryVoltage1);
Serial.print("Battery Voltage2: ");
Serial.println(batteryVoltage2);
```

Figure 15. Screenshot of the code used to check battery voltages.

The system uses the AZ1117I low dropout voltage regulator to step down the voltage level to 3.3V for the board. This regulator was chosen for its current limit of 1.35A and maximum input voltage of 18V. In addition, this component was readily available as part of the stock parts that could be used in the board design. The main power supply of the system was designed to use the batteries detailed above; however, the board is capable of being powered via USB-C connection. The inclusion of the USB-C connector was mainly for ease of programming the system, but it is still accessible to the user if the need arises. The schematic for the power input and the voltage step down is seen in Figure 16.

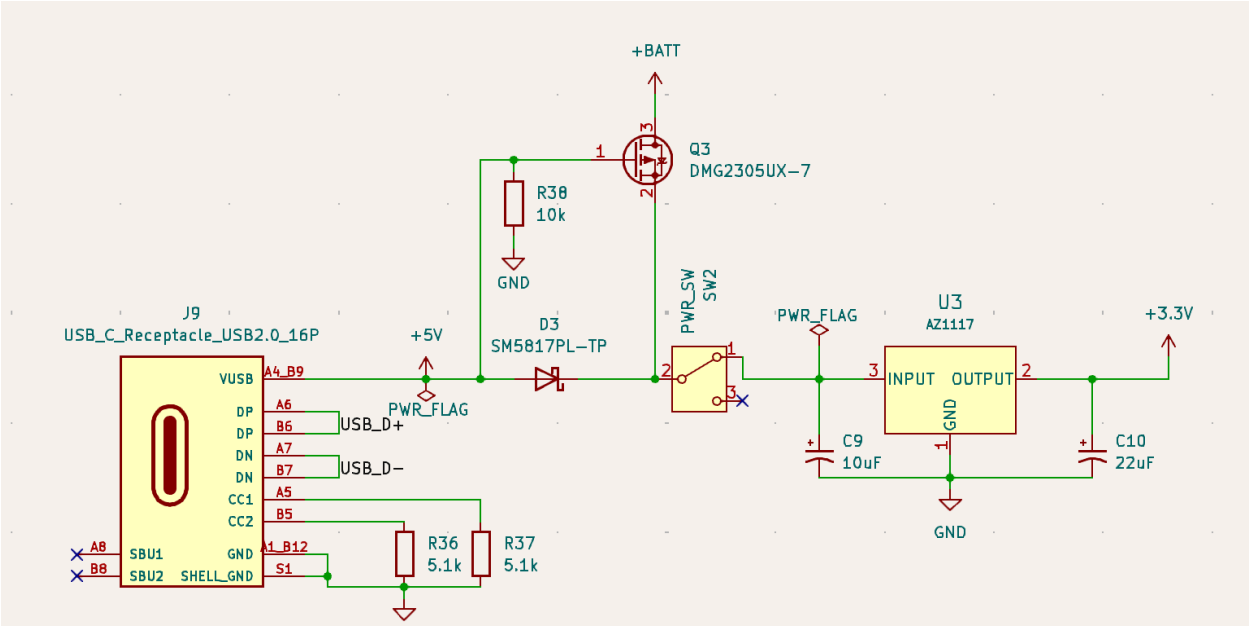


Figure 16. Schematic of power supply and voltage regulator.

The LDO voltage regulator uses decoupling capacitors (with values chosen according to the datasheet recommendations) at the input and output terminals. The USB-C connector was a stock component that was readily available for use for building the board, and the pulldown resistors were chosen according to the recommendation for the part. The power line has a physical switch that creates an open circuit to turn the ESP32 on or off. The battery power input and USB power input are configured to prioritize using USB power over battery power. If both the USB and battery inputs are plugged in, then the Schottky diode (SM5817PL-TP in the above figure) is forward biased, and the gate of the p-channel MOSFET (DMG2305UX-7) sees the 5V from the USB. This causes the MOSFET to not allow current flow from the batteries. If the USB power is disconnected, however, then the body diode of the MOSFET allows current to flow and the drain to source is forward biased. This allows the batteries to feed the input of the voltage regulator.

Testing Subsystem 2: Power

Batteries

Testing was done on the batteries themselves using a multimeter. A fully charged battery measured about 4.2V. Once the batteries were depleted past a certain range, their over-discharge protection would activate and the voltage would read around 2.3V or less.

Voltage Regulator

The output of the voltage regulator was tested using a multimeter. When powering the board with either the USB-C connection or the batteries, the voltage at the output *terminal of the regulator read a constant value of 3.23V.*

Power LEDs

The LED power indicators were also tested. First, the green LED turned on with either USB or battery power (it always turns on when the board has its nominal 3.3V). The red LED was also tested by removing the batteries and running the board with only USB power. Because the analog pins are connected to the battery input and not the USB input, the red LED would power on. To continue this test, the threshold voltage was raised from 4.2V to just over 7V. Even though the battery voltage was sufficient to power the board, the red LED was now powered on. The threshold voltage was reset to 4.2V after this test.

3.5 Detailed Design/Operation of Subsystem 3: User Interface

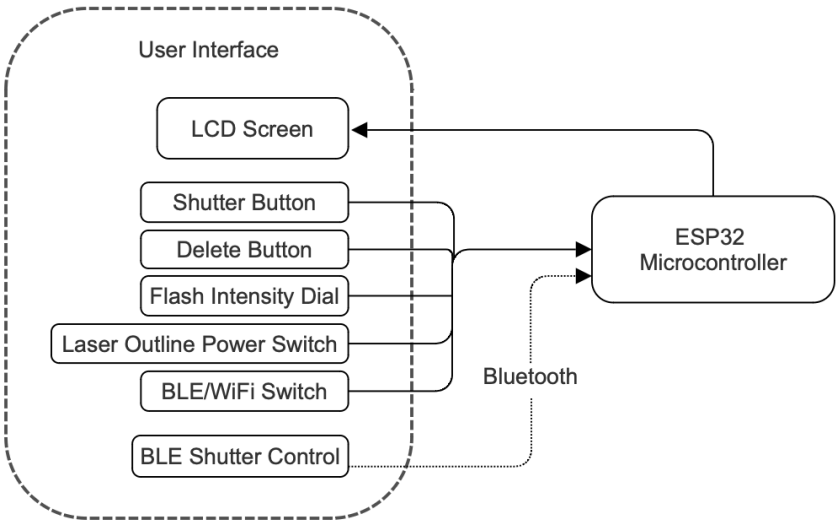


Figure 17. User Interface Subsystem diagram.

Engineering Decisions

A potentiometer was chosen to act as the dimmer for the flash LED (see Figure 5 above). This was a 10K ohm potentiometer chosen for the ease of use by the consumer. Turning the potentiometer counterclockwise increases its resistance; turning the dial changes the amount of voltage that is dropped across the LED. The greater the potentiometer resistance, the dimmer the LED. Once the voltage across the LED drops a little below 3.2 V, the flash no longer powers on, acting as a way for the user to turn off the flash for a photo. The programming logic is explained in the section on subsystem 1.

The viewfinder for this project was the Waveshare 2 inch LCD 240 x 320 resolution Display Module. This display was chosen for its SPI communication protocol, its accessible library, and its ability to display color images. The schematic of its connection to the ESP32 is seen in Figure 18.

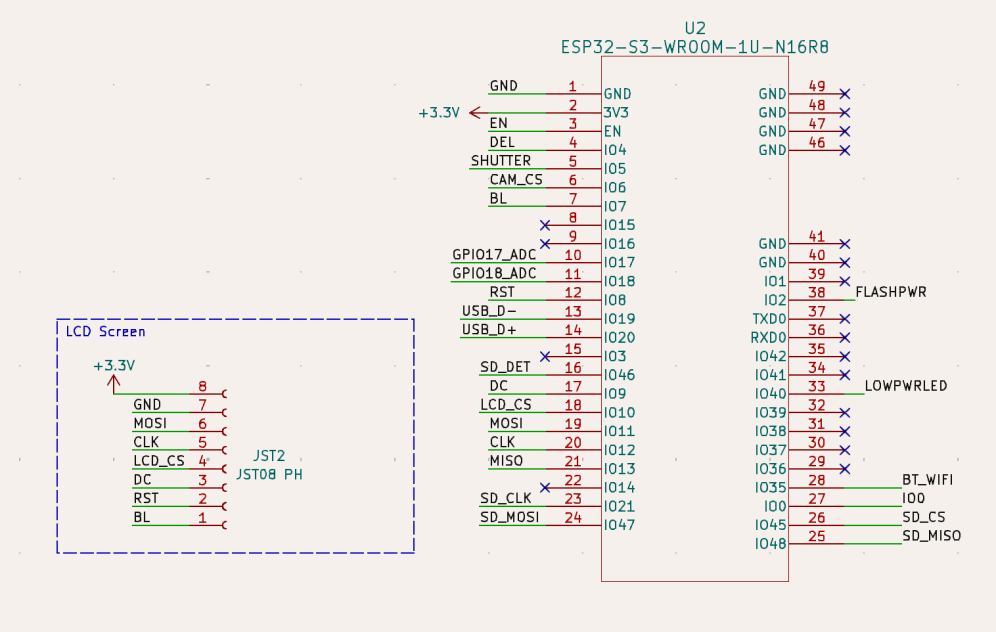


Figure 18. Schematic of LCD screen connections to ESP32.

The LCD screen was programmed to display both a live feedback of what the camera sees and the most recent image captured. See Figure 9 above for the flowchart of the programming logic.

One key feature of our user interface was the ability to remotely trigger the camera using a Bluetooth-enabled shutter button. Initially we tested a commercial Bluetooth shutter remote that advertised compatibility with mobile devices. However, we discovered that this device operated over classical Bluetooth, while our ESP32-S3 microcontroller only supports Bluetooth Low Energy (BLE). As a result, the ESP32 could not detect or pair with the shutter.

We then acquired a second Bluetooth remote that supported BLE. This remote successfully established a connection with the ESP32 and advertised characteristics that we could interact with through the NimBLE and BLEServer libraries. We were successfully able to connect the remote to the ESP32 Server, but found that pressing the button on the BLE remote did not trigger the onWrite() callback. We attempted different UUID configurations and monitored serial output, but the ESP32 never acknowledged an incoming write request from the device.

Instead of relying on BLE shutter hardware, we transitioned to using the LightBlue iPhone app, which provides a manual BLE interface for sending values to specific characteristics. We programmed the ESP32 to advertise a custom UART-style BLE service with a write characteristic. In LightBlue, we were able to manually send the

hexadecimal value 0x01 to this write characteristic, which successfully triggered the onWrite() callback and set a flag in the firmware to act as a shutter press.

WiFi/Bluetooth Switch

Key to Pixel's usability is allowing the system to toggle between WiFi mode (for uploading images to the web server) and Bluetooth mode (for remote shutter triggering via BLE). See Figure 19 for the schematic of the switch. To support this functionality, we initially used a button and interrupt function to carry out the switching. However, for better user experience and less integration issues with the interrupt and debounce, we implemented a physical toggle switch, wired to GPIO pin 35 of the ESP32-S3. We wrote a script for the microcontroller to continuously monitor the state of this pin and transition between modes accordingly.

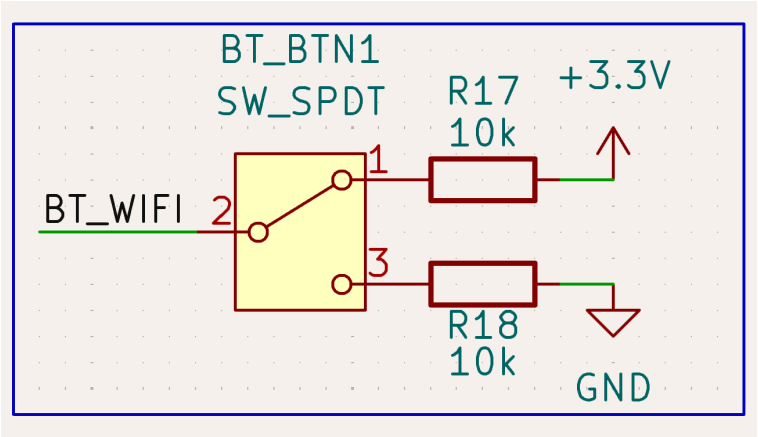


Figure 19. Schematic of BT/WiFi switch.

Two tactile push buttons were used as part of the user interface. The first was the shutter button used to capture an image; the second was the delete button used to tell the system to not save an image that was just captured. The schematic of these buttons is seen in Figure 20.

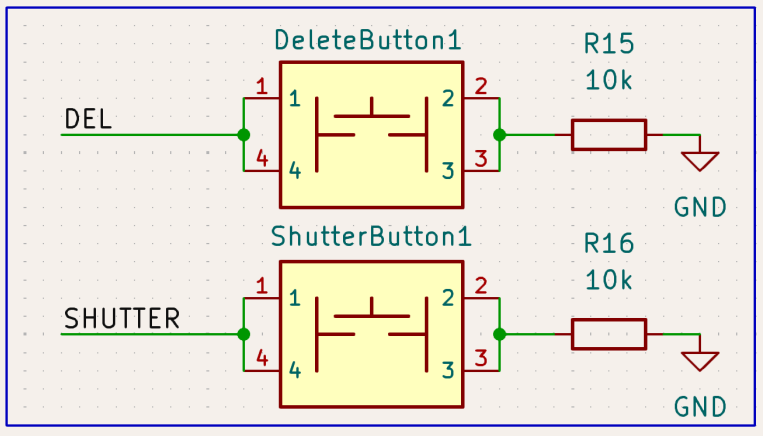


Figure 20. Schematic of UI buttons.

The buttons were programmed using interrupts. See Figure 9 for the flowchart detailing main programming logic.

The buttons, switches, and dial were connected to the board using external wires. This was done so that they could be positioned in an ergonomic manner. The camera case was designed in SolidWorks and 3D printed using ABS-M30 filament. The dimensions of the case are 14.1 cm long, 7.3 cm wide, and 10 cm high. The UI layout of the case is seen in Figure 21. The shutter button was placed on the back of the camera near the screen because placing it on the top was too difficult for the user to press. With the button on the back, it is near the thumb and feels natural to use.

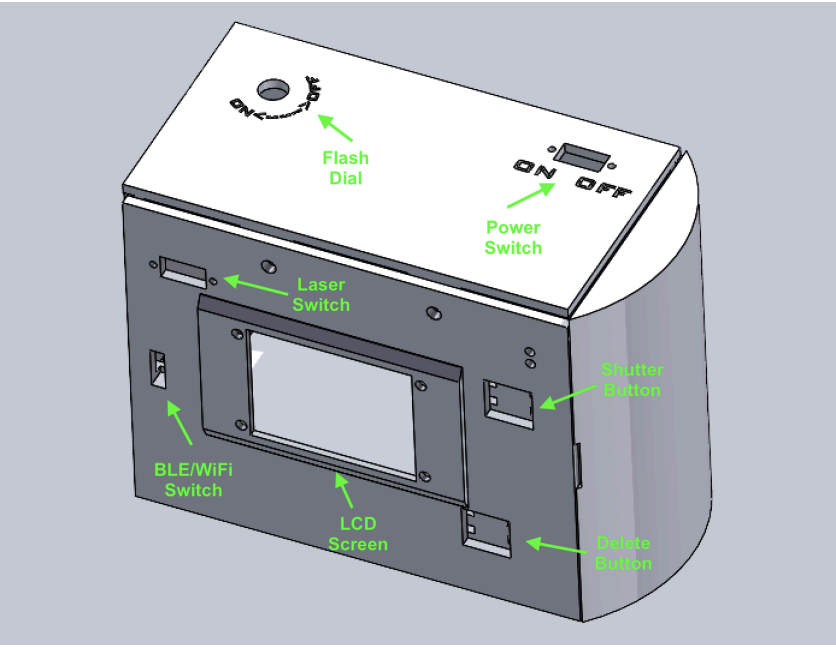


Figure 21. SolidWorks assembly with UI element locations labeled.

Subsystem 3 Testing

Camera to Display

To test the live view and the captured image streams from the camera to a display, a variety of different screen options and libraries were tested first. An image that was previously downloaded to a laptop was uploaded initially as a test image. An OLED, which was initially thought to provide better resolution and speed than an LCD, was seen to be relatively easy to program but difficult to find online in color in a usable size for the application. Then, the LCD screen that was used in the first semester of the Senior Design course was tested by wirin the display to a correct set of SPI pins on the ESP32. This display had more pixels, was able to depict images in color, and was an adequate size for the camera case. Figure 22, below, shows this initial testing of the OLED and LCD screens with a previously downloaded image of the dome at the University of Notre Dame. Additional displays, such as the Adafruit Qualia Display for RBG-666, that had increased resolution and processing speeds through parallel interfacing were also tested. However, these displays were found to require significantly more pins on the ESP32; this increase in pins would have meant that a second board for just the display would have been necessary. Adding a second board would increase the risk for mistakes and interference, so it was decided that these parallel interfacing displays were not the best fit for this project. Because the LCD offered better resolution and more realistic options online than the OLED, the Adafruit ST7789 display was chosen for the final prototype.

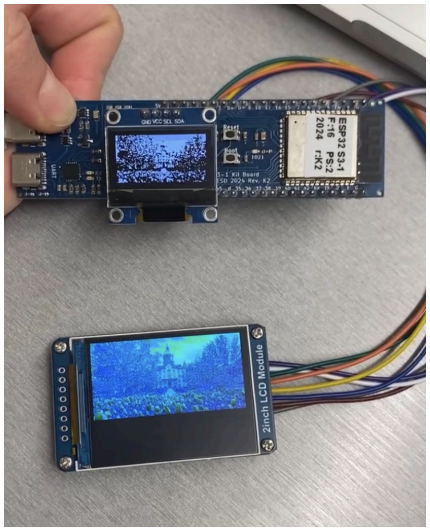


Figure 22. Picture of OLED (top) versus LCD (bottom) image depiction.

Once a display had been chosen based on a static image, the camera could be interfaced with the display. The ArduCam_Mega library was still utilized for image capture, and the Adafruit_ST7789 library was used for the display. Initially, the tft.drawPixel function was used to stream a live camera feed to the display. However, this function only added one pixel at a time, so it was slow to refill the screen each time the camera field of view updated. Also, this function was dependent on the size of the display, which did not match the dimensions of the camera image. Therefore, some of the camera's field of view was being cropped and thin black bars of empty pixels were filling the sides of the screen, which can be observed in Figure 23 below. To address these issues, a different function, the tft.drawRGBBitmap function, was utilized so that multiple pixels could be drawn at one time with one SPI call through the use of an optimized buffer. This function also allowed for the livestreamed and captured images to be drawn in the exact pixel dimensions of the LCD, which was 240x320 pixels, so the entire display was filled. Various other display libraries, such as the LovyanGFX library, were experimented with, but they were found to interfere with the Bluetooth libraries or the other SPI devices.

Figure 23. LCD Screen Image Depiction with the drawPixel Function

Buttons to Camera

While the camera could be told to capture an image by refreshing the local web server page for the microcontroller, this method of image capture was not feasible for the functional prototype in which the user could press buttons to take images. The buttons were initially implemented by connecting them directly to the ESP32 with the breadboard. Then, when these specific pins were set to high, an image would be taken. However, we realized that this system was not robust for the specific application. Because of this, software debugging was added to the code to ensure that it had been at least a defined amount of time since the last button press. Additionally, resistors were

added between the buttons and the microcontroller to ensure that the buttons were not floating or fluctuating when a press was not occurring. While this system was sufficient in minimal testing, falling interrupts based on the button presses were later implemented to ensure that when a button was pressed, the camera instantly captured or deleted an image based on the button. The attachInterrupt function was utilized to set the interrupts to the specific capture and delete button pins, which were configured as INPUT_PULLUP pins. Because these were set as pullup pins, when the button was pressed, the signal went low, and the falling edge triggered an interrupt. Later, during the system integration testing portion, different button flags and their orderings were utilized and tested.

Potentiometer for Flash Brightness

Originally the LED flash was to be powered with a flash driver. The MP3412 flash driver was ordered along with an inductor and input/output capacitors as recommended by the datasheet. The feedback pin of the driver used 0.3 ohms according to the equation given by the datasheet. Changing the duty cycle of a square wave sent from the ESP32 to the input voltage pin of the driver was to change the LED brightness; however, testing found that the dimness hardly changed—in addition to the current through the LED being too low to emit the desired brightness. See Figure 24 for the testing setup.

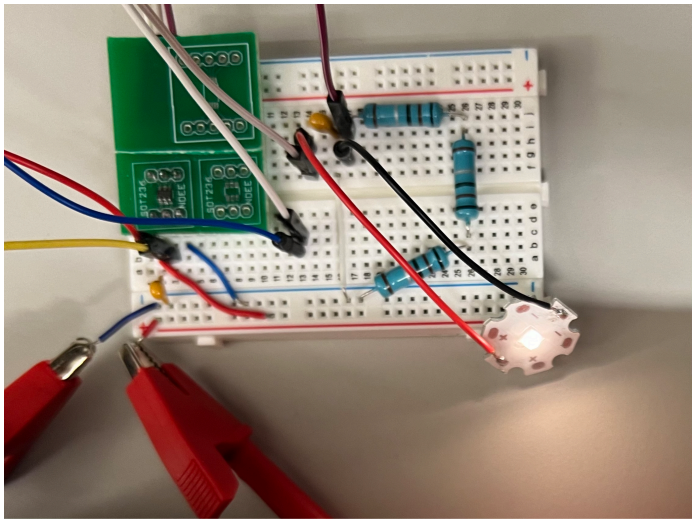


Figure 24. Testing setup for the MP3412 flash driver.

After this, we realized that a simple potentiometer and MOSFET would achieve the desired effect while being simpler to implement. A 10k ohm potentiometer was selected for the flash. The testing for the potentiometer was done by wiring the LED and potentiometer as described under subsystem 1. First, a photo was taken with the potentiometer set to the “ON” position with the flash at full brightness. Another photo was then taken with the dial turned to a point that achieved half brightness. Lastly, the

potentiometer was turned to the “OFF” position and a photo was captured. Each test produced a consistent brightness according to the dial setting, and so the functionality of this feature was able to be verified.

Bluetooth Shutter

To validate the functionality, we powered the device and manually toggled the switch between its two positions. In each mode, we wrote tests for the following modes:

WiFi Mode:

- The ESP32 disconnects from BLE and initializes WiFi connection using the stored SSID and password
- Upon successful connection, the web server initializes and becomes accessible from a browser through the ESP32’s IP address
- The captured images are streamed live and saved directly to the SD card

Bluetooth Mode:

- The ESP32 turns off WiFi and initializes a BLE server advertising a custom UART-style service
- BLE clients can discover and connect to the ESP32
- A 0x01 writes to the designated BLE characteristic and triggers the image capture flag
- Upon receiving the BLE “shutter signal”, the ESP32 automatically disconnects BLE, reconnects to WiFi, and goes on to capture the image, save it to the SD card, and serve it over WiFi

We also implemented a 500ms debounce delay to mitigate false positives from mechanical bounce. We were successfully able to implement the above logic and procedure. The switch reliably triggered mode transitions, and the ESP32 correctly shut down the corresponding radio stack before initializing the other, which we discovered was vital to the reliability of the system. In BLE mode, sending the correct 0x01 signal caused the ESP32 to reconnect to WiFi, take a picture and resume normal server operation without requiring a manual restart or reset. This seamless switch was essential to our self-timer framing and remote shutter workflow.

3.6 Detailed Design/Operation of Subsystem 4: Cloud

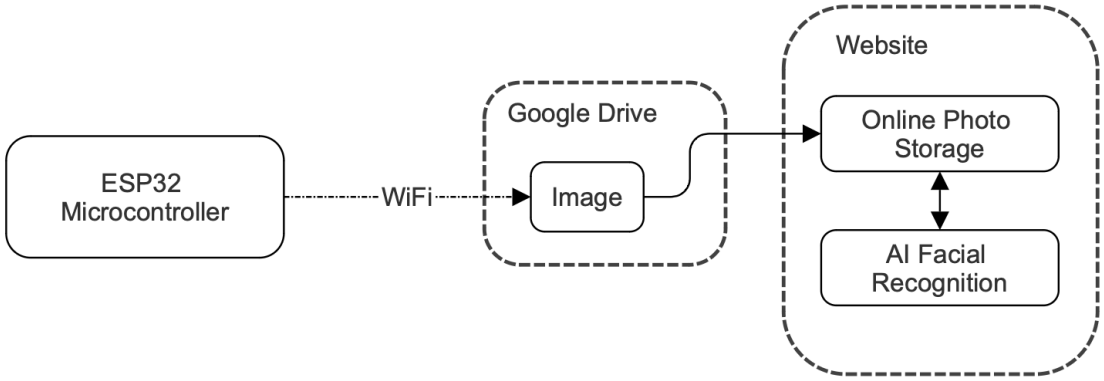


Figure 25. Cloud Subsystem diagram.

Engineering Decisions

Google Drive was chosen to be the central hub for storing and sending images because of the ease of access and how the same folder can be shared with many people. Once taken, the microcontroller sends the images to a web server, a Python script fetches new images on the server and uploads them to the drive folder, and then another Python script downloads the images and displays and sorts them through a Flask web app. The Flask framework was selected for its simplicity and usability. The routing structure made it easy to add new pages and the built-in development server was helpful when testing. Flask apps can be run locally, which removed concerns surrounding hosting and online storage requirements. On the next page, Figure 26 outlines the flow of the web app, with the rectangles representing separate pages and the diamonds representing functions.

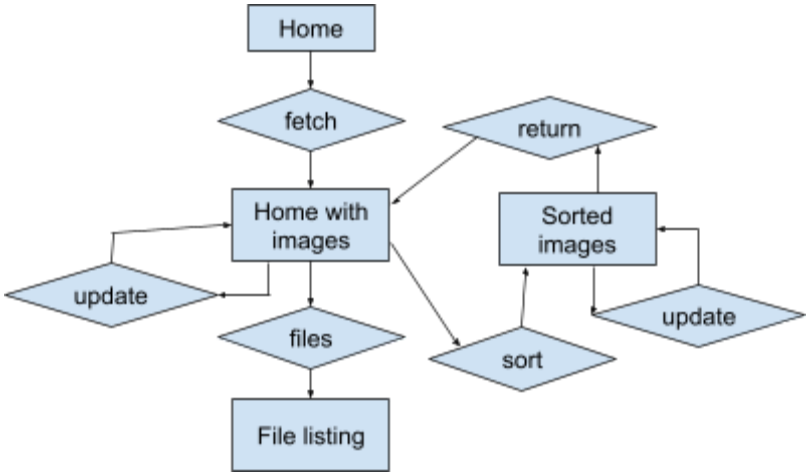


Figure 26. Flask web app workflow

The AI script was inspired by the examples on Python Package Index (here). Python was used because of the existing facial recognition libraries, including DeepFace and OpenCV. The code uses a dictionary to keep track of faces and works by iterating over all images and trying to extract a face. If there are no faces, it saves the image under the no_faces dictionary entry. If there is a face, it compares the face to the faces existing already in the dictionary. If it finds a match, it appends the image to the corresponding person, and if not, it creates a new entry. The two main parameters that needed to be set in the AI algorithm were the detector backend and the confidence level. The detector backend was set to “retinaface” which is said to prioritize accuracy over speed, and the confidence level of face matching was set to be at least 90 percent. After testing, below 90 resulted in incorrect matchings while above 90 seemed to afford no additional accuracy.

Subsystem 4 Testing

The primary testing that took place when creating this subsystem was programmatically finding ways to move images from one place to another in the cloud. For instance, a script was written just to download an image from a Google Drive folder using the folder ID. Another script was written to move an image from a web server to a drive folder. Once these individual pieces were deemed working, they were strung together to form the complete workflow outlined in Figure 25.

The AI sorting code was tested by running it with a set of test images featuring two distinct people and landscapes. The confidence level and detector backend were adjusted until these images were correctly sorted. On the next page, Figure 27 shows the sorted images output by the script. It’s worth noting here that the images used to

test the algorithm were very high quality and clearly featured faces, which was not the case with the photos taken on the group's final digital camera.

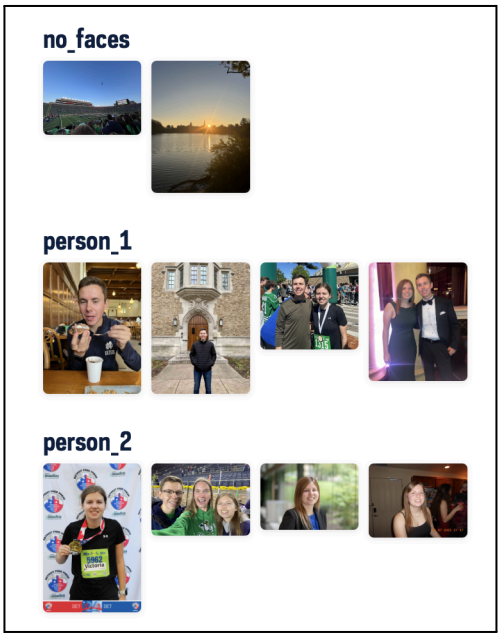


Figure 27. Output of sorting algorithm with test images

4 Interfaces

Most of the details of the interfaces between each subsystem are covered in the sections above. Each subsystem interacts with the ESP32 as the main hub (or perhaps just the source in the case of Subsystem 4) for the image capture and handling.

5 System Integration Testing

5.1 Description of Testing for Integrated Set of Subsystems

To validate the full functionality of the PIXEL camera system, we performed integration testing after all major subsystems had been tested and verified independently (as described in section 5). We first tested on a breadboard with the development board, then transitioned to the printed PCB. The goal of this phase was to ensure interoperability between the camera module, LCD screen, SD card, power system, wireless communications, user interface and cloud platform.

Shared SPI Bus Integration

In our system, there are three SPI devices: camera module, LCD screen, and microSD card. We tested them on the kitboard, attaching each device to SPI-designated pins on the microcontroller. Independently, they all functioned properly, but when they were put on the same SPI bus, we ran into issues having them all function seamlessly. One key focus of the integration was implementing a dual SPI bus architecture. The VSPI was assigned to the ArduCam Mega 5MP camera and the ST7789 LCD display. The HSPI was assigned to the SD card. Since SPI does not support concurrent communication by default, only one device must be active on the VSPI bus at a given time. This was achieved by managing the Chip Select (CS) lines in software. For example, when capturing an image from the camera, the LCD’s CS pin is pulled high. When drawing to the LCD, the camera’s CS is deactivated. This SPI integration testing ensures reliable data transfer. During testing, debug prints in the serial monitor confirmed that each device’s data transmission began only when it was the sole active SPI peripheral on that particular bus. We validated that the image data streamed from the camera to RAM did not interfere with LCD preview functionality. We tested back-to-back operations: capturing an image from the camera, displaying it on the LCD, and saving it to the SD card. We visually confirmed the live view feed on the LCD continued to operate smoothly even after repeated photo captures and SD card writes. This integration is shown below in Figure 28.

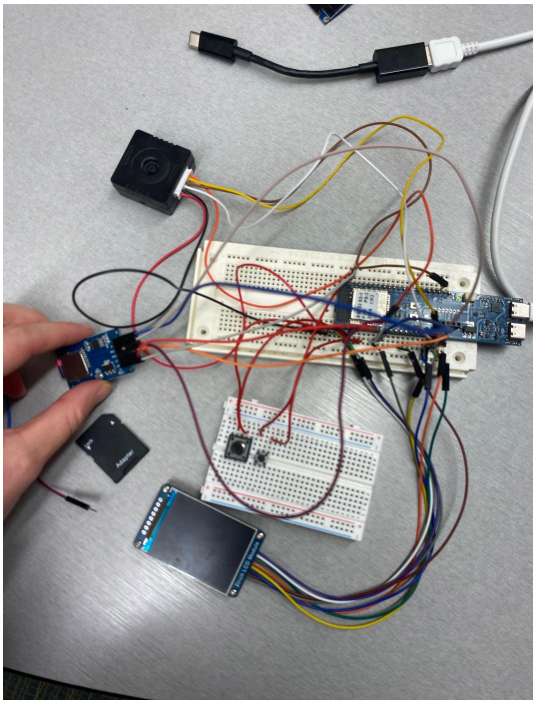


Figure 28. Camera Module, SD card, and LCD integration

Image to SD card, live LCD screen, and Web Server with Cloud Integration

A critical part of integration testing involved the end-to-end image pipeline: from photo capture to local storage, on-device display, and wireless transmission. We tested the camera module on the kitboard to capture an image using SPI and store the JPEG data in RAM, then write the image to the SD card via another SPI bus, ensuring proper CS line control. Tests confirmed that images are saved with consistent file sizes and no corruption, even with repeated use. We printed file sizes directly to the serial monitor to confirm. The LCD screen also was tested by observing the screen to make sure it provides both a real-time preview and post-capture feedback at the same time. Using the same SPI bus, the LCD was able to render frames from the camera during preview mode. Integration testing resulted in smooth transitions between preview and capture modes. Upon image capture, we tested that the system was able to initialize an HTTP web server accessible over WiFi. Captured images can be accessed from a client browser though a REST endpoint (/image). So to test this functionality, we could simply type in the ESP32's IP address into a browser and see if we see a captured image. The external python script and cloud management were also integrated, running them on a laptop to ensure that after a successful image capture, a script can upload the picture from the web server to the Google Drive and then perform facial recognition tagging and album sorting. This ensures that the full data pipeline–from hardware to cloud–functions

seamlessly. During testing, image access and uploads succeeded consistently, even across network restarts.

Integrating Camera Flash and User Buttons into Data Pipeline

The flash LED is controlled by a GPIO pin through an N-channel MOSFET and powered through the main battery line. During image capture, the flash is enabled for a brief, predefined duration (200 ms) to coincide precisely with the camera shutter. Different duration times were tested to try to get to an optimal time frame for the flash to be off. Integration tests verified that the flash fires reliably only during capture. We integrated a potentiometer, wiring it directly to the appropriate pin, so that the brightness is adjustable, which was also tested to confirm that flash intensity responds correctly to user input.

Two physical buttons were integrated into the system and tested with the data pipeline on the kitboard. The shutter button on GPIO5 triggers photo capture which was successfully tested to initiate photo-to-cloud path. The delete button on GPIO4 was confirmed to cancel a captured image during the review period before saving. Both buttons were attached via interrupts to ensure low-latency response. Testing confirmed that the shutter button correctly initiates full capture-display-save sequence previously tested, the delete button interrupts the post-capture wait period and returns to live view without writing to the SD card, and the button debounce and race condition handling were robust under repeated presses. Printing to the serial monitor allowed us to confirm whether an image was deleted or saved. System-level testing confirmed the integrity of this pipeline whenever the shutter button is pressed:

1. The camera module captures the image over SPI
2. The flash is turned on and off in sync with the shutter
3. The image is decoded and displayed on the LCD screen
4. If not deleted, the image is written to SD card and sent to the web server
5. Python script detects new image and moves it to Google Drive
6. Image is sorted from Google Drive

Integration of BLE WiFi Switch into Data Pipeline

In Bluetooth mode, a BLE GATT server advertises a custom UART-style service. When the user sends a 0x01 signal via the LightBlue app, the ESP32:

1. Triggered the onWrite() callback
2. Set a flag to exit BLE
3. Switched to WiFi mode
4. Captured and saved an image

5. Sent image to Web Server

During integration testing, we had to make sure that the active wireless stack is cleanly shut down, the new mode is initialized with proper event handling, and the server or BLE advertising resumes automatically, with full functionality. A physical toggle switch on GPIO35 allowed manual switching between WiFi and BLE modes, and the system monitors this pin state continuously. Printing to the serial monitor allowed us to confirm which protocol was active and verify when WiFi had been safely connected and disconnected. Whether using physical buttons or BLE commands, the camera should behave identically: flash goes off, image is displayed, saved, and served. Switching modes should not interrupt or corrupt ongoing tasks. We tested that power consumption and memory usage remained within the ESP32-S3 operational limits. Even though we got it working with the development board, when switching to the printed PCB, we ran into issues having the switch on GPIO35. Analyzing the ESP32-S3 datasheet, we hypothesized that GPIO35 could not properly function as an input pin, as there was some SPI protocol interference. Therefore, we tried moving the switch to GPIO38 by first setting that pin HIGH with an external power supply and then pulling the pin LOW to ground and seeing if the ESP32 could detect and print the state of the pin to the serial monitor. When this was confirmed, we cut the line on GPIO35 and re-soldered the switch to GPIO38. The system then functioned properly and we were able to switch back and forth between bluetooth and WiFi modes.

Laser Outline Integration Test

When using the breadboard, we were able to test the laser outline feature by using an external power supply. However, when switching to the printed PCB, the lasers, which were on the power supply line, when turned on, caused a system reset. We realized that they were drawing too much power on the board, so to mitigate this problem, we cut the laser line and added a third battery to our camera that was just used to power on the lasers. This made their integration to the rest of the subsystems seamless, because they were a part of a separate circuit.

To ensure the lasers accurately outlined the camera’s field of view, we physically aligned them with the ArduCam module’s capture angle. We estimated their positioning in the 3D-printed camera housing and iteratively adjusted their mounts through hand-filing until the laser box matched the camera’s frame without being able to see the outline in the captured picture. We observed the field of view of the camera through the web server.

PCB Total System Integration Testing

After the PCB had been printed and assembled, we began full system testing by verifying that all components powered up correctly. In the first iteration of our PCB, the green indicator LED illuminated when connected to USBC, but we were not able to communicate with the microcontroller. Using a multimeter, we measured the output of the voltage regulator and found that it had not successfully stepped down the 5V USB-C power to the 3.3V input that the microcontroller (and other components on our board) required. This indicated that something was wrong with our voltage regulator. Upon further inspection, we realized we had ordered a step-up converter instead of a step-down converter. Recognizing the importance of stable power delivery across all components, we redesigned and ordered a second version of the PCB with a corrected power section using the appropriate regulator. We also flipped the footprint of the microSD card slot to make it easier for users to insert and remove the card from the edge of the camera casing.

The second iteration of the board was delivered and assembled. We again began full system testing by verifying that all components powered up correctly and saw the green indicator LED illuminated when connected to USBC. This time, we were able to successfully communicate with the microcontroller, starting our testing with a simple “Hello World” program. To test the buttons and switches, we externally mounted the physical components on the printed 3D case and then soldered wires to the appropriate header pins on the PCB to be able to test their functionality. Then we methodically carried out the subsystem integration testing procedure as outlined above.

5.2 How Testing Shows Design Requirements are Met

System integration testing successfully demonstrated that the PIXEL camera system meets the functional, performance, and usability requirements established in the design specification:

Camera and Imaging Requirements

During integration testing, photos were successfully captured after shutter press using the ArduCam Mega module, stored on the SD card, and displayed on the LCD screen. The LCD shows continuous live feedback until an image is captured, satisfying the core purpose of a live preview system for framing. Image clarity and resolution were verified by comparing the file size and quality to datasheet expectations.

A physical switch enables the user to turn on the laser feature. Laser framing was tested in both dark and well-lit environments. Physical placement and angular alignment

of lasers were confirmed to match the camera field of view using calibration images and web streaming.

Connectivity and Data Management

BLE and WiFi were successfully initialized using a toggle switch. Captured photos were served through an onboard web server and uploaded to Google Drive using a Python script. The script carried out facial recognition and robustly sorted images into user folders easily accessible through the website.

Captured images were saved to the SD card with or without WiFi connection.

Power and Indicator Requirements

Power testing with a multimeter showed a stable 3.3V at the output of the voltage regulator. When the battery voltage dropped below our indicated threshold voltage of 4.8V, the onboard red LED lit up, confirming correct threshold detection. The system continued operating until the cutoff was met. All components received the correct power requirements. The camera battery was able to be charged via an external circuit.

MOSFETs and regulators were tested over long run sessions without overheating or discharging dangerously, meeting our power system requirements.

Safety and Accessibility

Our chosen lasers were rated below 5 mW (Class IIIA). Laser brightness was sufficient to see the framing without bringing along safety risks.

Each button was tested. Interrupts triggered correctly on press, with debounce timing validated through repeated presses. The remote button in BLE mode functionality was also validated through testing.

Practicality of Device

The original requirement was for compact dimensions of camera. However, to accommodate the ESP32-S3, battery pack, laser mounts, LCD, and other circuitry, the case was scaled up. However even with the size increase, the camera remains easy to hold, stands stably, fits into the standard bag, and maintains usability as a compact camera with easy-to-press buttons. The batteries are able to be easily charged. The camera’s adjustable flash was tested and demonstrated to go off at the appropriate time—when a picture is taken. The camera also was able to successfully save images to an SD card and upload over WiFi when available. BLE remote functionality was tested up to 25 feet away from the physical camera.

6 User Manual / Installation Manual

6.1 How to Install the Product

Because our camera can function as its own device, there is not much installation necessary for the user. If the user were to be handed the camera or take it out of a box, all they would need to do is turn the on/off switch to the one mode and begin taking pictures. These images would automatically be saved to the SD card, which could then later be seen by viewing the SD card from a laptop. Also, the switch for the lasers can instantly be used when the device is powered on. However, to use the Bluetooth button mode, the user must install the LightBlue app so that they can send the correct BLE signal to the camera. This app can easily be installed for free from the iOS or Android application stores. Additionally, to use the automatic uploading and AI facial recognition features, the user must be able to run Python scripts. If the user has a Google Drive account, then they can use the Google Colaboratory service to run these scripts and view the images through Google Drive. However, if the user does not have a Google Drive account, they must make one, which is free to create and utilize for purposes such as Google Colaboratory.

6.2 How to Set Up the Product

Once the user has the camera, they must first enter their specific Wi-Fi credentials. This task can be done by editing the ESP32 code so that the ssid, or first term in the WiFi.begin function call, is the network name being utilized and the password, or second term in the WiFi.begin function call, is the password for that network. Then, the code can be uploaded to the camera, and the IP address of the microcontroller will be printed to the serial monitor. Next, the user must verify that the printed IP address matches the IP address in the Python script for connecting the server to the Google Drive folder. If the IP address does not match, the user should edit the IP address field in that Python script to match the newly printed address. Now that the code has been updated, the user can turn on the camera and upload this modified code. All of the functions local to the camera itself, such as the lasers, flash, SD card, buttons, and Bluetooth switch, will work automatically once the camera is turned on.

6.3 How to Tell if the Product is Working

When the user turns the power switch to “ON”, they should see the display backlight turn on. Likewise, they should see the green power LED shine through its corresponding hole in the case. Within seconds of the power being turned on, the user should see the display begin to depict the live view from the camera. If the user flips the laser switch to on, the laser frame should light up with four green lasers creating a rectangular frame. If the shutter button is pressed, the user should see the screen

freeze on the image that was just captured. If the delete button is pressed within five seconds of the image being captured, then the screen should return to the live view mode. However, if the delete button is not pressed within five seconds of the image capture, then the user should see the display return to the live view mode for the camera, and the images should be uploaded to the SD card and the server. If the Bluetooth/WiFi switch is in Bluetooth mode, then the user should be able to send the hexadecimal value of 01 to the camera, which will trigger the capture of an image. If the potentiometer is turned fully on, whenever an image is captured, whether through the physical button or the bluetooth shutter, the flash should go off. The user should be able to remove the SD card and verify the captured images, as well as they should be able to remove the batteries to recharge them whenever the battery monitoring LED lights up red. If the user turns the power switch to “OFF”, the screen should turn off and the buttons and switches should not trigger any sort of response.

6.4 How to Troubleshoot the Product

If the power switch is turned to “ON” and the power LED does not light up, the user should try different batteries or verify that the batteries are properly connected. If the power switch is turned on, but the LCD screen does not light up or display a live camera view, the user should wait a few seconds to give the full setup time to run. If the screen is still not turned on within a few minutes, the power switch should be turned off and the camera should be left alone for a few minutes. Then, the power switch can be turned back on, and if the display still does not respond, new batteries should be tried in the device. If the WiFi connection cannot be established or the images are not uploading to the server, the user should try a different network, take the device outside, or try using the camera in a less crowded room or building. If images are not able to be accessed on the SD card, the user should try to format their SD card to be the FAT32 type by directly connecting it to their laptop and editing the SD card’s settings. If the images are not able to be viewed on the server and pulled to the Google Drive folder, the user should try re-running the Python script and verify that their laptop’s WiFi connection matches the WiFi network to which the ESP32 is connected.

7 To-Market Design Changes

While this device works as a functional prototype for addressing the image access and remote photo capture limitations of handheld digital cameras, it is not yet a fully usable commercial product. The first design change to bring this product to a market would be to implement the WiFi provisioning feature. At the moment, the user must enter WiFi credentials into a C++ coding file each time that they change WiFi networks. However, connecting the camera to a laptop and rerunning code is not practical for a commercial product that is meant to be portable and easy to use. A provisioning app, such as the ESP BLE Prov App, allows the user to connect to the camera from their phone and enter the WiFi credentials of their specific location over Bluetooth. However, the use of a provisioning app was interfering with the Bluetooth/WiFi switch on the camera. More robust code that efficiently turns Bluetooth off as soon as it is not being used so that the WiFi can connect must be developed to bring this camera to a market.

Additionally, the camera is limited as a commercial device because it is larger and less convenient than many digital cameras on the market today. Many people already carry their phones around with them wherever they go for general communication purposes, so the fact that these phones already have a camera makes customers less likely to buy and bring along a second camera device. We left extra space on our PCB so that the USB and SD card could both align with the edges of the camera case. However, we ended up adding extra room on the side of the case to improve battery and SD card access, so our overall device ended up larger than most handheld digital camera devices. To make this product more ideal for a market, we could shrink the case size so that it aligns with the edges of the PCB and the batteries, USB, and SD card are all easily accessible to the user without requiring extra space. We could also reduce the vertical height of the PCB by rearranging components, allowing for more space for internal wiring.

Finally, to fully utilize our camera for automatic image upload and facial recognition, Python scripts must be run while the camera is being used. These scripts have only been tested from laptops, but it is not practical for a user to carry around a laptop just so that they can use a camera. However, Python scripts can be run in Google Colaboratory, which can be run from a phone. Further testing of the server interactions and facial recognition software can be performed on a phone with the current Python scripts so that a user does not need a laptop to use all of the camera features. Then, these scripts can be incorporated into an app using a development software, such as Xcode, so that the user only needs the camera and an app on their phone. While a second device is still necessary for the full uploading potential of this camera, the majority of users have their phone with them at all times anyway or are not bothered by waiting to upload the images from the SD card to a Google Drive folder for facial

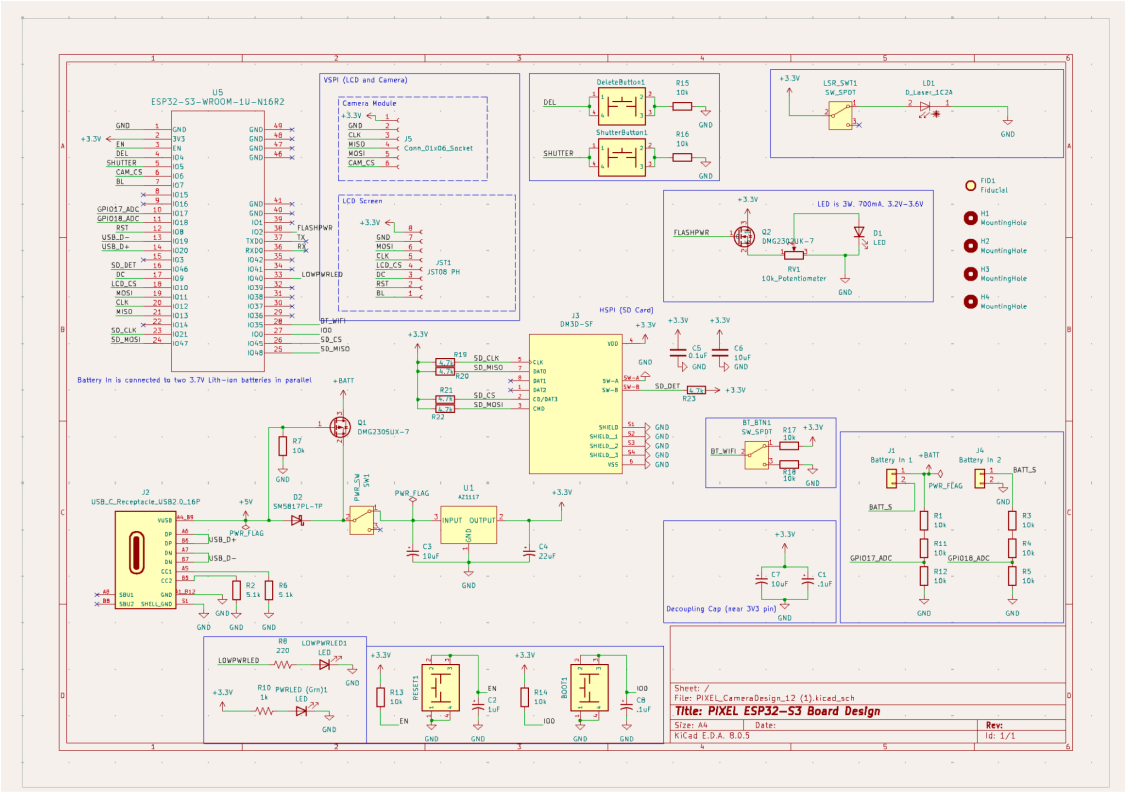
recognition processing. Clearly, the camera is not in its most ideal form. However, each of its limitations—which are due to time limitations for further programming development and case redesign—have a solution that can very reasonably be carried out.

8 Conclusions

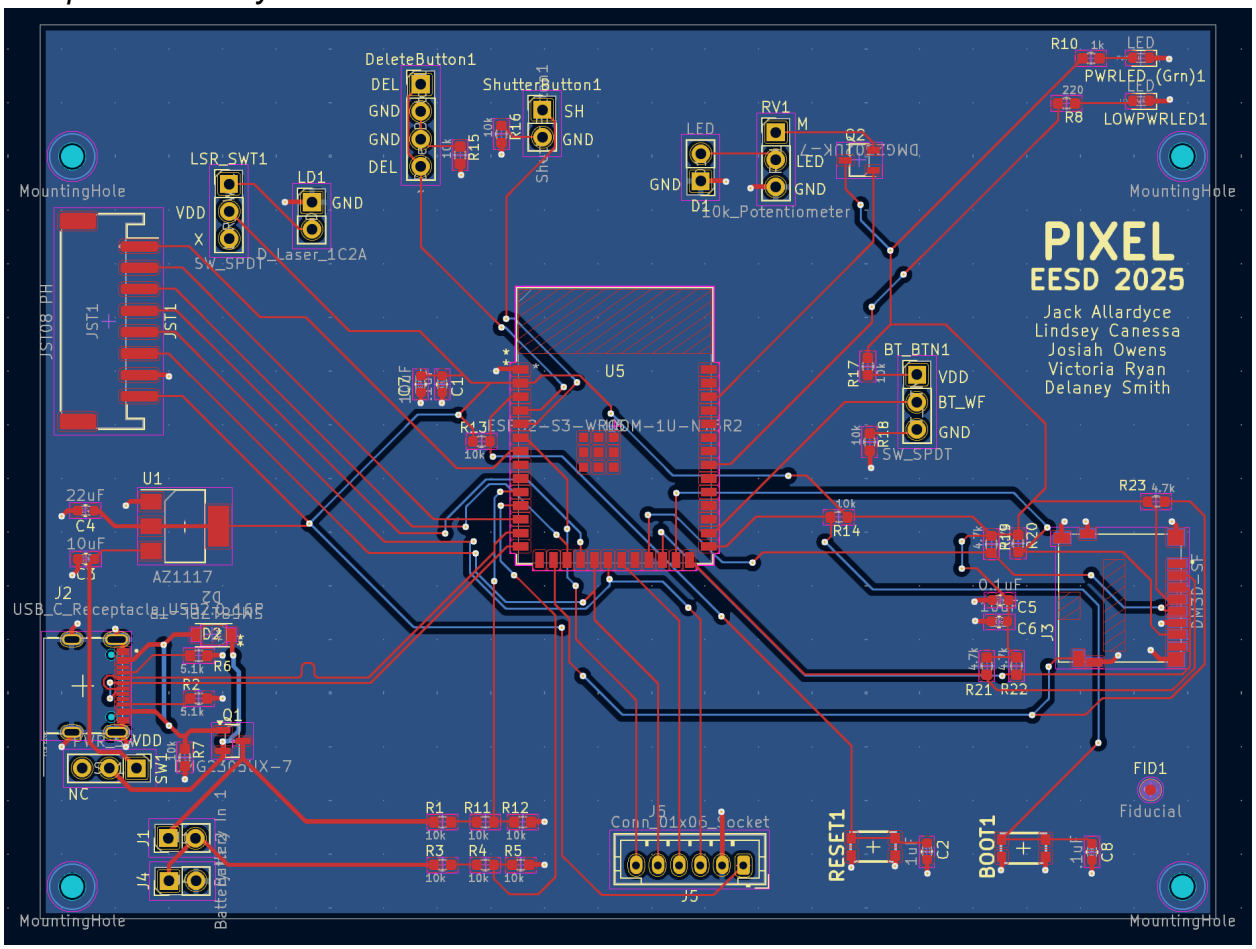
P.I.X.E.L. successfully delivers an innovative digital camera experience by addressing limitations and inadequacies of traditional cameras, including inconvenient image access, poor remote capture functionality, and lack of framing. Through integration of WiFi-enabled image uploading, AI-based facial recognition, laser-assisted framing, adjustable flash, and remote shutter functionality, this prototype not only fulfills its initial requirements but also shows potential for both practical and commercial use. Simply put: PIXEL is fun. This camera brings people together and inspires a new love of capturing the moment. Through the development process, careful engineering decisions were made for hardware selection, subsystem design, and user experience. Moving forward, we could make improvements on the size of the camera, app-based WiFi provisioning, and tighter integration with mobile platforms to carry out the cloud server portion of the product. Overall, this project demonstrates the successful application of an embedded system to create a user-friendly camera that combines the nostalgia of vintage digital photography with modern smart features. With PIXEL, every shot is in frame. Just point, shoot, and smile.

9 Appendices

Complete Schematic



Complete PCB Layout



Relevant Hardware Datasheets/Documentation:

- ArduCam Mega 5MP Camera Module with Autofocus Lens: Waveshare 2-inch LCD Module
- LED Flash:

JH-3535WW12L48-T8B Important parameter

Continuous Forward Current	700mA
Peak Forward Current	1000mA
Reverse Voltage	5V
Power Dissipation	3W
Electrostatic Discharge	1000V
Operating Temperature Range	-25°C to +60°C
Storage Temperature Range	-35°C to +80°C
Lead Soldering Temperature	260°C
Forward Voltage	Warm White 2.8-3.2V
Luminous Flux	Warm White 160-190Lm
Color Temperature	2800K-3200K
Reverse Current	10μA
Viewing Angle	120deg
Recommend Forward Current	700mA
Test Condition	700mA

- Green Laser Line:

  Direct Diode 3V 3.7V 520nm Green Line Laser Module
  Wave length: 520nm. Output power: Class IIIa – less than 5mW
  DC3-3.7V operation .Working Temperature Range:-20 °C ~ + 40 °C
  Green Line laser,not DOT .Dimensions: D12 x 35mm
  1PCS Adjust Focus Green Diode Laser Module

- ESP32-S3-WROOM-1-N16R8

Complete ESP32 Code:
(Camera, LCD, SD Card, WiFi and Server, Buttons, Bluetooth/WiFi Switch, Battery Monitoring)

```
// P.I.X.E.L. Senior Design team
// ESP32 Code
// Camera Control, Display, SD card, Bluetooth Switching

// BLE Libraries
#include <BLEDevice.h>
#include <BLEUtils.h>
#include <BLEServer.h>
#include <BLE2902.h>

// Libraries for WiFi and Peripherals
#include "ArduCam_Mega.h"
#include <SPI.h>
#include <WiFi.h>
#include <WebServer.h>
#include <Adafruit_GFX.h>
#include <Adafruit_ST7789.h>
#include <JPEGDecoder.h>
#include <FS.h>
#include <SD.h>
#include "WiFiProv.h"   // if provisioning is not used, then do not need this

//Potentiometer Flash Pin
#define POT_PIN 2

//WiFi BLE Button
#define SWITCH_PIN 38  // GPIO pin for the physical button

//Camera control buttons
#define BUTTON_PIN 5
#define DELETE_BUTTON 4

// VSPI Pins (LCD & Camera)
#define TFT_CS    10
#define TFT_DC    9
#define TFT_RST   8
#define TFT_MOSI  11
#define TFT_CLK   12
#define TFT_BL    7
```

```
#define CAM_CS     6

// HSPI Pins (SD Card)
#define SD_CS 45
#define SD_SCK 21
#define SD_MISO 48
#define SD_MOSI 47

// LCD & Camera Configuration Sizes
#define LCD_WIDTH  240
#define LCD_HEIGHT 320
#define CAM_WIDTH  320
#define CAM_HEIGHT 320

// Center 320x320 image on 240x320 display
#define X_OFFSET  -40
#define Y_OFFSET  0

//Battery monitoring
const int analogPin1 = 17; // ESP32 ADC pin
const int analogPin2 = 18;
const float R1 = 10000.0; // 10k
const float R2_3 = 20000.0;  // Two 10k in series = 20k
const float ADC_MAX = 4095.0; // ESP32 12-bit ADC max value
const float VREF = 3.3; // Reference voltage
const float threshold = 4.8; // Combined battery threshold voltage
#define LED 40 // battery monitoring LED on pin 40

// BLE UUIDs for UART-style service (used by AB Shutter-style buttons)
// matches app
#define SERVICE_UUID        "6E400001-B5A3-F393-E0A9-E50E24DCCA9E"
#define CHARACTERISTIC_UUID "6E400002-B5A3-F393-E0A9-E50E24DCCA9E"

// Global Variables
// State Variables
volatile bool buttonPressed = false;
volatile bool deletePressed = false;
bool captureMode = false;
bool isWifiOn = false;
bool isBluetoothOn = false;
volatile bool shouldSwitchToWiFi = false;
int lastSwitchState = -1;
```

```
volatile bool captureRequested = false;

//Timing Variables
unsigned long captureTimestamp = 0;
const int captureDisplayTime = 5000;
unsigned long lastDebounceTime = 0;
unsigned long debounceDelay = 500;

// Interrupt function for capture button
void IRAM_ATTR handleButtonPress() { buttonPressed = true; }

// Interrupt function for delete button
void IRAM_ATTR handleDeletePress() { deletePressed = true; }

// For streaming the most recent image saved to SD card to the web
String lastSavedFilename = "";
bool sdInitialized = false;
void handleImageRequest();    // declared here for web handler registration

// BLE characteristic pointer (BLE write handler)
BLECharacteristic* pCharacteristic;

// Object Declarations
Adafruit_ST7789 tft = Adafruit_ST7789(TFT_CS, TFT_DC, TFT_RST);
ArduCam_Mega myCamera(CAM_CS);
SPIClass sdSPI(HSPI);
WebServer server(80);   //HTTP server on port 80

// BLE write callback for AB shutter
class MyCallbacks : public BLECharacteristicCallbacks {
 void onWrite(BLECharacteristic* pChar) {
     std::string value = std::string(pChar->getValue().c_str());
     if (value.length() > 0 && value[0] == 0x01) {
         Serial.println("AB Shutter BLE button pressed!");
         shouldSwitchToWiFi = true;  // start wifi capture mode
     }
 }
};

void setupBLE() {
 Serial.println("Starting BLE peripheral...");
 WiFi.mode(WIFI_OFF);    //Need to disable WiFi to avoid conflicts
```

```
 BLEDevice::init("ESP32_Shutter");   //BLE device name
 BLEServer* pServer = BLEDevice::createServer();
 BLEService* pService = pServer->createService(SERVICE_UUID);

 pCharacteristic = pService->createCharacteristic(
     CHARACTERISTIC_UUID,
     BLECharacteristic::PROPERTY_WRITE
 );
 pCharacteristic->setCallbacks(new MyCallbacks());   //Button handler

 pService->start();
 BLEAdvertising* pAdvertising = BLEDevice::getAdvertising();
 pAdvertising->start();    //Begin advertising

 Serial.println("BLE ready. Connect with iPhone and send 0x01.");
 isBluetoothOn = true;
 isWifiOn = false;
}

// Initialize SD card
void initSDCard() {
 sdSPI.begin(SD_SCK, SD_MISO, SD_MOSI, SD_CS);
 if (!SD.begin(SD_CS, sdSPI, 1000000)) {     //can adjust SPI speed here
   Serial.println("[ERROR] SD Card init failed");
 } else {
   Serial.println("[OK] SD Card initialized");
 }
}

// Save captured image to SD card
void saveImageToSD(uint8_t *jpegBuffer, uint32_t totalLength) {
 String filename = "/IMG_" + String(millis()) + ".jpg";  //Assign unique filename
 File file = SD.open(filename, FILE_WRITE);

 if (!file) {
   Serial.println("Failed to open file for writing!");
     return;
 }
   file.write(jpegBuffer, totalLength);  //Write image data to file
   file.close();
```

```
  Serial.println("Image saved: " + filename);

  //Store the name for later streaming
  lastSavedFilename = filename;
  Serial.println("lastSavedFilename updated to: " + lastSavedFilename);

  server.begin();   // start web server
  Serial.println("Web server started! Visit http://ESP32_IP/image");
  server.on("/image", handleImageRequest);    // set image route
}

// Stream live camera view to display
void streamLiveView() {
 if (captureMode) return;   // skip if image is being captured

 digitalWrite(TFT_CS, HIGH);   //disable LCD
 digitalWrite(CAM_CS, LOW);    //enable camera

 if (myCamera.takePicture(CAM_IMAGE_MODE_QVGA, CAM_IMAGE_PIX_FMT_RGB565) !=
CAM_ERR_SUCCESS){    //QVGA is resolution 320x240
     Serial.println("Failed to capture live view!");
     return;
 }

 const int imgWidth = 320;
 const int imgHeight = 240;

 uint32_t totalLength = myCamera.getTotalLength();
 uint16_t* frameBuffer = (uint16_t*)malloc(imgWidth * imgHeight * sizeof(uint16_t));
 if (!frameBuffer) {
     Serial.println("[ERROR] Failed to allocate frame buffer!");
     return;
 }

 uint32_t receivedLength = 0;
 uint8_t imageData[128];
 int pixelIndex = 0;

 // Read raw image data and convert to RGB565 pixels
 while (receivedLength < totalLength && pixelIndex < imgWidth * imgHeight) {
     uint8_t readLen = myCamera.readBuff(imageData, sizeof(imageData));
     receivedLength += readLen;
```

```
    for (int i = 0; i < readLen; i += 2) {
        uint16_t color565 = (imageData[i] << 8) | imageData[i + 1];
        frameBuffer[pixelIndex++] = color565;
    }
 }

 digitalWrite(CAM_CS, HIGH);   //disable camera
 digitalWrite(TFT_CS, LOW);     //enable LCD

 // Clear screen and draw full 320x240 image at (0,0) — assuming landscape mode
 tft.drawRGBBitmap(0, 0, frameBuffer, imgWidth, imgHeight);

 free(frameBuffer);    //release memory
}

// Capture and Display Image, Show on Display, Delete, Save to SD
void captureAndDisplay() {
 captureMode = true;
 Serial.println("Capturing Image...");

 digitalWrite(POT_PIN, HIGH);
 delay(200);

 digitalWrite(TFT_CS, HIGH);
 digitalWrite(CAM_CS, LOW);

 if (myCamera.takePicture(CAM_IMAGE_MODE_QVGA, CAM_IMAGE_PIX_FMT_JPG) !=
CAM_ERR_SUCCESS) {
    Serial.println("Failed to take picture!");
    return;
 }

 //pinMode(POT_PIN, OUTPUT);
 digitalWrite(POT_PIN, LOW);

 uint32_t totalLength = myCamera.getTotalLength();
 Serial.print("Image Size: ");
 Serial.println(totalLength);

 uint8_t *jpegBuffer = (uint8_t *)malloc(totalLength);
 if (!jpegBuffer) {
```

```
    Serial.println("Memory allocation failed!");
    return;
}

Serial.println("Memory allocated");

uint32_t receivedLength = 0;
while (receivedLength < totalLength) {
    uint8_t readLen = myCamera.readBuff(jpegBuffer + receivedLength, 128);
    receivedLength += readLen;
    Serial.println(receivedLength);
}

Serial.println("JPEG Image Captured.");

// Display the captured image
if (JpegDec.decodeArray(jpegBuffer, totalLength)) {
    Serial.println("JPEG Decoding Successful.");

    digitalWrite(CAM_CS, HIGH);
    digitalWrite(TFT_CS, LOW);
    tft.fillScreen(ST77XX_BLACK);

    while (JpegDec.read()) {
      uint16_t *pImg = JpegDec.pImage;
      tft.drawRGBBitmap(JpegDec.MCUx * JpegDec.MCUWidth, JpegDec.MCUy *
JpegDec.MCUHeight,
                pImg, JpegDec.MCUWidth, JpegDec.MCUHeight);
      yield();
    }
} else {
    Serial.println("JPEG Decoding Failed.");
}

unsigned long startTime = millis();
while (millis() - startTime < captureDisplayTime) {
  if (deletePressed) {
    deletePressed = false;
    buttonPressed = false;  //Prevent second capture
    shouldSwitchToWiFi = false;  //Just in case still having double presses
    Serial.println("Delete button pressed, returning to live view.");
    free(jpegBuffer);
```

```
      captureMode = false;
      tft.fillScreen(ST77XX_BLACK);  // Clear screen before resuming live view
      return;
    }
  }

  // If the delete button wasn't pressed, save the image
  saveImageToSD(jpegBuffer, totalLength);
  free(jpegBuffer);
  Serial.println("Captured Image Saved. Returning to live view.");
  captureMode = false;
  tft.fillScreen(ST77XX_BLACK);
  buttonPressed = false;
  deletePressed = false;
  shouldSwitchToWiFi = false;
}


// Serve Last Image over WiFi to the Server
void handleImageRequest() {
  Serial.println("Serving last saved image over WiFi...");

  if (lastSavedFilename == "") {
      server.send(404, "text/plain", "No image captured yet.");
      return;
  }

  File imageFile = SD.open(lastSavedFilename, FILE_READ);
  if (!imageFile) {
      server.send(500, "text/plain", "Failed to open saved image.");
      return;
  }

  Serial.print("Streaming file: ");
  Serial.println(lastSavedFilename);

  WiFiClient client = server.client();
  if (!client.connected()) {
      imageFile.close();
      return;
  }
```

```
 client.println("HTTP/1.1 200 OK");
 client.println("Content-Type: image/jpeg");
 client.println("Connection: close");
 client.println();

 uint8_t buffer[128];
 while (imageFile.available()) {
     size_t len = imageFile.read(buffer, sizeof(buffer));
     client.write(buffer, len);
 }

 imageFile.close();
 Serial.println("Finished streaming image.");
}

void setup() {
 Serial.begin(115200);
 delay(1000);
 pinMode(LED, OUTPUT);
 pinMode(POT_PIN, OUTPUT);
 digitalWrite(POT_PIN, LOW);

 Serial.println("[BOOT] Initializing camera...");
 if (myCamera.begin() != CAM_ERR_SUCCESS) {
   Serial.println("[ERROR] Camera init failed");
   return;
 }
 myCamera.setAutoFocus(true);    // autofocus on
  initSDCard();

 SPI.begin(TFT_CLK, -1, TFT_MOSI, TFT_CS);

 Serial.println("Initializing LCD...");
 tft.init(LCD_WIDTH, LCD_HEIGHT);
 tft.setRotation(1);               //landscape mode
 tft.fillScreen(ST77XX_BLACK);
 pinMode(TFT_BL, OUTPUT);
 digitalWrite(TFT_BL, HIGH);

 pinMode(BUTTON_PIN, INPUT_PULLUP);
 attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), handleButtonPress, FALLING);
```

```
 pinMode(DELETE_BUTTON, INPUT_PULLUP);
 attachInterrupt(digitalPinToInterrupt(DELETE_BUTTON), handleDeletePress, FALLING);

 pinMode(SWITCH_PIN, INPUT);
 int initialState = digitalRead(SWITCH_PIN);
 Serial.print(initialState);
 lastSwitchState = initialState;
 if (initialState == HIGH) {
   if (!isWifiOn && WiFi.status() != WL_CONNECTED) {

     WiFi.begin("SDNet", "CapstoneProject");
     Serial.print("[WIFI] Connecting");
     while (WiFi.status() != WL_CONNECTED) {
       delay(500);
       Serial.print(".");
     }
     Serial.println("\n[WIFI] Connected!");
     Serial.println(WiFi.localIP());
     isWifiOn = true;
     isBluetoothOn = false;
     }
   } else {
       setupBLE();
       isWifiOn = false;
       isBluetoothOn = true;
   }
}

void loop() {
 int currentSwitchState = digitalRead(SWITCH_PIN);

 if (currentSwitchState != lastSwitchState) {
   delay(100);  // debounce
   currentSwitchState = digitalRead(SWITCH_PIN);
   if (currentSwitchState != lastSwitchState) {
     lastSwitchState = currentSwitchState;

     // TOGGLE between WiFi and BLE, no matter high or low
     if (isWifiOn) {
       Serial.println("Switch flipped: WiFi -> BLE");

       WiFi.disconnect(true, true); // Disconnect from WiFi
```

```
      WiFi.mode(WIFI_OFF);
      delay(300);

      setupBLE();
      isBluetoothOn = true;
      isWifiOn = false;
    } else {
      Serial.println("Switch flipped: BLE -> WiFi");

      if (isBluetoothOn) {
        BLEDevice::deinit();
        delay(300);
      }

      WiFi.begin("SDNet", "CapstoneProject");
      Serial.print("[WIFI] Connecting");
      while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
      }
      Serial.println("\n[WIFI] Connected!");
      Serial.println(WiFi.localIP());

      server.on("/image", handleImageRequest);
      server.begin();
      Serial.println("[SERVER] Web server started!");

      isWifiOn = true;
      isBluetoothOn = false;
    }
  }
}

// Handle BLE shutter press asking for Wi-Fi
if (shouldSwitchToWiFi) {
  shouldSwitchToWiFi = false;

  Serial.println("Switching from BLE to WiFi because shutter button was pressed...");

  if (isBluetoothOn) {
      BLEDevice::deinit();
      delay(300);
```

```
  }

  WiFi.begin("SDNet", "CapstoneProject");
  Serial.print("[WIFI] Connecting");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\n[WIFI] Connected!");
  Serial.println(WiFi.localIP());

  server.on("/image", handleImageRequest);
  server.begin();
  Serial.println("[SERVER] Web server started!");

  isWifiOn = true;
  isBluetoothOn = false;

  captureAndDisplay();
}

if (WiFi.status() == WL_CONNECTED) {
  server.handleClient();
  if (buttonPressed) {
    buttonPressed = false;
    deletePressed = false;
    captureAndDisplay();
  }
}

if (!captureMode) {
  streamLiveView();
}

float voltage1 = analogReadMilliVolts(analogPin1) / 1000.0; // Get voltage directly
float batteryVoltage1 = voltage1 * (1 + (R1 / (R1+R2_3))); // Adjust for voltage
divider

float voltage2 = analogReadMilliVolts(analogPin2) / 1000.0; // Get voltage directly
float batteryVoltage2 = voltage2 * (1 + (R1 / (R1+R2_3))); // Adjust for voltage
divider
```

```
 if (batteryVoltage1 + batteryVoltage2 < threshold) {
     digitalWrite(LED, HIGH);
 } else { // blink if low battery
      digitalWrite(LED, LOW);
 }

 Serial.print("Battery Voltage1: ");
 Serial.println(batteryVoltage1);
 Serial.print("Battery Voltage2: ");
 Serial.println(batteryVoltage2);

 delay(100);
}
```

Python Script for Server to Google Drive:

```
import requests
import os
import datetime
import time
import random
import hashlib
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive

# need to authenticate google drive access
ESP32_IP = "192.168.10.114"
IMAGE_URL = f"http://{ESP32_IP}/image"
SAVE_DIR = "images"
GOOGLE_DRIVE_FOLDER_ID = "1j-JN4hXzCYdJ7iE1mYVw--ezV-WDLH6k"

gauth = GoogleAuth()
gauth.LocalWebserverAuth()
drive = GoogleDrive(gauth)

if not os.path.exists(SAVE_DIR):
    os.makedirs(SAVE_DIR)
# Store the hash of the last uploaded image to check for duplicates
last_image_hash = None

# function for retry attempts to connect to web server
def retry(operation, retries=3, base_delay=2, on_exception=Exception):
    for attempt in range(1, retries + 1):
        try:
            return operation()
        except on_exception as e:
            print(f"[Retry {attempt}] Error: {e}")
            time.sleep(base_delay * attempt + random.uniform(0, 1))
    print(f"[FAIL] Operation failed after {retries} attempts.")
    return None

# function to compute hash of an image to check for duplicates
def compute_hash(filepath):
    sha = hashlib.sha256()
    with open(filepath, "rb") as f:
        for chunk in iter(lambda: f.read(4096), b""):
```

```
            sha.update(chunk)
    return sha.hexdigest()

# function to grab image from web server
def fetch_image():
    #add time stamp metadata
    timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
    image_path = os.path.join(SAVE_DIR, f"esp32_image_{timestamp}.jpg")

    def do_fetch():
        print("Fetching image from ESP32...")
        # apply appropriate headers
        response = requests.get(IMAGE_URL, headers={"Accept": "image/jpeg"},
stream=True, timeout=10)
        if response.status_code != 200:
            raise RuntimeError(f"Failed to fetch image, status {response.status_code}")

        content_type = response.headers.get("Content-Type", "")
        # make sure the image is a JPEG
        if "image/jpeg" not in content_type:
            raise RuntimeError(f"Invalid content type: {content_type}")
        # open the image in chunks
        with open(image_path, "wb") as file:
            for chunk in response.iter_content(chunk_size=4096):
                if chunk:
                    file.write(chunk)

        print(f"Image saved to {image_path}")
        return image_path

    return retry(do_fetch)

# function to upload the image to google drive folder
def upload_image(image_path):
    def do_upload():
        print("Uploading to Google Drive...")
        # create new blank file in folder
        file_drive = drive.CreateFile({'title': os.path.basename(image_path),
'parents': [{'id': GOOGLE_DRIVE_FOLDER_ID}]})
        file_drive.SetContentFile(image_path)
        # upload fetched image and print path
        file_drive.Upload()
```

```
        print(f"Uploaded: {image_path}")
        return True

    success = retry(do_upload)
    # delete local file to free up storage
    if success and os.path.exists(image_path):
        os.remove(image_path)
        print(f"Deleted local file: {image_path}")

# Main loop
while True:
    path = fetch_image()
    if path:
        # compute hash to search for duplicates
        img_hash = compute_hash(path)
        # do not want to upload the same image multiple times to the google drive
        if img_hash == last_image_hash:
            print("Image is identical to previous. Skipping upload.")
            os.remove(path)
        else:
            # update hash and upload image to google drive if we can fetch new image
            last_image_hash = img_hash
            upload_image(path)
    else:
        # print statement for usability
        print("Skipping upload due to fetch failure.")
    # set system to sleep to avoid timing errors
    time.sleep(10)
```

Flask web app main code (app.py):

```
from flask import Flask, render_template, url_for, redirect

import requests

import os

from deepface import DeepFace

from utils import get_filename, update_images, update_sort, extract_datetime

from pathlib import Path


app = Flask(__name__)

# global variables

web_app_url =
"https://script.google.com/macros/s/AKfycbw2KM-GIc6C7mTgzi7TRHD8AgMC4O5yFYrvp4h5Tzk_Bj
TTYsJdJw036nqDaJjhoCD7/exec"

# dictionary for detected faces

face_db = {}  # { "person_X": ["image1.jpg", "image2.jpg"] }

@app.route("/")

def home():

   folder = Path('static/images')

   # if images are already saved, display them

   # otherwise, display generic home page

   if folder.exists():

       image_folder = os.path.join('static', 'images')

       image_names = os.listdir(image_folder)

       image_info = []

       for name in image_names:

           if name.endswith((".jpg", ".png")):

               image_info.append({

                   'url': url_for('static', filename=f'images/{name}'),

                   'filename': name

               })
```

```
        image_info.sort(key=extract_datetime, reverse=True)
        return render_template('home_with_images.html', images=image_info)
    return render_template('home.html')
@app.route('/run-download', methods=['GET'])
def download():
    # get images from google apps script
    response = requests.get(web_app_url)
    image_urls = response.json()
    # create folder to store images
    os.makedirs("static/images", exist_ok=True)
    # download images
    for url in image_urls:
        response = requests.get(url, allow_redirects=True)
        cd = response.headers.get('Content-Disposition')
        filename = get_filename(cd) or 'unknown.jpg'
        with open(f"static/images/{filename}", "wb") as f:
            f.write(response.content)
    return redirect(url_for('home'))
@app.route('/files')
def list_files():
    files = os.listdir('static/images')
    return render_template('files.html', files=files)
@app.route('/run-sort',methods=['GET'])
def run_sort():
    global face_db
    # path to images
    image_folder = "static/images"
```

```
    # list all images

    image_files = [f for f in os.listdir(image_folder) if f.endswith(('.jpg',
'.png','.JPG'))]

    # if image file is already in the face_db, take it out of the list

    image_files = [img for img in image_files if img not in face_db]

    # item for images with no faces

    face_db["no_faces"] = []

    # iterate over all images

    for image_file in image_files:

        image_path = os.path.join(image_folder, image_file)

        # detect faces

        # use retinaface for better accuracy

        # only take faces with high confidence

        try:

            faces =
DeepFace.extract_faces(image_path,detector_backend="retinaface",enforce_detection=Fals
e)

            faces = [face for face in faces if face.get("confidence", 1.0) > 0.9]

            if not faces:  # no faces found

                face_db["no_faces"].append(image_file)

                continue

            for face in faces:

                face_img = face["face"]

                # compare with known faces

                matched_person = None

                for person, saved_images in face_db.items():

                    if person == "no_faces":  # skip "no_faces" category

                        continue

                    reference_image = os.path.join(image_folder, saved_images[0])

                    # compare face embeddings
```

```
                    result = DeepFace.verify(image_path, reference_image,
enforce_detection=False)
                    if result["verified"]:  # face matches
                        matched_person = person
                        break
                if matched_person:
                    face_db[matched_person].append(image_file)
                else:
                    new_person = f"person_{len(face_db)}"
                    face_db[new_person] = [image_file]
        except Exception as e:
            print(f"Skipping {image_file}: {e}")
    # remove duplicates
    for person in face_db:
        face_db[person] = list(set(face_db[person]))
    return render_template('sorted.html', data=face_db)
@app.route('/run-update', methods=['POST'])
def run_update():
    new = update_images(web_app_url)
    return redirect(url_for('home'))
@app.route('/run-update-sort', methods=['POST'])
def run_update_sort():
    global face_db
    updated_faces = update_sort(face_db,web_app_url)
    face_db = updated_faces
    return redirect(url_for('sorted'))
@app.route('/sorted')
def sorted():
    return render_template('sorted.html', data=face_db)
```

```
if __name__ == "__main__":
    port = int(os.environ.get("PORT", 5000))
    app.run(host="0.0.0.0", port=port)
```

Flask web app functions (utils.py):

```
import re
from pathlib import Path
import requests
from deepface import DeepFace
import os
from datetime import datetime
def extract_datetime(image):
    name = image['filename']
    base = os.path.splitext(name)[0]  # removes .jpg or .png
    ts = base.replace('esp32_image_', '')  # '20250501_140409'
    dt_str = ts[:8] + ts[9:]  # '20250501140409'
    return datetime.strptime(dt_str, '%Y%m%d%H%M%S')
# function to get actual file name
def get_filename(cd):
    if not cd:
        return None
    fname = re.findall('filename="(.+)"', cd)
    if len(fname) == 0:
        return None
    return fname[0]
# update photos
def update_images(url):
    # get images from google apps script
    response = requests.get(url)
```

```
    image_urls = response.json()
    new_images = []
    folder = Path('static/images')
    # download images
    for url in image_urls:
        response = requests.get(url, allow_redirects=True)
        cd = response.headers.get('Content-Disposition')
        filename = get_filename(cd) or 'unknown.jpg'
        file_path = folder / filename
        if not file_path.exists():
            new_images.append(filename)
            with open(f"static/images/{filename}", "wb") as f:
                f.write(response.content)
    return(new_images)
def update_sort(face_db,url):
    # sort only new images
    response = requests.get(url)
    image_urls = response.json()
    folder = Path('static/images')
    # download images
    for url in image_urls:
        response = requests.get(url, allow_redirects=True)
        cd = response.headers.get('Content-Disposition')
        filename = get_filename(cd) or 'unknown.jpg'
        file_path = folder / filename
        if not file_path.exists():
            with open(f"static/images/{filename}", "wb") as f:
                f.write(response.content)
```

```
    image_folder = "static/images"
    all_files = os.listdir('static/images')
    # check for images not yet sorted
    used_files = set()
    for images in face_db.values():
        used_files.update(images)
    new_images = []
    for f in all_files:
        if f not in used_files:
            new_images.append(f)
    # iterate over new images
    for image_file in new_images:
        image_path = os.path.join(image_folder, image_file)
        # detect faces
        # use retinaface for better accuracy
        # only take faces with high confidence
        try:
            faces =
DeepFace.extract_faces(image_path,detector_backend="retinaface",enforce_detection=Fals
e)
            faces = [face for face in faces if face.get("confidence", 1.0) > 0.9]
            if not faces:  # no faces found
                face_db["no_faces"].append(image_file)
                continue
            for face in faces:
                face_img = face["face"]
                # compare with known faces
                matched_person = None
                for person, saved_images in face_db.items():
```

```
                    if person == "no_faces":  # skip "no_faces" category
                        continue
                    reference_image = os.path.join(image_folder, saved_images[0])
                    # compare face embeddings
                    result = DeepFace.verify(image_path, reference_image,
enforce_detection=False)
                    if result["verified"]:  # face matches
                        matched_person = person
                        break
                if matched_person:
                    face_db[matched_person].append(image_file)
                else:
                    new_person = f"person_{len(face_db)}"
                    face_db[new_person] = [image_file]
        except Exception as e:
            print(f"Skipping {image_file}: {e}")
    # remove duplicates
    for person in face_db:
        face_db[person] = list(set(face_db[person]))
    return(face_db)
```