ECE 395

Fall 2024

Project Obelisk Final Report

Braden Nigg(bnigg2) and Ben Malone(bmalo4)

Project Overview

Project Obelisk aimed to recreate and enhance the key tag unlocking systems commonly used for business security. Our variation eliminates the physical key tag, replacing it with a “custom key” powered by image recognition using artificial intelligence. An example of a “custom key” is really any object you could think of; the back of your phone, an apple, a rubber duck, ect. The final system allows users to train custom keys dynamically and generates an output signal to activate devices, such as a servo motor, which we used to simulate a door lock mechanism.

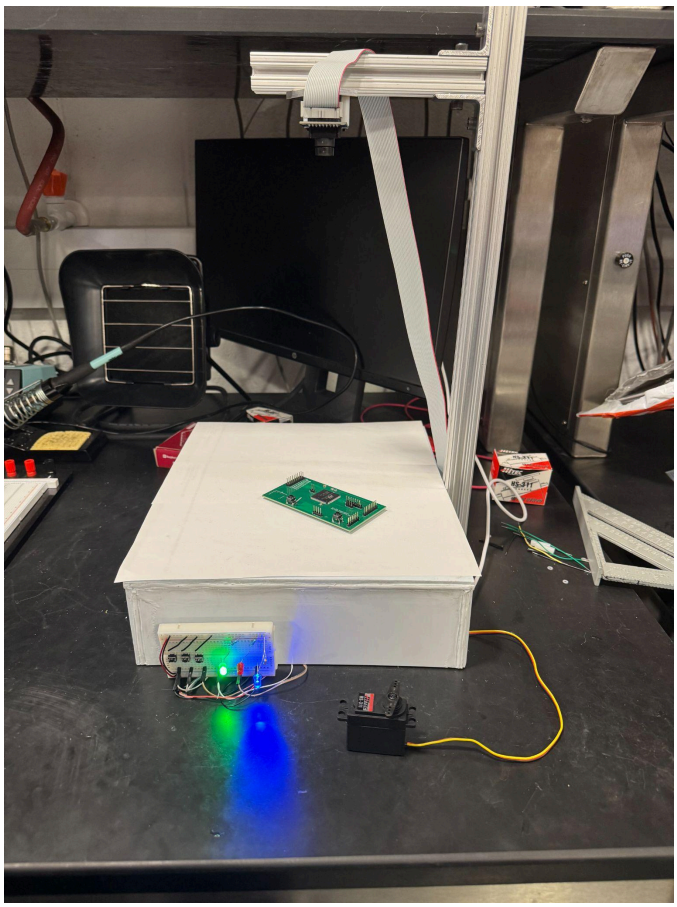


Table of Contents

System Architecture

Components

The core component for object capture in Project Obelisk is the OV7670 camera. This camera supports a maximum resolution of 640x480 at 30 fps and offers various configurable settings for color, lighting, and scaling adjustments. It also includes an onboard algorithm for automatic image calibration, enhancing its adaptability to different environments.

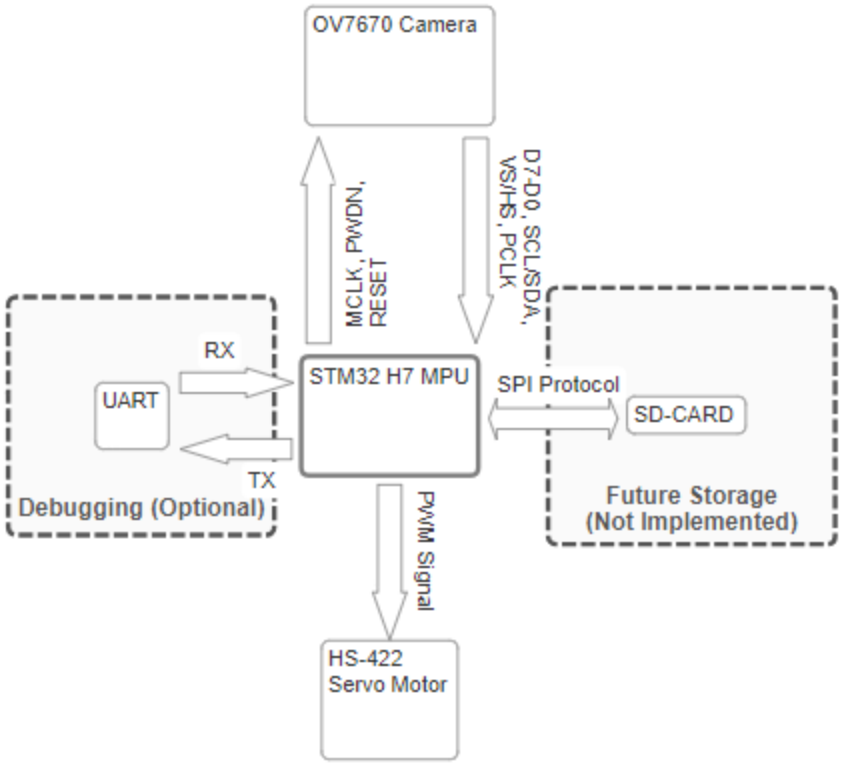
Project Obelisk leverages the STM32 H7 series processor, selected for its high-frequency internal clock, which is crucial for driving the OV7670's master clock (MCLK). Additionally, its 2 MB of embedded SRAM provides sufficient capacity to run the lightweight AI algorithm utilized in the project.

The system's output component is the HS-422 servo motor. While the motor was chosen primarily for demonstration purposes in this project, those replicating the design should consider the required torque for their specific application. For our implementation, the HS-422 proved suitable for showcasing the unlocking mechanism.

Debugging Components

We recommend using a UART device to transmit image captures to a more powerful system for analysis and validation during the calibration stages. Key considerations include the maximum baud rate supported by the UART device and the processing capability of the receiving system. For efficient transmission, we suggest a baud rate of at least 500,000 bits per second for lower resolution modes and exceeding 1 Mbit/s for full-resolution image captures. It is important to make sure the external system and the UART have matching baud rates to prevent corrupt data transmission.

Block Diagram



Hardware Design

Camera

The camera operates with Vcc and Vdd-IO expected at 3.0V, but it has an absolute maximum rating of 4.5V. We used 3.3V as the standard, which simplified the design by eliminating the need for additional voltage step-down circuitry.

The camera accepts a master clock input (MCLK) with a frequency range of 10–48 MHz; we chose the standard 24 MHz. However, testing revealed that the camera can operate at lower frequencies by leveraging its internal PLL (Phase Lock Loop), as described in the software section.

The pixel clock is an output signal that indicates when pixels are ready, with one pixel corresponding to two clock cycles. The camera also outputs Vsync (VS) and Hsync (HS) signals for synchronization. The Hsync signal can be modified to Href through register settings if required. In our design, these pins are not utilized as we implemented image capture based on a single snapshot. However, the Vsync signal can be valuable if the project is expanded to

support a continuous live feed. Additionally, Hsync can be useful for debugging purposes, such as identifying line numbers or implementing line-based data transmissions.

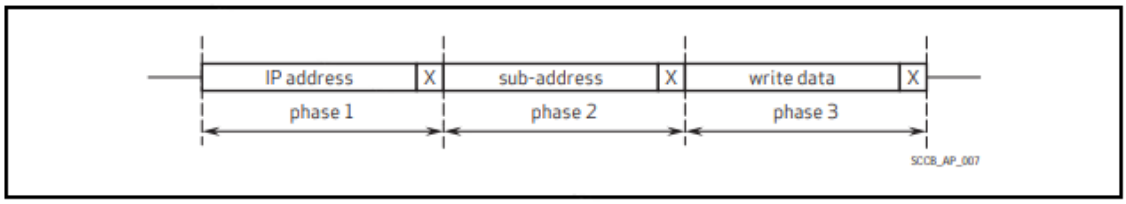
Additionally, the camera provides Power Down (PWDN) and Reset pins, which can be used for power savings and setup control.

SCCB Communication

The OV7670 camera uses the SCCB (Serial Camera Control Bus) protocol for hardware register communication. SCCB is very similar to the I2C protocol, with the key difference being that the SDA line can float in SCCB, whereas I2C requires a pull-up resistor. Despite this difference, the OV7670 can be successfully interfaced using I2C, as STM32 microcontrollers provide convenient drivers for I2C communication. By utilizing the transmit and receive functions provided by the STM32, it is possible to match the transmission sequence expected by the camera.

To ensure proper communication, pull-up resistors must be used for the SCL and SDA lines. A pull-up resistor value of 10kΩ is recommended, but it should not be lower than 4.7kΩ to maintain signal integrity. Additionally, the MCLK (Master Clock) signal must be connected to the camera; otherwise, I2C communication will not function correctly.

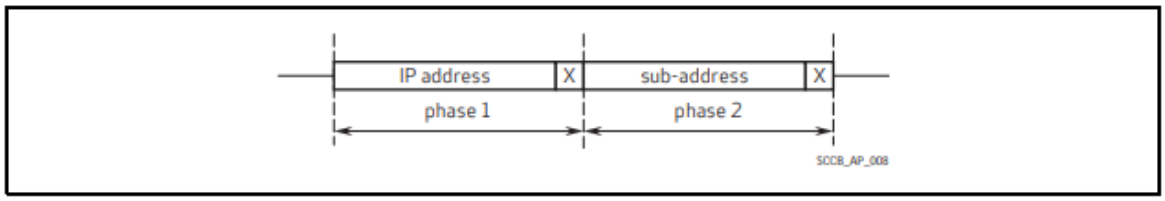
Figure 3-5 3-Phase Write Transmission Cycle



The OV7670 requires a 3-phase write cycle to modify its internal registers. First, a device address byte is sent, where the OV7670's address is 0x42, with bit 0 set to 0 to indicate a write operation. If bit 0 is set to 1, it would instead indicate a read operation. Second, the subaddress byte specifies the register address to be written. Finally, a write data byte containing the desired data is transmitted to the selected register. By following this sequence, the OV7670 can reliably receive configuration data.

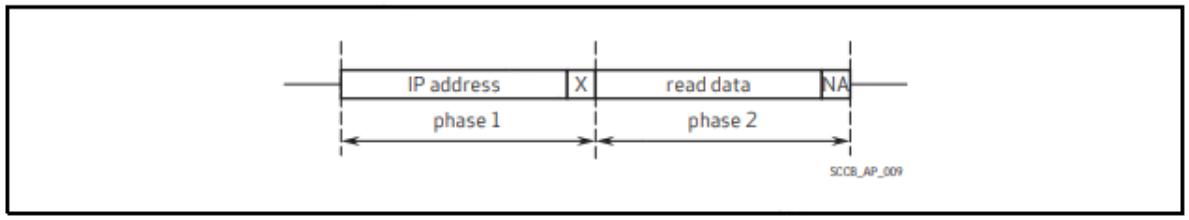
Reading data from the OV7670 camera follows a process similar to the write sequence but involves a 2-phase write followed by a 2-phase read. The first two bytes of the read process are identical to the 3-phase write sequence. First, the device address byte (0x42) with bit 0 set to 0 indicates a write operation, followed by the subaddress byte, which specifies the register address to be read.

Figure 3-6 2-Phase Write Transmission Cycle



After the subaddress is written, the process transitions to the read phase. Here, the device address is sent again, but this time bit 0 is set to 1, signaling a read operation. The camera then responds with the read data byte, which contains the value of the register specified in the earlier write phase. By following this sequence, the register value can be successfully read from the OV7670 camera.

Figure 3-7 2-Phase Read Transmission Cycle



SCCB Spec. Documentation

Servo Motor

The servo motor features a simple hardware interface with a Vcc requirement of 5V, a ground connection, and a signal pin for motor control. The signal pin expects a 50 Hz PWM signal for operation. To generate this signal, we used a timer, as the required frequency is relatively slow compared to the system clock. The motor changes position based on pulse width of the PWM, and will be leveraged in the software section.

Microcontroller

DCMI/DMA

One of the key features of the high-performance STM32 processors is the DCMI (Digital Camera Memory Interface) driver, which simplifies the high-speed GPIO constraints when gathering image data. Enabling DCMI allows for fine control over synchronization, DMA configuration, clock signal polarity, and the specification of captured bytes and lines.

In our approach, we opted to read all bytes to capture complete pixel information and read all lines, as this allows for easier software manipulation of the image if needed. We also chose

DMA to reduce the number of read/write operations from the camera, as the pixel clock operates at speeds nearly matching that of the processor.

DMA also supports interrupts, which trigger a callback when the DMA transfer is complete, known as the frame event callback. For synchronization, we selected hardware synchronization, as the OV7670 camera generates its own hsync and vsync signals based on its internal clock and state. Finally, we specified the input as 8-bits, corresponding to D7-D0 on the OV7670, which is crucial for correctly aligning the DMA request in memory.

Clocks

The MCU is configured to run at 168 MHz, which is more than sufficient for our use case. The primary requirement is that the AHB bus is at least 2.5 times faster than the pixel clock to ensure DMA can operate properly, as outlined in the STM32 DMA Configuration documentation. The main clock is derived from the internal high-speed clock (HSI), eliminating the need for an external oscillator. However, for greater precision, especially when working with PWM signals above 16 MHz, using an external oscillator may be preferable if available.

A Reset and Clock Control (RCC) option is used to generate a master clock out (MCO). This is necessary as timers are not clean enough at high frequencies. The MCO is configured at 24MHz and is a PLL divider down from the HSI clock.

GPIOs and Serial Wire

Refer to the configuration document for full specifications. The I/O is primarily used to provide user control over the project's state and to manage specific camera I/O functions. SWDIO and SWCLK are used for programming purposes and for debugging.

Software Development

Camera Register Setup

Below is a list of registers settings we used, there are many more and we highly recommend reading documentation. All registers are given by name and specific hex values can be found in documentation corresponding to name.

QVGA Resolution

Register	Register Value	Description
COM3	0x04	DCW - Enabled
COM14	0x19	Manual scaling and PCLK controls

SCALING_XSC	0x3A	Scales X-Dir of Image Array
SCALING_YSC	0x35	Scales Y-Dir of Image Array
SCALING_DCWCTR	0x11	Downsampling and scaling control
SCALING_PCLK_DIV	0xF1	Pixel clock divider control
SCALING_PCLK_DELAY	0x02	Pixel clock delay control

Table 2-2 of OV7670 Implementation Guide

RGB565 and Color Calibration

Register	Register Value	Description
COM7	0x04	Camera control register, enables DCW (digital chroma width) processing.
RGB444	0x00	Disable RGB444 settings
COM1	0x00	Camera control register, disables CCIR656 format setting
COM15	0xD0	Control for output format and compression, sets RGB output.
COM9	0x6A	Automatic gain control (AGC) settings, 128x Gain Ceiling
COM13	0x40	Control for gamma correction, set saturation
MTX 1	0xB3	Matrix coefficient 1, part of YUV to RGB conversion
MTX 2	0xB3	Matrix coefficient 2, part of YUV to RGB conversion
MTX 3	0x00	Matrix coefficient 3, part of YUV to RGB conversion
MTX 4	0x3D	Matrix coefficient 4, part of YUV to RGB conversion
MTX 5	0xA7	Matrix coefficient 5, part of YUV to RGB conversion
MTX 6	0xE4	Matrix coefficient 6, part of YUV to RGB conversion

Derived from Table 8-2 of OV7670 Implementation Guide

Registers below the bolded line can be adjusted based on calibration needs. Registers above the bolded line are required for RGB565 to function properly.

Frame Correction

Register	Register Value	Description
HSTART	0x15	Horizontal start position for the image frame. Affects where the capture begins.
HSTOP	0x03	Horizontal stop position for the image frame. Defines the end of the capture region in the horizontal direction.
HREF	0x00	Horizontal reference. Combines the start and stop positions to synchronize the horizontal scanning.
VSTART	0x03	Vertical start position for the image frame. Determines when the capture begins vertically.
VSTOP	0x7B	Vertical stop position for the image frame. Defines the end of the capture region in the vertical direction.
VREF	0x00	Vertical reference. Combines the vertical start and stop positions to synchronize the vertical scanning.

Derived from Table 8-2 of OV7670 Implementation Guide

This section may vary for each individual camera, so debugging and finding the correct values may be necessary. Essentially, it adjusts the window for capturing pixel data, where the start and stop values can behave in a modulo fashion.

Frame Rate Control

Frame rate is crucial for continuous capture mode, but modifying the pixel clock is equally important, as synchronization signals depend on it. We chose to match the pixel clock to our master clock, which runs at 24 MHz. The camera features an internal PLL that can either be configured or bypassed. We decided to use the PLL because it produces a more stable, square pixel clock compared to the input master clock. The following equation shows how the pixel clock frequency is determined based on the input master clock.

$$f_{\text{int\_clk}} = \frac{f_{\text{clk}} \times \text{PLL\_MULT}}{2 \times \text{CLRK}[5:0] + 1}$$

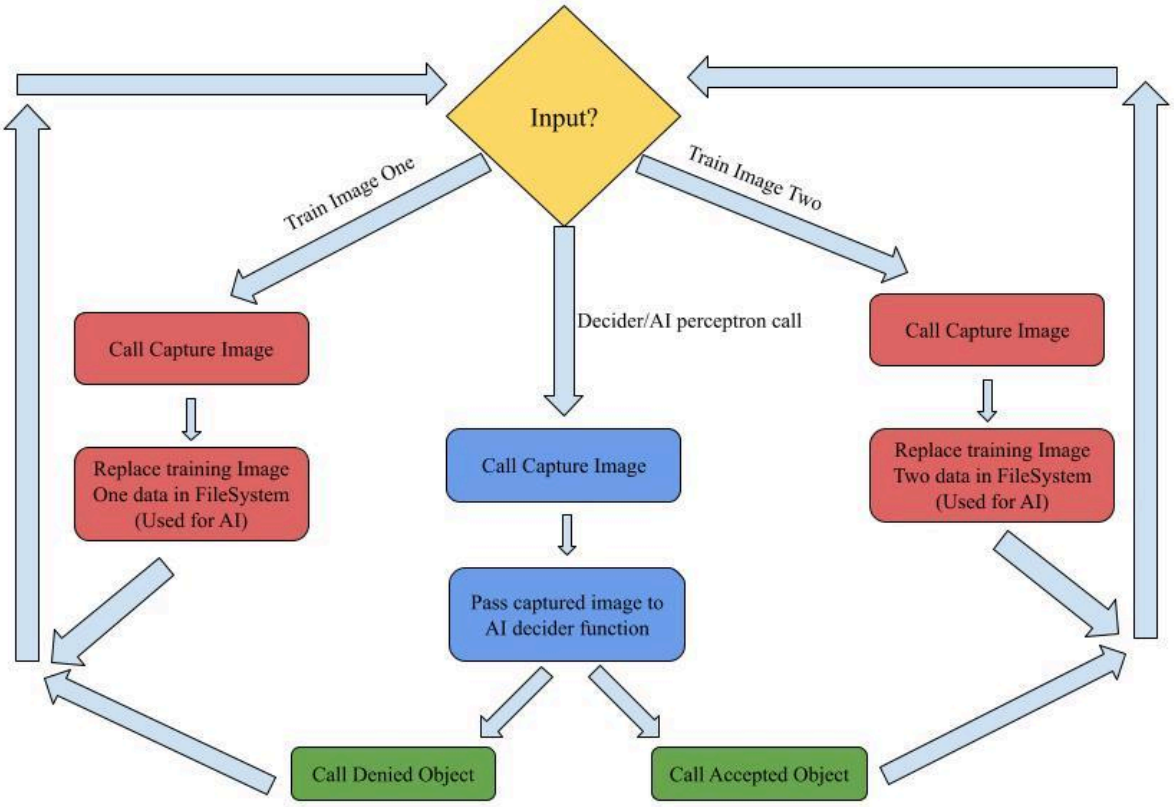
To match obtain our 24MHz clock the following registers are manipulated:

Register	Register Value	Description
DLBV	0x7A	Input clock x4, enable PLL
CLKRC	0x01	Cancels out the x4 multiplier

Derived from Table 3-2 of OV7670 Implementation Guide

Control Flow

Our system runs continuously polling button GPIOs while in a ready state. Once a GPIO is pressed, a decoding function determines the expected next state, training one of the two images or activates the decider for if the inputted key is valid. Below is a flow graph of our system's high level control loop:



Interrupts and Memory Management

Below is the memory layout for how our configured DCMI/DMA will place RGB565 values. Each word contains two pixels of complete information, as each pixel has 2 bytes of data.

Pixel X (D7 - D0)		Pixel X + 1 (D7 - D0)	
RRRRRGGG	GGGBBBBB	RRRRRGGG	GGGBBBBB

With 2 MB of available embedded SRAM, we opted to create a global image buffer for the two trained images and the test image. Each image is approximately 230 KB in size when converted

to RGB888, totaling 691 KB. We also have a buffer for the current capture, which is linked to our DMA. This buffer is 153.6 KB in size, as each pixel requires two bytes from the camera. In total, 844 KB of space is required for storing the images.

The image capture works by signaling an interrupt when the DMA transfer is complete. We override the FrameEventCallback function to add our custom interrupt handler. This is where we signal that the image is ready for the intended purpose, as specified in the control loop.

Artificial Intelligence

Training Set

The AI was trained on a much larger system. This allowed exploration of possibilities because of additional speed and memory, and for the ability to verify accuracy using lots of storage space for fake test images. Originally we attempted to explore the possibility of using a binary output neural network(perceptron), but because of memory constraints without off-chip memory, we had to come up with a different solution. Our solution was something like a baby-form of SURF or SIFT computer vision algorithms, which detect a variety of features about objects in images and make a decision upon them.

Algorithm and Tuning

To be more specific, our algorithm runs an edge detection matrix multiplication on the image data in order to isolate the parts of the image which contain the object. From here a number of very defining features can be detected; object size in the specific frame, a number of different color average values, some features about the specific shape of the object. Lots of hard-tuning and auto-tuning went into finding optimized epsilon values for each of the different features, but in essence each object has a relatively personalized set of features within a reasonable epsilon which is additionally supported with a second reference image.

To ensure accuracy of the feature set method though, more data was needed to ensure that true key objects were accepted, and false images were denied(to a reasonable accuracy). So we created an image scraping tool using google images and selenium, gathered around a thousand images with a description matching the white-background environment, and tested and tuned our epsilon values again to increase accuracy. This tool and our accuracy analysis training code/tool are available in the linked github.

AI limitations

While we put a lot of time into programming our AI code section, there are still strong limitations, as there are with all AI. Our less than 2MB ram shippable AI code is certainly no different and therefore some things won’t work. Most notably, the area in which images are taken from the camera must be where the entire object rests AND where the training images are taken. Otherwise the decider will hallucinate badly. Additionally, it is advised to avoid very non-unique

objects, or objects lacking a specific color. For instance, if you trained the decider on your black wallet, there might be significant security issues considering how many black square shaped objects exist. Even worse, an all white object which could trick the edge detector, completely ruining all other feature detection mechanisms, obviously causing hallucinations.

Servo Angle Control

As mentioned in the hardware section, the servo control input requires a 50 kHz PWM signal. To generate this signal, we need to configure the prescaler (PSC) and auto-reload register (ARR) of the timer.

$$f_{\text{PWM}} = \frac{f_{\text{clk}}}{(PSC + 1) \times (ARR + 1)}$$

For a 50 kHz PWM, we set PSC = 168-1 and ARR = 20-1. With these values, the timer produces the desired 50 kHz signal. Note these values can be of different multiples.

Next, to control the angle of the motor, we adjust the duty cycle. The following equation allows us to achieve the desired duty cycle based on the PWM period.

$$\text{Duty Cycle } (\%) = \frac{\text{CCR}}{\text{ARR}} \times 100$$

The key factor is that the servo motor's pulse width corresponds to the angle: a 2 ms pulse represents 180 degrees, and a 1 ms pulse represents 0 degrees. By manipulating the capture control register (CCR), we can generate the appropriate pulse width, thus controlling the motor's angle based on the desired state in the program.

Challenges and Mitigation to Demo

Issues

Nucleo Prototype

The project encountered several issues during the camera calibration stages, many of which were beyond our control. Late in development we suspected that our STM32 Nucleo prototype board had a short circuit early in development. However, the board was able to program and respond to modifications for several weeks alluding us to think it was code related.

Later, we faced an issue while erasing part of the flash memory during programming. Online sources suggest this could be a command byte issue in the bootloader. However, we encountered the same problem during the PCB building stage, when the board was shorted indicating the protoboard was certainly damaged. When this error occurred on the prototype

board, we could temporarily bypass it by over-volting the board, though this led to unstable behavior and program crashes.

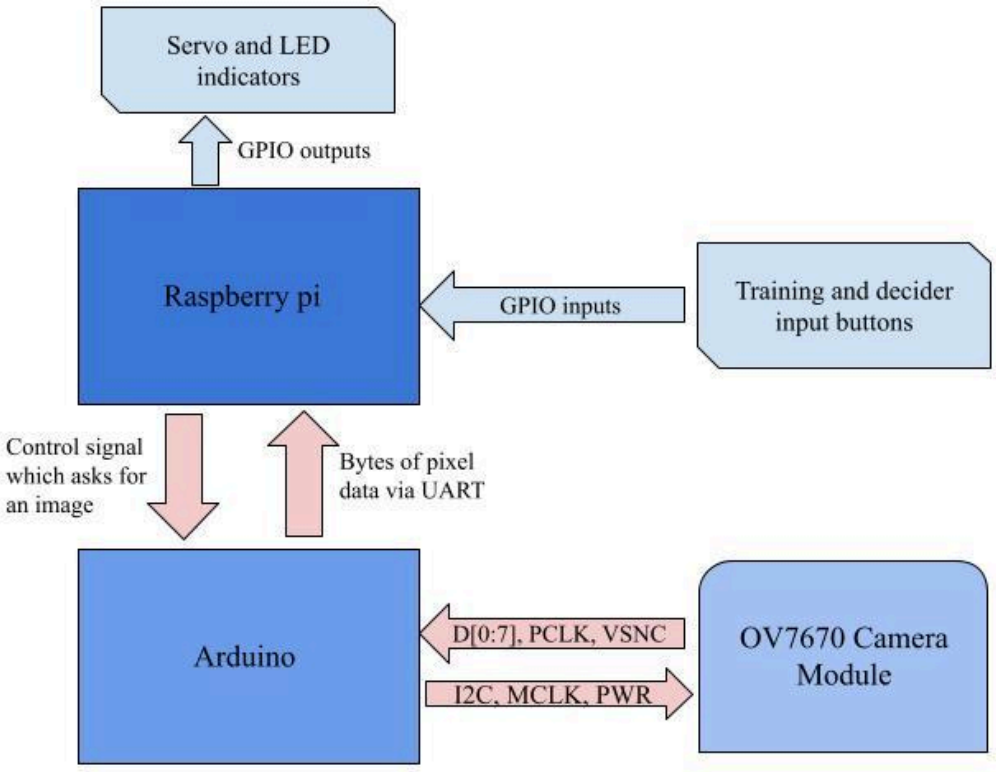
Custom PCB

In the PCB building phase, we successfully powered two PCBs, but after some testing, we suspected issues with both. The first PCB worked for a short time before encountering a debug authentication error during programming. This required us to drive the boot pin to boot from a different region. Further testing revealed that the board was shorted, causing the current to drop and leading to unstable program behavior. We are unsure if the debug authentication was an issue related to the hardware configuration or an advanced feature pertaining to the high end STM32 H7 chip.

The second PCB had a significant ground issue, which manifested as a triangular sine wave on the oscilloscope. Due to these challenges, we eventually ported our code and knowledge to an Arduino for demo purposes.

Arduino Setup

Arduino-Raspberry Pi Demo Block Diagram



Our Arduino-based demo prototype required a different approach due to the limited onboard memory compared to the STM32 chip. Specifically, the Arduino lacked sufficient memory to store even a single image buffer, which made running the AI on-chip impossible. To work around this, we sent pixel data byte by byte through the UART serial line to a Raspberry Pi, which handled the memory constraints. Although we initially faced challenges with byte alignment, we were able to resolve them successfully.

Once the data reached the Raspberry Pi, it took over the processing, running a simple control loop and executing the AI code on the images received from the camera. For GPIO control, the Raspberry Pi provided a straightforward infrastructure. Instead of sending control packets back to the Arduino, we managed the limited GPIO outputs directly from the Raspberry Pi.

Key Differences in Arduino Code

Given the limited time constraints (about 2 days) of porting our STM32 infrastructure to an arduino uno R3 we opted for merging a premade OV7670 library with our code. As much of the camera is configured through registers, there were few adjustments to be made to these settings. However, the arduino diverged from our system with not supporting DMA. The library way of gathering pixel data was to manually poll the pixel clock GPIO pin and send over UART within a time constraint. We largely left this alone as we needed a working product for the camera. The downside of this approach is very slow image capture with the combination of a slowed pixel clock and UART speeds.