# System Architecture of the PIXEL MIRROR: Real-time Multitasking, and Virtual Gloving

Larry Hui[1], Steph Akakabota[1], Kyle Nelson[1], Wen Cao[1]

[1]*Department of Mechanical Engineering, University of California, Berkeley*

***Abstract*—*Pixel Mirror* is a real-time computer-vision art project. We used an Apple M5 Macbook Pro's webcam and fed it into a computer vision (CV) pipeline on the same laptop; `YOLOv8-seg` instance segmentation CNN extracts a binary silhouette of every person in frame, `MediaPipe Hands` extracts fingertip positions and streams it over a 1 Mbaud USB-CDC link to an ESP32 Feather V2 driving a Waveshare RGB-Matrix-P2 $64 \times 64$ HUB75 LED panel. Our system uses two display modes selected by a physical button on the ESP32: a silhouette mode in which the person's shape is lit and a 10 kΩ potentiometer interpolates the color from white to red, and a gloving mode in which up to ten fingertips are rendered as colored $3 \times 3$ dots on a black field. A `PyQt6` GUI hosts the live camera, the silhouette, a colored $64 \times 64$ LED preview that mirrors the physical panel, and a control surface with sliders, buttons, and link telemetry. The full vision-to-pixel latency sits between 50 and 70 ms on an Apple M5 MacBook Pro at 25–30 fps, the link delivers $\geq 99\%$ of frames on the first attempt.**

## I. BACKGROUND AND MOTIVATION

### A. Engineering as Art

Mechanical engineering can be artwork (I know, shocking!). PIXEL MIRROR is built around that claim: a low-resolution LED canvas that turns a passer-by's movements into real-time pixel art, the way an exhibit might in a museum. The viewer walks past, their silhouette appears on a $64 \times 64$ panel, and a small physical control surface, one potentiometer and one button, lets them change what the panel does with their shape. The project leans on the vocabulary of engineering coursework (instance segmentation, framed serial protocol, ADC smoothing, panel DMA) alongside the vocabulary of visual art (silhouette, gesture, color wash, palette), and the claim is that the two are describing the same activity. What we are after is engineering used as a *medium* of art.

The course objective was to build a real-time, multi-tasking embedded system, which in our case, conveniently leverages computer vision and a physical output device, exposed through a GUI. We chose to make a **mirror**: a display that gives the viewer back to themselves, reduced to the lowest information bandwidth at which a recognizable human form can survive. The constraint is the point. A 4096-pixel 2-bit image of you is barely a person. Make it move, react, and shift color in real time, and the panel stops being a screen and becomes a presence.

A second motivation was *systems integration as a primary deliverable*. The project combines three subsystems that are each nontrivial on their own: a deep-learning CV chain, a multi-threaded GUI, and an ESP32. With this, the hard problems live at the intersection. The CV is a thin layer over a pretrained model, the firmware drives a well-documented panel through a vendor library, and the GUI is a standard Qt application; the engineering content sits mostly in the protocol design, the threading model, and in the failure modes that show up when those three things have to agree about the states and making the integration very challenging.

### B. Project Goals

1) Build a real-time computer-vision processing system around an instance-segmentation network. `YOLOv8-seg` restricted to the COCO (dataset) `person` class, running on the laptop, with end-to-end vision-to-pixel latency in the tens of milliseconds.
2) Expose interactive visual filtering through a GUI. Sliders for confidence and LED threshold, buttons for camera reset and re-calibration, plus a live LED preview that mirrors what the physical panel is showing.
3) Run all host–ESP32 traffic over one bidirectional UART link. A single UART serial link carries mask frames host→ESP32 *and* sensor telemetry ESP32→host, with framed packets and CRC protection so the communication layer survives noise, retries, and an entire hardware pivot.

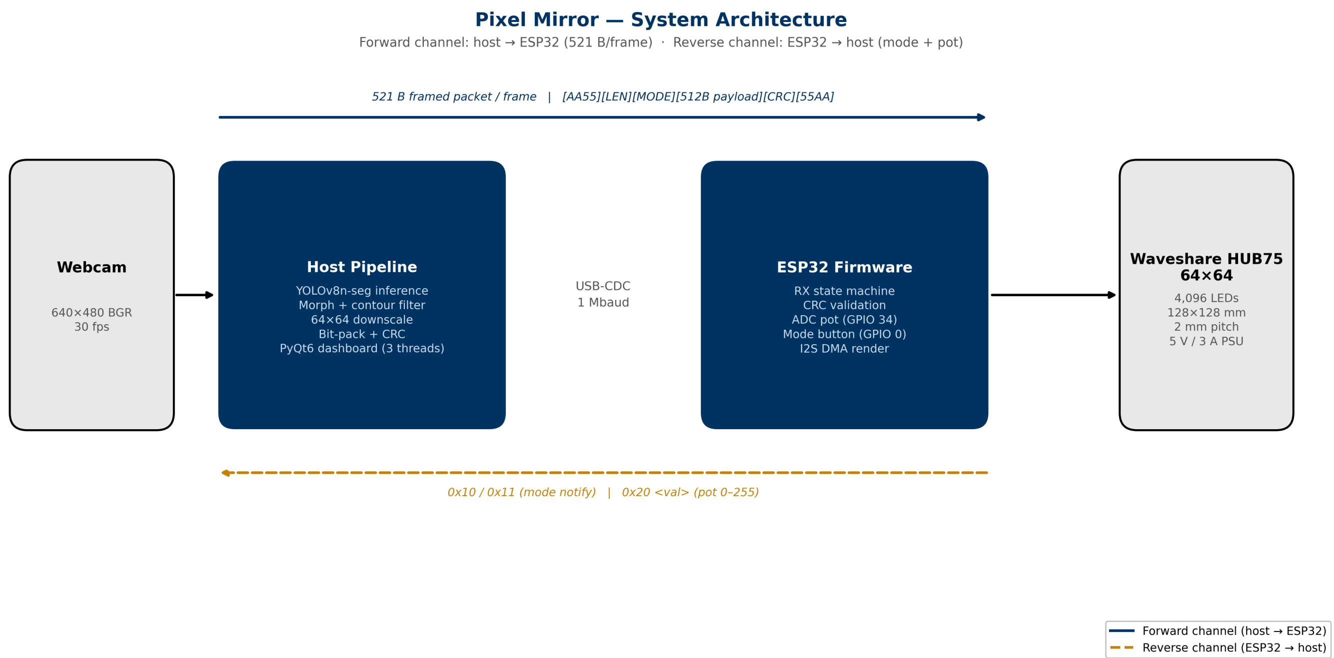


**Figure 1:** A computer and a microcontroller talk to each other in real time and the computer continuously sends silhouettes, while the microcontroller simultaneously sends back control signals and actuates the LED board.

### *C. System Architecture*

The system decomposes into three logical nodes connected by a single serial link:

- **Image-capture node: the laptop camera.** The laptop's built-in webcam captures the raw $640 \times 480$ frame and hands it to the host process. There is no external camera, capture card, or USB cam in the loop. The same frame later reaches the GUI (for display) and the LED panel (after compression and transport), so the camera is in effect the source for two consumers.
- **Host node: Python on the laptop.** Runs `YOLOv8-seg`, applies a per-person bounding box, builds an annotated silhouette, then downsizes to a $64 \times 64$ pixelated frame. Hosts the `PyQt6` dashboard, which renders the original feed, the cleaned silhouette, and a colored LED preview side-by-side. Sends the processed pixel data to the display node over USB serial.
- **Display node: ESP32 + button + potentiometer +** $64 \times 64$ **LED panel.** Receives framed pixel data from the host and renders to the HUB75 panel. A push-button toggles between two modes (silhouette wash and fingertip glove). The potentiometer controls the silhouette wash intensity (white to red) in real time. Both control inputs stream **back** to the host, which is how the GUI's LED preview ends up matching the physical panel rather than guessing at it.

Figure 1 summarizes the three-node system architecture.

### *D. Computer-vision Stack*

The vision side runs every frame in roughly this order, on the GUI's worker thread (`Project_GUI.py` and the standalone reference loop `vision_send.py`):

1) **Frame capture.** OpenCV `VideoCapture` opens the laptop's built-in webcam (camera index 0) at $640 \times 480$. Backend selection is portable across OSes: AVFoundation on macOS, V4L2 on Linux i.e. NVIDIA Jetson, with 'CAP_ANY' as a fallback. No external USB camera or capture card is in the loop.
2) **YOLOv8 instance segmentation.** A `yolov8n-seg.pt` checkpoint ( 6 MB) is loaded once at startup through Ultralytics. Each frame is passed to 'model.predict(..., classes=[0], conf=...)', restricting detections to the COCO 'person' class and to a configurable confidence threshold. The model emits both bounding boxes and per-instance masks at the model's internal resolution ($160 \times 160$ prototype $\times$ 32 coefficients in `YOLOv8-seg`).

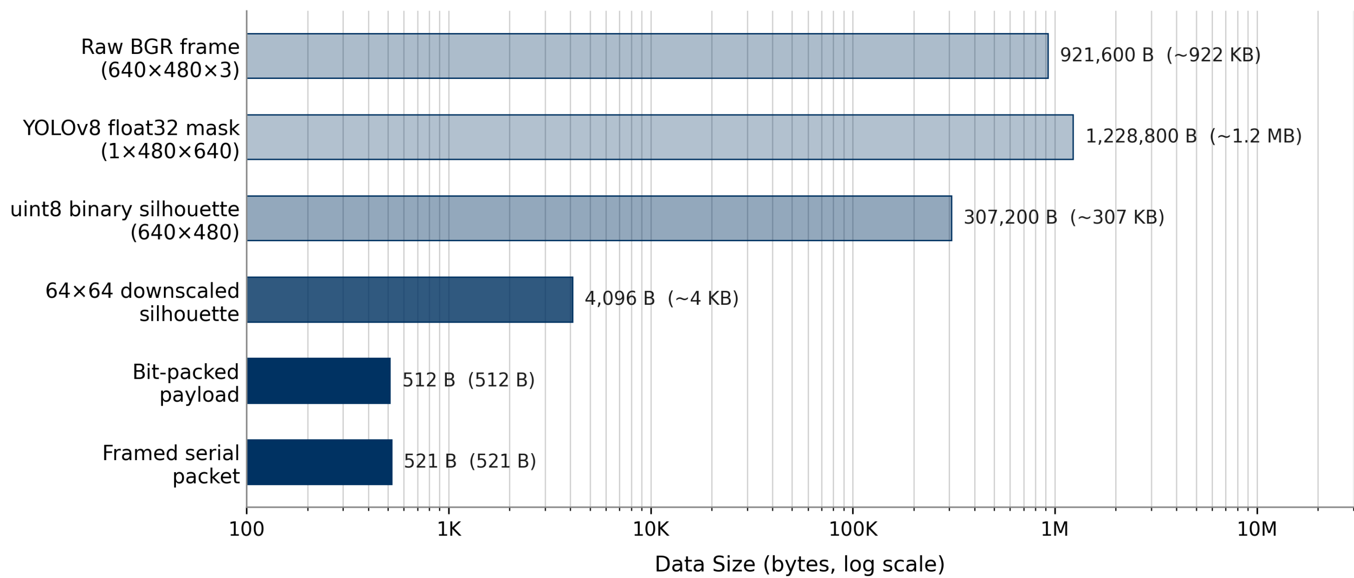


**Figure 2:** How much data each processing stage produces, on a log scale. A raw camera frame is nearly 1 MB; the packet sent to the ESP32 is just 521 bytes—a 1,770× reduction. This figure quantifies bandwidth

3) **Silhouette assembly.** Each per-person mask is resized to the camera frame, then used a logical OR mask to get a single binary silhouette. A 5×5 elliptical morphological close smooths jagged mask edges.
4) **Contour cleanup.** `findContours` extracts external contours; contours below 0.2% of frame area are dropped as stray bits, the rest are re-filled into a clean silhouette. This is what gets sent to the panel and what the GUI displays in the "Silhouette" viewport.
5) **Downscale to** $64 \times 64$**.** The clean silhouette is resampled with `INTER_AREA` (an area-weighted resize, which preserves coverage better than nearest-neighbor when shrinking), then re-thresholded at 96 to recover a clean binary image at panel resolution. The downsize **is** the pixelation.
6) **MediaPipe Hands.** Independently, the same frame is converted to RGB and run through `mp.solutions.hands` (up to two hands). The five fingertip landmarks per hand (IDs 4, 8, 12, 16, 20) are mapped from the model's normalized coordinates directly to the $64 \times 64$ grid, then assigned a color from a cycling 5-color palette that rotates over time (Figure 3). In silhouette mode MediaPipe is skipped entirely, which claws back about 6–8 ms per frame.
7) **Serial transmission.** Depending on the ESP32's reported mode, either the $64 \times 64$ binary mask or the fingertip list is packed and sent over the framed serial protocol. The host gates on ‘send_complete_signal‘, so at most one frame is ever in flight (see §3). reduction achieved across these stages: a 1,770× reduction from the 921 KB raw frame down to a 521-byte framed serial packet.

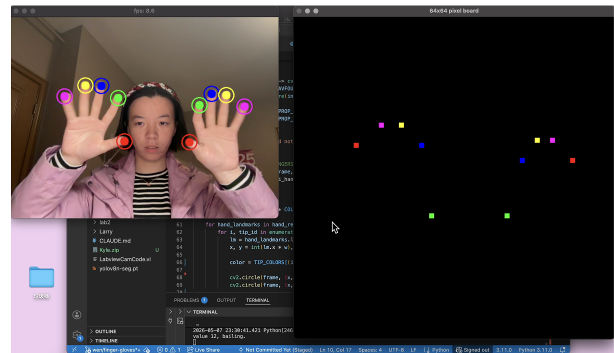

**Figure 3:** Gloving mode tracking up to ten fingertips concurrently.

### E. Hardware

The ESP32 talks to the panel over HUB75 through the ‘ESP32-HUB75-MatrixPanel-DMA‘ library; that library uses the $I^2S$ peripheral in DMA mode to clock pixels out

| Part | Role |
| --- | --- |
| M-series MacBook | `YOLOv8-seg`, MediaPipe, PyQt6 GUI, serial host |
| Laptop built-in webcam (640×480, opened at index 0) | Image source |
| Adafruit ESP32 Feather V2 (PICO-MINI-02, 8 MB flash, 2 MB PSRAM) | Frame RX, panel driver, button + pot I/O |
| Waveshare RGB-Matrix-P2 64×64 HUB75E, 2 mm pitch, 4096 LEDs | Output |
| 10 kΩ linear potentiometer on GPIO 33 (ADC1_CH5) | Color-blend control |
| Tactile pushbutton on GPIO 34, with external 10 kΩ pull-up to 3.3 V (GPIO 34 is input-only and has no internal pull-up) | Mode toggle |
| Separate 5 V / 3 A+ DC PSU for panel; ESP32 powered from MacBook USB | Avoids brownout on full-white frames |
| 1000–2000 $\mu$F electrolytic across panel V+/GND | Soaks turn-on inrush |
| USB CDC, 1,000,000 baud, 8N1 | Host ↔ ESP32 link |

TABLE I: Hardware Subsystems, Parts, and Roles

at panel-refresh rates with almost no CPU cost. Three GPIOs that DevKitC tutorials assume are free turn out to be unavailable on the Feather V2: GPIOs 16 and 17 are reserved for on-module PSRAM, and GPIO 23 is simply not broken out. The A, D, and CLK signals were remapped to GPIOs 21, 20, and 22 respectively. The full pin map and a known-good wiring/bring-up procedure are documented in `Kyle/firmware/WIRING.md`, including the GPIO 12 strapping-pin caveat (it must not be pulled high at boot) and the power-rail separation that keeps a fully lit panel from browning out the dev board.

### F. Technical Challenges

Four parts of the build were a lot harder than the rest, and most of the architectural choices we made were driven by them.

- **LabVIEW → PyQt6 GUI.** The original plan was a LabVIEW GUI. The room we expected to demo in—the big LED screen at the Grimes Engineering Center—has a control laptop with LabVIEW already installed and licensed, which made the choice obvious at the time. We pursued it for two weeks. The problem was coupling. The same project needed to load the CNN, decode camera frames at frame rate, talk to a serial port at 1 Mbaud, and render a custom pixel-art aesthetic. Doing all of that in LabVIEW means stitching together LabVIEW Vision, a Python node for YOLO, a custom serial VI, and a hand-built UI. We rewrote the GUI in `PyQt6`, and the whole stack collapsed into one Python process, one thread model, one debugger. The Grimes laptop now runs the Python build directly; the LabVIEW dependency went away with the rewrite.
- **OpenCV masking + contour detection → `YOLOv8-seg`.** The first CV approach was background subtraction (`MOG2/KNN`) (Gaussian Mixture Model / k-Nearest Neighbors) plus contour filtering—the typical way to find the moving blob. It worked in a controlled lighting setup and fell apart the moment anyone moved the camera, opened a window, or changed shirt color. `YOLOv8-seg` is class-aware (it knows what a person is, not what "something that moved" is) and motion-independent (it works on a still subject). It is also far more forgiving of lighting changes. The cost—about 25 ms of inference per frame on CPU—is paid willingly.
- **Wiring the $64 \times 64$ LED matrix board.** HUB75 has thirteen logic lines plus four ground/power lines, and the Adafruit Feather V2 hides three GPIOs that DevKitC tutorials assume are free (16, 17 reserved for PSRAM; 23 not broken out at all). We remapped A/D/CLK and documented the full pin map in `Kyle/firmware/WIRING.md`. Other gotchas—the GPIO 12 strapping pin, panel-vs-ESP32 power-rail separation, the 1000–2000 µF bulk cap across V+/GND, and the optional 74HCT245 level shifter for 3.3 V→5 V—are documented in that same file. Once it was on paper, bring-up at a new venue is about a 30-minute procedure.
- **Grimes deployment vs MVP—the same $64 \times 64$ LED panel runs both.** The minimum-viable build runs the $64 \times 64$ panel locally on the bench next to the laptop. The Grimes Engineering Center deployment runs the **same** Python code, with the **same** wire protocol and the **same** ESP32 firmware underneath. Keeping the wire format identical across both targets—512 bytes, 1 Mbaud, framed and ACK'd—meant the Grimes deployment is a hardware swap rather than a software port.

The system also began life on different hardware: an

NVIDIA Jetson Nano with a PS3 Eye camera, `MOG2/KNN` background subtraction, and a WS2812B 108×108 NeoPixel grid driven by FastLED. That archived path lives at `Kyle/_archive/` and is preserved for the historical record but no longer compiled or extended. The pivot to HUB75 + YOLO happened around mid-semester for two reasons that compounded: (a) Waveshare LED Display updates at 800 kbit/s are bandwidth-starved on a 108×108 grid (one frame is roughly 35 ms of raw bit-banging, before any compute), and (b) background subtraction is fragile under live-demo conditions. `YOLOv8-seg` side-stepped both, and the HUB75 panel refreshes at hundreds of Hz off DMA. The protocol had been written generically enough that the pivot did not require rewriting the wire format—only updating the payload size from 1458 bytes to 512.

## II. GUI, Real-Time, Interrupts, and Multitasking

The user-facing application is a single `PyQt6` desktop application: `Kyle/vision/dashboard/Project_GUI.py` (Fig. 4). We picked Qt over LabVIEW because the project had to do four things at once: load a deep-learning model, process OpenCV images at frame rate, talk to a serial port, *and* render a pixel-art aesthetic. `PyQt6` gives us all of that in one process, and the Qt threading model (QThread + signals/slots with `QueuedConnection`) maps cleanly onto the producer/consumer structure we needed. The GUI is the only operator surface in the build. There is no LabVIEW anywhere.

### A. GUI Layout

The window opens at 1280 × 980 minimum. From top to bottom it has: a checkered top border; a header block with the title “PIXEL MIRROR” in 56 pt Fixedsys, the course string, and the team names; a row of three live viewports (*Original Feed*, *Silhouette*, and *LED Preview (*64 × 64*)*); and a footer row with sliders (CONFIDENCE 0–100%, LED THR 0–255), buttons (AI SCENE ANALYSIS, RESET CAMERA, RE-CALIBRATE), and a system-status panel that reports FPS, people count, ESP32 connection state, current mode, pot percentage, and a running ACK/NAK counter.

Twelve draggable pixel-art emoji decorations float over the layout. They are real `QLabels` with custom mouse event handlers, so the user can grab and drag them around the window. A small thing, but it makes the decoration feel like part of the interface rather than wallpaper.

### B. Real-Time Event Sources

Four event sources fire concurrently, each on its own primitive (Qt signal, hardware timer, or DMA interrupt). They are logically independent, which is what makes the threading model tractable:

- **Panel refresh.** Hardware timer + I$^2$S DMA interrupt inside `ESP32-HUB75-MatrixPanel-DMA`. Refreshes the panel hundreds of times per second from the ESP32 framebuffer. CPU cost ≈ 0.
- **New frame from host.** UART RX interrupt feeds a byte-driven `pollFrame()` state machine. When a complete CRC-valid frame arrives, the framebuffer is overwritten and the dirty bit is set.
- **Mode toggle.** Active-low GPIO 34 polled every loop, with a 50 ms software debouncer. Flips `currentMode` and emits `0x10/0x11` over serial.
- **Pot adjustment.** 12-bit ADC sampled every loop through an EWMA filter ($\alpha = 0.1$). Pot-change notifications are rate-limited to 50 ms. Updates the white→red lerp applied to the silhouette.
- **Filter adjustment.** Qt `valueChanged/clicked` signals on the GUI thread, written directly to the worker thread’s attributes. The worker reads the new threshold on its next frame iteration.
- **GUI render.** `change_pixmap_signal` emitted by the vision worker thread, received as a queued event on the GUI thread. Repaints the three viewports and updates the status panel labels.

### C. Real-Time Threading Model

`PyQt` runs a single-threaded event loop, so *anything* that blocks for more than  16 ms on the GUI thread shows up as dropped frames. YOLO inference takes  25 ms per frame on CPU, MediaPipe takes another 6–8 ms, and OpenCV camera reads block until a frame is available. None of that can run on the GUI thread.

`VisionWorker` is a `QThread` that owns the camera, the YOLO model, the `MediaPipe Hands` instance, and the serial `SerialSender`. Its `run()` method is the main per-frame loop, and on each iteration it emits two signals: `change_pixmap_signal` carries three numpy arrays (annotated camera, silhouette, LED preview), and `status_signal` carries FPS, people count, ESP32 mode, pot value, connection state, and ack/nak counts. Because Qt queues the connection between a thread-owned signal and a GUI-thread slot by default, the GUI thread sees those signals as ordinary Qt events and handles them in

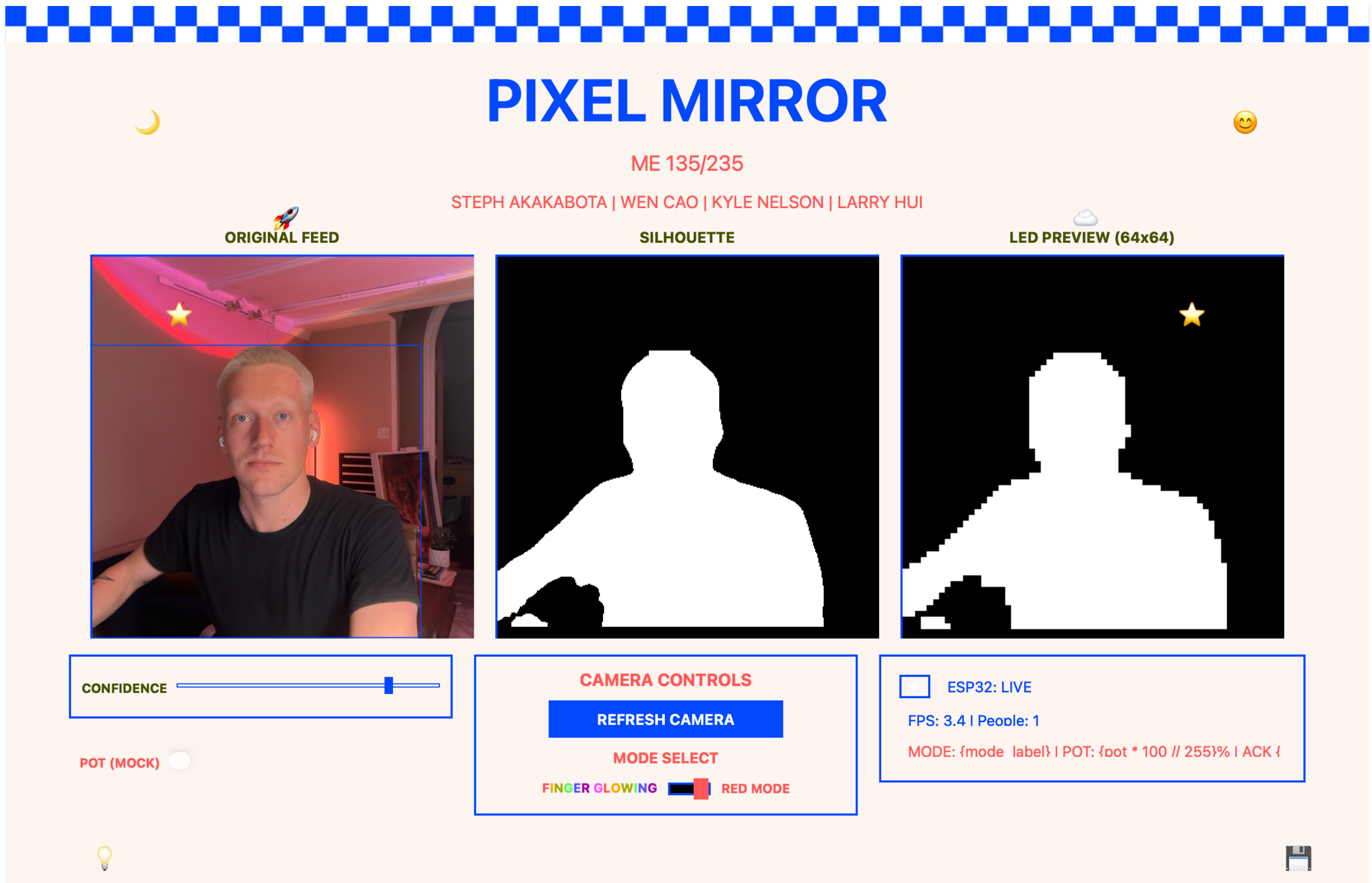


**Figure 4:** The PyQt6 GUI showing the live camera feed, the silhouette mask, and the $64 \times 64$ LED preview side by side.

its own time. The worker never blocks waiting for the paint.

A second throttle sits in front of the link itself. The vision worker gates transmission on `TX_MIN_INTERVAL_S = 1.0 / 30.0` (`Kyle/vision/vision_send.py:51`, `Project_GUI.py:47`): even when YOLO inference finishes in under 33 ms, the host waits until at least 33.3 ms have elapsed since the last frame before pushing the next one. This complements the one-frame-in-flight ACK gate—the gate bounds queue depth, the interval bounds frame rate. Together they pin host throughput to exactly the rate the panel can usefully consume and the rate the 50 ms ACK timeout was sized for.

| Stage | Cost |
|---|---|
| Camera capture + colorspace convert | ∼5 ms |
| `YOLOv8-seg` inference (CPU) | ∼25 ms |
| Mask cleanup + downscale + bit-pack | ∼3 ms |
| `MediaPipe Hands` (mode 1 only) | ∼6–8 ms |
| Serial wire time at 1 Mbaud (521 B) | ∼5 ms |
| Firmware ingest + render + ACK | ∼2 ms |

TABLE II: End-to-end vision-to-pixel latency budget. Vision-to-pixel lands between 50 and 70 ms at 25–30 fps. The ACK timeout is 50 ms.

### *D. Latency Budget*

End-to-end vision-to-pixel latency was measured across a 1,000-frame demo run on an M-series MacBook. The dominant contributors are shown in Table II.

### E. Multitasking on the ESP32 Side

The firmware (`POT_working/src/main.cpp`) runs on a single FreeRTOS task, the PlatformIO `loop()`, and time-slices it cooperatively. Each iteration walks six steps: reads the button and debounces; pumps the byte-by-byte receive state machine (`pollFrame`); reads and EWMA-smooths the pot ADC; emits a pot notification if the smoothed value has changed and 50 ms have elapsed; repaint the panel if the dirty bit is set; and checks the 5-second watchdog. The render-only-when-dirty optimization is what lets the loop tolerate 30+ Hz frame arrival without dropping pot updates. A typical idle iteration comes back in under 200 µs.

### F. Interrupts

The projects mainly utilize three interrupt sources.

1) New image data was driven by serial communication. This interrupt triggers whenever new data arrives via UART. Instead of making the processor constantly waste time checking if new data has arrived, the incoming data itself trips the interrupt to make sure that no incoming data is lost or dropped.
2) Ability to switch between the color interpolation mode and the gloving mode. Whenever we click the switch button on the circuit, it is able to switch freely between the two modes and show on the GUI.
3) Potentiometer filter. This interrupt is triggered by a change in the GPIO pin connected to a potentiometer. We apply the filter by physically turning the knob, and it will also be shown by the GUI interface by how dark the red shadow is in the pixel display.

### G. Bonus: A Refined Serial Worker (`serial_worker.py`)

A later cut of the serial layer ditches the QThread-owning-everything pattern for a `QObject + moveToThread` design. A private `_Core` lives on a dedicated worker thread and owns the `SerialSender`; the public `SerialWorker` proxy on the GUI thread re-emits private signals over `QueuedConnection` to land in the core's slots. The GUI gates emissions on `send_complete_signal` to keep a strict one-frame-in-flight invariant. Without that gate, the internal queue is unbounded, so it can swell whenever ACK retries push the per-frame cost above the 33 ms frame budget. The code is committed and unit-tested via `tests/test_protocol.py`; we just haven't cut it over yet.

## III. Communication Protocol

The link between host and ESP32 is a single USB-CDC channel running at **1,000,000 baud, 8-N-1**. On top of that one byte stream the protocol carries three separate concerns: bulk image data from host→ESP32, sensor telemetry from ESP32→host, and an ACK/NAK reliability layer running over both.

The Python side is `vision/serial_protocol.py` (`SerialSender`, `build_frame`, `pack_mask`, `pack_fingertips`, `crc16_ccitt`). The ESP32 side is the `pollFrame()` state machine and the `Serial.write` paths in `firmware/POT_working/src/main.cpp`. The two were written together, and the pytest suite at `vision/dashboard/tests/test_protocol.py` keeps them byte-for-byte in sync.

### A. Why a Framed Custom Protocol

USB-CDC at 1 Mbaud is a byte stream, not a packet stream. The naive option, which is to write 512 bytes per frame and hope, works fine on a bench with one device and one cable. It fails the first time the host opens the port mid-frame, the cable is jostled, or a USB buffer underruns. Any protocol sitting on top of USB-CDC has to answer four questions per frame: *where does it start, where does it end, is it intact, and did it arrive?* The 9-byte framing envelope below answers all four for a 521-byte mode-0 packet, an overhead of about 1.7%.

Text-based alternatives also came up. A 64×64 binary mask is 4,096 bits of *packed* data, and a base-64-then-newline encoding more than doubles the wire size. At 1 Mbaud every byte costs 10 µs, so the 512-byte payload alone burns about 5 ms of wire time. Doubling that pushes the frame budget past what the 30 fps target can survive (Figure 2).

A mode-0 frame is $2 + 2 + 1 + 512 + 2 + 2 = 521$ bytes on the wire, which is about 5.21 ms at 1 Mbaud with 8N1 framing. Add a typical ACK round-trip of about 1 ms and the link is busy for roughly 6.2 ms per frame against a **33 ms budget at 30 fps**. The link is roughly **5× over-provisioned** for the bit rate the application needs. That headroom is what lets the link absorb the occasional NAK retransmit without dropping a frame the user actually sees. Mode-1 frames are essentially free on the wire (maximum 51 bytes of payload, total ∼60-byte packet), so the only real cost in gloving mode is the host's MediaPipe inference time.

### B. Reliability Layer: ACK/NAK

Once the ESP32 finishes parsing a frame, it writes back exactly one response byte: `0x06` (ACK) when the frame arrived and the CRC matched, or `0x15` (NAK) when the CRC didn't match or a sync error occurred. On the host side, `SerialSender._send_packet` waits up to 50 ms for the ACK and will retry up to three times before giving up on a frame. Frame-by-frame stats (`frames_sent`, `frames_acked`, `frames_naked`) are exposed back to the GUI and displayed in the status panel.

### C. Sideband Telemetry: ESP32 → Host

Three notification bytes share the same RX stream from the ESP32 back to the host:

- `0x10`—Mode just switched to mask mode
- `0x11`—Mode just switched to fingertip mode
- `0x20 <v>`—Smoothed pot value $v \in [0, 255]$, emitted at up to 20 Hz when the pot moves

Because all four sideband codes are disjoint from the start-sync byte `0xAA`, no escape character is needed: at byte 1 of any response the host can already tell which of the five things it is looking at.

### D. Error Recovery

The firmware handles three independent failure modes:

- **Single-bit corruption.** The CRC catches every odd-bit-count flip; the ESP32 responds with NAK and the host retries. The framebuffer is *not* zeroed on a NAK—the previous good frame stays on the panel, so transient noise reads as a brief freeze rather than a black flash.
- **Frame de-sync.** The `pollFrame()` state machine arms a 100 ms timeout as soon as it sees a leading `0xAA`. If the rest of the frame never shows up, the state resets to `RX_WAIT_AA`, and the next valid `0xAA 0x55` re-acquires sync, typically within 6 ms at 1 Mbaud.
- **Host crash / disconnect.** The ESP32 runs a 5-second watchdog on `lastFrameMs`: if no valid mask frame arrives in that window while the panel is in mask mode, the framebuffer is zeroed and the panel blanks.

## IV. Impressive Parts

The visually impressive part of this project is the panel. The engineering content is mostly in the transport, the threading, and the firmware control loop.

### A. Threaded GUI with a Real Back-Pressure Contract

The threading model above sounds obvious, but the subtle part is the back-pressure contract on the serial worker. PyQt's signal/slot connections enqueue into per-thread mailboxes, and those mailboxes are unbounded. If the GUI emits a 30 fps stream of `send_mask` requests and the worker's per-frame cost climbs past 33 ms (e.g., one NAK + one retry at 50 ms ACK timeout), the queue just grows. The fix is simple: the GUI never emits a new `send_mask` until it has seen `send_complete_signal` from the previous one. That pins the queue depth at 1 and bounds host-to-panel latency to one frame's worth of transport, no matter how lossy the link gets.

### B. ESP32 Receive State Machine

The receive path in `main.cpp` is a classic byte-driven state machine (`RX_WAIT_AA → RX_WAIT_55 → RX_LEN_HI → ... → RX_END_AA`), with two refinements that aren't obvious from the state list: sync recovery without dropping the next frame (when the state machine detects a CRC or sync error, it returns to `RX_WAIT_AA` and keeps scanning *the bytes already in the UART buffer*—no flush), and frame-timeout based on the first sync byte (a 100 ms timer fires and resets the state if no further bytes arrive after the leading `0xAA`).

### C. Pot Loopback for Closed-Loop GUI Preview

The ESP32 *sends its pot value back* to the host (`0x20 <byte>`), at up to 20 Hz. The host caches that value in `SerialSender.esp32_pot`, and the GUI's LED-preview viewport uses it to tint the on-screen $64 \times 64$ silhouette with the same white→red lerp the panel itself is showing. The hardware is the single source of truth here: the host doesn't *guess* what color the panel is, the panel *tells it*.

### D. Optional Gemini Scene Description

The dashboard's "AI SCENE ANALYSIS" button grabs the current camera frame, JPEG-encodes it, and POSTs the result to the Gemini 2.0 Flash multimodal endpoint. The response gets printed to the console—deliberately, because the GUI's aesthetic is minimalist and a popup would break the look. It's a tongue-in-cheek bridge between two kinds of "vision": a 6 MB YOLO model that emits a 4096-pixel mask, and a foundation model that

writes a paragraph of natural language about the same frame.

### *E. GUI Screen Capture*

A live capture of the running dashboard is included as `gui_screenshot.png` next to this report. The reference layout is also visible in `Steph/Screenshot 2026-05-07 212224.png`, the design-review screenshot from an earlier iteration of the GUI.

## V. Reflection and Version 2.0 Updates

With fifteen weeks of hindsight, almost every structural choice we would change sits on the software side. The hardware stack—YOLOv8 + ESP32 + HUB75 + pot + button—we would keep.

**Move the host code to C++ from day one.** We already have a working C++ port (`Larry/vision_fast.cpp`), and the delta between it and the Python version is framework, not algorithm. A native build also opens up Apple's Neural Engine via Core ML, which would push inference latency below 10 ms.

**Replace the bit-packed mask with a per-pixel palette index.** A 4-bit palette index per pixel—16 simultaneous on-panel colors—still fits in 2048 bytes per frame, comfortably under what 1 Mbaud sustains at 30 fps. One change, and we get gradient silhouettes, body-part coloring, and motion trails without touching the cable, the panel, or the framing geometry.

**Switch the firmware loop to FreeRTOS tasks.** Our current cooperative `loop()` works, but it makes adding anything to it fraught. A cleaner structure is four pinned tasks—panel render, serial RX, sensor poll, control—talking through ring buffers. The ESP32 has two Xtensa cores. We are using one.

**Replace USB CDC with ESP-NOW or UDP-over-WiFi.** USB tethering was the right call for the first build, but it ties the panel to a desktop. ESP-NOW between the host's USB-Wi-Fi dongle and the ESP32 would decouple the panel from the laptop entirely. Our frame structure already survives unreliable transport: the host's retry path and the firmware's watchdog were designed with that in mind.

**Capability handshake at boot.** Right now the host *assumes* the firmware supports modes 0 and 1, and *assumes* the panel is 64×64. A capability handshake at link-up—a `0x30` "hello" from the firmware reporting {firmware version, panel size, supported modes, max payload}—would let the host adapt automatically whenever we swap firmware revisions or, eventually, drive a different panel size. Two lines of code on each side. The payoff is that the host becomes generic over the display device, instead of hard-coded to one panel.

**Virtual ESP32 simulator over a `pty` pair.** By a wide margin, the biggest dev-velocity cost on this project was the host-firmware round-trip. Every protocol change required a re-flash; every protocol bug required a logic analyzer. A v2 would ship a Python "virtual ESP32" that speaks the wire format over a `pty` pair, validates framing and CRC, and renders the mask to a Pygame window. From the host's point of view, nothing would change. Most protocol work could then happen on a plane, with the laptop closed-loop against itself, and the physical panel becomes a confirmation step rather than a step in the build loop. The same harness unblocks a fuzz suite for the RX state machine—randomly corrupt byte streams, verify recovery—that we never wrote.

**Sliding-window ACKs instead of one-frame-in-flight.** Right now the GUI's back-pressure contract is: emit one mask → wait for `send_complete_signal` → emit the next. A 4-frame sliding window with selective NAK would amortize the round-trip cost across several in-flight frames. At our current frame budget the difference is invisible. At 60+ fps it would start to matter.

**One vision file, not three.** Right now we have three near-duplicates—`Larry/vision.py`, `Kyle/vision/vision.py`, and `Kyle/vision/vision_send.py`—all sharing roughly ∼80% of the same YOLO logic. The fix is to factor a shared `vision_core.py` out into one place and let both folders import from it.

**Continuous integration on the protocol tests.** `Kyle/vision/dashboard/tests/test_protocol.py` exists and passes locally, but nothing runs it on push. A two-line GitHub Action calling `pytest`, plus a one-line `pio check` on the firmware, would have caught at least one bit-packing regression we ended up finding by hand.

If there is one meta-lesson worth carrying forward, it is this: **freeze the wire format first, vary everything around it**. The frame layout written in week 5 has survived a pivot from MOG2 to YOLO, from WS2812B 108×108 to HUB75 64×64, from a Jetson host to a MacBook host, and from a single-mode mask protocol to a two-mode mask+fingertips protocol with sideband sensor telemetry. Everything else we built has been rewritten at least once. The byte schema has not.

# VI. Summary of Deliverables

The zip submission contains:

- `Kyle/vision/dashboard/`—PyQt6 GUI, serial worker, protocol tests
- `Kyle/vision/`—standalone vision script (`vision.py`), unified sender (`vision_send.py`), serial protocol (`serial_protocol.py`)
- `Kyle/firmware/POT_working/`—PlatformIO project, ESP32 firmware (`src/main.cpp`), platformio config, build README
- `Kyle/firmware/WIRING.md`—bring-up procedure, pin map, and a troubleshooting table
- `Larry/vision.py`, `Larry/vision_fast.cpp`, `Larry/led_board.cpp`, `Larry/main.cpp`—Python and C++ reference implementations of the CV system, plus an HUB75 hello-world for the panel
- `Steph/Project_GUI.py`, `Steph/Screenshot 2026-05-07 212224.png`—GUI design-review iteration and screen capture
- `yolov8n-seg.pt`—pretrained YOLOv8 nano segmentation checkpoint (∼6.7 MB)
- `gui_screenshot.png`—screen capture of the dashboard while running

This build has no LabVIEW component. Operator control runs entirely through the PyQt6 dashboard (§2), and the hostESP32 link is the framed binary protocol from §3.

# VII. Acknowledgments

**Larry Hui** did the hardware: the HUB75 panel assembly, the ESP32-to-panel ribbon, the power rail with bulk-capacitor decoupling, and the button and potentiometer wiring. He also wrote the original `vision.py` silhouette workflow the production sender is built on, the C++ port (`vision_fast.cpp`), and the standalone HUB75 hello-world (`led_board.cpp`) we used to verify panel bring-up. **Wen Cao** built the full MediaPipe fingertip system to make the gloving mode work and designed the mode-1 protocol payload. **Steph Akakabota** built the retro PyQt6 frontend the demo runs on. **Kyle Nelson** wrote the serial protocol, the ESP32 firmware, and the bring-up documentation (`WIRING.md`), and was responsible for integrating the four pieces. Finally, the project owes a lot to the open-source communities behind YOLOv8, MediaPipe, and the ESP32-HUB75-MatrixPanel-DMA library; without those three libraries we could not have shipped this in one semester.

# References

[1] G. Jocher, A. Chaurasia, and J. Qiu. *Ultralytics YOLOv8*, version 8.0.0, 2023. https://github.com/ultralytics/ultralytics
[2] C. Lugaresi *et al.* “MediaPipe: A Framework for Building Perception Pipelines.” *arXiv:1906.08172*, 2019.
[3] “ESP32-HUB75-MatrixPanel-I2S-DMA,” open-source PlatformIO library. https://github.com/mrfaptastic/ESP32-HUB75-MatrixPanel-DMA
[4] Waveshare Electronics. *RGB-Matrix-P2 64×64 wiki and datasheet.* https://www.waveshare.com/wiki/RGB-Matrix-P2-64x64
[5] G. Bradski. “The OpenCV Library.” *Dr. Dobb’s Journal of Software Tools*, 2000.
[6] Riverbank Computing. *PyQt6 Reference Guide.* https://www.riverbankcomputing.com/static/Docs/PyQt6/

# VIII. Appendix A—Code Listings

The code listings referenced in this report are included verbatim in the submission. Of those, the five excerpts below are the ones most worth reading in full.

**Note:** For the final submission, all code was consolidated into the `Kyle/` folder of the repository. This is a packaging choice, not an authorship claim—multiple contributors wrote the code found there. The `Name/` folder convention was individual experimentation space during the project.

## *A.1 CRC-16/CCITT-FALSE and Frame Builder (Python)*

From `Kyle/vision/serial_protocol.py`. The same CRC is mirrored on the firmware side; the two were validated against each other with a round-trip test before bring-up.

```
54 def crc16_ccitt(data: bytes, init: int = 0xFFFF) -> int:
55     """CRC-16/CCITT-FALSE: poly=0x1021, init=0xFFFF,
56        no reflect, no xorout."""
57     crc = init
58     for byte in data:
59         crc ^= byte << 8
60         for _ in range(8):
61             if crc & 0x8000:
62                 crc = (crc << 1) ^ 0x1021
63             else:
64                 crc <<= 1
65             crc &= 0xFFFF
66     return crc
67
68
69 def build_frame(mode: int, payload: bytes) -> bytes:
70     """Wrap payload in the full framed packet."""
71     body = bytes([mode]) + payload
72     crc = crc16_ccitt(body)
73     length = len(payload)
```

```
74      return (FRAME_START + struct.pack(">H",
            length) + body + struct.pack(">H",
            crc) + FRAME_END)
```

Listing 1: A.1: CRC-16/CCITT-FALSE and frame builder (Kyle/vision/serial_protocol.py)

*A.2 Bit-Packed Mask Encoder (Python)*

From `Kyle/vision/serial_protocol.py`. Reduces a 64×64 `uint8` mask to 512 bytes MSB-first row-major in one `numpy.packbits` call.

```
68  def pack_mask(mask: np.ndarray) -> bytes:
69      """Pack a (64, 64) {0,1} or {0,255}
            mask
70         into 512 bytes, MSB-first row-major.
               """
71      if mask.shape != (PANEL_SIZE,
            PANEL_SIZE):
72          raise ValueError(
73              f"Expected ({PANEL_SIZE}, {
                    PANEL_SIZE}), "
74              f"got {mask.shape}")
75      arr = np.ascontiguousarray(mask, dtype=
            np.uint8)
76      if arr.max() > 1:
77          arr = (arr > 0).astype(np.uint8)
78      packed = np.packbits(arr.flatten(),
79                           bitorder="big").
                                 tobytes()
80      if len(packed) != PAYLOAD_BYTES:
81          raise ValueError(
82              f"Pack produced {len(packed)}
                    bytes, "
83              f"expected {PAYLOAD_BYTES}")
84      return packed
```

Listing 2: A.2: Bit-packed mask encoder (Kyle/vision/serial_protocol.py)

*A.3 ESP32 RX State Machine (C++)*

From `Kyle/firmware/POT_working/src/main.cpp`. The state machine is the firmware's only structural concurrency point; render and input scan run between RX byte arrivals.

```
139 // States: WAIT_START0, WAIT_START1,
        READ_LEN_H, READ_LEN_L,
140 //         READ_MODE, READ_PAYLOAD,
        READ_CRC_H, READ_CRC_L,
141 //         READ_END0, READ_END1
142 while (Serial.available()) {
143     uint8_t b = Serial.read();
144     switch (rx_state) {
145         case WAIT_START0:
146             if (b == 0xAA) rx_state =
                    WAIT_START1; break;
147         case WAIT_START1:
148             rx_state = (b == 0x55) ?
                    READ_LEN_H : WAIT_START0;
149             break;
150         case READ_LEN_H:
151             payload_len = (uint16_t)b << 8;
152             rx_state = READ_LEN_L; break;
153         case READ_LEN_L:
154             payload_len |= b;
155             rx_state = READ_MODE; break;
156         case READ_MODE:
157             rx_mode = b;
158             crc_acc = crc16_step(0xFFFF, b)
                    ;
159             payload_idx = 0;
160             rx_state = (payload_len > 0) ?
161                         READ_PAYLOAD :
                                READ_CRC_H;
162             break;
163         case READ_PAYLOAD:
164             rxbuf[payload_idx++] = b;
165             crc_acc = crc16_step(crc_acc, b
                    );
166             if (payload_idx >= payload_len)
167                 rx_state = READ_CRC_H;
168             break;
169         case READ_CRC_H:
170             frame_crc = (uint16_t)b << 8;
171             rx_state = READ_CRC_L; break;
172         case READ_CRC_L:
173             frame_crc |= b;
174             rx_state = READ_END0; break;
175         case READ_END0:
176             rx_state = (b == 0x55) ?
                    READ_END1 : WAIT_START0;
177             break;
178         case READ_END1:
179             if (b == 0xAA && frame_crc ==
                    crc_acc) {
180                 memcpy(framebuf, rxbuf,
                        payload_len);
181                 Serial.write(0x06);   //
                        ACK
182                 last_frame_ms = millis();
183             } else {
184                 Serial.write(0x15);   //
                        NAK
185             }
186             rx_state = WAIT_START0;
187             break;
188     }
189 }
```

Listing 3: A.3: ESP32 RX state machine (Kyle/firmware/POT_working/src/main.cpp)

*A.4 Pot Smoothing and Host Mirror (C++)*

From `Kyle/firmware/POT_working/src/main.cpp`. EWMA smoothing on the potentiometer, with a notification sent only on quantization-step changes so the reverse channel does not flood.

```
334 // Per-loop pot scan
335 int raw = analogRead(POT_PIN);
336 float norm = raw / 4095.0f;
337 pot_ema = 0.1f * norm + 0.9f * pot_ema;
            // alpha = 0.1
338 uint8_t quantized = (uint8_t)(pot_ema *
        255.0f);
```

```
339
340 if (quantized != last_quantized) {
341     last_quantized = quantized;
342     Serial.write(0x20); ↩
                                   // pot-notify
343     Serial.write(quantized);
344 }
```

Listing 4: A.4: Pot EWMA smoothing and loopback notify (Kyle/firmware/POT_working/src/main.cpp)

### A.5 Qt-Threaded Serial Worker (Python)

The pattern that keeps the GUI thread free of blocking I/O: a `QObject` subclass moved onto a `QThread` and accessed exclusively via signals.

```
197 class SerialWorker(QObject):
198     frame_sent = pyqtSignal(bool)          #↩
         success
199     esp32_notification = pyqtSignal(int)   #↩
         mode / pot
200
201     def __init__(self, port: str):
202         super().__init__()
203         self._sender = SerialSender(port=↩
             port)
204
205     @pyqtSlot(bytes, int)
206     def send_frame(self, payload: bytes, ↩
         mode: int):
207         ok = self._sender.send(mode, ↩
             payload)
208         self.frame_sent.emit(ok)
209         for note in self._sender.↩
             drain_notifications():
210             self.esp32_notification.emit(↩
                 note)
211
212
213 # Wiring in the GUI:
214 worker = SerialWorker(port="/dev/cu.↩
     usbserial-XXXX")
215 thread = QThread()
216 worker.moveToThread(thread)
217 thread.start()
218 # self._send_frame_signal.emit(payload, ↩
     mode)
```

Listing 5: A.5: Qt-threaded serial worker (Kyle/vision/dashboard/serial_worker.py)