ShrimpHub Challenge Final Report

System Design, Implementation & Engineering Evidence

Team Name: Desyn Studio
Team Leader Name: Tamal Krishna Chhabra

January 11, 2026

Contents

1 Task 1 — Timing Keeper 3
1.1 Overview . . . 3
1.2 Requirements & Success Criteria . . . 3
1.3 Hardware & Wiring . . . 3
1.4 System Architecture . . . 3
1.5 Engineering Highlights: Optimization & Robustness . . . 4
1.5.1 Real-time (RTOS) Architecture . . . 4
1.5.2 Optimization . . . 4
1.5.3 Resource Management . . . 4
1.5.4 Error Handling . . . 4

2 Task 2 — Priority Guardian 5
2.1 Overview . . . 5
2.2 Requirements & Success Criteria . . . 5
2.3 System Architecture & Task Priorities . . . 5
2.4 Engineering Highlights: Optimization & Robustness . . . 5
2.4.1 Real-time Architecture & Innovation . . . 5
2.4.2 Optimization . . . 5
2.4.3 Resource Management . . . 6
2.4.4 Function & API-level Explanations . . . 6
2.5 Logging & Evidence Format . . . 6

3 Task 3 — Window Synchronizer 7
3.1 Overview . . . 7
3.2 Requirements & Success Criteria . . . 7
3.3 System Architecture . . . 7
3.4 Engineering Highlights: Optimization & Robustness . . . 7
3.4.1 Real-time Architecture (Super Loop) . . . 7
3.4.2 Optimization . . . 7
3.4.3 Error Handling . . . 7
3.5 Timing & Debouncing Strategy . . . 8

4 Task 4 — The Silent Image (Steganography Decoder) 9
4.1 Overview . . . 9
4.2 Requirements & Success Criteria . . . 9
4.3 System Architecture . . . 9
4.4 Engineering Highlights: Optimization & Robustness . . . 9
4.4.1 Innovation: Hybrid Compute Architecture . . . 9
4.4.2 Optimization . . . 9
4.4.3 Robustness & Error Handling . . . 9
4.5 Steganography Analysis Strategy . . . 10

1 Task 1 — Timing Keeper

1.1 Overview

Objective: Read timing arrays for Red/Green/Blue channels from MQTT topic `shrimphub/led/timin`
and reproduce the illumination pattern on an RGB LED with strict timing accuracy (±5 ms), no drift over 5 minutes. Provide reproducible artifacts and video proof with stopwatch and serial logs.

1.2 Requirements & Success Criteria

- **Subscribe** to `shrimphub/led/timing/set`.
- **Parse JSON payload** containing integer arrays for red, green, and blue.
- **Implement on/off cycles** with durations in milliseconds using strict timing.
- **Repeat pattern continuously** until replaced by a new array.
- **Program must not drift** over 5 minutes.
- **Provide video**: stopwatch visible + serial log + LED visible.

1.3 Hardware & Wiring

Components: ESP32 DevKit V1, RGB LED, OLED Display (SSD1306, I2C).
Wiring:

- **RED_LED**: GPIO 25
- **GREEN_LED**: GPIO 26
- **BLUE_LED**: GPIO 27
- **SDA/SCL**: GPIO 21 / 22

Note: The code logic assumes Active-Low LEDs (LOW = ON, HIGH = OFF).

1.4 System Architecture

The solution uses **FreeRTOS** to separate network handling, display updates, and precise LED timing.

1. **MQTT Task (Network):** Maintains connection and handles incoming messages.
2. **Display Task (Low Priority):** Updates the OLED with status.
3. **Three LED Tasks (High Priority):** Independent tasks for Red, Green, and Blue channels using `vTaskDelayUntil` for drift-free precision.
4. **Semaphores:** Mutexes protect access to shared timing arrays.

1.5 Engineering Highlights: Optimization & Robustness

1.5.1 Real-time (RTOS) Architecture

The system employs a **Preemptive Multitasking** model. The LED tasks are assigned higher priorities (3) than the Display task (2) and System/Network tasks. This ensures that even if the Display task takes 50ms to render the OLED, or the MQTT task is processing a packet, the LED toggle will occur exactly on the millisecond tick.

Innovation: We utilize **Task Notifications** (`xTaskNotifyGive`) instead of heavy binary semaphores for signaling new patterns, reducing context switch overhead.

1.5.2 Optimization

- **Drift Elimination:** We strictly use `vTaskDelayUntil` rather than `vTaskDelay`. The former calculates the next wake time based on the *scheduled* previous wake time, not the actual wake time. This mathematically eliminates cumulative error (drift).
- **Local Buffering:** Each LED task copies the global timing array into a local stack-allocated array to minimize mutex hold time to microseconds.

1.5.3 Resource Management

- **Static JSON Allocation:** We use `StaticJsonDocument<2048>` instead of Dynamic to prevent heap fragmentation.
- **Fixed Stack Sizes:** Tasks are allocated 4096 bytes of stack, tuned to accommodate `printf` formatting without wasting RAM.

1.5.4 Error Handling

- **Mutex Protection:** All shared data access is guarded by `xSemaphoreTake`.
- **Network Resilience:** The `MQTTTask` runs an infinite loop with connection checking. If WiFi drops, the LED tasks **continue running** the current pattern uninterrupted.

2 Task 2 — Priority Guardian

2.1 Overview

Objective: Maintain a continuous low-priority rolling-average calculation on numbers published to stream while immediately handling high-priority "CHALLENGE" messages. When a CHALLENGE arrives, the device must publish an ACK to `<ReefID>` within 250 ms and visually signal via LED.

2.2 Requirements & Success Criteria

- **Subscribe** to `krillparadise/data/stream` (background).
- **Subscribe** to `<TeamID>` (distress).
- **Latency:** ACK must be sent $\leq$ 250 ms.
- **Concurrency:** Rolling average continues without blocking the Distress response.

2.3 System Architecture & Task Priorities

The system uses a **Priority-Based Preemptive Scheduling** model with FreeRTOS:

1. **Priority 3 (Highest) - taskDistress:** Unblocks on signal, preempts everything, handles ACK/LED immediately.
2. **Priority 2 (Medium) - taskDispatcher:** Routes incoming messages to queues; keeps network alive.
3. **Priority 1 (Lowest) - taskBackground:** Computes rolling averages only when high-priority tasks are idle.

2.4 Engineering Highlights: Optimization & Robustness

2.4.1 Real-time Architecture & Innovation

We implemented a **"Three-Tier Priority Guardian"** pattern.

- **Decoupling:** The MQTT Dispatcher (Tier 2) acts as a high-throughput router, doing almost zero work.
- **Preemption:** The Distress Handler (Tier 3) consumes 0% CPU until a challenge arrives, at which point it instantaneously preempts the system.

2.4.2 Optimization

- **Queue IPC:** We use FreeRTOS Queues as buffers. The background task processes data at its own pace (Backpressure) while the network task fills the queue.
- **Memory Efficiency:** We use `snprintf` with pre-allocated buffers instead of `String` concatenation.

2.4.3 Resource Management

Backpressure Handling: The background queue has a finite length. If the network floods, we use a timeout of 0ms on the send, effectively implementing *Load Shedding* to prevent OOM crashes.

2.4.4 Function & API-level Explanations

- `mqttCallback`: Minimalist design. Copies payload → Pushes to Queue. Keeps network unblocked.
- `taskDistress`: Critical path. Wakes on queue signal, toggles LED, captures `millis()`, publishes ACK, logs delta.

2.5 Logging & Evidence Format

```
[Background] msg=10.000 count=1 avg=10.00 time_ms=5000
[Dispatcher] Distress received at ms=5500 payload=CHALLENGE
[Distress] recv_ms=5500 ack_attempt_ms=5505 delta=5ms publish=OK
```

3 Task 3 — Window Synchronizer

3.1 Overview

Objective: Detect a short "window open" event broadcast by the broker, and physically press a hardware push-button during that window with ±50 ms tolerance. Implement debouncing and reliably publish synchronization status.

3.2 Requirements & Success Criteria

- Listen for `{"status":"open"}` on `<window_code>`.
- Detect physical button press.
- **Sync Criterion:** `abs(press_time - window_open_time) <= 50ms`.
- **Debounce:** $\geq$ 20ms.

3.3 System Architecture

The implementation uses a **Super-Loop Polling Architecture** (Single Threaded). Unlike complex multi-tasking solutions, this approach minimizes overhead to ensure the loop runs fast enough (typically <1ms per iteration) to capture button presses accurately.

3.4 Engineering Highlights: Optimization & Robustness

3.4.1 Real-time Architecture (Super Loop)

While Tasks 1 and 2 use FreeRTOS, Task 3 intentionally uses a **Bare Metal Super Loop**.

- **Zero Overhead:** Context switching takes microseconds. Polling a GPIO register takes nanoseconds.
- **Rationale:** Detecting a button press within a millisecond window is best done by checking the pin as frequently as possible.

3.4.2 Optimization

- **Non-blocking Timers:** Strictly use `millis()` checks. Zero `delay()` calls in the main loop ensuring the processor is always available.
- **Immediate Interrupt Logic:** The `mqtt_callback` sets `windowOpen = true` immediately, bringing the software-defined "Window Start" as close to packet arrival as possible.

3.4.3 Error Handling

- **State Cleanup:** Timeout logic ensures the system self-heals (resets to IDLE) if a window opens but no button is pressed.
- **JSON Safety:** Malformed packets are safely ignored using error code checks.

3.5 Timing & Debouncing Strategy

- **Clock:** `millis()` is used for all timestamps.
- **Debouncing:** Button changes are ignored if `millis() - lastDebounceTime < 20`.

4 Task 4 — The Silent Image (Steganography Decoder)

4.1 Overview

Objective: Request a hidden artifact via MQTT, reconstruct the received 64x64 PNG, and use steganographic analysis to extract a hidden message based on pixel relationships.

4.2 Requirements & Success Criteria

- **Request:** Publish `{"request":"...", "agent_id":"..."}`.
- **Receive:** Base64 PNG.
- **Reconstruct:** Save as valid PNG.
- **Crack:** Find the hidden URL inside the image.

4.3 System Architecture

The solution uses a hybrid approach:

- **ESP32 (Network Bridge):** Handles the MQTT request/response cycle.
- **Python (Compute Node):** Performs the heavy image decoding and steganographic analysis (`task4_phase3-4_analyze.py`).

4.4 Engineering Highlights: Optimization & Robustness

4.4.1 Innovation: Hybrid Compute Architecture

We recognized that breaking LSB or relational steganography is computationally expensive and requires complex libraries (`PIL`, `numpy`) unavailable on microcontrollers. **Innovation:** We treat the ESP32 strictly as an IoT Interface, while offloading the "Brain" work to a host machine. This mirrors real-world Edge-to-Cloud architectures.

4.4.2 Optimization

- **Vectorized Analysis:** The Python script uses `numpy` arrays to perform pixel operations (whole-matrix operations) instead of iterating pixels, optimizing the search space by orders of magnitude.
- **Base64 Buffer Management:** On the ESP32, the MQTT buffer is increased to 2048 bytes to handle large chunks.

4.4.3 Robustness & Error Handling

- **Image Integrity:** Python validates the PNG signature before analysis.
- **Multi-Method Brute Force:** The script iteratively tries LSB, Channel Difference, and Ratio methods, catching exceptions and proceeding automatically.

4.5 Steganography Analysis Strategy

The pipeline checks multiple hiding schemes:

1. **LSB Analysis:** Checking specific bit planes (0-2).
2. **Channel Ratios:** Comparing Red vs Green vs Blue intensities.
3. **XOR/Difference Maps:** Visualizing differences between adjacent pixels.