PAROL6 Firmware Migration Strategy - v2 (2026 Update)

Goal Description

The PAROL6 robot is migrating its low-level hardware controller from an ESP32 (serial relay) to a **Teensy 4.1** acting as a deterministic, real-time servo controller.

This document outlines the professional-grade architectural roadmap to achieve a **1 kHz deterministic control loop** with high-resolution QuadTimer encoder feedback, while strictly preserving the existing ROS 2 `parol6_hardware` interface, serial commands, and thesis validation metrics.

This is a targeted migration that moves intelligence *down the stack*—from ROS-driven logic to embedded-controlled logic—enabling high-bandwidth, low-latency trajectory execution suitable for welding and precision robotic tasks.

The architecture prioritizes temporal determinism over algorithmic complexity, ensuring all control paths have bounded execution time suitable for real-time guarantees. The architecture remains compatible with future fixed-point implementations if floating-point determinism becomes a constraint.

🛑 Key Architectural Locks & Decisions

Non-Goals (Thesis Scope Limits):

- Model-predictive control (MPC)
- Full rigid-body dynamics computation
- Multi-axis torque control/compliance

Before implementation begins, we explicitly lock the following design decisions:

1. **Interpolation Strategy**: Locked to **Linear position + velocity feedforward**.
    - *Why*: At a 1 kHz inner loop (effectively 40x upsampling from a 25 Hz ROS command), linear interpolation is deterministic, predictable, and exceptionally smooth. Complex models like Cubic Splines are deferred as post-thesis future work.
2. **Control Law**: Locked to **Velocity Command = Feedforward Velocity + Kp * Position Error**.
    - *Why*: Simple, proven, and computationally cheap for a 1 kHz loop. Prevents scope creep into complex model-predictive territory.
3. **Velocity Estimation**: Locked to an **Alpha-Beta Observer (Filter)**.
    - *Why*: Welding paths demand tangential smoothness, not just raw bandwidth. An alpha-beta filter is a minimal-cost 2-state observer that drastically outperforms simple IIR smoothing and negates the need for a complex (and non-deterministic) Kalman filter.
4. **RTOS Usage**: Locked to an **Optional Background Model**.
    - *Why*: An RTOS (e.g., FreeRTOS) may be used for the transport layer, logging, or diagnostics. However, the RTOS is optional and never involved in the servo loop timing chain. The core 1 kHz servo loop **must** remain driven by a strict Hardware Timer ISR.
5. **Transport Protocol**: Locked to **Current ASCII Protocol** (`<SEQ...>`/`<ACK...>`).

6. **Tooling Ecosystem**: Locked to **PlatformIO + Docker (`teensy_loader_cli`)**.
    - *Why*: Arduino IDE is non-reproducible. ESP-IDF Makefiles lose the Teensy core USB stack. PlatformIO executing inside the `parol6-ultimate` Docker image guarantees an identical compiler toolchain for every developer, perfectly mirroring the established ESP32 build flow via a one-click `flash_teensy.sh` script.

Performance Targets (KPIs)

These are the strict engineering metrics the new firmware must hit to validate the thesis:

- **Inner Control Loop Rate**: 1 kHz (1 ms period)
- **Control Jitter**: < 50 µs
- **Encoder Noise**: < 0.05°
- **End-to-End Latency**: < 15 ms (command received to motor action)
- **Packet Loss**: 0 packets (validated via the `SEQ` metric logging)
- *Future Capability*: 100 Hz ROS command rate over 480 Mbps USB High-Speed.

Proposed Changes: Architectural Layers

Each architectural layer is designed for isolated validation using the ESP32 PWM simulator, enabling hardware-in-the-loop testing without mechanical dependencies. The Alpha-Beta observer operates entirely in the signal processing layer, independent of the control ISR scheduling, ensuring estimator upgrades do not perturb control timing.

1. Transport & Application Layer (RTOS Safe)

- **Action**: Isolate serial parsing. The parser runs in the main loop (or an RTOS task) and pushes target ROS waypoints into a **lock-free ISR-safe queue**.
- **Future**: This layer is cleanly swapped for USB CDC later without touching the control math.

Implementation Example:

```
// SerialTransport.h
#include <CircularBuffer.h> // Common Arduino/Teensy ISR-safe buffer

struct RosCommand {
    uint32_t seq;
    float positions[6];
    float velocities[6];
    uint32_t timestamp_us;
};

class SerialTransport {
public:
    void init(uint32_t baud) {
        Serial.begin(baud);
    }
```

```
    // Called in main loop() or RTOS Task (Non-blocking)
    void process_incoming(CircularBuffer<RosCommand, 10>& cmd_queue) {
        while (Serial.available()) {
            char c = Serial.read();
            if (c == '\n' || c == '\r') {
                if (rx_pos_ > 0) {
                    rx_buf_[rx_pos_] = '\0';
                    RosCommand new_cmd;
                    if (parse_string(rx_buf_, new_cmd)) {
                        cmd_queue.push(new_cmd); // Push to ISR-safe queue
                    }
                    rx_pos_ = 0;
                }
            } else if (rx_pos_ < MAX_BUF - 1) {
                rx_buf_[rx_pos_++] = c;
            }
        }
    }

    void send_feedback(uint32_t seq, const float current_pos[6]) {
        // Formats <ACK, seq, p1, p2...> and Serial.write()s it
    }

private:
    static const int MAX_BUF = 128;
    char rx_buf_[MAX_BUF];
    int rx_pos_ = 0;

    bool parse_string(const char* str, RosCommand& cmd) {
        // Implementation of strtok-based parse for <SEQ, J1...>
        return true;
    }
};
```

2. Supervisor / Safety Layer (NEW)

- **Action**: Introduce a safety state machine containing command timeouts, encoder anomaly detection, and motor runaway stops.

Implementation Example:

```
// Supervisor.h
class SafetySupervisor {
public:
    enum State { INIT, NOMINAL, SOFT_ESTOP, FAULT };

    void update(uint32_t current_time_ms, const float actual_velocities[6])
{
        // 1. Check Watchdog (Command Timeout)
        if (current_time_ms - last_cmd_time_ms_ > COMMAND_TIMEOUT_MS) {
            trigger_fault(SOFT_ESTOP, "Command Timeout: No ROS data");
```

```
        }

        // 2. Check Kinematic Limits (Runaway Velocity)
        // Velocity thresholds evaluated using observer-estimated velocity
        for (int i = 0; i < 6; i++) {
            if (abs(actual_velocities[i]) > MAX_SAFE_VELOCITY_RAD_S) {
                trigger_fault(FAULT, "Runaway Velocity Detected");
            }
        }
    }

    void feed_watchdog(uint32_t time_ms) { last_cmd_time_ms_ = time_ms; }
    bool is_safe() const { return current_state_ == NOMINAL; }
    State get_state() const { return current_state_; }

private:
    State current_state_ = NOMINAL;
    uint32_t last_cmd_time_ms_ = 0;
    const uint32_t COMMAND_TIMEOUT_MS = 100; // ROS sends at ~40ms
    const float MAX_SAFE_VELOCITY_RAD_S = 10.0f;

    void trigger_fault(State state, const char* reason) {
        if (current_state_ != FAULT) { // Don't override hard faults
            current_state_ = state;
            // Log over serial/debug port immediately
        }
    }
};
```

3. Trajectory Interpolation Layer

- **Action**: Interpolate between the 25 Hz ROS points and the 1 kHz servo tick using deterministic linear math. Interpolation duration is derived from the inter-arrival time of ROS waypoints to maintain temporal consistency.

Implementation Example:

```
// Interpolator.h
class LinearInterpolator {
public:
    // Called when a new ROS command is popped from the queue
    void set_target(float target_pos, float target_vel, uint32_t
current_time_ms, uint32_t expected_duration_ms) {
        start_pos_ = current_pos_;
        target_pos_ = target_pos;

        // Dynamically compute step delta based on inter-arrival time
expectation
        step_delta_ = (target_pos_ - start_pos_) /
(float)expected_duration_ms;
```

```
        feedforward_vel_ = target_vel;
        steps_remaining_ = expected_duration_ms;
    }

    // Called precisely every 1ms inside the Control ISR
    void tick_1ms(float& out_pos_cmd, float& out_vel_ff) {
        if (steps_remaining_ > 0) {
            current_pos_ += step_delta_;
            steps_remaining_--;
        } else {
            // Reached target, hold position, zero feedforward velocity
            current_pos_ = target_pos_;
            feedforward_vel_ = 0.0f;
        }

        out_pos_cmd = current_pos_;
        out_vel_ff = feedforward_vel_;
    }

private:
    float current_pos_ = 0.0f;
    float target_pos_ = 0.0f;
    float step_delta_ = 0.0f;
    float feedforward_vel_ = 0.0f;
    uint32_t steps_remaining_ = 0;
};
```

4 & 5. Signal Processing & Control Layer (The 1 kHz Servo Loop)

- **Action**: A dedicated Alpha-Beta observer for smooth velocity estimation, feeding a strict 1 kHz Timer ISR executing the locked `P + Feedforward` control law. The ISR contains no dynamic memory allocation, logging, or string-based branching to guarantee bounded execution time.

Implementation Example:

```
// AlphaBetaFilter.h
class AlphaBetaFilter {
public:
    AlphaBetaFilter(float alpha, float beta, float dt)
        : alpha_(alpha), beta_(beta), dt_(dt) {}

    // Called at 1kHz. Unwraps raw absolute encoder angles and filters.
    void update(float raw_encoder_angle_rad) {
        // 1. Handle MT6816 Multi-turn wrapping (e.g. crossing 2PI
boundary)
        float delta = raw_encoder_angle_rad - last_raw_angle_;
        if (delta > PI)  turn_offset_ -= 2.0f * PI;
        if (delta < -PI) turn_offset_ += 2.0f * PI;
        last_raw_angle_ = raw_encoder_angle_rad;

        float unwrapped_measurement = raw_encoder_angle_rad + turn_offset_;
```

```
        // 2. Observer Prediction Step
        estimated_pos_ += estimated_vel_ * dt_;

        // 3. Innovation (Measurement Error)
        float residual = unwrapped_measurement - estimated_pos_;

        // 4. Observer Update Step
        estimated_pos_ += alpha_ * residual;
        estimated_vel_ += (beta_ / dt_) * residual;
    }

    float get_position() const { return estimated_pos_; }
    float get_velocity() const { return estimated_vel_; }

private:
    float estimated_pos_ = 0.0f;
    float estimated_vel_ = 0.0f;
    float last_raw_angle_ = 0.0f;
    float turn_offset_ = 0.0f;
    float alpha_, beta_, dt_;
};

// Main.cpp (Hardware Timer ISR Context)
// Rule: NO dynamic memory allocation, NO string generation, NO serial
printing inside ISR.
void run_control_loop_isr() {
    float current_velocities[6]; // Used to feed supervisor

    for (int i = 0; i < 6; i++) {
        // 1. Read Hardware Abstraction Layer
        float raw_pos = encoder_hal[i].read_angle();

        // 2. Update Observer
        observer[i].update(raw_pos);
        float actual_pos = observer[i].get_position();
        float actual_vel = observer[i].get_velocity();
        current_velocities[i] = actual_vel;

        // 3. Get 1ms Interpolated Setpoint
        float cmd_pos, cmd_vel_ff;
        interpolator[i].tick_1ms(cmd_pos, cmd_vel_ff);

        // 4. Strict Control Law (P + FF)
        float pos_error = cmd_pos - actual_pos;
        float velocity_command = cmd_vel_ff + (Kp[i] * pos_error);

        // 5. Motor Hal Output
        if (supervisor.get_state() == SafetySupervisor::NOMINAL) {
            motor_hal[i].set_velocity(velocity_command);
        } else {
            motor_hal[i].set_velocity(0.0f); // Fast stop on fault
        }
    }
```

```
    // Update supervisor with highest priority state
    supervisor.update(millis(), current_velocities);
}
```

6. Hardware Abstraction Layer (HAL)

- **Action**: Strict boundaries separating the `QuadTimer` capture logic and `FlexPWM` step generation from the generic signal and control math.

Implementation Example:

```
// EncoderHAL.h
// Pure virtual interface allowing macros() simulation or QuadTimer actuals
class EncoderHAL {
public:
    virtual void init() = 0;
    virtual float read_angle() = 0; // Returns raw radians [0, 2PI)
};

// QuadTimerEncoder.h (Phase 3 Target)
class QuadTimerEncoder : public EncoderHAL {
public:
    void init() override {
        // Complex i.MXRT1062 register configuration for edge capture
        // e.g. TMR1_CTRL0 = ...
    }

    float read_angle() override {
        // Read captured high/low ticks directly from hardware registers
        // Zero CPU overhead.
        uint32_t high_ticks = TMR1_CAPT0;
        uint32_t period_ticks = TMR1_CAPT1;
        float duty = (float)high_ticks / (float)period_ticks;
        return duty * 2.0f * PI;
    }
};
```

Phased Migration Plan (Optimized for Risk Reduction)

Phase 1 — Refactor Architecture (Current Priority)

- Break the existing `realtime_servo_teensy.ino` prototype into defined modules (Transport, Supervisor, Interpolation, Control, HAL).
- Maintain the existing `micros()` encoder logic temporarily.

Phase 1.5 — ESP32 Hardware PWM Injection (Accelerated)

- Implement `PwmCaptureEncoder.h` using Teensy `attachInterrupt()` to measure PWM pulse widths.
- Assign 6 physical digital input pins on the Teensy.
- Develop an ESP32 firmware utilizing the LEDC peripheral to generate 6 synthetic, noisy PWM signals representing robotic joint movement.
- Wire the 6 ESP32 PWM outputs to the 6 Teensy interrupt inputs.
- *Validation Required*: Hook an oscilloscope to `ISR_PROFILER_PIN` and prove that the 1 kHz ISR timing remains strictly bounded (< 15 µs) even while the CPU is bombarded with 6 simultaneous external hardware interrupts. This proves the Alpha-Beta filter can handle real hardware edge-capture latency without starving the control loop.

Phase 2 — Increase Loop to 1 kHz

- Bump the `IntervalTimer` to 1000 Hz.
- Implement the linear interpolation and Alpha-Beta observer.
- Validate the control logic runs at 1 kHz without CPU starvation using the ESP32 synthetic PWM simulator.

Phase 3 — QuadTimer Encoder HAL

- Implement the i.MXRT1062 specific `QuadTimer` capture logic.
- Establish strict Pin Zoning Architecture to prevent QuadTimer/FlexPWM XBAR collisions as the system scales to include E-stops, limits, and step generation.
- *Validation Required*: Explicitly validate the MT6816 PWM frequency mode and duty encoding scheme (including stability across temperature, as magnetic encoders drift) via oscilloscope/simulator before merging.
- Swap out the `micros()` HAL for the `QuadTimer` HAL.

Phase 4 — STEP/DIR Hardware Integration & Safety (Current)

- **3-Stage FlexPWM Rollout**:
  1. **Stage 1 - Dummy Carrier**: Generate a fixed 50 kHz PWM on one pin. Measure ISR jitter. Verify `< 1 µs` deviation holds.
  2. **Stage 2 - Velocity-Controlled Axis**: Implement `velocity_command -> pulse frequency` for ONE motor. Validate direction flip timing (~2 µs guard).
  3. **Stage 3 - Multi-Axis Scheduler**: Synchronize 6-axis generation and shared time bases.
- **MotorHAL Architecture**: Implement strict layer isolation:
  - `ActuatorModel` (Math): Gear ratios (`rad/s -> steps/s`).
  - `StepScheduler` (Timing): Phase accumulation and frequency register mapping.
  - `FlexPWMDriver` (Hardware Layer): Pin muxing, dead-time, and raw IP bus register writes.
- Integrate the legacy Limit Switch map (Zone 4) discovered from the open-loop firmware (`LIMIT1` to `LIMIT6` with explicit LOW/HIGH trigger polarities for active inductive sensors).
- **Homing & Calibration Layer**: Implement the rigorous 2-stage limit sweep (Fast Seek → Hit → Backoff → Slow Seek) as a distinct architectural layer that establishes the URDF zero-frame before the `ControlLaw` is permitted to engage. This layer enforces the *coupled mechanical dependencies* found in legacy `main.cpp` (e.g., J6 must be moved out of the way before J5 can safely home without a physical collision).

- Formally freeze the ASCII Serial Protocol specification (`<SEQ,pos,vel...>`) including decimal precision and field order. The Transport Layer assumes a byte-stream agnostic interface supporting UART, USB CDC, or bulk endpoints to prevent ROS ↔ MCU drift.

Phase 5 — USB Transport Evolution

- Migrate the ROS 2 host to use high-speed libusb.
- Replace the UART `Serial.read()` backend with the Teensy 480 Mbps PHY, increasing the ROS control and feedback rate to 100 Hz.

The Role of the ESP32 Simulator

The ESP32 PWM Simulator is not just a stepping stone; it is a **long-term strategic asset**. It will be maintained as a regression/CI tool to pump synthetic, perfectly known PWM trajectories into the Teensy. This allows us to quantify the exact phase lag and jitter of the Alpha-Beta filter mathematically, without physical motor dynamics skewing the data.

Future Work & Research Extensions (Post-Thesis Roadmap)

To extend the platform beyond the thesis baseline and elevate it to an industrial-grade robotics platform, the following extensions are explicitly identified as future work:

Control & Motion Stability

- **Embedded Motion Profiling**: Implementation of S-curve planners (jerk limiting) inside the trajectory interpolation layer to reduce torch vibration and smooth cornering behaviors.
- **Time-Optimal Splines**: Transitioning from linear to cubic Hermite splines or jerk-limited profiles as processing margins allow.
- **Adaptive Gain Scheduling**: Adjusting velocity feedforward and proportional gains continuously based upon the payload or specific joint inertia (e.g., J1 gearbox vs J5 direct transmission).

Estimation & Sensing

- **Encoder Linearity Calibration**: Construction of an embedded Look-Up Table (LUT) mapping intrinsic Magnetic Encoder Non-Linearity (INL) and applying runtime sine/cosine corrections for flawless kinematic fidelity.
- **Multi-Sensor Fusion Architecture**: Providing a structured observer graph to accept not just encoder readings, but also external tracker feeds, vision corrections, or force-torque metrics.

Real-Time Infrastructure

- **Host ↔ MCU Clock Synchronization**: Implementing a ping-pong timestamping or "PTP-lite" protocol to synchronize the ROS PC and Teensy 4.1 clocks. This allows for timestamped joint states (`<ACK,seq,p,v,timestamp_us>`) and rigorous latency/drift analysis.
- **Deterministic Telemetry Logging**: A specialized mode that logs timestamped observer states at the absolute 1 kHz control tick (bypassing USB transport jitter) to provide immaculate datasets for thesis evaluation.

- **DMA-based Encoder Capture**: Refactoring the QuadTimer capture to stream directly into memory via Direct Memory Access (DMA), completely offloading the CPU from signal processing.

Safety & Reliability (Industrial Hardening)

- **Blackbox Fault Logging**: Implementing a circular telemetry buffer storing the last 5 seconds of control states that automatically dumps over serial upon a fault (thermal, tracking error, or stall).
- **Torque Estimation**: Implicitly estimating joint torques by monitoring step-loss or tracking error growth, enabling soft dynamic limits.

System Integration

- **High-Rate ROS 2 Transport over USB HS**: Moving entirely from UART serial strings to a zero-copy transport or micro-ROS configuration taking full advantage of the i.MXRT1062 480 Mbps PHY.