Firmware Developer Guide

This document outlines the architectural boundaries and strict rules regarding the `parol6_firmware` implementation.

The firmware is designed as a **Production-Grade, Hard Real-Time Servo Controller**. It elevates the intelligence of the low-level embedded system to enable deterministic, 1 kHz control loops critical for tasks like welding.

1. Directory Structure

The `parol6_firmware/src` folder is strictly modularized:

- **`transport/`**: Isolates Serial/USB ASCII parsing. Translates raw bytes into lock-free generic structs (`RosCommand`).
- **`safety/`**: The `SafetySupervisor`. Manages E-Stops, runaway limits, and command timeouts. Must sit *outside* the math equations and strictly execute *before* physical motor outputs.
- **`observer/`**: The Estimator. `AlphaBetaFilter` for velocity prediction/correction on noisy encoders.
- **`control/`**: The strict math execution. `LinearInterpolator` and the PID loops.
- **`hal/`**: Hardware Abstraction Layer. Translates standard angle math (radians) into physical i.MXRT1062 register commands (`QuadTimerEncoder`).

2. Hard Real-Time Rules

To maintain the 1 kHz deterministic control loop and the < 50 µs jitter parameter, the following rules apply **strictly** inside the `run_control_loop_isr` and all functions called by it:

1. **NO Dynamic Memory Allocation:** You must not use `new`, `malloc`, `std::vector::push_back`, or anything that calls the heap inside the ISR. All classes must be statically allocated.
2. **NO Blocking Serial Prints:** Do not call `Serial.print()` inside the control loop. It will cause catastrophic jitter. If you need debug tracing, push data to a lock-free telemetry queue and let the main loop serialize it over USB.
3. **NO Heavy Floating-Point Division:** While the Teensy has an FPU, division is slower than multiplication. Where possible in filters, multiply by the inverse (e.g., `value * 0.5f` instead of `value / 2.0f`).
4. **Hardware Timers Only:** `micros()` and `millis()` software timers must not be used to drive the core timing logic. Use the dedicated ARM internal hardware timers (`IntervalTimer`) configured specifically for the 1ms tick.

3. The `PlatformIO` CLI Environment

We use PlatformIO for pure build determinism. The `platformio.ini` enforces:

- `-O3`: Aggressive compilation optimization.
- `-ffast-math`: Relaxes strict IEEE math standards for faster FPU execution (safe for our control bounds).
- `-std=gnu++17`: Modern C++ conventions.

Do not submit code relying on "magic" Arduino includes. Use standard library headers (`<stdint.h>`, `<math.h>`) when building generic logic classes under `src/` to retain simulator compatibility.

4. Simulator Compatibility

Where possible, generic math modules (`Interpolator`, `AlphaBetaFilter`) should be developed without hardware dependencies so they can be compiled and validated separately in the `esp32_benchmark_idf` simulator system for pure mathematical regression testing without robot mechanics.

5. Code Concepts Glossary

The Monotonic Tick

Inside `run_control_loop_isr()`, you will see `system_tick_ms++;`. We do not use Arduino's `millis()` inside the hardware timer because `millis()` relies on the internal SysTick interrupt. Managing interrupts within interrupts can lead to unpredictable latency. A local, monotonic variable incrementing at 1 kHz guarantees perfectly equidistant time steps.

Output Saturation (Clamping)

The `Control Law` implements `velocity_command = cmd_vel_ff + (Kp * pos_error)`. Mathematical spikes (caused by a noisy encoder or a sudden large waypoint jump) could command the motor to an unsafe infinite velocity. The variable is immediately passed through a `MAX_VEL_CMD` threshold to physically saturate the hardware request.

Innovation & Unwrapping

In the `AlphaBetaFilter`, the code continually checks if `delta > M_PI`. Magnetic encoders report absolute angles (e.g., 0 to 2π). When a joint spins past the zero point, the reading snaps from 6.28 to 0.00. The filter catches this mathematical discontinuity using the `M_PI` bounds and subtracts 2π, presenting a continuous, unwrapped multi-turn angle to the control loop.

ISR Array Caching & The Safety Execution Order (`sense -> estimate -> supervise -> act`)

`run_control_loop_isr()` computes the mathematical commands (`commanded_velocities[i]`) for *all* 6 axes before applying physical voltages. **Why?** In a safety-critical system, the architecture must strictly enforce: **sense** (encoders) → **estimate** (filter) → **supervise** (safety checks) → **act** (motor output). If Axis 0 passes safety, but Axis 5 triggers a runaway fault, applying voltages sequentially would cause Axis 0 to jump momentarily before the system halted. By caching the math, calculating safety, and *then* applying the outputs, the entire arm halts synchronously.

Zero-Interrupt Hardware PWM Capture (Phase 3 QuadTimers)

In Phase 1.5, we used `attachInterrupt` to capture PWM edges in software. However, software interrupts suffer from physical invocation latency and instruction jitter, which disrupts the 1 kHz control loop when 6 interrupts fire randomly.

In Phase 3, we migrated to the **i.MXRT1062 QuadTimers using Gated Count Mode (CM=6)**. This entirely removes the CPU from the capture process:

1. **Autonomous Counting:** The hardware timer is configured to count incredibly fast internal IP Bus Clock ticks *only when the physical input pin is HIGH*.
2. **Synchronous Polling:** When our strict 1 kHz control ISR fires, it reads the hardware timer register (`tmr_->CH[ch_].CNTR`) instantly without waiting for an edge.
3. **Cyclic Derivation:** By capturing the `ARM_DWT_CYCCNT` (CPU cycle counter) simultaneously with the QuadTimer counter, we know precisely how much real-time elapsed between reads. The duty cycle is calculated mathematically as `(High Ticks) / (Expected Ticks in Elapsed Time)`.
4. **Result:** Zero interrupts, zero jitter, and native 14-bit resolution encoder tracking.

6. Program Flow & State Machine

The firmware operates as a strict state machine to prevent uncalibrated kinematics from damaging the robot.

Boot Sequence (`setup()`)

1. **Clock Initialization**: The CPU frequency is maximized to 600 MHz.
2. **Peripheral Config**: Hardware IPs (UART, FlexPWM for motors, QuadTimers for encoders) are configured via static register setups. No dynamic objects are created.
3. **Hardware Sanity Check**: Tests communication with MT6816 encoders and TMC5160 stepper drivers over SPI.
4. **State Transition**: System enters the `UNHOMED` state. The 1 kHz Control Interrupt is started, but the `ControlLaw` is explicitly gated and outputs zeros.

Homing Sequence (`HomingCalibration Layer`)

The firmware enforces a strict order of operations based on mechanical dependencies (e.g., J6 cannot move if J5 is un-homed due to physical collision boundaries).

1. **J6 Clearance**: Axis J6 executes a fast-seek to its limit switch, backs off, and slow-seeks to establish an absolute mathematical zero.
2. **Coupled Alignment**: Once J6 is clear, J5 and J4 execute their sweeps.
3. **Base Alignment**: J1, J2, and J3 execute their sweeps simultaneously.
4. **State Transition**: The `Interpolator` states are synchronized to the now-known absolute encoder positions. The system transitions to `SAFE_OPERATIONAL` and begins accepting ROS commands.

Steady-State Operational Flow

Once homed, execution splits into two radically decoupled domains.

Background / Main Loop (Asynchronous)

- **Role**: Parses incoming `115200` baud Serial/USB streams into `<SEQ,pos,vel>` structs.
- **Flow**: Validated packets are pushed into the `Lock-Free RX Queue`.
- **Frequency**: Operates as fast as possible (best-effort), typically executing > 1 MHz.

Foreground / 1 kHz ISR (Strictly Deterministic)

- **Role**: The physical manifestation of movement.
- **Flow**:
    1. Safely pops the latest target from the `RX Queue`.
    2. Injects the target into the `LinearInterpolator` to generate a 1ms micro-waypoint.
    3. Executes `sense -> estimate -> supervise -> act` logic sequence perfectly every 1000 µs.
- **Safety**: If the `RX Queue` runs empty for > 100ms (ROS crash or disconnect), the ISR traps into a `Soft E-Stop`, smoothly decelerating the interpolator to 0 velocity and triggering a fault state.