Firmware Architecture & Data Flow

This document visualizes the hard real-time architecture of the PAROL6 Teensy 4.1 firmware.

1. Top-Level Module Relationships

The firmware is designed around a strict separation of concerns, decoupling the asynchronous ROS 2 communication from the highly deterministic 1 kHz control mathematics.

System Diagram

Module Responsibilities

Sub-System	Thread Context	Responsibilities	Constraints
`SerialTransport`	Main Loop / BG	UART `<SEQ...>` Parsing, Feedback emitting	Transport layer is byte-stream agnostic and supports UART, USB CDC, or bulk USB endpoints.
`RX Queue`	Cross-Thread	Holding validated generic `RosCommand` structs	Lock-free array
`HomingCalibration`	Main Loop / ISR	Executes multi-stage limit sweeps to establish absolute URDF zero-frame	Transitions system from `UNHOMED` to `SAFE_OPERATIONAL`. Must complete before ControlLaw engages.
`SafetySupervisor`	Main Loop / ISR	E-Stops, runaway limits, watchdog timeouts	MUST execute *before* Motor Output dispatch.
`AlphaBetaFilter`	1 kHz ISR	Predicts vel/pos from noisy raw angles and unwraps `M_PI` bounds	A Kalman filter was rejected due to variable matrix operations and non-deterministic execution time on embedded hardware without hardware acceleration. Requires encoder sampling > 2x max mechanical speed to avoid unwrap aliasing.
`LinearInterpolator`	Main Loop / ISR	Up-samples 25 Hz ROS commands to smooth 1 ms deltas	Pure kinematic domain (radians only).
`EncoderHAL`	1 kHz ISR	Extracts physical timer registers to radians	Absolute determinism required
`ControlLaw`	1 kHz ISR	`cmd_vel_ff + (Kp * pos_error)` clamped to `MAX_VEL_CMD`	Float math (FPU accelerated). All math uses single-precision floats with bounded execution time; migration to fixed-point is structurally possible without architectural changes.
`ActuatorModel / MotorHAL`	1 kHz ISR	Applies exact mechanical gear ratios (e.g., J3 `18.095`) to convert radians to steps, dispatches pulses	Separates actuation gear dynamics from pure kinematic math.

> **Architectural Note on FlexPWM & Stepper Drivers (e.g., MKServo42C)** The firmware uses the i.MXRT1062 `FlexPWM` peripheral to generate STEP signals. It is crucial to understand that we are **NOT** using "Analog-style PWM" (where duty cycle controls voltage/speed). Instead, we are using FlexPWM as a **Hardware Metronome**.
>
> - **Frequency** controls velocity.
> - **Duty Cycle** is irrelevant (fixed to a safe 2-5µs pulse width).
> - **Edge Count** controls position.
>
> This approach completely offloads step pulse generation from the CPU, preventing ISR jitter or contention that would occur if we attempted to bit-bang GPIO (`digitalWrite`) for 6 axes simultaneously.

---

2. The 1 kHz ISR Execution Pipeline

The 1 ms tick is the heart of the system. To guarantee jitter remains < 50 µs, the execution sequence inside the `run_control_loop_isr()` function is rigidly ordered to compute all math *before* making safety decisions and applying physical outputs.

ISR Timeline

```
--- BEGIN 1ms INTERRUPT ---
  TIME  | ACTION
  00 µs | Read Hardware System Tick (system_tick_ms++)
  02 µs | [Loop Axis 0 to 5: Math Phase]
        |   -> read_angle() from EncoderHAL
        |   -> AlphaBetaFilter::update(raw_pos) (Unwrap M_PI, Innovation step)
        |   -> Interpolator::tick_1ms() (Gets cmd_pos, cmd_vel_ff in Radians)
        |   -> Compute: pos_error = cmd_pos - estimated_pos
        |   -> Compute: cmd_vel = cmd_vel_ff + (Kp * pos_error)
        |   -> Clampcmd_vel to safely bounded MAX_VEL_CMD
        |   -> Cache cmd_vel in local array
  ~15 µs| [Math Computations Conclude]
        |
  16 µs | SafetySupervisor::update(tick_ms, all_velocities)
        |   -> Checks for Runaway/Timeouts
        |
  18 µs | [Loop Axis 0 to 5: Output Phase]
        |   -> ActuatorModel::convert_radians_to_pulses(cmd_vel, gear_ratio)
        |   -> supervisor.is_safe() == true ? Apply pulses to MotorHAL : Apply
0.0f
  ~25 µs| return;
--- END 1ms INTERRUPT ---
```

3. Data Ownership & Thread Safety Rules

Because the system blends an asynchronous background loop (serial parsing) with a pre-emptive foreground ISR (hardware timer), data ownership is strictly enforced to prevent race conditions without relying on heavy RTOS mutexes taking down the ISR.

1. **`CircularBuffer<RosCommand>`**: Acts as the sole locking boundary. The `MainLoop` owns pushing. `MainLoop` temporarily calls `noInterrupts()` to pop items safely before the ISR can strike.

2. `Interpolator`: Owned by the ISR (`tick_1ms`). Setpoints (`set_target`) are injected by the `MainLoop` only during the safe periods between queue pops.
3. `AlphaBetaFilter`: Exclusively owned by the ISR. The `MainLoop` is allowed to *read* the state (for background 10 Hz telemetry) but must wrap the read in `noInterrupts()` to prevent reading a torn float if the 1ms tick interrupts the copy operation.
4. **Cache Coherency**: Cortex-M7 D-cache effects are avoided by utilizing Tightly Coupled Memory (TCM) for all critical ISR data arrays, ensuring that variables shared across the `noInterrupts()` boundary do not suffer from cache-miss latency or stale lines.

4. Migration Context: Previous Firmwares vs. Current Architecture

The new `parol6_firmware` completely restructures the approach taken in the previous two iterations (`realtime_servo_control` for ESP32 and `realtime_servo_teensy`). Our primary design driver was eliminating jitter and enabling advanced trajectory filtering for welding ops.

Evolution of the Control Loop

Firmware	**`realtime_servo_control` (Legacy ESP32)**	**`realtime_servo_teensy` (Legacy Teensy)**	**`parol6_firmware` (Current Teensy 4.1)**
Execution Context	FreeRTOS Task (`vTaskDelayUntil`)	Hardware `IntervalTimer` ISR	Hardware `IntervalTimer` ISR
Control Rate	500 Hz	500 Hz	**1 kHz (1000 Hz)**
Timing Jitter	Susceptible to RTOS preemption	Very Low (< 30 µs)	**Strictly Bounded (< 15 µs)**
Signal Processing	None / Raw positional	None / Raw positional	**AlphaBeta Observer Filter (noise rejection)**
Interpolation	Basic / Host Dependent	Basic	**Dynamic Linear Interpolation (Feedforward)**
Data Threading	Synchronous RTOS tasks	Polled main loop `elapsedMicros`	**Lock-Free `CircularBuffer<RosCommand>`**
Safety Engine	Mingled with control math	Mingled with control math	**Isolated State Machine (Timeouts & Clamping)**

Key Takeaways: The ESP32 firmware suffered from context-switching overhead because the FreeRTOS control task competed with the WiFi/Bluetooth stacks internally, causing jitter. The legacy `realtime_servo_teensy` fixed the jitter by using a hardware ISR (500 Hz) but lacked intelligence (no filters, no rigorous interpolators). The current architecture operates at double the bandwidth (1 kHz) while cleanly decoupling the noisy serial traffic (`CircularBuffer`) from a highly mathematical, noise-rejecting (`AlphaBetaFilter`) real-time core.

5. ROS 2 Communication Strategy

Communication between the ROS 2 Host PC and the Teensy 4.1 maintains compatibility with the legacy `parol6_hardware` interface but is optimized for the deterministic architecture.

- **Protocol**: ASCII over UART (115200 Baud) or USB CDC.
- **Command Format**: `<SEQ, J1, J2, J3, J4, J5, J6, V1, V2, V3, V4, V5, V6>` emitted at ~25 Hz by `ros2_control`.

- **Feedback Format**: `<ACK, SEQ, J1, J2, J3, J4, J5, J6, V1, V2, V3, V4, V5, V6>` emitted at ~10 Hz back to ROS.
- **Decoupling**: The 25 Hz asynchronous stream from ROS is cleanly decoupled from the 1 kHz control loop via a **Lock-Free Circular Queue**.
- **Data Integrity**:
  - The `SerialTransport` parses bytes in the background `main()` loop to avoid blocking.
  - Full, validated `RosCommand` structs are pushed to the Queue.

6. Real-World Profiling Metrics & Academic KPIs (Phase 1.5 & Phase 3)

To mathematically prove the determinism of this architecture, a bare-metal software profiler leveraging the ARM Cortex-M7 cycle counter (`ARM_DWT_CYCCNT`) was injected directly into the 1 kHz `run_control_loop_isr()`.

Academic KPI Definitions

- **Latency**: Defined explicitly as `timestamp_motor_step - timestamp_ros_command_rx`. Measured via embedded timestamp echoing in the `<ACK>` packet and host correlation. Target: `< 15 ms`.
- **Control Jitter**: Defined strictly as the *max deviation of ISR start time from the ideal 1 ms schedule*, measured using the DWT cycle counter over 10^6 consecutive cycles. Target: `< 50 µs`.

Test Conditions (Hardware-in-the-Loop):

- Teensy 4.1 executing full Alpha-Beta filter math, linear interpolator, and safety supervisor.
- Concurrently parsing incoming 115200 baud serial commands from ROS.
- Externally bombarded by 6 simultaneous, independent 1 kHz physical PWM signals from an ESP32 simulator.

Phase 1.5 Results (using Software `attachInterrupt`):

- **Measured Nominal ISR Execution Time**: `~6 µs`
- **Measured Peak ISR Execution Jitter**: `Up to 15 µs` (due to software interrupt collisions).

Phase 3 Final Results (using Hardware QuadTimers):

- **Measured Maximum ISR Execution Time**: severely restricted to `1-2 µs` thanks to Gated Count Mode zero-interrupt capture.
- **Peak ISR Jitter**: `< 1 µs` (below logic analyzer measurement resolution).
- **Available CPU Headroom per 1 ms (1000 µs) Tick**: `> 998 µs`
- **Control Stability**: With 1 kHz sampling and `ControlLaw` (P+FF), the closed-loop bandwidth is conservatively bounded below 50 Hz, ensuring absolute stability under typical stepper mechanical dynamics.

This absolutely confirms that the transition to the Teensy 4.1 eliminates real-time processing bottlenecks. The architecture is fully capable of driving all 6 axes with advanced filtering algorithms while maintaining strict determinism.

7. Clock Domains & Safety Architecture

Understanding time is critical for the `SafetySupervisor` and latency calculations. The architecture operates across three distinct clock domains.

Clock Domains

1. **ROS Host Clock (Observational):** The PC's system time. Used by `tf2` and `joint_states`. Not trusted by the MCU for real-time actuation.

2. **MCU Monotonic Hard-Tick (Control Truth):** Incremented perfectly at 1 kHz inside the hardware ISR. This absolute tick drives the `ControlLaw`, the `Interpolator`, and triggers `SafetySupervisor` protocol timeouts.
3. **ARM DWT Cycle Counter (Diagnostic):** A 600 MHz register used exclusively for microsecond profiling (`ARM_DWT_CYCCNT`). It measures ISR execution time and computes encoder capture latency.

(Note: Latency correlation is achieved by the MCU echoing its Monotonic Tick or DWT stamp inside the `<ACK>` packet back into the ROS Observational domain).

Safety Execution & Failure Mode Matrix

The architecture uses a strictly ordered execution phase: `sense -> estimate -> supervise -> act`.

Failure Mode	Detection Mechanism	Architectural Reaction
ROS Link Disconnect	`SafetySupervisor` Watchdog timeout (e.g., > 100ms since last `RosCommand`)	Soft E-Stop (Command velocity forced to 0.0)
Encoder Glitch / Noise spike	`AlphaBetaFilter` residual bounds and innovation clamping	Rejection of bad data point; velocity outputs physically saturated
ISR Execution Overrun	Embedded DWT cycle measurement flag	Catastrophic Fault State; `ControlLaw` disabled, hardware driver `ENABLE` pin cut
Joint Runaway (Limit tripped)	Zone 4 physical safety pin check	Trajectory aborted, error state broadcast over Serial Transport