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

```
+---------------------+
|   ROS 2 Host PC     | (125 Hz Trajectory Generation)
+---------------------+
           |
           | <UART 115200 Baud / USB CDC>
           |
+--------------------------------------------------------------+
| TEENSY 4.1 MCU (i.MXRT1062)                                  |
|                                                              |
|  [ BACKGROUND APPLICATION LOOP ]                             |
|  +-------------------------+      +-----------------------+  |
|  | SerialTransport         | ---> | RX Queue (Lock-Free)  |  |
|  +-------------------------+      +-----------------------+  |
|                                              |               |
|  +-------------------------------------------v------------+  |
|  |                  Main() Command Router                 |  |
|  +--------------------------------------------------------+  |
|         | Sets Targets                  | Feeds Watchdog     |
|         v                               v                    |
|  +-------------------------+      +-----------------------+  |
|  | LinearInterpolator (x6) |      | SafetySupervisor      |  |
|  +-------------------------+      +-----------------------+  |
|         |                               |                    |
| ~ ~ ~ ~ | ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ | ~ ~ ~ ~ ~ ~ ~ ~ ~ ~|
|         |                               v                    |
|  [ PREEMPTIVE 1 kHz ISR  ------------------------------ ]    |
|  | +-----------------+    +---------------------------+ |    |
|  | | EncoderHAL (x6) | -> | AlphaBetaObserver (x6)    | |    |
|  | +-----------------+    +---------------------------+ |    |
|  |                              |                       |    |
|  |  +---------------------------v--------------------+  |    |
|  |<-| Control Law: Feedforward + (Kp * Pos Error)    |  |    |
|  |  +---------------------------v--------------------+  |    |
|  |                              |                       |    |
|  |  +---------------------------v--------------------+  |    |
|  |  | Output Saturation (Clamping)                   |  |    |
|  |  +---------------------------v--------------------+  |    |
|  |                              |                       |    |
|  |  +---------------------------v--------------------+  |    |
|  |  | Motor Output Dispatcher (Safety Gated)        |  |    |
|  |  +-------------------------------------------------+  |    |
|  [ ---------------------------------------------------- ]    |
+---------------------------------------------------------------+
```

Module Responsibilities

Sub-System	Thread Context	Responsibilities	Constraints
`SerialTransport`	Main Loop / BG	UART `<SEQ...>` Parsing, Feedback emitting	Non-blocking bytes processing
`RX Queue`	Cross-Thread	Holding validated generic `RosCommand` structs	Lock-free array
`SafetySupervisor`	Main Loop / ISR	E-Stops, runaway limits, watchdog timeouts	Only blocks outputs, never math
`AlphaBetaFilter`	1 kHz ISR	Predicts vel/pos from noisy raw angles and unwraps `M_PI` bounds	Must not contain divisions `(/)`
`LinearInterpolator`	Main Loop / ISR	Up-samples 25 Hz ROS commands to smooth 1 ms deltas	Must dynamically adapt to delta_t
`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)

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)
          |   -> 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]
          |   -> supervisor.is_safe() == true ? Apply cmd_vel : 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. Migration Context: ESP32 vs. Teensy 4.1

The previous iterations of the PAROL6 firmware utilized an ESP32 micro-controller. The migration to the Teensy 4.1 was driven by the strict requirements of hard real-time execution and deterministic control loops for advanced trajectory tracking (e.g., welding).

Key Differences

Feature	Former ESP32 Firmware	Current Teensy 4.1 Firmware
Primary Role	Serial relay and simple step generation	Deterministic 1 kHz servo controller
Control Intelligence	Heavy reliance on ROS/Host PC	Intelligence moved "down the stack" to MCU
Control Loop	Soft real-time (RTOS/Task-based)	Hard real-time (Strict 1 kHz Hardware ISR)
Interpolation	Basic (or purely host-driven)	1 kHz linear interpolation with Feedforward
Safety	Host-dependent or delayed	Independent, < 1ms response isolated from math

Feature	Former ESP32 Firmware	Current Teensy 4.1 Firmware
Clock Domain	Non-uniform (`millis()` / RTOS ticks)	Unified, pure hardware-driven `system_tick_ms`

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.
  - Main loop safely peeks the Queue inside a momentary `noInterrupts()` block to push setpoints to the `LinearInterpolator`, ensuring the 1 kHz ISR never reads half-written floats.