CONTROL SYSTEM FOR DRONE STABILIZATION AND PRECISION MOTION

TECHNICAL REPORT

TEAM 24

DRONA AVIATION™

DRONA AVIATION PVT. LTD.

Inter IIT Tech Meet 14.0

Contents

Contents

1 Approach Outline

1.1 Problem Understanding

Indoor nano-drones are increasingly used for research and education because they are safe, portable, and suitable for small environments. The Pluto series by Drona Aviation is one such platform, offering programmability for hands-on UAV experimentation. However, without GPS or external positioning systems, accurate hovering and motion control depend entirely on onboard sensing and control algorithms. With lightweight hardware and limited sensing accuracy, factors like lighting variations affecting optical flow, ToF noise, IMU drift, and small vibrations can cause noticeable drift or instability. This makes reliable hover and precise 10–20 cm user-triggered micro-movements challenging across varying indoor conditions. The goal is to design an onboard stabilization and precision motion-control system that fuses optical flow, ToF, IMU, and barometer data to compute stable position and velocity estimates. These feed a cascaded Position → Velocity → Attitude controller capable of autonomous takeoff, hands-free hover, disturbance rejection, and centimeter-level motion with smooth acceleration and minimal overshoot. The solution must run entirely on the MagisV2 firmware within its microcontroller limits, remain robust up to 2 meters altitude, and deliver repeatable, deterministic behavior across flights.

1.2 Final Approach

In the final implementation, we followed the problem statement by using the provided sensor drivers and wiring them into the existing MagisV2 fusion and control points, instead of creating standalone custom drivers.

1. **Altitude Fusion (VL53L1X + Barometer)**
   The VL53L1X ToF sensor is used as the primary low-altitude reference and fused with the barometer in `altitudehold.cpp`.

   - XVision is initialised once in `main.cpp` using long-range mode.
   - In the altitude loop, ToF readings are retrieved, tilt-compensated, and combined with the barometer to estimate:
     - `EstAlt` (filtered altitude)
     - `VelocityZ` (vertical rate)
   - These values feed the cascaded altitude controller:
     - Outer loop: altitude error → desired vertical speed
     - Inner loop: vertical-speed error → throttle adjustment

   This replaces legacy VL53L1X code and ensures alignment with the required `LaserSensor_L1` interface.

2. **Optical Flow Fusion (PAW3903 + Altitude)**
   Horizontal motion estimation now uses the updated `paw3903_opticflow` driver, with velocity calculations moved into `opticflow.cpp`.

   - `paw3903_init()` configures the sensor at startup.

- During updates, motion burst data (`deltaX`, `deltaY`, `squal`) is read.
- Low-quality frames are rejected using an `squal` threshold.
- Valid optical-flow deltas are filtered and scaled using current height to compute:
  - `VelocityX`, `VelocityY`
- These velocities are integrated to obtain:
  - `PositionX`, `PositionY`

These estimates are used by the FLOW controller to correct drift and maintain stable hover.

3. **Debugging and Validation**
   Validation was performed using Monitor/PlutoMonitor in the standard `GRAPH:` telemetry format.

   - Logged variables included:
     - Altitude: `EstAlt`, `VelocityZ`, ToF height
     - Horizontal: `VelocityX/Y`, `squal`, and intermediate flow terms
   - Logging confirmed correct altitude fusion while revealing remaining optical-flow drift issues, particularly asymmetric response along the Y-axis.

1.3 Background Study

The background study focused on understanding control and state-estimation methods suitable for indoor nano-quadrotors, which rely entirely on onboard sensing. Prior work shows that cascaded PID control remains a very effective architecture for quadrotor stabilization, offering modularity and reliable real-time performance on constrained microcontrollers. Because the drone must operate without GPS, accurate horizontal velocity estimation is critical. Studies highlight that optical flow fused with IMU data significantly improves indoor velocity accuracy by compensating for yaw-induced distortions and texture limitations. Research on indoor flight fusion techniques further stresses the need for adaptive, confidence-based integration of ToF, barometer, and IMU measurements to maintain stability under noise and sensor dropouts. A broader survey of UAV sensor characteristics reinforces the importance of robust, low-latency fusion to counter IMU drift, ToF noise at higher altitudes, and barometric fluctuations. These insights collectively justify the use of a fused OF–IMU–ToF–Baro estimator driving a cascaded control pipeline for stable hover and precise micro-movements in indoor environments.

2 System Architecture

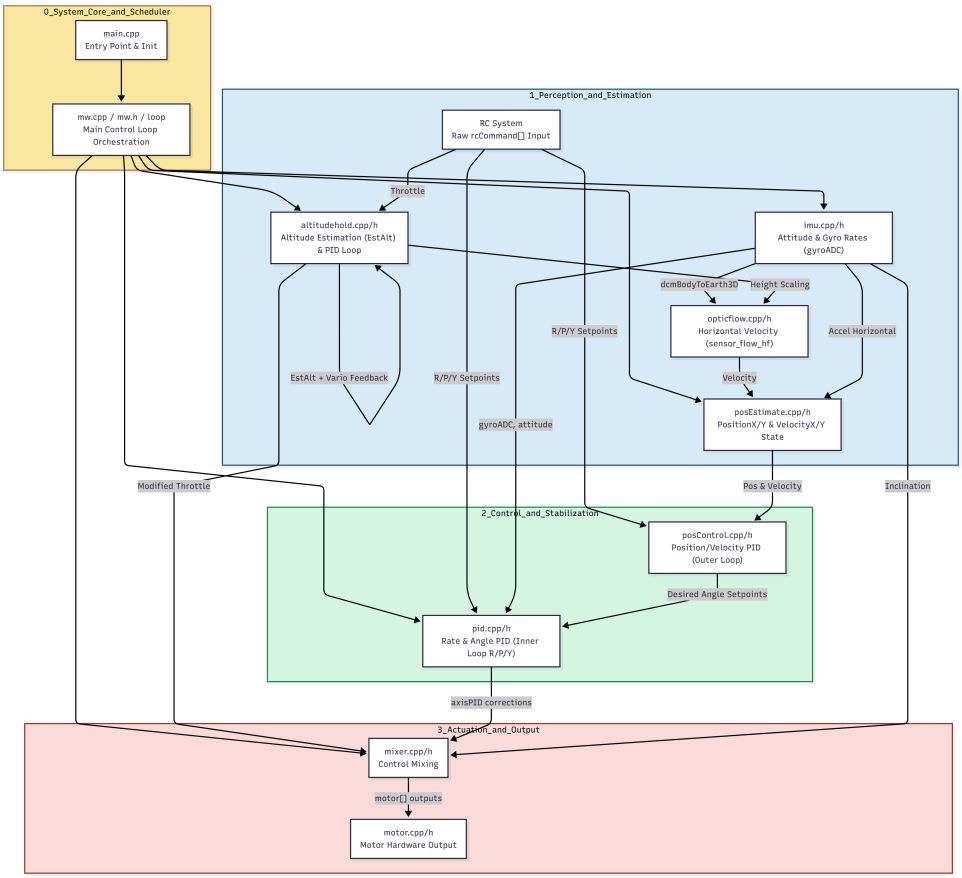

Figure 1: System Architecture Diagram

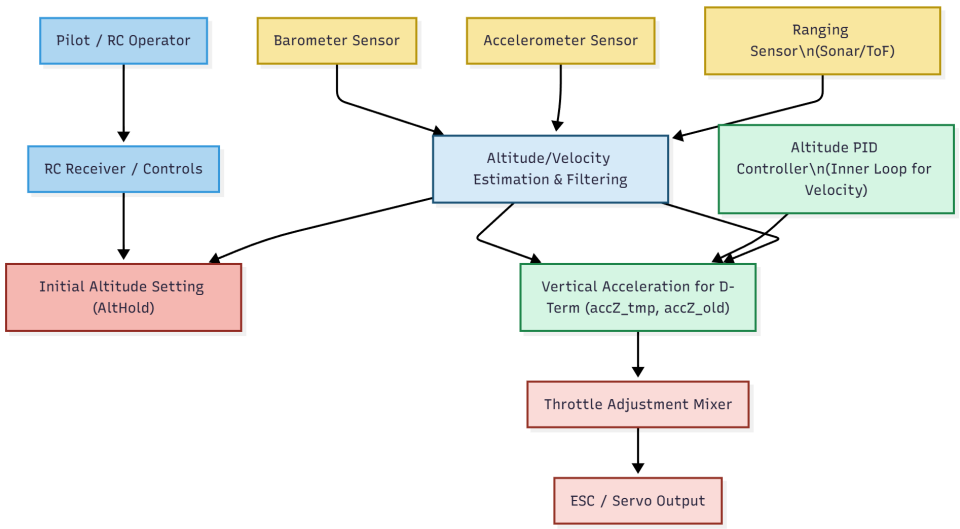

Figure 2: AltitudeHold Flowchart

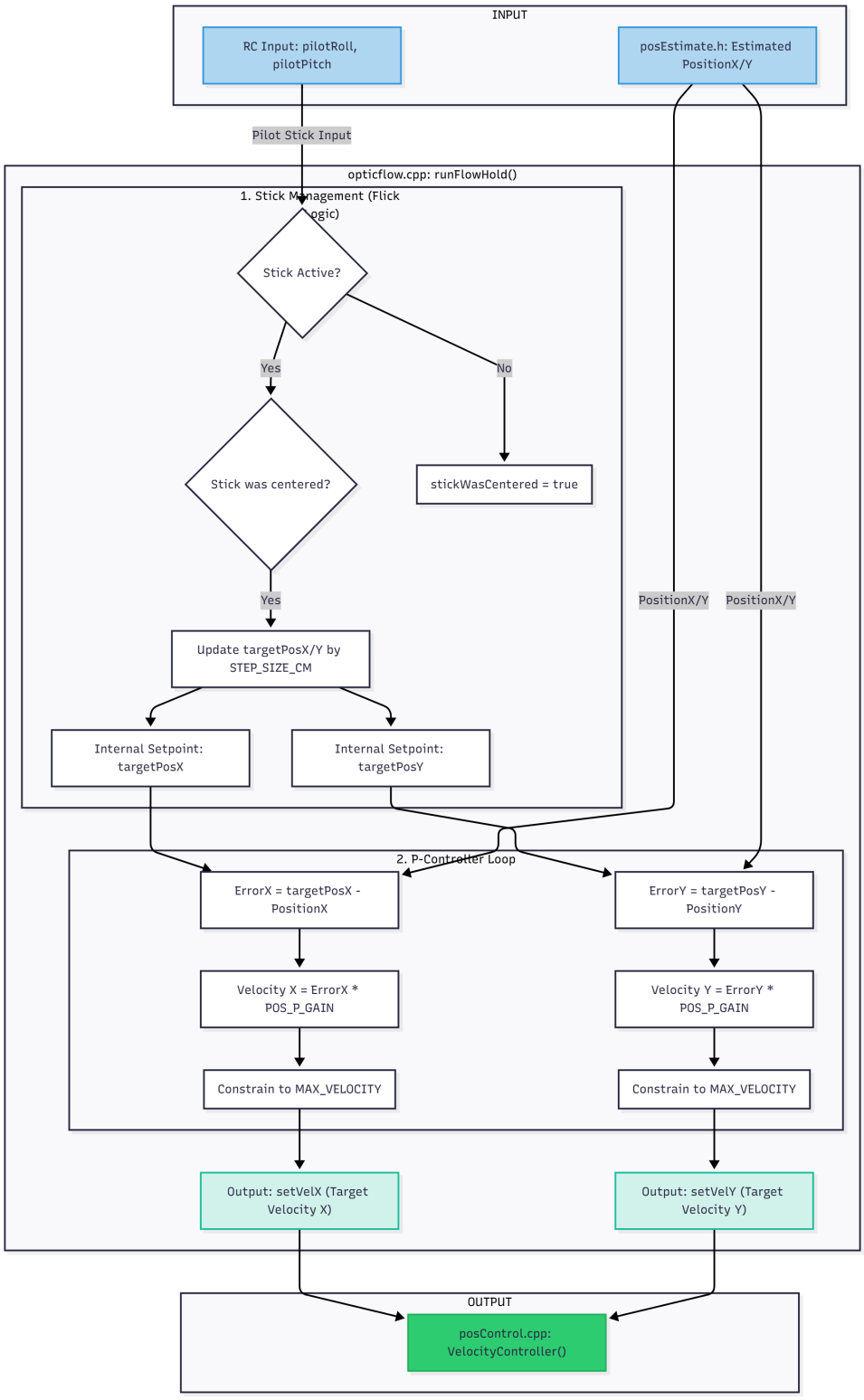

Figure 3: Opticflow Flowchart

Overview

The updated flight control implementation introduces a structured separation between the hardware sensor layer and high-level navigation logic, enabling reliable horizontal positioning and scalable flight capabilities.

1. **Optic Flow Driver (`PAW3903_opticflow.cpp`)**

   - Acts as the hardware abstraction interface, replacing the legacy driver and providing standardized measurement output.
   - Reads raw surface motion from the PAW3903 optical flow sensor and computes values required for horizontal motion estimation.
   - **Primary Outputs:**
     - `flowrate` – raw optical motion converted into measurable frame displacement.
     - `bodyrate` – rotational compensation term based on gyro data from `gyro.cpp`.
     - `last_ms_update` – timestamp used for consistent *dt* computation to prevent stale-state usage.
   - Operates purely as a sensor-side data producer and does not apply filtering or control logic. Its data feeds into the state estimation and control layers.

2. **Opticflow Control Logic (`opticflow.cpp`)**

   - Implements the horizontal position hold and motion control behaviour.
   - **Data Dependencies:**
     - `rcCommand[ROLL/PITCH]` from `rc_control.h` representing pilot intent.
     - `PositionX/Y` from `posEstimate.h` representing current estimated position state.
     - Additional references such as `LaserSensor_L1` (height scaling) and `acc_1G` (gravity constant).
   - **Core Function:** `runFlowHold()`
     - Implements a *"flick-to-move"* behaviour where stick deflections past a threshold update the target position only if the stick was previously centered.
     - Applies cascaded proportional-type control where:

       $$vel\_cmd = P \cdot (targetPos - Position)$$

     - Writes the velocity-setpoint-derived output back into:

       `rcCommand[ROLL/PITCH]`

       replacing the direct pilot angle request with controller-generated commands.
     - These modified commands propagate into `pid.cpp`, where the attitude and rate loops generate appropriate Roll/Pitch motor responses.

3. **Integration With Altitude Control**

   - Horizontal velocity and position control operate in coordination with the `altitudehold.cpp` module, which maintains vertical flight stability.

- The altitude controller fuses barometer and rangefinder measurements to compute:

  `EstAlt`, `VelocityZ`

- A dedicated PID loop generates:

  `altHoldThrottleAdjustment`

  which supplements or replaces manual throttle input.

- Together, the horizontal velocity outputs from `opticflow.cpp` and the vertical throttle adjustments from `altitudehold.cpp` feed into the `mixer.cpp` stage, producing the final motor PWM signals for flight.

3 Sensor Fusion - Methods and Outputs

3.1 Altitude Hold System (ToF and Baro fusion)

This altitude-hold module incorporates several control-theoretic and sensor-fusion improvements to achieve stable, accurate altitude estimation and smooth closed-loop behavior.

- Adaptive Throttle Capture for Smooth Mode Engagement
  To ensure a disturbance-free transition into altitude-hold, the controller records the pilot's current throttle at the moment the mode is activated:

```
initialThrottleHold = rcData[THROTTLE];
```

  This removes any discontinuity in commanded thrust, preventing vertical jumps and providing a smooth engagement into closed-loop altitude control.

- VL53L1X Time-of-Flight (ToF) Sensor Integration
  Support for the VL53L1X high-range ToF sensor enables accurate low-altitude measurement. The code retrieves fresh distance measurements (in cm), applies tilt compensation, and integrates the values into the altitude estimator:

```
if (isTofDataNew_L1() && !isOutOfRange_L1()) {
    ToF_Height = (float)NewSensorRange_L1 / 10.0f;

    tilt = degreesToRadians(calculateTiltAngle(&inclination) / 10);
    if (tilt < 25.0f)
        ToF_Height *= cos_approx(tilt);
}
```

  This provides precise, low-noise absolute altitude data within the sensor's valid range.

- Deterministic Sensor-Fusion Hierarchy
  A clear fusion priority is implemented to ensure reliable altitude estimation:

```
if (ToF_Height > 5.0f && ToF_Height < 350.0f && !isOutofRange_L1())
    {
    baro_offset = filtered - EstAlt;
    correctedWithTof(ToF_Height);
```

```
} else {
    correctedWithBaro(Baro_Height - baro_offset, dt);
}
```

Fusion logic:

- ToF is used whenever valid
- Barometer is used as fallback
- A dynamic barometric offset stabilizes transitions

This produces a stable altitude estimate across both low-altitude and high-altitude regimes.

- Tilt Compensation for Accurate Vertical Height
  To correct slant-range measurements from the downward ToF sensor, altitude is multiplied by the cosine of the aircraft's tilt angle:

```
ToF_Height *= cos_approx(tilt);
```

This accounts for geometric distortion during roll/pitch motion and ensures the estimator receives true vertical altitude.

- Third-Order Complementary Filter Enhancements
  The altitude estimator uses a high-order complementary filter derived from ArduPilot. It performs IMU-based prediction followed by sensor-based correction:

```
_accel_correction_hbf_z += _position_error_z * _k3_z * dt;
_velocity_z             += _position_error_z * _k2_z * dt;
_position_correction_z  += _position_error_z * _k1_z * dt;

float velocity_increase_z = (accel_ef_z + _accel_correction_hbf_z)
   * dt;
_position_base_z += (_velocity_z + velocity_increase_z * 0.5f) * dt
   ;
_position_z      = _position_base_z + _position_correction_z;
_velocity_z      += velocity_increase_z;
```

To maintain estimator stability during aggressive vertical motion, the system automatically triggers a reset when large velocity spikes occur:

```
if (abs(VelocityZ) > 200) {
    AltRequired = 1;
}
```

These mechanisms ensure drift-resistant altitude estimation based on accelerometer integration with continuous error correction.

- Kalman Filter Smoothing of Altitude and Vertical Velocity
  A lightweight 1-D Kalman filter is applied to both altitude and vertical velocity to reduce noise while retaining dynamic responsiveness:

```
float filteredVel = kalmanFilterUpdate(&velHoldFilter, VelocityZ);
float filteredAlt = kalmanFilterUpdate(&altHoldFilter, EstAlt);

VelocityZ = lrintf(filteredVel);
EstAlt  = lrintf(filteredAlt);
```

This smoothing step enhances the performance of the downstream PID velocity and altitude loops.

- Real-Time Telemetry for Estimator and Controller Tuning

```
Monitor_Println("GRAPH:alt_est_cm",  (double)EstAlt);
Monitor_Println("GRAPH:vel_z",       (double)VelocityZ);
Monitor_Println("GRAPH:tof_height_cm", (double)ToF_Height);
```

This assists in tuning complementary-filter gains, Kalman parameters, and altitude/velocity PID coefficients.

3.2 Optical Flow System (PAW3903 + ToF-Based Horizontal Velocity)

The module extends MagisV2 by estimating horizontal velocity and position using the PAW3903 optical-flow sensor plus ToF altitude. Instead of IMU + baro for vertical motion, it uses pixel motion + height + timing to compute ground-relative XY velocity.

1. PAW3903 Optical-Flow Integration
   The PAW3903 provides frame-to-frame pixel motion:
   - paw3903_read_motion_burst(flowData)
   - deltaX, deltaY: signed pixel deltas
   - squal: signal quality

   A quality threshold discards low-quality frames, reducing noise.

2. ToF-Based Height Scaling
   Pixel deltas alone don't give true speed — they're scaled with height and loop time.

```
float getLaserAltitudeMeters() {
    float dist_m = Dist / 1000.0f;
    return constrainf(dist_m, 0.05f, 2.0f);
}
```

   Core relation used in calculateSensorFlow():

$$v_{x,y} \propto \frac{filtered\_flow_{x,y} \cdot altitude \cdot FLOW\_SCALER}{\Delta t}$$

   where:
   - filtered_flow is the low-pass filtered pixel delta,
   - altitude is the laser height in meters,
   - t is the loop time,
   - FLOW_SCALER converts the result into cm/s.

3. Deterministic Processing Pipeline

   (a) Timimg

- Compute deltaT
- Reject too small/large values
- Update last_calc_time

(b) Optical flow read
- If read fails → skip this cycle

(c) Quality gating
- Discard if squal ¡ threshold

(d) Pixel filtering
- Convert pixel deltas to float
- Low-pass filter with SENSORFLOW_LPS

(e) Height
- Use getLaserAltitudeMeters()
- Store in opticflowHeight

(f) Velocity estimation
Compute:

```
vel_x_est = (filtered_flow[0] * altitude * FLOW_SCALER) /
   deltaT;
vel_y_est = (filtered_flow[1] * altitude * FLOW_SCALER) /
   deltaT;
```

(g) Deadband
- Small values near zero are forced to 0 → prevents drift

(h) State update (only when armed)
- Convert velocities to cm/s → update:
  - VelocityX, VelocityY
- Integrate over time → update:
  - PositionX, PositionY
- When disarmed → reset all values

(i) Outputs to the Controller
The module exposes:
- Velocities: VelocityX, VelocityY (cm/s)
- Positions derived from integrating velocity
- These help the controller:
  - Reduce drift
  - Improve horizontal hover

(j) Telemetry for Tuning (Teleplot/PlutoMonitor)
Exposed variables:
- Estimated velocities:
  - VelX_Bench, VelY_Bench (continuous cm/s)
  - VelX_Calc, VelY_Calc (integer controller values)
- Raw sensor data:

  - FlowX_Raw, FlowY_Raw (pixel deltas)
  - LaserAlt (m)
  - Squal (signal quality)

Used to check:

- Correct sign and scaling
- Velocities 0 when drone is still
- Relationship between height, SQUAL, and noise
- Tuning:
  - FLOW_SCALER
  - SENSORFLOW_LPS
  - Deadband values

4 Control Architecture Strategy

4.1 Cascaded Control System

To achieve the stability requirements outlined in the problem statement, we implemented a Cascaded Control Architecture. We implemented a Cascaded P-P (Proportional-Proportional) Control Strategy. We utilize a Proportional controller for the outer position loop to generate velocity setpoints, and a Proportional gain for the inner velocity loop to generate attitude setpoints for the underlying high-frequency Attitude PID loop of the Magis firmware. This method decouples the position dynamics from the attitude dynamics, allowing for robust handling of both stations-keeping and dynamic maneuvers. The control pipeline consists of two primary loops:

- Outer Loop (Position Controller):
  Calculates the desired velocity required to minimize the distance error between the drone's current position and its target.

- Inner Loop (Attitude Controller):
  Converts the desired velocity into pitch and roll angles to accelerate the drone.

Mathematical Formulation:
Let $e_p(t)$ be the position error at time t:

$$e_p(t) = P_{target} - P_{current}$$

The desired velocity V_{cmd} is computed using a Proportional (P) controller. We utilize a P-only controller for the outer loop because the inner velocity loop naturally acts as a damping factor, preventing overshoot:

$$V_{cmd} = K_{p\_pos} \cdot e_p(t)$$

This command is then constrained to a safe maximum velocity ($V_{\max}$) to prevent runaway conditions:

$$V_{cmd} = \text{clamp}(V_{cmd}, -V_{\max}, V_{\max})$$

Finally, the velocity command is converted into an attitude (tilt) command θ_{cmd} using a velocity-to-angle gain K_{v2a}:

$$\theta_{cmd} = K_{v2a} \cdot V_{cmd}$$

This architecture was chosen over a single PID loop because optical flow sensors provide velocity data directly. By treating velocity as the primary state for the inner loop, we achieve faster disturbance rejection than if we relied solely on position estimates, which suffer from integration drift over time.

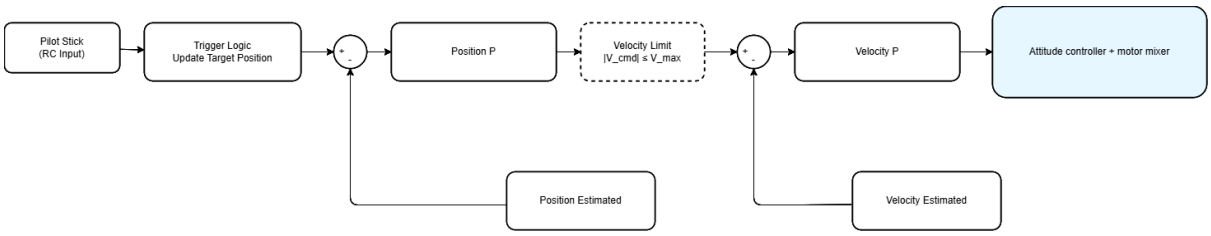

Figure 4: Control Architecture

4.2 Micro-Movement Logic

To satisfy the requirement of "User-Triggered Micro-Movements" where stick input acts as a discrete displacement command, we implemented a Trigger-and-Hold logic. Instead of mapping stick position to velocity (which causes continuous drift), our system detects stick deflection as a binary event.
When the input exceeds a safety deadband (T_{stick}), the global positional setpoint is instantaneously updated by a fixed quantum:

$$P_{target}^{new} = P_{target}^{old} \pm STEP\_SIZE\_CM$$

where $STEP\_SIZE\_CM$ is fixed at 15 cm.

Safety Precision:
To prevent "runaway" behavior where holding the stick causes continuous motion, we implemented a Input Latch. Once a step is triggered, the system ignores all further inputs until the pilot physically releases the stick to the center. This guarantees that one stick flick equals exactly one 15 cm movement, ensuring deterministic navigation. The Cascaded PID loop then automatically handles the acceleration and deceleration, settling the drone at the new target.

5 PID Tuning

5.1 Tuning Strategy

The tuning process utilized Sequential Loop Closing, prioritizing the innermost, fastest control loops first. We calibrated the Flow Scalar to set the physical scale, then tuned the Inner Loop (Velocity-to-Angle Gain) for optimal damping, and finally tuned the Outer Loop (Position P-Gain) to govern the speed of the position correction.

5.2 Final Gains

Parameter	Name	Value
Flow Scalar (Calibration)	`FLOW_SCALER`	0.001f
Velocity-to-Angle Gain (Inner Loop)	`VEL_TO_ANGLE`	3.0f
Position P-Gain ($P \rightarrow V$) (Outer Loop)	`POS_P_GAIN`	1.5f

Table 1: Final Gains

6 Experiments and Results

6.1 Altitude Hold Performance

Indoor hover tests were conducted with `ALT-HOLD` enabled, using the fused barometer + XVision ToF altitude estimate (`EstAlt`) as the primary input to the altitude controller.

The drone demonstrated the ability to maintain any commanded height within the **0.5–2 m** operating band. After receiving the takeoff command, it rose smoothly to approximately **1 m** and stabilised. From this point, the pilot could increase or decrease the hover altitude using the throttle stick, and the aircraft reliably settled at the new target height.

Telemetry logs from Teleplot show a clean altitude profile: a smooth climb during takeoff followed by a stable plateau at the commanded height. Only minor fluctuations were present, indicating that the cascaded altitude controller (altitude error $\rightarrow$ vertical-velocity setpoint $\rightarrow$ throttle correction) effectively rejects disturbances and maintains a steady hover.

6.2 Horizontal Drift with Optical Flow

With the PAW3903 optical-flow module enabled, the drone exhibited minimal horizontal drift during hover. Visual checks against floor markers indicated a displacement of approximately **10–20 cm** over the duration of each hover test.

This behaviour corresponds well with the optic-flow-derived velocity estimates, which remained close to zero when the drone was stationary. This verifies that:

- **SQUAL gating** and low-pass filtering effectively remove poor-quality or noisy flow frames.
- The tuned `FLOW_SCALER` and deadband values prevent small biases from accumulating into large position errors.

Overall, the combination of the altitude-hold system and optic-flow-based horizontal stabilisation produced a stable, near "hands-off" hover. The remaining drift was slow, minor in magnitude, and well within the acceptable limits for this project.

Figure 5: Vertical Velocity (VelocityZ)

Figure 6: Estimated Altitude (EstAlt)

7 Failure Handling

7.1 Identified Failure Modes

The primary instabilities addressed by the control system fall into sensor corruption and loss of control authority:

1. **Velocity Saturation & Runaway:** Large position errors command impossible speeds, risking structural failure.
2. **Sensor Data Corruption:** Optic Flow data returns unreliable or non-physical readings (low SQUAL / stuck bits), which directly corrupts velocity estimation.
3. **Z-Axis Sensor Failure:** Loss of the primary Laser ToF reference, making horizontal scaling ($V \propto h$) unreliable.

7.2 Mitigation Strategy

1. The desired velocity (V_{cmd}) is strictly clamped ($\pm MAX\_VELOCITY$) to ensure the control system never demands an unsafe tilt angle, mitigating saturation risk (F1).
2. **Frame Rejection:** The `calculateSensorFlow` function ignores corrupted frames by checking the SQUAL value (if `squal < 15`), preventing unreliable data from polluting the velocity estimates.
3. **Fusion Fallback:** The Z-axis estimation implements graceful degradation by automatically switching its height reference from the Laser ToF (primary) to the secondary Barometer when the laser signal is lost or deemed unreliable.

8 Limitations and Improvements

Limitations

Our system is currently limited by sensor data quality and processing latency:

1. Persistent Optic Flow Asymmetry : During flight testing, the Y-axis optical flow velocity (sensor flow y) consistently remained positive, even when the drone was moved in the negative direction.

2. The implemented Optic Flow velocity logic relies on raw sensor data, which is highly susceptible to noise and corruption. The final flight code uses a workaround (ignoring low SQUAL readings) that risks losing stability when sensor performance degrades.

Improvements

1. Closed-Loop Velocity Controller: Convert the inner Velocity stage from the current P-controller to a PI-Controller (Proportional-Integral). This would add an integral term to actively eliminate any steady state velocity errors (like slow drift from wind) that the P-term alone cannot fully counter.

2. Implement a failsafe where prolonged low SQUAL readings or continuous flow corruption automatically cause a slow, controlled descent to the ground, rather than simply ignoring the data.

References

[1] DronaAviation, "DA-InterIIT25-resources," GitHub repository. Available: https://github.com/DronaAviation/DA-InterIIT25-resources.

[2] D. Belfadel, J. Cain, D. Haessig, and C. Chibane, "Optical Flow and IMU Fusion for Drone Horizontal Velocity Control."

[3] N. Gageik, M. Strohmeier, and S. Montenegro, "An Autonomous UAV with an Optical Flow Sensor for Positioning and Navigation."

[4] M. Serebryakov, "Modelling and Simulation of Quadcopter Control Using Cascaded PID Controller in Matlab."

[5] MathWorks, "Model a Quadcopter Based on Parrot Minidrones," *Aerospace Blockset Documentation*, [Online]. Available: `https://in.mathworks.com/help/aeroblks/quadcopter-project.html`.

[6] YouTube, "Drone Simulation and Control by MATLAB," YouTube Playlist, [Online]. Available: `https://www.youtube.com/playlist?list=PLn8PRpmsu08oOLBVYYIwwN_nvuyUqEjrj`.