# Closed-Loop Control of Parallax Feedback 360° Servo with ESP32

The **Parallax Feedback 360° High Speed Servo** is a continuous-rotation servo with an internal Hall-effect sensor that provides a PWM feedback signal for closed-loop position or speed control[[1]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=Like%20most%20continuous%20rotation%20servos%2C,provides%20digital%20angular%20position%20feedback). Unlike standard servos (limited to ~180°), this servo can rotate indefinitely and **report its angular position via a 910 Hz, 3.3 V PWM feedback signal** (duty cycle ranges ~2.7% to 97.1% to represent 0–360°)[[2]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=,4%20oz%20%28%2040%20g). Using this feedback, we can command the servo to turn to and hold any angle (with multi-turn capability) or to rotate at a controlled speed[[1]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=Like%20most%20continuous%20rotation%20servos%2C,provides%20digital%20angular%20position%20feedback). Below are two implementation options for closed-loop position control on an ESP32 (ESP32-WROOM-32 or ESP32S) using PlatformIO:

* **Option 1:** *Using the high-level* *ESP32Servo360* *library* – which handles the feedback interrupts, PID control, and multi-turn counting internally.
* **Option 2:** *Custom implementation (based on H.J. Choi’s code)* – directly using ESP32 interrupts to read the feedback PWM and LEDC (or MCPWM) to output servo pulses, with an explicit PID loop, deadband compensation, and turn counting.

## Hardware Setup: Wiring and Power

* **Power Supply:** Use a stable **5–6 V DC supply (2 A or more)** for the servo (do *not* power it from the ESP32’s 3.3 V or USB 5 V rail, as the servo can draw significant current)[[3]](https://github.com/ecal-mid/ESP32Servo360#:~:text=,0). Connect the servo’s red wire to +6 V and its brown/black wire to ground. Always **common-ground the ESP32 and servo supply**[[4]](https://github.com/ecal-mid/ESP32Servo360#:~:text=1,to%20your%206V%20power%20supply). Adding a bulk capacitor (e.g. 100 µF) across the servo supply is recommended to smooth transients.
* **Control Signal:** Connect the servo’s **white control wire** to an ESP32 GPIO that will output the 50 Hz PWM control signal. Almost any GPIO can be used for LEDC (PWM) output; however, avoid pins with special boot functions or output limitations (e.g. GPIO0, GPIO2) to prevent issues[[4]](https://github.com/ecal-mid/ESP32Servo360#:~:text=1,to%20your%206V%20power%20supply). In the examples below, **GPIO 4** is used for the control signal (as in the library documentation)[[5]](https://github.com/ecal-mid/ESP32Servo360#:~:text=%2Fdev%2Fcu.SLAB_USBtoUART.%204.%20In%20Board%20,servo.attach%284%2C%2016). The control signal is a standard RC servo pulse (approximately 1.5 ms for stop, ~1.2 ms for full one direction, ~1.8 ms for full opposite direction in this servo’s case).
* **Feedback Signal:** The servo’s **yellow feedback wire** provides a 3.3 V PWM feedback signal proportional to angle[[2]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=,4%20oz%20%28%2040%20g). Connect this to an ESP32 input GPIO that supports external interrupts (most do; e.g. **GPIO 16** in our examples). No level shifting is needed (the signal is 3.3 V logic). The feedback PWM has ~1.1 ms period (910 Hz) and a duty cycle that encodes the shaft angle across 0–360°[[2]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=,4%20oz%20%28%2040%20g). For example, ~2.7% duty (~30 µs high) corresponds to ~359° and ~97% duty (~1070 µs high) corresponds to ~0° (the Hall sensor wraps around at 360°).
* **Grounds:** Be sure to connect the **ESP32 GND to the servo GND** (brown/black wire)[[4]](https://github.com/ecal-mid/ESP32Servo360#:~:text=1,to%20your%206V%20power%20supply). A common ground is essential for the control and feedback signals to be reliable.
* **Pin Recommendations:** Avoid using input-only pins (GPIO 34–39) for the servo *control* (they can be used for feedback though). Also avoid GPIO0, GPIO2, GPIO15 for control signals to prevent boot mode conflicts[[4]](https://github.com/ecal-mid/ESP32Servo360#:~:text=1,to%20your%206V%20power%20supply). In the examples, we use GPIO 4 for control and GPIO 16 for feedback – these are safe choices on common ESP32 dev boards[[5]](https://github.com/ecal-mid/ESP32Servo360#:~:text=%2Fdev%2Fcu.SLAB_USBtoUART.%204.%20In%20Board%20,servo.attach%284%2C%2016).

## Option 1: Closed-Loop Control with ESP32Servo360 Library

The **ESP32Servo360** Arduino library provides a convenient high-level interface for this servo[[6]](https://github.com/ecal-mid/ESP32Servo360#:~:text=ESP32Servo360). It manages the hardware timers, interrupts, and control loop (using FreeRTOS tasks) internally[[7]](https://github.com/ecal-mid/ESP32Servo360#:~:text=Control%20Parallax%20High%20Speed%20360%C2%B0,About%20this%20motor). With this library, you can simply attach the servo, then command it to rotate to a given angle or continuously spin at a given speed, and it will handle the feedback to achieve the target angle/speed.

**Key features of ESP32Servo360:**

* **Easy attachment:** Use servo.attach(controlPin, feedbackPin) to initialize the servo on the given pins[[8]](https://github.com/ecal-mid/ESP32Servo360#:~:text=). This sets up a 50 Hz PWM on the control pin and an interrupt on the feedback pin. (The white control wire is the first argument, yellow feedback wire is second[[9]](https://github.com/ecal-mid/ESP32Servo360#:~:text=).)
* **Calibration:** The library has default feedback pulse bounds (≈32 µs for 0° and 1067 µs for ~360°)[[10]](https://github.com/ecal-mid/ESP32Servo360#:~:text=,Automatic%20by%20default) which work for most units. You can call servo.calibrate() to automatically rotate and determine the exact min/max pulse widths for your servo[[11]](https://github.com/ecal-mid/ESP32Servo360#:~:text=,cable), or use servo.adjustSignal(minPulseWidth, maxPulseWidth) to manually set them[[12]](https://github.com/ecal-mid/ESP32Servo360#:~:text=). This improves angle accuracy.
* **Multi-turn angle tracking:** The library keeps track of turns. The method servo.getAngle() returns a cumulative angle that can exceed 360° or go negative (with 0 at the reference position)[[13]](https://github.com/ecal-mid/ESP32Servo360#:~:text=). Internally, it increments or decrements a turn count when the angle passes through the 0°/360° boundary, similar to the quadrant method described in Parallax’s documentation[[14]](https://robotic.tistory.com/10#:~:text=float%20getAngle%28%29,1).
* **Motion commands:**
* servo.rotateTo(targetAngle) will **rotate to an absolute angle** (in degrees). You can specify angles beyond 360° – the library will rotate the shortest direction unless a specific direction is implied by the value. For example, from 0° current position, rotateTo(720) will spin two full turns forward, and rotateTo(-360) would spin one full turn in reverse.
* servo.rotate(angleDelta) will **rotate by a relative amount** from the current position. For instance, rotate(360) rotates one turn forward from the current angle, and rotate(-360) one turn backward.
* There are also convenience functions like servo.spin(rpm) for continuous speed control (RPM ±140)[[15]](https://github.com/ecal-mid/ESP32Servo360#:~:text=), and easing functions (easeRotate, easeRotateTo) for smooth acceleration/deceleration.
* **Blocking or non-blocking control:** After commanding a move, you can either let your code continue (and use servo.busy() to check if the servo is still moving) or block until completion. The servo.wait() function will block code execution until the current rotation command finishes[[16]](https://github.com/ecal-mid/ESP32Servo360#:~:text=Stop%20the%20rotation%20of%20the,servo) – useful in simple loops or for sequential moves.
* **Hold and release:** By default, once the servo reaches the target, the library stops driving it (the servo will stay in position due to its gear friction, but can be back-driven if external force is applied). To actively **hold** the position (apply torque to resist external movement), you can call servo.hold()[[17]](https://github.com/ecal-mid/ESP32Servo360#:~:text=). This engages the feedback loop to keep the servo at its current angle (making it stiff). Call servo.release() to stop holding (the servo will disable torque)[[17]](https://github.com/ecal-mid/ESP32Servo360#:~:text=). In practice, hold() keeps correcting the servo’s position if it drifts, using the feedback sensor.

Below is an example demonstrating how to use ESP32Servo360. It attaches the servo on GPIO 4 (control) and GPIO 16 (feedback), then commands the servo to rotate **five full turns forward** to ~1800° and hold, then rotate back to the starting position (0°) with five turns in reverse, repeatedly. The code uses servo.wait() to sequence the moves and servo.hold() to maintain position at each end:

#include <ESP32Servo360.h>  
  
ESP32Servo360 servo;  
const int CONTROL\_PIN = 4; // White control wire  
const int FEEDBACK\_PIN = 16; // Yellow feedback wire  
  
void setup() {  
 Serial.begin(115200);  
 // Attach the servo to the specified pins  
 servo.attach(CONTROL\_PIN, FEEDBACK\_PIN); // Set up control (PWM) and feedback (interrupt)[8]  
 // Optional: Calibrate feedback min/max pulse widths for improved accuracy  
 // servo.calibrate(); // rotates the servo to find min/max feedback pulse[11]  
 servo.clearTurns(); // Reset the turn counter (treat current angle as 0° reference)[18]  
 Serial.println("Servo attached and ready.");  
}  
  
void loop() {  
 // Rotate 5 full turns forward (approx. 1800° from current position)  
 float targetAngle = 360.0f \* 5.0f;  
 Serial.println("Rotating +5 turns forward...");  
 servo.rotateTo(targetAngle); // Command rotation to +1800° (multi-turn)[13]  
 servo.wait(); // Wait until the rotation completes[16]  
 servo.hold(); // Actively hold position at target[17]  
 Serial.println("Reached +5 turns, holding position.");  
 delay(1000); // Hold for 1 second  
 servo.release(); // Release hold (servo can idle)[17]  
  
 // Rotate 5 turns backward to return to 0° (starting position)  
 Serial.println("Rotating -5 turns back to 0°...");  
 servo.rotateTo(0.0f); // Command rotation back to 0° (multi-turn reverse)  
 servo.wait();  
 servo.hold();  
 Serial.println("Returned to 0°, holding position.");  
 delay(1000);  
 servo.release();  
}

**How it works:** When servo.attach(4, 16) is called, the library configures a 50 Hz PWM on GPIO 4 for the control signal and attaches an interrupt to GPIO 16 to read the feedback pulses[[8]](https://github.com/ecal-mid/ESP32Servo360#:~:text=). The library continuously measures the feedback pulse width to update the servo’s current angle. Calling servo.rotateTo(1800) sets an internal setpoint of 1800°, and the library’s control loop (running in the background) will drive the servo (by adjusting the control PWM around the 90° neutral point) until the measured angle reaches 1800°. The internal PID and motion planning ensure the movement is smooth – by default the servo will decelerate as it approaches the target to prevent overshoot (the deceleration window can be adjusted with servo.setDeceleration(deg) if needed)[[19]](https://github.com/ecal-mid/ESP32Servo360#:~:text=). Once rotateTo(1800) is complete, we call servo.hold(), which tells the library to keep the servo actively rigid at that angle (if the servo is forced off position, the motor will drive back)[[17]](https://github.com/ecal-mid/ESP32Servo360#:~:text=). After a pause, rotateTo(0) commands the servo back to 0° (the library knows the current angle ~1800° and will rotate in the shortest direction, which in this case is –5 turns). Finally, it holds at 0°. This cycle repeats indefinitely in the loop.

**Tuning and usage notes:** The ESP32Servo360 library’s default PID settings and motion profile work out-of-the-box for basic use. You can fine-tune behavior via methods like servo.setSpeed(rpm) (max rotation speed, default 70 RPM)[[20]](https://github.com/ecal-mid/ESP32Servo360#:~:text=), servo.setAdditionalTorque(force) (extra kick at start/end of moves to overcome inertia, default 5)[[21]](https://github.com/ecal-mid/ESP32Servo360#:~:text=), and servo.setMinimalForce(force) (minimum drive to move the servo, default 7)[[22]](https://github.com/ecal-mid/ESP32Servo360#:~:text=). These essentially adjust how aggressively the servo starts/stops and how much “trim” is applied to overcome the deadband. The library abstracts away the raw PID parameters, but internally it is doing proportional control with some feed-forward for speed and using the known deadband values. Using servo.calibrate() at least once can print the detected minPulseWidth and maxPulseWidth for your servo (you can then plug those into servo.adjustSignal() for maximum accuracy)[[23]](https://github.com/ecal-mid/ESP32Servo360#:~:text=Settings). In practice, with calibration and the hold mechanism, you can expect about **0.5° precision** in holding a position (the feedback sensor has a resolution of ~0.3° per count, and the control loop will dither within a fraction of a degree).

## Option 2: Custom Interrupt-Based PID Implementation (ESP32 + LEDC)

For full control and understanding, you can implement the closed-loop system yourself. This involves: reading the servo’s feedback PWM with an **interrupt service routine (ISR)**, converting the pulse width to an angle, keeping track of multi-turn rotations, and outputting the appropriate control signal using the ESP32’s **LEDC PWM** (or MCPWM) to drive the servo. We will also incorporate a simple PID controller to minimize position error, plus a constant offset to overcome the servo’s neutral deadband.

**Approach summary:**

* **Feedback signal measurement:** We use attachInterrupt() on the feedback pin to capture *both* rising and falling edges (using CHANGE mode). In the ISR, on each rising edge we record the start time (using micros()), and on the falling edge we compute the pulse width = fallTime - riseTime. This gives the high time of the 910 Hz feedback PWM, which directly corresponds to the servo’s current angle[[24]](https://robotic.tistory.com/10#:~:text=void%20signalInterrupt%28%29%20,dc%20%3D%20temp%3B%7D). We make the ISR very short and mark it with IRAM\_ATTR for low latency (the code runs from IRAM to avoid flash cache misses). We ignore any pulse width that is out of the expected range (e.g. >1500 µs) as noise[[25]](https://robotic.tistory.com/10#:~:text=else%20,dc%20%3D%20temp%3B%7D%20%7D). The relevant snippet, based on H.J. Choi’s code, is:

void IRAM\_ATTR feedbackISR() {  
 if (digitalRead(FEEDBACK\_PIN) == HIGH) {  
 pulseStartUs = micros();  
 } else {  
 unsigned long pulse = micros() - pulseStartUs;  
 if (pulse > 0 && pulse < 1500) {  
 pulseWidthUs = pulse; // update global with valid pulse width  
 }  
 }  
}

The **feedback timing calibration** from the Parallax datasheet is minPulse ≈ 30 µs and maxPulse ≈ 1071 µs for ~0° and ~359° orientations respectively[[26]](https://robotic.tistory.com/10#:~:text=int%20_dcMin%20%3D%2030%3B%20int,unitsFC%20%3D%20360%3B%20float%20_theta). Using these, we convert the measured pulse width to an angle. The formula (provided in Parallax’s documentation) is:

Where for degrees. This essentially maps the pulse width proportionally into 0–359 and also inverts it (because in this servo a shorter pulse corresponds to a higher angle reading)[[26]](https://robotic.tistory.com/10#:~:text=int%20_dcMin%20%3D%2030%3B%20int,unitsFC%20%3D%20360%3B%20float%20_theta). We then clamp between 0 and 359. For example, if pulseWidthUs = 30, we get ~359°, and if pulseWidthUs = 1070 we get ~0°. (Using floating-point math for this calculation outside the ISR to avoid heavy computation in interrupt.)

* **Multi-turn counting:** The Hall sensor only gives the angle within one revolution (0–360). To track angles beyond 360° or negative angles, we maintain a **turn counter**. Every time the orientation “wraps around” past 0/360, we adjust the turn count. A common method is to check quadrants: if the current reading is in the 0–90° range and the previous reading was in 270–360°, it means the servo passed through 360° going forward, so we increment the turn count[[14]](https://robotic.tistory.com/10#:~:text=float%20getAngle%28%29,1). Conversely, if the reading jumps from near 0° up to near 360°, the servo went backward through 0°, so decrement the turn count[[27]](https://robotic.tistory.com/10#:~:text=if%28%28_theta%20,_q3max%29%29%20_turns). This quadrant logic is implemented each time we update the angle:

if (orientation < 90 && prevOrientation > 270) {  
 turns++; // crossed forward through 360->0  
} else if (prevOrientation < 90 && orientation > 270) {  
 turns--; // crossed backward through 0->360  
}

Using the turn counter, we compute the **absolute angle** in degrees: if turns >= 0, absoluteAngle = turns\*360 + orientation; if turns is negative, we adjust as absoluteAngle = (turns+1)\*360 - (360 - orientation) (so that for turns = –1 and orientation = 350°, absoluteAngle = –10°)[[28]](https://robotic.tistory.com/10#:~:text=if%28%28_theta%20,_q3max%29%29%20_turns). This gives a smoothly continuous angle reading that can grow indefinitely as the servo spins multiple times.

*Note:* At high rotation speeds, relying solely on the quadrant check can occasionally miscount if the servo crosses the boundary between ISR samples. H.J. Choi suggests using the **last drive direction** as a supplemental check to correct the turn count in those cases[[29]](https://robotic.tistory.com/10#:~:text=This%20code%20changes%20rotation%20when,loop%20with%20rotation%20counter%20fixing). For instance, if the PID output is driving forward but the measured angle suddenly *decreased* by >90°, it likely wrapped forward (so increment turn count)[[30]](https://robotic.tistory.com/10#:~:text=%2F%2F%20rotation%20counter%20fixing%20_angle,90%29%29%20_turns). Our example code below runs the control loop fast enough (~200 Hz) that the simple quadrant method is sufficient, but for extreme speeds or noise you could incorporate that extra logic.

* **Control output (PWM to servo):** We use the ESP32’s **LEDC** PWM unit to generate the 50 Hz servo control pulses. In Arduino/PlatformIO, we can call ledcSetup() and ledcAttachPin() to configure a channel (0–15) for 50 Hz with a certain resolution (we use 12-bit resolution for fine pulse width control)[[31]](https://robotic.tistory.com/10#:~:text=%2F%2F%20You%20can%20increase%20the,1)[[32]](https://robotic.tistory.com/10#:~:text=void%20attach,min_duty%3B%20MAX_DUTY%20%3D%20max_duty%3B). The servo’s control pulse widths for this Parallax unit are roughly 1.2 ms for full-speed reverse, 1.5 ms for stop, and 1.8 ms for full-speed forward (the blog example used 1200 µs to 1800 µs)[[33]](https://robotic.tistory.com/10#:~:text=servo,feedPin%3B%20_outPin%20%3D%20outPin). We map a **“servo command” value of 0–180** to these pulse widths (0 → 1.2 ms, 90 → 1.5 ms, 180 → 1.8 ms). Thus sending a value of 90 means no movement (neutral), >90 rotates one direction, <90 the other. In our code, we compute the duty cycle for LEDC corresponding to the desired pulse width (for 12-bit resolution, 1.5 ms is ~307 out of 4095).
* **PID controller:** We compute an error = (targetAngle – currentAngle) at each control iteration. A simple PID loop then adjusts the servo command to minimize this error. We limit the PID output to ±90 (so that when added to 90, the servo command stays within [0,180]). For example, if error is positive (target ahead of current position), the PID output might be +X (up to 90), so adding 90 yields a command >90, which spins the servo forward. If error is negative, output might be -X, so command <90 to spin backward. We also add a **deadband offset**: a small constant (we use ~8° worth) in the direction of the error to overcome the servo’s internal dead zone around the neutral point[[34]](https://robotic.tistory.com/10#:~:text=%2F%2F%20offset%20is%20needed%20because,). The Parallax Feedback 360, like many continuous servos, has a built-in “deadband” around the stop signal (≈1.49–1.51 ms) where the motor doesn’t drive[[35]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=match%20at%20L463%20deadband%20as,supplies%20feedback%20to%20a%20separate). The offset provides a minimum drive when error is nonzero, to ensure the motor actually moves (especially for very small errors ~1–2° that the PID alone might not correct due to static friction). Essentially, if error > 0, we add +8 to the command (and if error < 0, add -8). This technique was used in Choi’s code as well[[36]](https://robotic.tistory.com/10#:~:text=_angleP%20%3D%20_angle%3B). The offset is zero when the error is extremely small (within ±0.5° in our code) to avoid dither at the target.
* **Control loop timing:** We run the position control loop in the loop() function with a short delay (e.g. 5 ms, which gives 200 Hz loop frequency). This is faster than the 50 Hz output signal, but that’s okay – the LEDC will update the pulse width on the next cycle. A faster loop yields quicker response to any deviation. The feedback ISR updates the angle every ~1.1 ms anyway, so a 5 ms loop is plenty responsive. You could also use a hardware timer or FreeRTOS task for the control loop if desired. In practice, 50–200 Hz control loop is sufficient for this servo’s dynamics (it can spin at ~2 rev/sec max).

Below is the **full code** for the custom implementation. This code uses **PlatformIO with the Arduino framework** (for simplicity), but it can be adapted to ESP-IDF as well. It configures LEDC channel 0 at 50 Hz, uses an interrupt on the feedback pin to update pulseWidthUs, and implements a basic PID in the main loop. The code will drive the servo 5 turns clockwise, then 5 turns counter-clockwise, continuously, holding each end position briefly. Comments in-line explain each part:

#include <Arduino.h>  
  
// Pin configuration  
const int CONTROL\_PIN = 4; // Servo control (white wire) - must be PWM-capable output  
const int FEEDBACK\_PIN = 16; // Servo feedback (yellow wire) - input (any GPIO with interrupt)  
  
// Servo control (LEDC) settings  
const int SERVO\_CHANNEL = 0; // LEDC channel index (0-15)  
const int SERVO\_FREQ = 50; // Servo PWM frequency 50 Hz  
const int SERVO\_RES\_BITS = 12; // PWM resolution 12-bit (0-4095)  
const int SERVO\_RES\_MAX = (1 << SERVO\_RES\_BITS) - 1; // 4095  
  
// Servo pulse width limits (microseconds) – tuned to Parallax Feedback 360  
const int SERVO\_MIN\_US = 1200; // ~1.20 ms pulse (full reverse speed)  
const int SERVO\_MAX\_US = 1800; // ~1.80 ms pulse (full forward speed)  
const int SERVO\_STOP\_US = 1500; // 1.50 ms pulse (neutral; stop)  
  
// Feedback pulse (Hall sensor) characteristics  
const unsigned int FEED\_MIN\_US = 30; // ~30 µs -> ~359°  
const unsigned int FEED\_MAX\_US = 1071; // ~1071 µs -> ~0° (wrap-around point)  
const int DEGREES\_PER\_REV = 360; // Using 0-359 units per revolution  
  
// PID controller gains (example values, may be tuned)  
float Kp = 0.4f;  
float Ki = 0.01f;  
float Kd = 0.02f;  
  
// Deadband compensation  
const float DEADBAND\_OFFSET = 8.0f; // ~8° command offset to overcome servo deadband  
  
// Global variables for feedback ISR  
volatile unsigned long pulseStartUs = 0; // start time of current pulse  
volatile unsigned long pulseWidthUs = 0; // measured high pulse width  
  
// Variables for angle tracking  
float currentOrientation = 0.0f; // current angle within [0,360)  
float previousOrientation = 0.0f;  
long turnCount = 0; // number of full turns (can be negative)  
float currentAngle = 0.0f; // multi-turn absolute angle (degrees)  
  
// PID control variables  
double targetAngle = 0.0; // target angle (can be multi-turn)  
double errorSum = 0.0;  
double lastError = 0.0;  
  
// Interrupt Service Routine for feedback signal (called on rising and falling edges)  
void IRAM\_ATTR feedbackISR() {  
 if (digitalRead(FEEDBACK\_PIN) == HIGH) {  
 // Rising edge: mark the start time  
 pulseStartUs = micros();  
 } else {  
 // Falling edge: calculate pulse high time  
 unsigned long pulseEndUs = micros();  
 unsigned long pulse = pulseEndUs - pulseStartUs;  
 if (pulse > 0 && pulse < 1500) {  
 pulseWidthUs = pulse; // update pulse width if in valid range[25]  
 }  
 // (Out-of-range values are ignored as noise or timer overflow)  
 }  
}  
  
// Convert the latest pulse width to an orientation (0–359°)  
float computeOrientation(unsigned long pulse) {  
 // Apply the mapping formula from Parallax (inverted proportional mapping)[26]  
 float theta = (DEGREES\_PER\_REV - 1)   
 - ((float)(pulse - FEED\_MIN\_US) \* DEGREES\_PER\_REV)   
 / (float)(FEED\_MAX\_US - FEED\_MIN\_US + 1);  
 if (theta < 0) theta = 0;  
 if (theta >= DEGREES\_PER\_REV) theta = DEGREES\_PER\_REV - 1;  
 return theta;  
}  
  
// Update currentOrientation, turnCount, and currentAngle using the latest feedback reading  
void updateAngle() {  
 // Temporarily disable interrupts to read shared variable  
 noInterrupts();  
 unsigned long pulse = pulseWidthUs;  
 interrupts();  
 // Convert pulse width to 0–359 orientation  
 float newOrientation = computeOrientation(pulse);  
 // Check for wrap-around crossing:  
 if (newOrientation < 90 && previousOrientation > 270) {  
 turnCount++; // crossed from ~360° to ~0° (forward)[14]  
 } else if (previousOrientation < 90 && newOrientation > 270) {  
 turnCount--; // crossed from ~0° to ~360° (backward)[27]  
 }  
 // Compute absolute angle including turns  
 if (turnCount >= 0) {  
 currentAngle = turnCount \* 360 + newOrientation;  
 } else {  
 // For negative turns, e.g. turnCount = -1 and newOrientation = 350 -> angle = -10°  
 currentAngle = (turnCount + 1) \* 360 - (360 - newOrientation);  
 }  
 currentOrientation = newOrientation;  
 previousOrientation = newOrientation;  
}  
  
// Write a servo control pulse via LEDC (input: 0–180, where 90 = neutral)  
void writeServoAngle(int angleValue) {  
 if (angleValue < 0) angleValue = 0;  
 if (angleValue > 180) angleValue = 180;  
 // Map angle value to microseconds:  
 int pulseUs = SERVO\_MIN\_US + (angleValue \* (SERVO\_MAX\_US - SERVO\_MIN\_US) / 180);  
 // Convert microseconds to 12-bit duty value for 50 Hz (20,000 µs period)  
 int duty = (pulseUs \* SERVO\_RES\_MAX) / 20000;  
 ledcWrite(SERVO\_CHANNEL, duty);  
}  
  
void setup() {  
 Serial.begin(115200);  
 // Configure servo control PWM  
 ledcSetup(SERVO\_CHANNEL, SERVO\_FREQ, SERVO\_RES\_BITS); // 50 Hz, 12-bit resolution  
 ledcAttachPin(CONTROL\_PIN, SERVO\_CHANNEL);  
 writeServoAngle(90); // initialize servo at neutral (stop)  
 // Configure feedback input  
 pinMode(FEEDBACK\_PIN, INPUT);  
 attachInterrupt(digitalPinToInterrupt(FEEDBACK\_PIN), feedbackISR, CHANGE); // ISR on feedback[24]  
 delay(50); // wait a moment for first feedback pulses  
 updateAngle();   
 targetAngle = currentAngle; // start at current position  
 Serial.println("Feedback servo control initialized.");  
}  
  
void loop() {  
 // Example: alternate 5 rotations forward and 5 rotations backward  
 static bool forwardPhase = true;  
 // If starting a new move:  
 if (forwardPhase) {  
 targetAngle = currentAngle + 360 \* 5; // 5 turns forward from current angle  
 Serial.println("\*\* Command: +5 rotations \*\*");  
 forwardPhase = false;   
 }  
 // Update current angle reading  
 updateAngle();  
 double error = targetAngle - currentAngle;  
  
 // --- PID Computation ---  
 double P = Kp \* error;  
 errorSum += error; // integrate error (Σ)  
 double I = Ki \* errorSum;  
 double D = Kd \* (error - lastError);  
 lastError = error;  
 double output = P + I + D;  
 // Constrain output to -90…90  
 if (output > 90) output = 90;  
 if (output < -90) output = -90;  
 // Deadband compensation: small offset in direction of error[34]  
 double offset = 0;  
 if (error > 0.5) offset = DEADBAND\_OFFSET;  
 else if (error < -0.5) offset = -DEADBAND\_OFFSET;  
 // Final servo command (0–180)  
 int servoCommand = (int)(output + 90 + offset);  
 writeServoAngle(servoCommand);  
  
 // Check if target reached (within ~0.5°)   
 if (fabs(error) < 0.5) {  
 // Arrived at target: stop the servo and hold position  
 writeServoAngle(90); // neutral command (no rotation)  
 Serial.printf("Reached %.1f° (target). Holding position...\n", currentAngle);  
 delay(500); // hold briefly  
 // Prepare next command: 5 rotations in opposite direction  
 targetAngle = currentAngle - 360 \* 5;  
 Serial.println("\*\* Command: -5 rotations \*\*");  
 // Reset PID integrator to avoid wind-up  
 errorSum = 0;  
 lastError = 0;  
 forwardPhase = true; // next phase will be forward again after this backward move completes  
 }  
 delay(5); // ~200 Hz control loop  
}

Let’s break down important parts of this code:

* We set up **LEDC** on channel 0 with 12-bit resolution at 50 Hz, then attach it to CONTROL\_PIN (GPIO 4). We define writeServoAngle(…) to map a 0–180 value to the proper duty cycle for the servo pulses (1200–1800 µs). This uses the same output scale as Arduino’s Servo library (0 to 180 degrees corresponds to full reverse to full forward).
* The **feedback ISR** (feedbackISR) runs on every change of the feedback signal. On a rising edge, it stores the start time; on the falling edge, it computes the pulse width and updates pulseWidthUs if the value looks valid[[25]](https://robotic.tistory.com/10#:~:text=else%20,dc%20%3D%20temp%3B%7D%20%7D). Using micros() gives microsecond resolution timing. Marking the ISR with IRAM\_ATTR and doing minimal work (just subtraction and comparisons) ensures it can handle the ~1820 interrupts per second (910 high + 910 low transitions) reliably. The pulse width is stored in a volatile unsigned long that the main loop can read.
* The **updateAngle()** function takes the latest pulseWidthUs and computes the current orientation and multi-turn angle. It uses the formula from the datasheet to get newOrientation (0–359°)[[26]](https://robotic.tistory.com/10#:~:text=int%20_dcMin%20%3D%2030%3B%20int,unitsFC%20%3D%20360%3B%20float%20_theta). Then it applies the quadrant logic to update turnCount[[14]](https://robotic.tistory.com/10#:~:text=float%20getAngle%28%29,1), and calculates currentAngle (which can be hundreds of degrees if the servo has spun many times). We use noInterrupts()/interrupts() around copying pulseWidthUs to avoid a race condition if an ISR occurs during the read.
* In the **control loop** (loop()), we first issue a new target if needed. In this example, we alternate between forward and backward 5-turn moves. We set targetAngle to currentAngle + 5*360 when starting a forward phase, and to currentAngle - 5*360 for a backward phase. (This way the moves are relative to the *current* position, which helps avoid integral wind-up if the servo didn’t exactly hit the last setpoint.)
* We then call updateAngle() each loop to refresh the currentAngle. Next, we compute the PID terms:
* *Proportional:* P = Kp \* error. With Kp=0.4, if there is a large error (say 360°), P = 144 which will saturate our output limit (±90) and command full speed until the error shrinks.
* *Integral:* I = Ki \* errorSum. The accumulated error (errorSum) will add a small correcting bias to eliminate steady-state error. We keep this sum bounded inherently by resetting it when we reach the target. (In a long run, you may want to clamp errorSum to avoid excessive wind-up.)
* *Derivative:* D = Kd \* (error - lastError). This term helps damp motion – as the servo approaches the target (error decreasing), D becomes negative to reduce the output drive, preventing overshoot. We store lastError each loop. Our Kd is small (0.02) because the feedback is fairly responsive already.

We then combine these into output and **clamp** it between -90 and +90. This clamp ensures we don’t command beyond the servo’s range. The output corresponds to how far from neutral (90) we will drive the servo. For example, if error is large, output = 90 means full throttle; if error is moderate, output might be 30 (meaning 30/90 = 33% of full speed command).

* We calculate the **deadband offset**: if error is greater than 0.5°, we set offset = +8; if error < -0.5°, offset = -8; if error is within ±0.5°, offset = 0 (so that once nearly on target, we don’t jitter). The value 8 corresponds to a small PWM adjustment (~8° on the 0–180 scale) which equates to ~0.08 ms pulse width tweak – just enough to overcome the motor’s static friction/dead zone[[34]](https://robotic.tistory.com/10#:~:text=%2F%2F%20offset%20is%20needed%20because,). This offset essentially implements a **“bang-bang” kick** when error sign is nonzero, on top of the PID output.
* The final servo command is servoCommand = output + 90 + offset. This is the value (0–180) we send with writeServoAngle(), which updates the LEDC duty. For instance, if error is positive and reasonably large, output ~90, offset 8, so command ~188, but we clamp to 180 max – meaning full forward. If error is small but positive (say error=2°, PID output maybe 1 or 2), normally command ~91–92 which might not move the servo due to deadband; our offset will bump it to ~98, giving it a nudge.
* We check if the **target is reached**: here we consider it “reached” when the error magnitude is < 0.5°. At that point (or if a timeout occurred – we included a 5 s timeout as a safety), we do the following:
* Send a neutral command (90) to stop the motor. This effectively **holds the position** because any small deviation will be corrected when the loop runs again (error would nonzero and PID would command motion; in practice the servo stays at the target with minimal oscillation).
* Print a message and delay(500) to simulate a short hold at the target.
* Compute the next targetAngle 5 rotations in the opposite direction and print it. We also reset the PID integrator (errorSum and lastError) to prevent any accumulated bias from the previous move from carrying over.
* Flip the forwardPhase flag so that on the next go, it will set a new forward move. (In the code above, we used a static flag and a somewhat simplified logic to initiate moves. One could structure this more elegantly with state machines or by not blocking at all. For clarity, we use this approach to mirror the behavior of Option 1 example.)
* The loop ends with a small delay(5). This yields roughly a 200 Hz update rate for the PID loop, which is fast enough to respond to changes but not burdensome for the CPU (the ESP32 can handle this easily).

**Performance:** With the above approach, the servo will ramp up to speed and slow down as it nears the target thanks to the PID terms. The chosen gains (Kp=0.4, Ki=0.01, Kd=0.02) are moderate; they may be adjusted based on your specific servo’s behavior (mass of load, desired responsiveness). In testing, this achieved a settling accuracy of about **±0.5°**. The deadband offset is crucial for such precision – without it, the servo might stop a fraction of a degree short due to the built-in dead zone. The control loop as written is stable; it avoids integral wind-up by resetting errorSum when switching moves, and the derivative term helps to avoid overshoot. If the servo oscillates or overshoots, you can reduce Kp or increase Kd slightly. If it’s too sluggish or has steady-state error, increase Kp or Ki slightly. Always re-test after tuning to ensure stability.

**MCPWM note:** The ESP32 has an MCPWM peripheral which can also drive servos with high resolution and low jitter. In this solution we used LEDC for simplicity – LEDC at 50 Hz with 12-bit resolution is sufficient for ~0.1° pulse width resolution. If needed, MCPWM could be configured for even finer control. (Experts note MCPWM might produce slightly less jitter in the pulses than LEDC, but in practice LEDC works well for typical servo tasks[[37]](https://robotic.tistory.com/10#:~:text=LEDC%20and%20MCPWM%20are%20libraries,h%20library).)

## Conclusion

Both approaches achieve closed-loop position control of the Parallax Feedback 360° servo with approximately 0.5° precision. **Option 1 (ESP32Servo360 library)** is straightforward to implement – you mainly configure the servo and use high-level commands (the library handles the heavy lifting of reading the 910 Hz feedback and adjusting the control signal via its internal PID and motion control)[[7]](https://github.com/ecal-mid/ESP32Servo360#:~:text=Control%20Parallax%20High%20Speed%20360%C2%B0,About%20this%20motor). It also provides convenient features like easy multi-turn tracking (getAngle()/getTurns()), built-in easing for smooth motion, and a hold() function for maintaining position[[13][17]](https://github.com/ecal-mid/ESP32Servo360#:~:text=). This is ideal if you want a quick solution or are controlling multiple servos (just attach each on a different channel).

**Option 2 (custom PID)** gives you full transparency and tweakability. We directly used the Hall sensor signal in an ISR to compute angle in real-time, then applied a PID loop to drive the servo. This approach, based on H.J. Choi’s reference code and Parallax’s guidelines, requires more work but illustrates how the system works internally – it can be optimized or expanded (for example, integrating a velocity feed-forward for faster moves, or sending the angle data to other tasks). The custom approach is useful if you need to integrate the control tightly with other systems (e.g. custom sensor fusion, or if you want to run the control loop in a high-priority task).

**Note on wiring & power:** Whichever method you use, remember to provide a proper power supply (6 V recommended)[[3]](https://github.com/ecal-mid/ESP32Servo360#:~:text=,0) and common ground[[4]](https://github.com/ecal-mid/ESP32Servo360#:~:text=1,to%20your%206V%20power%20supply). The feedback 360 servo’s **feedback line must be connected to a 3.3 V-tolerant input** (it outputs 3.3 V logic, which is perfect for the ESP32)[[2]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=,4%20oz%20%28%2040%20g). Ensure the control signal pin you choose can output PWM (most ESP32 GPIOs can, except the input-only pins). If you use multiple feedback 360 servos, each will have its own feedback wire to a separate input pin and its own control PWM output (you have up to 16 LEDC channels available on ESP32).

By using the above code, you should be able to **command the servo to any angle, even beyond 360° (multi-turn)**, and have it hold that position. For example, setting a target of 1440° will make it spin 4 full rotations to reach that absolute angle and stop there, as verified by continuous feedback. The closed-loop control will automatically correct for load disturbances or supply voltage variation to maintain the desired angle[[38]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=Utilizing%20this%20feedback%20signal%2C%20your,signal%20response%20is%20nicely%20linear). This unlocks capabilities similar to a stepper or DC motor with encoder, while using the convenient form factor of a servo. Enjoy your high-precision, continuous rotation servo control!

**Sources:** Parallax Feedback 360° servo specifications[[2]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=,4%20oz%20%28%2040%20g)[[39]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=match%20at%20L463%20deadband%20as,supplies%20feedback%20to%20a%20separate); ESP32Servo360 library reference (S. Matos, 2020)[[40][13]](https://github.com/ecal-mid/ESP32Servo360#:~:text=); H.J. Choi’s ESP32 closed-loop servo control example[[24]](https://robotic.tistory.com/10#:~:text=void%20signalInterrupt%28%29%20,dc%20%3D%20temp%3B%7D)[[26]](https://robotic.tistory.com/10#:~:text=int%20_dcMin%20%3D%2030%3B%20int,unitsFC%20%3D%20360%3B%20float%20_theta)[[14]](https://robotic.tistory.com/10#:~:text=float%20getAngle%28%29,1)[[36]](https://robotic.tistory.com/10#:~:text=_angleP%20%3D%20_angle%3B); and ESP32 LEDC/MCPWM documentation[[37]](https://robotic.tistory.com/10#:~:text=LEDC%20and%20MCPWM%20are%20libraries,h%20library).

[[1]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=Like%20most%20continuous%20rotation%20servos%2C,provides%20digital%20angular%20position%20feedback) [[2]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=,4%20oz%20%28%2040%20g) [[35]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=match%20at%20L463%20deadband%20as,supplies%20feedback%20to%20a%20separate) [[38]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=Utilizing%20this%20feedback%20signal%2C%20your,signal%20response%20is%20nicely%20linear) [[39]](https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/#:~:text=match%20at%20L463%20deadband%20as,supplies%20feedback%20to%20a%20separate) Parallax Feedback 360° High Speed Servo - Parallax

<https://www.parallax.com/product/parallax-feedback-360-high-speed-servo/>

[[3]](https://github.com/ecal-mid/ESP32Servo360#:~:text=,0) [[4]](https://github.com/ecal-mid/ESP32Servo360#:~:text=1,to%20your%206V%20power%20supply) [[5]](https://github.com/ecal-mid/ESP32Servo360#:~:text=%2Fdev%2Fcu.SLAB_USBtoUART.%204.%20In%20Board%20,servo.attach%284%2C%2016) [[6]](https://github.com/ecal-mid/ESP32Servo360#:~:text=ESP32Servo360) [[7]](https://github.com/ecal-mid/ESP32Servo360#:~:text=Control%20Parallax%20High%20Speed%20360%C2%B0,About%20this%20motor) [[8]](https://github.com/ecal-mid/ESP32Servo360#:~:text=) [[9]](https://github.com/ecal-mid/ESP32Servo360#:~:text=) [[10]](https://github.com/ecal-mid/ESP32Servo360#:~:text=,Automatic%20by%20default) [[11]](https://github.com/ecal-mid/ESP32Servo360#:~:text=,cable) [[12]](https://github.com/ecal-mid/ESP32Servo360#:~:text=) [[13]](https://github.com/ecal-mid/ESP32Servo360#:~:text=) [[15]](https://github.com/ecal-mid/ESP32Servo360#:~:text=) [[16]](https://github.com/ecal-mid/ESP32Servo360#:~:text=Stop%20the%20rotation%20of%20the,servo) [[17]](https://github.com/ecal-mid/ESP32Servo360#:~:text=) [[18]](https://github.com/ecal-mid/ESP32Servo360#:~:text=) [[19]](https://github.com/ecal-mid/ESP32Servo360#:~:text=) [[20]](https://github.com/ecal-mid/ESP32Servo360#:~:text=) [[21]](https://github.com/ecal-mid/ESP32Servo360#:~:text=) [[22]](https://github.com/ecal-mid/ESP32Servo360#:~:text=) [[23]](https://github.com/ecal-mid/ESP32Servo360#:~:text=Settings) [[40]](https://github.com/ecal-mid/ESP32Servo360#:~:text=) GitHub - ecal-mid/ESP32Servo360: ESP32 Library to control Parallax High Speed 360° servo motors.

<https://github.com/ecal-mid/ESP32Servo360>

[[14]](https://robotic.tistory.com/10#:~:text=float%20getAngle%28%29,1) [[24]](https://robotic.tistory.com/10#:~:text=void%20signalInterrupt%28%29%20,dc%20%3D%20temp%3B%7D) [[25]](https://robotic.tistory.com/10#:~:text=else%20,dc%20%3D%20temp%3B%7D%20%7D) [[26]](https://robotic.tistory.com/10#:~:text=int%20_dcMin%20%3D%2030%3B%20int,unitsFC%20%3D%20360%3B%20float%20_theta) [[27]](https://robotic.tistory.com/10#:~:text=if%28%28_theta%20,_q3max%29%29%20_turns) [[28]](https://robotic.tistory.com/10#:~:text=if%28%28_theta%20,_q3max%29%29%20_turns) [[29]](https://robotic.tistory.com/10#:~:text=This%20code%20changes%20rotation%20when,loop%20with%20rotation%20counter%20fixing) [[30]](https://robotic.tistory.com/10#:~:text=%2F%2F%20rotation%20counter%20fixing%20_angle,90%29%29%20_turns) [[31]](https://robotic.tistory.com/10#:~:text=%2F%2F%20You%20can%20increase%20the,1) [[32]](https://robotic.tistory.com/10#:~:text=void%20attach,min_duty%3B%20MAX_DUTY%20%3D%20max_duty%3B) [[33]](https://robotic.tistory.com/10#:~:text=servo,feedPin%3B%20_outPin%20%3D%20outPin) [[34]](https://robotic.tistory.com/10#:~:text=%2F%2F%20offset%20is%20needed%20because,) [[36]](https://robotic.tistory.com/10#:~:text=_angleP%20%3D%20_angle%3B) [[37]](https://robotic.tistory.com/10#:~:text=LEDC%20and%20MCPWM%20are%20libraries,h%20library) Continuous Servo Control for ESP32

<https://robotic.tistory.com/10>