ChatGPT

ESP32 Motor Control and Interface Guide (Late 2025 Edition)

This guide provides a comprehensive walkthrough for using the ESP32 microcontroller with various motors and related hardware. We cover DC motors, servo motors, and stepper motors (e.g. NEMA 17 and 28BYJ-48), along with common driver modules (L298N, L9110S, A4988, DRV8825, ULN2003). Integration of peripheral devices like rotary encoders, potentiometers, and push buttons is also discussed. We include reliable, well-commented code examples (both simple and FreeRTOS-based) and recommend up-to-date libraries for motor control and peripheral management. Finally, we outline development workflow steps in the Arduino IDE and command-line, including how to install ESP32 support, libraries, and perform firmware flashing via USB, OTA, or using CLI tools like `esptool.py`. All content reflects current ESP32 best practices as of late 2025.

Motor Types and Control Examples

DC Motors (Brushed)

DC motors are simple two-wire motors that spin when a voltage is applied. To **reverse** the rotation, the polarity of the voltage must be flipped. In practice, an **H-bridge** driver circuit is used to electronically switch the polarity, allowing bidirectional control [1]. The motor's **speed** is controlled by adjusting the average voltage via PWM (Pulse Width Modulation) [2]. The ESP32 can generate PWM signals on any GPIO pin using its LEDC module or `analogWrite()` function (0-255 duty by default, where 255 = 100% duty cycle) [3].

Because the ESP32's GPIO cannot drive a motor directly (the motor draws more current and typically a higher voltage), a driver like the L298N or L9110S H-bridge is used as an interface. The driver takes low-current control signals from the ESP32 and outputs higher-current drive to the motor [4]. For a single DC motor, you typically need two digital output pins for **direction** (INA and INB), and one PWM-capable pin for **speed** (enable pin on the driver). For example, the L298N driver has pins IN1 and IN2 to control a motor's direction, and ENA (enable) to control speed via PWM [5]. By setting IN1 high/IN2 low or vice versa, you select the rotation direction, and by applying a PWM on ENA, you modulate the speed.

Figure: Wiring diagram of an ESP32 driving a pair of DC motors via an L298N dual H-Bridge module (two IN pins and one EN PWM pin per motor) [5] [6]. The ESP32's GPIOs provide logic signals to the driver, which in turn powers the motors from an external supply (note the shared ground). This setup allows control of both **direction** (through IN1..IN4 signals) and **speed** (through ENA/ENB PWM inputs) [5].

Below is a simple example of controlling a single DC motor with an ESP32 and an H-bridge driver. The code uses one GPIO (`IN1`) for forward/reverse and another (`IN2`) for the opposite direction input, plus a PWM output (`EN`) for speed control. It gradually ramps the motor speed up and down and then reverses direction:

```
// DC Motor control example with ESP32 and H-bridge (e.g., L298N or L9110S)
```

```
// Pin assignments (change these to your setup)
const int IN1 = 27;    // H-bridge input 1
const int IN2 = 26;    // H-bridge input 2
const int EN  = 14;    // H-bridge enable pin (PWM)

// Setup function
void setup() {
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(EN, OUTPUT);
  // Optional: configure PWM frequency and resolution if needed
  analogWriteFrequency(EN, 1000);   // 1 kHz PWM frequency for motor speed
control
  analogWriteResolution(EN, 8);     // 8-bit resolution (0-255 duty)
}

// Helper function to spin motor in a given direction at a given speed
(0-255)
void driveMotor(bool forward, uint8_t speed) {
  digitalWrite(IN1, forward ? HIGH : LOW);
  digitalWrite(IN2, forward ? LOW  : HIGH);
  analogWrite(EN, speed);
}

// Loop function: ramp speed up and down, then reverse
void loop() {
  // Ramp up speed forward
  for (int duty = 0; duty <= 255; duty += 5) {
    driveMotor(true, duty);
    delay(20);
  }
  // Ramp down speed forward
  for (int duty = 255; duty >= 0; duty -= 5) {
    driveMotor(true, duty);
    delay(20);
  }
  delay(500);

  // Ramp up speed in reverse
  for (int duty = 0; duty <= 255; duty += 5) {
    driveMotor(false, duty);
    delay(20);
  }
  // Ramp down speed in reverse
  for (int duty = 255; duty >= 0; duty -= 5) {
    driveMotor(false, duty);
    delay(20);
  }
  delay(500);
}
```

Explanation: In this code, `driveMotor(forward, speed)` sets the H-bridge inputs such that if `forward` is true, `IN1` is HIGH and `IN2` LOW (one polarity); if false, it inverts them to reverse polarity. The `EN` pin is driven with PWM via `analogWrite()`, where a value of 0 means motor off and 255 means full speed. We ramp the duty cycle up and down to smoothly accelerate and decelerate the motor. Using a 1 kHz PWM frequency is typical for DC motors; the ESP32's LEDC can support much higher frequencies as well [7]. At low duty cycles, you may hear the motor buzz due to the rapid on-off switching – this is normal PWM behavior [8]. Always remember to provide a suitable external power supply for the motor driver (do not power motors from the ESP32's 3.3V). Also ensure the grounds of the ESP32 and the motor power supply are common.

Servo Motors

Servo motors are closed-loop actuators with an internal control circuit that moves the output shaft to a given angle (usually between 0° and 180°). Standard hobby servos have three wires: **power** (typically 5V, red), **ground** (black or brown), and **signal** (yellow/orange/white) [9]. The ESP32 can control a servo by generating a 50 Hz PWM signal where the pulse width encodes the desired angle (e.g., ~1 ms pulse for 0°, ~2 ms for 180°). The ESP32's LEDC PWM controller or dedicated servo libraries can produce this signal on any GPIO pin (avoid strapping pins like GPIO 9-11 that are used for flash on some boards) [10].

To make controlling servos easier, you can use the **ESP32Servo** library, which mirrors the classic Arduino `Servo.h` API but is implemented using the ESP32's PWM capabilities [11]. This library allows up to 16 servo objects on different pins, and it abstracts the 50 Hz timing for you. After attaching a servo to a pin with `servo.attach(pin)`, you can simply call `servo.write(angle)` to set an angle between 0-180 degrees.

Wiring: Small servos (like SG90 or TowerPro 9g micro servos) can be powered from the ESP32's 5V (VIN) pin if your board is powered via USB, but for multiple or larger servos it's recommended to use an external 5V supply (with common ground) to avoid brownouts. Connect the servo's signal pin to an ESP32 PWM-capable GPIO (almost any pin), its power to 5V, and ground to the ESP32 ground.

Figure: Example hookup of an ESP32 to a small SG90 servo motor. The servo's three wires are connected to the ESP32: brown to GND, red to 5V (VIN), and orange to a signal pin (GPIO 13 in this example). In general, any free GPIO can be used for the servo signal (here we avoid GPIOs 9-11 due to ESP32 flash interface constraints) [10]. An external 5V supply should be used if driving many or high-torque servos.

Below is a code example that uses the ESP32Servo library to sweep a servo back and forth. This demonstrates basic servo control with the ESP32:

```
#include <ESP32Servo.h>

Servo myServo;
int servoPin = 13;  // GPIO to which servo is connected

void setup() {
  Serial.begin(115200);
  myServo.attach(servoPin);        // Attach the servo to the pin
  // Optionally, specify min and max pulse widths if calibration is needed:
  // myServo.attach(servoPin, 500, 2400);  // pulse width in microseconds for
0° and 180°
```

```
}

void loop() {
  // Sweep from 0 to 180 degrees
  for (int angle = 0; angle <= 180; angle += 5) {
    myServo.write(angle);
    delay(20);  // small delay to allow the servo to reach position
  }
  // Sweep back from 180 to 0 degrees
  for (int angle = 180; angle >= 0; angle -= 5) {
    myServo.write(angle);
    delay(20);
  }
  delay(500);  // pause between sweeps
}
```

Explanation: We include the ESP32Servo library and create a `Servo` object. In `setup()`, we attach it to `servoPin` (GPIO 13). The library by default generates the proper 50 Hz waveform with 1-2 ms pulses corresponding to 0-180°. In the loop, we simply increment the angle and write it to the servo, creating a sweep motion. A small delay is used between writes to control speed of motion (20 ms roughly yields a smooth motion given servo mechanics). The `Servo.write()` call internally maps the angle to the correct pulse width for the ESP32's PWM. This example uses the default calibration; if your servo's end stops are slightly off, you can specify custom min/max pulse widths in `attach()`. Remember that servo power draw can be significant when moving or holding position under load – ensure your 5V supply can provide adequate current (typical small servo can draw 100-250 mA or more under load).

Stepper Motors (Bipolar & Unipolar)

Stepper motors rotate in discrete steps, making them suitable for precise position control. They come in two main types: **unipolar** (e.g. 28BYJ-48, 5 wires) and **bipolar** (e.g. NEMA 17, 4 wires). The ESP32 can drive stepper motors using appropriate driver hardware. Steppers require activating coils in sequence; a driver or controller is used to simplify this. We'll consider two common scenarios:

- **28BYJ-48 + ULN2003:** The 28BYJ-48 is a 5V unipolar stepper often paired with a ULN2003 Darlington transistor driver board. The ULN2003 has 4 input pins that directly correspond to the four coils of the stepper [12]. To rotate the motor, the ESP32 must energize these inputs in a repeating sequence (either full-step or half-step sequence). No PWM is needed; instead, you set coils HIGH/LOW in the correct order with short delays for stepping speed.

- **NEMA 17 + A4988/DRV8825:** NEMA 17 is a class of bipolar stepper motors commonly used in CNC, 3D printers, etc. They require a bipolar driver. Modules like **A4988** or **DRV8825** are step/direction drivers that handle the coil driving and current limiting internally. You control these by providing a digital pulse on the STEP pin for each micro-step and a HIGH/LOW on the DIR pin for direction. These drivers make controlling a stepper much easier by internally sequencing the coils – "you only need to control two simple pins: one tells the motor to take a step, and the other tells it which direction to move [13]." The A4988 supports microstepping in 1/2, 1/4, 1/8, 1/16 steps, while the DRV8825 can go up to 1/32 microsteps and also tolerates higher supply voltage (45V vs 35V for A4988) [14]. Both accept 3.3V logic signals from the ESP32.

Example 1: Driving a NEMA17 (bipolar stepper) with an A4988 – We connect the A4988 driver's STEP and DIR pins to two ESP32 GPIOs (plus an optional ENable pin). The motor's two coils connect to the A4988 outputs (1A/1B and 2A/2B), and we provide a motor supply (e.g. 12V) to the driver. The A4988's logic Vdd can be 3.3V (it accepts 3–5.5V logic). A common setup is to tie the driver's `EN` low (enable always on), `RESET` and `SLEEP` pins tied together (so it's not sleeping), and microstep selection pins MS1,MS2,MS3 set as needed (e.g. all LOW for full-step). A crucial detail is to add a large electrolytic capacitor (≥47µF) across the driver's motor supply (VMOT and GND) to prevent voltage spikes [15] which could otherwise damage the driver.

To move the stepper, the ESP32 pulses the STEP pin at the desired rate. Each pulse causes the motor to advance by one microstep (or full step, depending on microstep setting). The direction pin sets the rotation direction (HIGH for clockwise, LOW for counterclockwise, for example) [16]. Below is a basic code to spin the motor one revolution in each direction:

```
// Stepper motor control with A4988 driver and ESP32
const int STEP_PIN = 14;    // Step pulse pin
const int DIR_PIN = 12;     // Direction pin
const int STEPS_PER_REV = 200;  // Motor steps per revolution (e.g. 200 for
1.8° step motor in full-step)

void setup() {
  pinMode(STEP_PIN, OUTPUT);
  pinMode(DIR_PIN, OUTPUT);
  digitalWrite(DIR_PIN, LOW);
}

void loop() {
  // Rotate one full revolution clockwise
  digitalWrite(DIR_PIN, HIGH);            // set direction CW
  for (int i = 0; i < STEPS_PER_REV; ++i) {
    digitalWrite(STEP_PIN, HIGH);
    delayMicroseconds(1000);              // pulse width (1 ms high)
    digitalWrite(STEP_PIN, LOW);
    delayMicroseconds(1000);              // pulse interval (controls speed)
  }
  delay(1000);

  // Rotate one full revolution counter-clockwise
  digitalWrite(DIR_PIN, LOW);             // set direction CCW
  for (int i = 0; i < STEPS_PER_REV; ++i) {
    digitalWrite(STEP_PIN, HIGH);
    delayMicroseconds(1000);
    digitalWrite(STEP_PIN, LOW);
    delayMicroseconds(1000);
  }
  delay(1000);
}
```

Explanation: This code toggles the STEP pin with 1 millisecond high, 1 ms low, resulting in a 500 Hz step rate (since each full cycle is 2 ms). At full-step mode and 200 steps per rev, this makes roughly 2.5

revolutions per second. We alternate the `DIR_PIN` to change direction between loops. You can adjust `delayMicroseconds()` to change speed – smaller delays yield faster rotation. Note that very short delays (high step rates) might require careful timing (at some point the motor can't keep up or torque drops). For higher step rates, consider using a dedicated library or hardware timer to generate pulses rather than `delayMicroseconds` busy-waiting. The ESP32 is quite fast, though, and this software loop can handle moderate speeds for one motor. The A4988 internally handles the current limiting and coil sequencing; "it takes care of the complex driving (like micro-stepping) and reduces the burden on the ESP32" [17]. Always set the current limit on the driver via its potentiometer according to your motor's coil rating to avoid overheating.

Alternative: You can use libraries like **AccelStepper** (Arduino library by Mike McCauley) or **StepperDriver** (by laurb9 on GitHub) to control step/dir drivers. These libraries provide acceleration control, non-blocking movement, and other conveniences. For example, using AccelStepper, one could initialize with `AccelStepper stepper(AccelStepper::DRIVER, STEP_PIN, DIR_PIN);` and then set speeds and move targets easily. This abstracts the pulsing into a simple `stepper.run()` call in the loop and handles ramping etc. For production use where smooth motion is needed, such libraries are recommended.

Example 2: Driving a 28BYJ-48 (unipolar stepper) with ULN2003 – The ULN2003 driver board has four inputs (often labeled IN1–IN4) that drive the stepper's coils. The typical stepping sequence for a 4-phase unipolar motor is either a full-step sequence (one coil on at a time) or half-step (alternating single and dual coil activation for smoother motion). For the 28BYJ-48, a common sequence (half-step) is 8 steps long: e.g. `IN1->IN1+IN2->IN2->IN2+IN3->IN3->IN3+IN4->IN4->IN4+IN1` and repeat. Each step in this sequence moves the motor by 5.625°/64 (in half-step mode) and due to the internal 64:1 gear reduction, it actually needs 2048 half-steps for one output shaft revolution [18] [19].

For brevity, here's a snippet that energizes the coils in a simple full-step sequence (4 steps) which is enough to drive the motor (though with a bit less smoothness than half-step):

```
// Unipolar stepper (28BYJ-48) control with ULN2003 driver
const int IN1 = 15;
const int IN2 = 2;
const int IN3 = 4;
const int IN4 = 16;
int stepIndex = 0;

// Full-step sequence for 28BYJ-48 (each element is a 4-bit pattern for
IN1..IN4)
const int stepSequence[4] = {
  0b1000, // energize IN1
  0b0010, // energize IN2
  0b0100, // energize IN3
  0b0001  // energize IN4
};

void setup() {
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(IN3, OUTPUT);
  pinMode(IN4, OUTPUT);
```

```
}

void singleStep(bool clockwise) {
  // Output the bit pattern to the 4 inputs
  int seq = stepSequence[stepIndex];
  digitalWrite(IN1, seq & 0x8 ? HIGH : LOW);
  digitalWrite(IN2, seq & 0x4 ? HIGH : LOW);
  digitalWrite(IN3, seq & 0x2 ? HIGH : LOW);
  digitalWrite(IN4, seq & 0x1 ? HIGH : LOW);
  // Increment or decrement step index
  stepIndex = (stepIndex + (clockwise ? 1 : -1) + 4) % 4;
}

void loop() {
  // rotate stepper continuously clockwise at ~10 rpm
  singleStep(true);
  delay(2);  // step delay in ms (adjust for speed; ~2ms per step ~ 500
steps/s)
}
```

In this snippet, `singleStep(clockwise)` energizes one coil at a time according to the sequence. A delay of 2 ms between steps will yield roughly 500 steps/s; for 2048 steps per rotation (half-step), that's about 0.25 rps (15 rpm). Decreasing the delay speeds up rotation, but if it's too fast, the motor may slip steps or stall due to torque limitations. You can also implement the more detailed half-step sequence (8 patterns) for smoother motion. Libraries like **Stepper** (built-in Arduino library) or custom code can manage the sequence for you. There are also specialized libraries (e.g. **AccelStepper** can also drive 4-wire steppers by initializing with `AccelStepper(AccelStepper::FULL4WIRE, IN1, IN3, IN2, IN4)` etc.).

Power considerations: The 28BYJ-48 is a 5V motor; the ULN2003 drops some voltage (Darlington transistors have a ~1V drop), so performance is best with a solid 5V supply. The ULN2003 has built-in clamp diodes to handle coil inductance, and LED indicators which are helpful for debugging coil states. The motor draws around 240 mA when one coil is energized (each coil ~50 Ω); ensure your 5V source can supply this. The ESP32 pins only provide the control signals; the actual motor current flows from the 5V source through the ULN2003 to ground.

Motor Driver Options (Overview and Comparison)

Various motor driver modules are available to interface between the ESP32 and motors. They differ in the motor types they support, current/voltage capacity, and control interface. Below we explain some commonly used drivers, how they connect to the ESP32, and their pros/cons:

L298N Dual H-Bridge

The **L298N** is a classic dual-channel H-bridge driver chip often mounted on a breakout module with a heatsink. It can drive two DC motors or one stepper motor and supports motor supply voltages from 5V up to about 35V. It uses bipolar transistors internally, which means it has a relatively large voltage drop across its outputs (around 2V lost at moderate currents) [20]. For example, if you supply 12V, the motor might effectively see only ~10V due to this drop. This makes L298N less efficient; it dissipates heat and

is not ideal for low-voltage motors unless you can afford the voltage loss. On the module, you'll usually find screw terminals for motor outputs and VMOT supply, and pin headers for logic connections:

- **Logic Inputs:** IN1, IN2, IN3, IN4 – these are the H-bridge inputs for channels A and B. They typically connect to ESP32 GPIOs. Each pair (IN1/2 or IN3/4) controls one motor's direction by steering current.
- **Enable Pins:** ENA, ENB – these enable the driver channels. On many L298N boards, ENA/ENB have jumpers tying them HIGH. To use PWM speed control, you should remove the jumpers and connect ENA and/or ENB to ESP32 PWM outputs [21].
- **5V logic supply:** The module often has an onboard 5V regulator. If your motor supply (VS) is >7V and <=12V, a jumper can be placed to enable the regulator, allowing the L298N to self-power its logic from VS (and even provide 5V out to power the ESP32 if needed, up to ~0.5 A) [22] [23]. If motor supply exceeds 12V or you remove the jumper, you must supply a separate 5V to the logic input (Vss pin) [24] [25].
- **Current:*** The L298N is rated for 2A per channel, but in practice 1A or less is advised without additional cooling, because it will run hot.

Pros: Widely available and low cost; can drive two DC motors or a stepper; easy to use for basic projects; has built-in diode protection and a regulator.
Cons: Inefficient (the bipolar transistors cause significant power loss as heat [20]); bulky compared to modern drivers; voltage drop reduces motor speed; not great for battery-powered applications due to waste.

The L298N is suitable for small robots or toys running off 9V–12V adapters or SLA batteries where a 2V drop is acceptable. For instance, with a 6V motor you might feed 7.5-8V into L298N to get ~6V effective [26]. If using with an ESP32, remember it's a 3.3V device – fortunately 3.3V is typically recognized as HIGH by L298N logic (which usually expects 5V logic, but 3.3V is generally sufficient to toggle it).

L9110S Dual H-Bridge

The **L9110S** is a smaller, low-voltage dual motor driver. It can handle motor supply from ~2.5V up to 12V and about 800 mA continuous per channel. It's based on a MOSFET H-bridge design (more efficient than L298N's BJT design). The module usually comes as a compact board with two driver ICs (each L9110S drives one channel). It's often used in small robot kits and projects with low-power motors (like toy DC motors or even small steppers).

Interface to ESP32 is straightforward: each motor channel has two inputs (often labeled A-IA, A-IB for channel A and B-IA, B-IB for channel B). These correspond to two half-bridge controls. To drive a motor, you set one input HIGH and the other LOW for a given direction (and swap for opposite direction). If both inputs are LOW or HIGH, the motor is braked (depending on the driver's internal design). For speed control, you can PWM one of the two inputs while holding the other low, effectively sending a pulsed voltage to the motor (the L9110S doesn't have a dedicated enable pin, but using PWM on an input pin works similarly).

Pros: Very simple and cost-effective; smaller footprint than L298N; lower voltage drop (MOSFET outputs); suitable for low-voltage battery-powered builds; can drive small stepper motors (it can control a 4-wire stepper by using both channels).
Cons: Lower current capability (~0.8 A) – not suitable for larger motors; no internal regulator (you need to supply it with logic voltage, typically 5V, but it **does** accept 3.3V logic HIGH as well); lacks the protection features of bigger drivers (no thermal shutdown in the basic chip).

The L9110 is great for tiny robots or mechanisms with modest current needs. Its simplicity (just a few control pins) is a big plus for beginners [27]. One advantage noted is its energy efficiency and low quiescent current, which can help battery life [28]. With an ESP32, simply connect the input pins to any GPIOs. Because the ESP32 PWM frequency can be high, you might want to choose a lower PWM frequency (e.g. a few kHz) for motors on L9110 to avoid switching losses or whining – but generally it will work across a broad range.

A4988 Stepper Driver

The **A4988** is a popular stepper motor driver module (used for bipolar steppers). It is typically used to drive **NEMA 17** size stepper motors as found in 3D printers and CNC machines. The A4988 requires a motor supply from 8V up to 35V and can supply up to ~1A continuously (2A peak with cooling) per coil [29]. It provides adjustable current limiting (via a trim pot), five microstepping modes (full, 1/2, 1/4, 1/8, 1/16) [30], and built-in protections (thermal shutdown, over-current, under-voltage, etc.) [31].

Interface: The A4988 significantly simplifies control of a stepper. Instead of energizing 4 coils in sequence manually, you only need **two pins** from the ESP32: STEP and DIR [13]. An optional EN (enable) pin can be used to disable the motor outputs (to save power or allow free rotation) – if not used, it defaults to enabled (active-low pin internally pulled low) [32] [33]. You can also wire the `RESET` and `SLEEP` pins together so the driver is not in reset or sleep mode (most breakout boards do this by default). The three microstep selection pins (MS1, MS2, MS3) can be tied HIGH/LOW to select the microstep resolution (they have internal pulldowns for default full-step if left unconnected) [34]. For example, setting MS1=MS2=MS3=HIGH puts it in 1/16 microstep mode. In practice, many use the driver in either full-step or 1/16 mode for maximum smoothness.

From the ESP32 side: You generate a pulse on STEP for each microstep you want the motor to move [35]. The pulse should be at least 1-2 microseconds long (A4988 datasheet recommends minimum pulse width ~1 µs and a minimum STEP low time ~1 µs as well). The DIR pin is simply set high or low and can be changed on the fly (it is read by the driver on the rising edge of each step pulse to determine direction). Because timing is critical for high speeds, using the ESP32's hardware timers or dedicated libraries (which can use interrupts) is beneficial if you need to generate very fast step pulses or control multiple steppers concurrently.

Pros: Very easy control interface (step/dir); offloads coil driving and current regulation – "the translator does all the complicated work for you" in sequencing the motor coils [13]; supports microstepping for smoother motion; widely available and inexpensive; can be interfaced directly to ESP32 3.3V signals (logic high threshold is low enough for 3.3V); includes protections that make it robust in many scenarios [31].
Cons: Limited current without cooling (for NEMA17 steppers that need >1A, the A4988 may overheat or require a heatsink/fan); no current feedback to MCU (open-loop positioning); needs careful adjustment of the current limit (via the potentiometer) to match the motor's current – if set too high, it will overheat, too low and you don't get full torque; also, *mixed decay mode* on A4988 can cause some audible noise or vibration at certain speeds (the DRV8825 and others have different decay modes that behave slightly differently).

Use case: A4988 is ideal for moderate-sized steppers where cost is a concern, like home projects with 3D printer axes or robotics. For the ESP32, one nice feature is you can leverage its dual-core to dedicate one core to generating steps (or use FreeRTOS tasks) to achieve very smooth motion even while other tasks (like WiFi) run on the other core – more on that in the RTOS section.

DRV8825 Stepper Driver

The **DRV8825** is another popular stepper driver, often seen as a "bigger brother" to the A4988. It is pin-compatible in many ways (the typical Pololu-style DRV8825 modules can plug into the same sockets as A4988 modules). The DRV8825 supports up to 45V motor supply and around 1.5A continuous (2.2A peak) per coil with adequate cooling [14] . It offers microstepping up to 1/32. The interface is the same concept (STEP, DIR, EN, MS1-3 for microsteps, etc.).

Differences from A4988: As noted, DRV8825 allows finer microsteps (1/32) [14] , higher supply voltage (useful for high inductance motors to get faster step rates), and a slightly higher current rating. It also uses a different current decay mode which can sometimes reduce audible noise and improve performance at some speeds, though in other cases it might produce different noise – users sometimes experiment with both to see which runs their particular motor more smoothly/quietly. The step pulse timing requirements differ slightly (DRV8825 needs a bit longer pulse, ~1.9 µs high per datasheet). Also, the DRV8825's current limit is set via a reference voltage similar to A4988 but the formula differs (so if swapping a A4988 with DRV8825, one must adjust the trim pot accordingly per the module's documentation).

Pros: Higher voltage and current headroom than A4988 (so it can drive larger steppers or run cooler at same current); 1/32 microstepping allows very smooth motion or fine resolution; pin-compatible with A4988 in many cases; has similar protection features (overheat, short, etc.).
Cons: Slightly more expensive typically; at very low step rates, the current waveform of DRV8825 can cause more audible noise ("whine") than A4988 in some motors (though at high speeds it's usually smoother – this is due to different decay modes); requires careful tuning of current like A4988. Also, the higher microstep means it can potentially produce *double* the number of interrupts if you are generating steps via software, but the ESP32 is powerful enough that 1/32 microstepping at moderate speeds is still easily handled.

In summary, if you have a choice: use A4988 for simpler or lower-power applications, and use DRV8825 when you need that extra current or voltage overhead. Both integrate well with ESP32 – just supply 3.3V to their logic (or 5V if available – they will treat 3.3V as HIGH fine) and ensure you share grounds.

ULN2003 Driver (Darlington Array)

The **ULN2003** isn't a motor driver IC in the same way as the above; it's a Darlington transistor array typically used to drive inductive loads like relays, solenoids, or unipolar stepper motors. The ULN2003A has 7 channels of NPN Darlington transistors with common emitters (ground) and open-collector outputs with clamp diodes. In the context of motors: a **ULN2003 driver board** usually refers to a little PCB with a ULN2003A chip and a header for a 5-wire stepper (like the 28BYJ-48) plus some LEDs. It allows an MCU (ESP32 in our case) to control the four coils of a unipolar stepper by sinking current through the ULN2003 transistors.

Each input of the ULN2003 corresponds to an output that can connect to one coil. When the input is driven HIGH (3.3V from ESP32), the ULN2003 outputs a LOW (ground) on that coil line, thus energizing the coil (the other end of the coil is tied to +5V). When input is LOW, the coil is not energized (output floats, coil sees no current). The ULN2003 essentially performs the high-current switching that the ESP32 cannot do, and its internal diodes protect against voltage spikes when coils are de-energized.

Pros: Very cheap and widely used for small motors; the ULN2003 board often comes with the 28BYJ-48 motor. It's straightforward to use – just 4 GPIOs to control a stepper's 4 coils. It can also drive small DC motors or other loads (each channel rated about 500 mA, though continuous load per channel should

be much less for thermal reasons). The ESP32's 3.3V outputs can directly interface (3.3V is enough to drive the Darlington inputs, which have a 0.7V drop and then drive the transistor).
Cons: Darlington transistors have a voltage drop (~1V or more) across them when conducting, which means some power is lost as heat and the motor sees a slightly lower voltage. For the 5V 28BYJ-48, this isn't usually a problem (it will run on ~4V effectively). The ULN2003 is not efficient compared to MOSFET drivers. Also, there's no current limiting – you rely on the coil resistance to limit current. For the intended motors (which are usually low current), that's fine. Additionally, ULN2003 has no fancy control features (no PWM built-in, no microstepping – any such control must be done in software by controlling the coil activation sequence).

Overall, the ULN2003 driver is perfectly adequate for the intended stepper (28BYJ-48) which draws around 240 mA per coil. If you needed to drive something like a unipolar stepper with higher current, you might need a different driver or use each ULN2003 channel in parallel (not usually recommended without balancing resistors).

Other driver options: Beyond the ones listed, there are more modern DC motor drivers like the **TB6612FNG** or **DRV8833** (dual MOSFET H-bridges with better efficiency than L298N), high-power drivers like **BTS7960** (for very large DC motors, 43A capability H-bridge), and dedicated servo driver boards (like PCA9685 16-channel PWM expander used for servos). Since this guide focuses on common beginner-friendly drivers, we haven't detailed those, but they are worth considering if your project demands specific performance (for example, TB6612FNG is a great drop-in upgrade from L298N for two DC motors at a few amps with much lower voltage drop).

Peripheral Hardware Integration

In robotics or motor-control projects with ESP32, you often need additional hardware for sensing and input. We will cover integration of **rotary encoders**, **potentiometers**, and **push buttons** – these can be used for feedback (e.g., measuring motor position or speed) or user inputs (joysticks, tuning knobs, etc.). Code examples and library recommendations are provided for each.

Rotary Encoders

A **rotary encoder** is a device that generates pulses as it is rotated. Common types are incremental encoders (often disk with slits or magnetic encoders) that provide quadrature signals (two out-of-phase signals A and B) which can be decoded to determine direction and count steps. They are used to measure angular position or speed, and are often attached to motor shafts (either built into motors for feedback or as separate components like a manual knob for user input).

The ESP32 is well-suited to reading encoders because it has a dedicated **Pulse Counter (PCNT) peripheral** that can count pulses in hardware with minimal CPU intervention. You can configure the PCNT unit to decode quadrature signals, freeing the CPU from handling every interrupt. There is an **ESP32Encoder** library available which wraps this functionality and supports up to 8 encoders using PCNT [36]. Using this library (or direct ESP-IDF PCNT API if you prefer) means you won't miss pulses even at high speeds – *"it doesn't skip counts like [simple polling] code does when you twist [the encoder quickly]"* [36].

For simpler needs (e.g., a low-speed knob that a user turns by hand), you can also use interrupts on the two encoder pins and do the decoding in software. The standard Arduino **Encoder** library by Paul Stoffregen can often be used with ESP32 as well, but it may be less efficient than using PCNT for fast rotations.

Wiring: A typical incremental encoder has 3 pins: A, B (the quadrature outputs) and GND (and possibly VCC if it's an active sensor). Connect A and B to two ESP32 GPIOs that can serve as inputs. Most GPIOs can be used with the PCNT or interrupts (there are a few strapping pins to avoid for input if possible, like GPIO34-39 are input-only but fine for this use). If the encoder is mechanical (like those low-cost rotary knobs), you may need pull-up resistors (many breakout boards have them) and you'll need to debounce the signals in software because mechanical contacts can bounce.

Reading using ESP32Encoder library: Below is a brief example of using the ESP32Encoder library to read a quadrature encoder:

```
#include <ESP32Encoder.h>

ESP32Encoder encoder;
const byte ENC_PIN_A = 33;
const byte ENC_PIN_B = 32;

void setup() {
  Serial.begin(115200);
  // Specify quadrature pins
  ESP32Encoder::useInternalWeakPullResistors = UP;  // enable internal pull-
ups
  encoder.attachHalfQuad(ENC_PIN_A, ENC_PIN_B);     // attach in half-
quadrature mode
  encoder.clearCount();                             // start count at 0
}

void loop() {
  int32_t count = encoder.getCount();
  Serial.println(count);
  delay(100);
}
```

In this code, we attach the encoder in half-quadrature mode (meaning it counts every transition on one channel while using the other to determine direction; full quadrature mode would count every edge on both channels for 4x resolution). The internal pull-ups are activated to keep the lines HIGH when the encoder switches are open (common for mechanical encoders). `getCount()` returns a signed count that increases or decreases as the encoder is turned. This library handles the PCNT configuration under the hood. It's very efficient – the counting is done in hardware and an interrupt only triggers on counter overflow (which at 16-bit can be configured to a large range, or you can use 32-bit logic in software to extend it).

If you don't want to use a library, you could configure PCNT via ESP-IDF calls or use attachInterrupt on the A and B pins and manually increment/decrement a counter in the ISRs. However, manual ISR decoding at high frequency can be tricky on ESP32 if WiFi or other tasks are active, since the latency might cause missed ticks – hence the recommendation to use PCNT hardware.

For **motor applications**, encoders are typically used for closed-loop control. For example, a DC motor with an attached encoder can form a DIY servo system where the ESP32 reads the encoder to adjust motor drive for reaching a target position or speed (this would involve a PID control loop in software,

possibly running in a high-priority task). That is beyond the scope of this guide, but know that ESP32 is capable of fairly advanced motor control when encoders are leveraged.

Potentiometers (Analog Inputs)

A **potentiometer** is a variable resistor, often used as a knob or slider that provides an analog value. The ESP32 has analog-to-digital converters (ADCs) on many of its input pins, allowing it to read the voltage from a potentiometer. Typically, one side of the potentiometer goes to 3.3V, the other to GND, and the wiper (middle pin) goes to an ESP32 ADC input. As you turn the knob, the wiper voltage changes between 0 and 3.3V.

Reading a potentiometer is straightforward using `analogRead(pin)`. On the ESP32, the ADC is 12-bit by default, so `analogRead()` will return a value from 0 to 4095 corresponding to 0 V to 3.3 V (assuming default attenuation). For example, if you have a 10k potentiometer:

```
const int POT_PIN = 34;  // any ADC-capable pin
void setup() {
  Serial.begin(115200);
}
void loop() {
  int raw = analogRead(POT_PIN);
  float voltage = (3.3 / 4095) * raw;  // approximate voltage (in theory)
  Serial.printf("Potentiometer raw: %d, approx voltage: %.2f V\n", raw,
voltage);
  delay(100);
}
```

Note: Make sure the pin you choose is ADC capable. ESP32 has two ADC units with various channels; not all GPIOs have ADC. Common ADC pins include 32-39 (ADC1) and 0,2,4,12-15,25-27 (ADC2). However, ADC2 pins cannot be used for analogRead when WiFi is active (ADC2 is shared with WiFi radio). So for simplicity, use ADC1 pins (32-39) for potentiometers or analog sensors to avoid conflicts.

The reading might be a bit noisy or non-linear; the ESP32 ADC isn't very high precision and by default has 11 dB attenuation (so it can measure up to ~3.3V). You can calibrate or smooth readings by averaging multiple samples if needed. But for tasks like reading a user knob to control motor speed or position, a direct `analogRead` mapped to a useful range is typically enough.

For example, you might use a potentiometer value to set a servo angle or motor speed. Simply map the 0-4095 to your desired output range (0-180 for servo angle, or 0-255 for PWM duty, etc.). Arduino provides `map()` function or you can do a scale multiply/divide.

In summary: Potentiometers provide an easy way to get analog input into the ESP32. No special libraries are needed (aside from the built-in analog support). Just remember that they should only be connected to the 3.3V of ESP32 (do *not* use 5V on a pot wiper into ESP32 ADC or you risk damage, since ESP32 ADC pins are not 5V tolerant). If you have a 5V-only pot setup (like a joystick module that often uses 5V), use a voltage divider or ensure the output never exceeds 3.3V.

Push Buttons and Switches (Digital Inputs)

Push buttons are simple digital inputs but often require careful handling for reliable operation. When a button is pressed or released, mechanical contacts can bounce (make/break rapidly) causing multiple transitions. Also, a floating input can pick up noise. The ESP32 (like Arduino) can use internal pull-up or pull-down resistors to stabilize an input.

A typical button circuit is to connect one side of the button to an input pin and the other side to ground, and enable an **internal pull-up** on that pin. This way, when the button is not pressed, the input reads HIGH (pulled up to 3.3V internally), and when pressed, it connects to ground and reads LOW. This is an **active-low** configuration. It's simple because you don't need an external resistor.

Basic code example (polling):

```
const int BUTTON_PIN = 21;
void setup() {
  pinMode(BUTTON_PIN, INPUT_PULLUP);
  Serial.begin(115200);
}
void loop() {
  if (digitalRead(BUTTON_PIN) == LOW) {
    Serial.println("Button pressed!");
    // ... do something, e.g., toggle a motor or change mode
    delay(200); // basic debounce: wait to avoid rapid repeats
  }
}
```

In this example, we configure the pin with `INPUT_PULLUP`. We then check for LOW which indicates pressed. The `delay(200)` is a crude debouncing (it will ignore any further presses for 200 ms). This can suffice for simple needs where response time isn't critical.

For more robust button handling (detecting short press, long press, multiple clicks, etc.), there are libraries available: - **OneButton** (by Matthias Hertel) – a library that simplifies handling single or multiple buttons and can detect single, double, long presses easily [37]. - **Bounce2** – a debouncing library that allows you to easily debounce without blocking, you then query `.fell()` or `.rose()` for state changes. - **Button2** – another modern library for ESP32/ESP8266 that handles multiple buttons and supports event callbacks for pressed, released, held, etc. [38]. - The **ESP32_Button** library (from Espressif) which even supports analog buttons (multiple buttons on one ADC via resistor ladder) [39] [40].

For production, using one of these can make the code cleaner and more reliable, especially if you need non-blocking behavior (no `delay`) and want to handle complex interactions.

Interrupts: You can attach an interrupt to a button pin to catch presses without polling. For example, `attachInterrupt(BUTTON_PIN, ISR, FALLING)` would call `ISR` when the button is pressed (since it goes from HIGH to LOW). In the ISR you might set a flag or increment a count. However, you must debounce either in hardware or in software (the ISR might fire multiple times due to bounce). A common trick is to use a timer or timestamp in the ISR: ignore new interrupts that happen within, say,

50 ms of the last one. Given that bounce typically settles within a few milliseconds, that works. Alternatively, have the ISR simply record the event and the main loop handles debouncing.

Recommendation: Use a library like OneButton if your project involves things like "press and hold 3 seconds to turn off" or "double-click to change mode" – it will save you time. If it's a simple toggle or momentary action, you can write a small state machine with debouncing.

Finally, ensure correct wiring: If using `INPUT_PULLUP`, connect the button to GND. If using `INPUT_PULLDOWN` (ESP32 has internal pull-downs as well), connect the other side to 3.3V. Active-low with pull-up is generally safer (less susceptible to noise when button is open).

Libraries for Motors and Peripherals

To manage complexity and improve reliability, leveraging well-maintained libraries is wise. Below is a list of recommended libraries (updated as of 2025) for various functionalities, along with alternatives and notes on each:

- **ESP32Servo** – *Servo motor control.* This library is specifically made for ESP32 (by Kevin Harrington/John Bennett) and extends the Arduino Servo API [41]. It uses the LEDC PWM hardware to generate stable 50Hz signals on any GPIO. It supports up to 16 servos. Pros: very easy to use (just like standard Servo library); allows tuning min/max pulse widths; well-tested. Alternative: you can manually use `ledcAttachPin` and `ledcWrite` or `analogWrite()` with 50 Hz frequency, but the library handles all that. There is also a newer Arduino official Servo library that might eventually include ESP32 support via the ArduinoCore (since analogWrite works, a servo could be done without a separate lib, but ESP32Servo remains the go-to in late 2025).

- **AccelStepper** – *Stepper motor control.* A versatile library for controlling stepper motors (supports 2-wire (step/dir) drivers or 4-wire coil driving) [42]. Pros: supports acceleration and deceleration, multiple motors, non-blocking operation (you call `.run()` frequently), and absolute or relative positioning commands. It's mature and commonly used. Cons: It's not the fastest for very high step rates (written for general Arduino use), but on ESP32 it generally runs fine. Alternatives:

- **StepperDriver** by laurb9 (on GitHub) – focused on step/dir drivers like A4988/DRV8825, providing both constant speed and accelerated movement modes [43]. It's efficient and simpler if you only use step/dir drivers.
- **FastAccelStepper** – a newer library aimed at high speed stepping, with ESP32 support (can generate very high frequency steps by using ESP32 RMT peripheral or interrupts) [44]. Useful if you need step rates in tens of kHz.

- For simple blocking motion, the basic Arduino `Stepper` library can be used for 4-wire steppers, but it lacks features (no acceleration control, and it will block during steps).

- **ESP32Encoder** – *Rotary encoder interface.* As discussed, this library efficiently uses the ESP32 PCNT hardware to read quadrature encoders with minimal CPU usage. It supports up to 8 encoders simultaneously [45]. Pros: Very reliable for high counts (no missed pulses), easy to use. Alternative:

- **AiEsp32RotaryEncoder** – another library that provides a high-level interface for rotary encoders, including support for click buttons often found on rotary knobs (the push-rotation encoders) [46]. It uses PCNT under the hood as well.
- **Encoder** (Paul Stoffregen's) – works if you prefer a simpler implementation and are fine with potentially missing very fast rotations.

- Direct use of PCNT via ESP-IDF functions if you need custom behavior.

- **Button handling libraries:**

- **OneButton** – as mentioned, great for managing single button with multiple click types [37].
- **Button2** – good for multiple buttons and has event callbacks, tailored for ESP devices [38].
- **Bounce2** – good for debouncing raw inputs in an update loop style.

- If using ESP-IDF directly, there's also an **ESP-IDF component** for buttons (esp_button) that handles common patterns.

- **Motor control (DC) libraries:** There isn't a single dominant DC motor Arduino library for ESP32, because controlling DC motors is often just a matter of analogWrite/ledc. However, you might encounter:

- **ESP32MotorControl** – a library that uses the MCPWM (Motor Control PWM) unit of the ESP32, allowing very fine control of up to 8 PWM signals with sync, deadtime insertion (for advanced motor drive like BLDC or synchronous rectification). This is more for advanced users (e.g., driving a 3-phase BLDC or controlling H-bridge with complementary signals). For typical hobby DC motors, usually you don't need this.

- **SimpleFOC** – an open source library for *Field Oriented Control* of BLDC and stepper motors, which does support ESP32. This is for closed-loop control of motors with encoders or magnetic sensors. Mentioned here in case one is exploring more advanced control of brushless gimbal motors or high-performance applications.

- **Networking/OTA libraries** (since OTA was asked):

- **ArduinoOTA** – official library for OTA updates via WiFi for Arduino on ESP32 [47]. Very useful to wirelessly upload new code.
- **AsyncElegantOTA** – a library that provides a nice web interface for OTA (you start a web server on ESP32 that allows uploading a new firmware file).

- **Bluetooth OTA** – not as widely used, but Espressif's **esp_ble_ota** or Nordic's UART DFU could be implemented. Most stick to WiFi OTA or wired updates.

- **FreeRTOS support libraries:** Generally, you don't need a separate library to use FreeRTOS on ESP32 (it's built-in). The Arduino-ESP32 integrates with FreeRTOS out of the box. However, some helper libraries or wrappers exist (for example, libraries for task management or wrapping FreeRTOS queues, like SafeString or ESP32Mail which uses RTOS under hood). For learning FreeRTOS patterns, you might refer to examples (e.g., how to use `xTaskCreate`, semaphores, etc., as shown in Espressif's docs or various tutorials).

The **key point** is to choose maintained libraries that abstract away low-level details. For instance, rather than manually coding a complex stepper acceleration profile, use AccelStepper or FastAccelStepper;

rather than struggling with rotary debounce logic, use ESP32Encoder. The libraries mentioned have active support and are compatible with ESP32 as of 2025.

FreeRTOS Multitasking on ESP32 (including RTOS Examples)

One of the strengths of the ESP32 is that it runs a **dual-core FreeRTOS** operating system under the Arduino environment. The Arduino `setup()` and `loop()` run as a task (on core 1 by default), but you can create additional tasks to run code in parallel. This is extremely useful for motor control projects – for example, you can dedicate one task to generating stepper motor pulses or reading an encoder at high speed, while another task handles communications or higher-level logic. By assigning tasks to different cores, you ensure time-critical motor control isn't blocked by WiFi or other activities. In fact, "by assigning the stepper motor control to one core and other logic to the second core, we can ensure that the motor runs smoothly without being interrupted by other tasks" [48].

Creating tasks: In Arduino (ESP32), you can use `xTaskCreate()` or `xTaskCreatePinnedToCore()` from the FreeRTOS API (already included) to start new tasks. Each task is essentially an infinite loop in a separate function. For example:

```
// Shared variables (if needed) should be volatile or protected by mutexes
volatile int speedCommand = 0;

// Task to control a motor (runs on core 0)
void MotorTask(void *pvParameters) {
  const int stepPin = 14;
  const int dirPin = 12;
  pinMode(stepPin, OUTPUT);
  pinMode(dirPin, OUTPUT);
  bool dir = false;
  for (;;) {
    // Generate steps at rate proportional to speedCommand
    if (speedCommand != 0) {
      digitalWrite(dirPin, dir ? HIGH : LOW);
      digitalWrite(stepPin, HIGH);
      // a short delay for pulse
      ets_delay_us(5);  // low-level delay, since vTaskDelay is too slow for
pulsing
      digitalWrite(stepPin, LOW);
      // Wait between steps:
      // Use vTaskDelay to yield to other tasks. If speedCommand is steps per
second:
      int delayMs = abs(1000 / speedCommand);
      if(delayMs < 1) delayMs = 1;
      vTaskDelay(delayMs / portTICK_PERIOD_MS);
    } else {
      // If speedCommand is 0, just yield CPU
      vTaskDelay(10 / portTICK_PERIOD_MS);
    }
    // (You might also check for notifications or use ulTaskNotifyTake for
more complex control)
  }
```

```
}

// Task to read an encoder or handle sensor input (runs on core 1)
void EncoderTask(void *pvParameters) {
  // Suppose we have an encoder on PCNT or a button to adjust speed
  pinMode(ENCODER_A, INPUT_PULLUP);
  pinMode(ENCODER_B, INPUT_PULLUP);
  // ... setup PCNT or attachInterrupt for encoder
  for (;;) {
    // Read encoder count or other sensor
    int32_t count = encoder.getCount();
    // Compute speedCommand based on count or input (this is just
illustrative)
    speedCommand = count;  // maybe set speed equal to count for demo
    vTaskDelay(50 / portTICK_PERIOD_MS);  // run this loop every 50ms
  }
}

void setup() {
  // ... other init ...
  // Create tasks:
  xTaskCreatePinnedToCore(MotorTask, "MotorTask", 2048, NULL, 1, NULL,
0);   // run on core 0
  xTaskCreatePinnedToCore(EncoderTask, "EncoderTask", 2048, NULL, 1, NULL,
1); // run on core 1
}

void loop() {
  // The main loop can still run (on core 1 alongside EncoderTask in this
setup)
  // You could use it for something non-critical, or just suspend it.
  vTaskDelay(1000 / portTICK_PERIOD_MS);
}
```

This pseudo-code outlines two tasks. The MotorTask is pinned to core 0 (which is usually otherwise used by WiFi and system tasks – here we assume WiFi is not heavily used, or we could instead pin motor to core 1 and run other logic on core 0; it depends on your needs). The EncoderTask is on core 1. The MotorTask generates step pulses using a combination of very short busy-wait (for pulse width) and `vTaskDelay` for the interval, which yields control to other tasks in between steps. The EncoderTask updates a shared variable `speedCommand`. Because `speedCommand` is shared, it's marked `volatile` to tell the compiler to not optimize access (for simple types like int, this is okay without a mutex if updates are atomic, but generally for multi-byte data a mutex or critical section is safer).

Important considerations in FreeRTOS multitasking:

- Use `vTaskDelay()` or `vTaskDelayUntil()` rather than `delay()` in tasks, because `delay()` in Arduino will block the entire loop task but not yield to lower priority tasks properly. `vTaskDelay` puts the task to sleep allowing other tasks to run.
- If you need very high timing precision (e.g., step pulses in the tens of kHz), consider using ESP32's **RMT** (remote control peripheral) or **MCPWM** to generate pulse trains, because doing it

purely in software might hit limits. But for many purposes, a task can generate a few kHz signals reliably, especially if isolated on its own core.

- **Task Priorities:** You can assign priorities (the parameter in `xTaskCreatePinnedToCore` before the task handle). Higher number means higher priority. If your motor control task must not be interrupted by, say, a web server task, give it a higher priority. But be careful: a high-priority task that never blocks (no delay) will starve other tasks. Always include a `vTaskDelay` or `vTaskDelay(1)` (which is like yield) in the loop or the task will take 100% CPU.
- **Inter-task communication:** FreeRTOS provides queues, semaphores, and direct task notifications. For example, instead of using a shared global for `speedCommand`, you could use a queue or `xTaskNotify()` to send the latest speed safely to the MotorTask. This avoids concurrency issues. The code above is simplified to illustrate the concept.
- **Dual core considerations:** By default, Arduino runs on core 1 and core 0 is often used for WiFi. If you pin a task to core 0 and also use WiFi/Bluetooth, you might get some contention. It can still work – FreeRTOS will schedule WiFi tasks and your task on core 0. If the motor control is very time-sensitive, some prefer to disable WiFi or run it on a separate core. A common approach is: core 1 for your application (including a motor task), and leave core 0 mostly for WiFi and background, or vice versa depending on needs. In any case, the ability to pin tasks ensures critical control loops can have a dedicated core if needed, **minimizing jitter** in timing [49].

Example RTOS use case: Smooth Stepper control – A stepper motor requires precise timing of pulses for smooth rotation. If you generate pulses in the main loop and the loop gets busy with printing to Serial or handling a web request, the pulses might hiccup. By offloading stepping to a high-priority task on one core, you isolate it. The medium article we referenced earlier demonstrated smooth motion by yielding periodically in the step loop to let lower priority tasks run briefly [50] [51]. Essentially, inside a long movement loop, inserting `vTaskDelay(1)` occasionally (e.g. every 1000 steps) allows WiFi and other tasks to not completely stall, while hardly affecting the motor motion timing. This technique is shown where they call `vTaskDelay(1)` every 1000 iterations to yield [52]. The result is a balance of real-time motor control and system responsiveness.

In summary, **FreeRTOS on ESP32** gives you the tools to create a **multitasking motor control system**. You can read sensors and control motors simultaneously without complex timer interrupt code, by simply running separate tasks. It is a more advanced topic, but even moderate familiarity with xTaskCreate and vTaskDelay can greatly improve the structure of a program that needs to do many things at once (like driving multiple motors and reading multiple encoders concurrently). The Arduino-ESP32 documentation and examples (such as the "ESP32 multitasking" tutorial [53]) are good resources to get started on this.

Development Workflow (Arduino IDE and CLI)

To work with ESP32 in the Arduino environment, you need to set up the **ESP32 board support** in Arduino IDE (or Arduino CLI). As of late 2025, the ESP32 Arduino core is very mature (version 3.x.x, built on ESP-IDF 5.x) [54], supporting many variants of the ESP32 (including newer SoCs like ESP32-S3, C3, etc., though our focus here is the classic ESP32).

Setting up Arduino IDE for ESP32:

1. **Install ESP32 Board Definitions:** In Arduino IDE, open **Boards Manager** and search for "ESP32 by Espressif Systems". Install the latest version. (If using Arduino IDE 1.x, you first add the URL `https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json` in File > Preferences > Additional Boards Manager URLs, then go

to Boards Manager and install). After installation, you should see a variety of ESP32 boards in the Tools > Board menu (e.g., “ESP32 Dev Module”, “WEMOS LOLIN32”, etc.). This process is well-documented on Espressif’s GitHub and sites like RandomNerdTutorials [55].

2. **Select the board and port:** In Arduino IDE, choose the appropriate ESP32 board definition that matches your hardware. For generic dev boards, “ESP32 Dev Module” works. Select the COM port (Windows) or `/dev/ttyUSBx` (Linux/macOS) corresponding to your ESP32. Install USB drivers if needed (common chips are CP2102 or CH340 – references to drivers are available if the port doesn’t show up [56]).

3. **Install required libraries:** Using the Library Manager (Sketch > Include Library > Manage Libraries), install any libraries mentioned above that you plan to use (e.g., ESP32Servo, AccelStepper, ESP32Encoder, OneButton, etc.). Many of these are available through the Library Manager by name. For example, search “ESP32Encoder” and install it [57], or “OneButton”, etc. If a library is not in the manager, you can usually download it from GitHub and place it in your `libraries` folder.

4. **Write or open your code:** You can start from scratch or use example sketches. The ESP32 core comes with numerous examples (File > Examples > *select ESP32 board* > you’ll see categories like WiFi, Bluetooth, etc., and basic examples like Blink). For motor control, you might not find built-in examples, but library examples (e.g., ESP32Servo has an example for sweep, AccelStepper has examples).

5. **Compile and Upload:** Click the Verify/Compile checkmark to build. If using Arduino IDE 2.x, you’ll get output in the console. Then click Upload. The first time, the Arduino core will compile everything (can take a minute or two). Subsequent compiles are faster. During upload, you should see it connecting at 115200 or higher baud and writing firmware. Most ESP32 dev boards have auto-reset circuitry: the IDE toggles DTR/RTS to reset the board into bootloader mode. If you see “Connecting...” and dots, and it doesn’t proceed, you might have to press the BOOT button on the board when “Connecting” appears, to force bootloader mode. Once programming starts (you see “Writing at 0x...”), you can release it.

6. **Monitor output:** Open the Serial Monitor at the appropriate baud (e.g., 115200) to see debug prints. *Tip:* The ESP32 often resets upon USB connection opening – it’s due to DTR toggling EN pin; if that behavior is undesired, some terminals allow you to disable DTR toggle.

Command-Line (CLI) development:

If you prefer not to use the Arduino GUI, you have a couple of options:

- **Arduino CLI:** Arduino offers `arduino-cli`, a command-line tool to compile and upload sketches. Once you’ve installed the ESP32 core as above, you can compile via CLI. For example:

```
arduino-cli compile --fqbn esp32:esp32:esp32dev MySketch.ino
arduino-cli upload --fqbn esp32:esp32:esp32dev -p /dev/ttyUSB0
MySketch.ino
```

This would compile and upload using the "ESP32 Dev Module" definition (`esp32dev`). You can get the exact FQBN (fully qualified board name) from `arduino-cli board listall` or from the IDE's output. Arduino CLI is great for automation or CI pipelines.

- **PlatformIO:** This is an alternative development environment that many use. It's not just CLI, it integrates with VSCode and other IDEs. PlatformIO handles downloading cores and libraries automatically via a configuration file. It's very convenient for advanced users, but has its own ecosystem. Since the question focuses on Arduino IDE and CLI, we'll not delve deep, but know that PlatformIO is a popular choice for ESP32 development as well.

- **ESP-IDF:** The Espressif IoT Development Framework is the official SDK for ESP32 in C (or C++). It has a different project structure (CMake based). If you require deep control, realtime performance, or want to use ESP32 features not exposed in Arduino, you might go with ESP-IDF. It provides its own `idf.py` (which wraps CMake and esptool). But mixing Arduino and IDF can be done (Arduino core can be used as a component in IDF projects). This is an advanced route and typically not needed for the scenarios in this guide.

Compilation notes: With the Arduino-ESP32 core v3.x, some API changes occurred (for instance, `ledcAttachPin` and `ledcSetup` were replaced with a single `ledcAttach()` in core 3.0) [58]. Most libraries have been updated for core 3.x compatibility by now. If you encounter deprecated function warnings or breaking changes, refer to the core's migration guide [59]. The current core aims to maintain Arduino compatibility while leveraging ESP-IDF improvements. For example, analogWrite is now fully supported and uses LEDC under the hood, meaning older code that manually called `ledcSetup` can be simplified to just `analogWrite()` [60].

Uploading speed and flash settings: In Arduino IDE's Tools menu for ESP32, you can often set the upload baud rate (default 921600 baud is fine), flash frequency (40MHz usually), partition scheme (e.g., minimal SPIFFS if you want more space for code, etc.), and core debug level. For most users, the defaults work. If you plan to do OTA, consider using a partition scheme that allots space for two apps (OTA requires space for the new firmware alongside the old).

Flashing Options: USB, OTA, and esptool.py

Once your code is ready, getting it onto the ESP32 can be done in several ways:

1. USB Serial Flashing: This is the standard method used by the Arduino IDE. The ESP32 comes with a built-in bootloader in ROM that listens on the serial port for flashing commands. When you hit upload, the IDE resets the ESP32 into this bootloader mode and sends the compiled binary via UART. Under the hood, the tool **esptool.py** is actually invoked to do this transfer [61]. It erases the necessary flash sectors and writes your program.

- This method requires a USB connection (or a serial adapter connected to GPIO0/2/15 and EN pins properly for boot mode). On dev boards, it's just the USB cable.
- Speed: typically 921600 baud upload. The entire flash (say a 1.3 MB sketch) might upload in a few seconds.
- If issues arise (failed to connect), check wiring or press BOOT manually as mentioned.
- After flashing, the ESP32 resets and runs your program.

2. OTA (Over-The-Air) Updates via WiFi: OTA is very convenient for remote or embedded devices where accessing USB is impractical. The process is: - You flash a program *once* via USB that contains an OTA

update mechanism (for instance, using the ArduinoOTA library or AsyncElegantOTA). - That program connects to WiFi and either waits for a connection from a PC or hosts a webpage for uploading new firmware. - Using ArduinoOTA library: You can do OTA directly from the Arduino IDE by selecting the network port (the device will advertise itself if using ArduinoOTA with mDNS). Then clicking upload sends the firmware over WiFi to the ESP32, which writes it to flash and reboots. This is integrated and easy. The ArduinoOTA library handles the flash partition switching and verification [47] . - Alternatively, using something like AsyncElegantOTA: you would open a web browser, go to the device's IP, and use a form to upload the new firmware file (.bin) which you compiled. This is also user-friendly (good for field updates where you might email a file to someone). - The ESP32 needs to have enough flash for two copies of the app (one slot for current, one for new). Most have 4MB, which is plenty if your app is <1.5MB. - OTA over WiFi is quite reliable; just ensure power is not lost during the update or the device could be stuck (the bootloader will remain intact but you might need to reflash via USB if OTA fails mid-way). - OTA over **Bluetooth**: There's no widely-used Arduino-supported method out-of-the-box, but Espressif had an example using BLE (the esp-idf supports BLE OTA, although not as common as WiFi OTA). For BLE, you might transfer the file via the BLE GATT to the device. Given the complexity, most stick to WiFi OTA.

3. Using esptool.py directly (CLI flashing): If you have a binary (e.g., from a CI build or compiled with Arduino CLI or PlatformIO), you can use **esptool.py** to flash it. This is the same tool Arduino IDE uses internally [61] . Common commands: - Erase entire flash: `esptool.py --port COM3 erase_flash` (helpful if switching frameworks or needing a clean slate) [62] . - Write firmware: `esptool.py --port COM3 write_flash 0x1000 firmware.bin`. However, a full ESP32 app has multiple parts (bootloader, partition table, app, etc.). Usually it's easier to let Arduino or PlatformIO handle offsets. If using the Arduino IDE's output, the `.bin` it produces is an *app image* that expects bootloader and partition table already present (they are flashed once when you burn bootloader). The Arduino IDE normally packages those, but if you use esptool manually, you might do:

```
esptool.py --port COM3 write_flash \
  0x1000 bootloader.bin \
  0x8000 partitions.bin \
  0x10000 my_program.bin
```

The addresses may vary (0x1000, 0x8000, 0x10000 are typical for ESP32 Arduino partitions). This is advanced usage. The `esptool.py` documentation provides exact commands for various scenarios [63] . - Esptool can also read flash, change the flash encryption or secure boot settings, etc. For general use, you might rarely need it outside of the Arduino toolchain, but it's good for troubleshooting (e.g., read logs from serial bootloader, or dump flash to see memory).

4. Other flashing methods: - **UART Bootloader over other interfaces:** Some people use an external USB-to-Serial converter (FTDI, etc.) for boards that don't have USB. You have to manually toggle GPIO0 (BOOT) and EN (reset) to enter bootloader mode. Typically pulling GPIO0 low and resetting puts it in upload mode. - **JTAG**: The ESP32 can also be programmed via JTAG (e.g., using OpenOCD). That's more for debugging and not common for Arduino users. - **ESP-Prog**: Espressif provides an "ESP-Prog" board for programming and debugging via the JTAG or serial lines. - **OTA via Internet**: It's possible to have the ESP32 check a server for updates (using HTTP client to fetch a new binary, then write to flash and switch OTA partition). This way, you can do unattended remote updates (e.g., your device on first boot checks your server for a new firmware). The ArduinoOTA library doesn't do this automatically (it's more for pushing from PC), but you can incorporate an HTTP-based OTA (there are libraries and examples for that as well).

After Flashing – Running the Program: The ESP32 will boot your program. If using the serial monitor, you might see the boot log (at 115200 baud by default) which prints chip info and memory info, then your program's output. If something goes wrong (esp32 doesn't start the sketch), common issues include wrong boot mode (GPIO0 held low?), or crashing due to incorrect power or code bugs (if it crashes, you'll see an exception dump). The board definitions by default enable the **watchdog timer** on both cores to reset if tasks hang too long. If you accidentally write a task that never yields, you might trigger the watchdog (resulting in a dump "Task watchdog got triggered"). This is a sign you need to put some vTaskDelay or yield in long loops.

Current APIs and Best Practices (2025): A few notes to ensure our guide is up-to-date with current practices:

- Use `analogWrite()` for PWM where possible – earlier guides for ESP32 might show the older LEDC functions, but now analogWrite is officially integrated [60], making code more portable from standard Arduino. You can still use LEDC for advanced control, but analogWrite with `analogWriteFrequency(pin, freq)` and `analogWriteResolution(pin, bits)` is convenient.
- Take advantage of **dual-core** by structuring tasks (as we did in RTOS section) rather than writing long delay loops that block everything.
- The Arduino-ESP32 core 3.x has some changes: for example, `attachInterrupt()` now supports an argument to attach on specific cores, and some WiFi and BLE APIs have improved. Ensure you have the latest core via Board Manager (the expected stable release of 3.x was around end of 2023 [58], and by 2025 we assume it's stable).
- For motor control, consider using the **MCPWM** hardware for complex drives (like controlling an H-bridge with high-frequency PWM and dead-time – this is useful if you were, say, building a sine-wave driver for a BLDC motor or controlling a half-bridge for an AC inverter). Arduino core now exposes some MCPWM functionality from ESP-IDF if you include the right headers.
- When using encoders or other peripherals, prefer using hardware features (PCNT for encoders, LEDC for PWM, etc.) to offload work from the CPU. The ESP32 is powerful, but efficient use of peripherals leads to smoother performance.
- **Power management:** If your project is battery-powered, note that by default the ESP32 in Arduino will not automatically go to deep sleep or such. But you can use `esp_deep_sleep` functions if needed. Also, heavy WiFi use will increase power draw – plan your tasks accordingly (e.g., turn off WiFi if not needed using `WiFi.mode(WIFI_OFF)`).
- **Thread-safe operations:** If you use multiple tasks that share data (like our simple `speedCommand`), use FreeRTOS synchronization primitives (Semaphore, Mutex, Queue) as needed to avoid race conditions. For instance, reading an encoder count from one task and writing to it from another should be protected. The Arduino core prints are thread-safe (Serial uses a lock internally), but your own variables are not unless you make them so.
- **Debugging:** Use `Serial.print` generously to debug logic, or even better, an actual debugger. The ESP32 can be debugged via JTAG if you have the setup, but often prints and occasionally the `esp_log` with different log levels (if you enable core debug) can help.

Finally, always test your system thoroughly – motors introduce noise (electrical and acoustic) and load, so ensure your ESP32's power supply is stable (use proper decoupling capacitors, especially when driving motors; motors can cause brownouts that might reset the ESP32 if the supply dips). Also be mindful of logic level shifting – all the mentioned drivers accept 3.3V logic except possibly some old L298N modules which might assume 5V logic; but generally 3.3V is recognized as HIGH by them. If not, a level shifter or using a 5V tolerant input on driver with a 5V output from ESP32 (not possible directly) would be needed – but again, most cases are fine.

Conclusion

Using the ESP32 for motor control opens up a world of IoT and robotics possibilities, combining powerful wireless capabilities with real-time control. In this guide, we covered how to drive DC motors, servos, and steppers with the ESP32, including example code for each. We discussed common driver hardware (L298N, L9110S, A4988, DRV8825, ULN2003), their interface details, and pros/cons, helping you choose the right one for your project. We also integrated peripherals like encoders (for feedback), potentiometers, and buttons (for inputs), which are often key in a complete control system.

Crucially, we highlighted up-to-date libraries that simplify development and improve reliability – from ESP32Servo and AccelStepper to ESP32Encoder and OneButton – leveraging the active Arduino & ESP32 community contributions. We demonstrated how FreeRTOS on the ESP32 allows for concurrent tasks, which is a powerful approach to handle complex projects (for example, reading sensors and controlling motors at the same time without hiccups). By pinning a critical task to a core or adjusting task priorities, you can achieve smooth, jitter-free motor operation [48] even while networking or other logic runs in parallel.

The development workflow section ensures you can set up the Arduino environment or use CLI tools like `arduino-cli` or `esptool.py` confidently to build and flash your code. Whether you upload via USB or OTA, the ESP32 provides flexibility in deployment. As of late 2025, the ESP32 Arduino core is robust and feature-rich, aligning closely with Espressif's official SDK improvements. Functions like `analogWrite()`, `analogRead()`, and others work intuitively [3], and many previously tedious tasks (PWM setup, etc.) have been streamlined.

With the information and examples provided here, you should be able to create **reliable, real-world motor control applications** using the ESP32. From a Wi-Fi controlled robot car with encoders on its wheels, to an internet-connected stepper motor slider controlled via a web interface, the combinations are endless. Use the proper drivers for the job, take advantage of existing libraries, and follow best practices for multitasking and power management. Happy making, and may your motors always spin under your command!

Sources: This guide referenced official documentation, open-source libraries, and reputable tutorials to ensure accuracy and currency. Key sources include Espressif's docs for API behaviors [3], community tutorials on motor drivers [20] [27], and examples of FreeRTOS usage on ESP32 [48], among others, to provide a well-rounded and authoritative set of recommendations. All example code has been written to reflect the latest APIs and has been explained line-by-line for clarity.

---

[1] [2] [4] [5] [6] [7] [8] [20] [21] [22] [23] [24] [25] [26] How to Control DC Motors Using ESP32 and L298N Module - Last Minute Engineers
https://lastminuteengineers.com/esp32-l298n-dc-motor-control-tutorial/

[3] [60] LED Control (LEDC) - - — Arduino ESP32 latest documentation
https://docs.espressif.com/projects/arduino-esp32/en/latest/api/ledc.html

[9] [10] [11] [55] ESP32 Servo Motor Web Server with Arduino IDE | Random Nerd Tutorials
https://randomnerdtutorials.com/esp32-servo-motor-web-server-arduino-ide/

[12] [18] [19] ESP32 Stepper Motor (28BYJ-48 and ULN2003 Driver) | Random Nerd Tutorials
https://randomnerdtutorials.com/esp32-stepper-motor-28byj-48-uln2003/

13 15 16 29 30 31 32 33 34 35 In-Depth: Control Stepper Motor with A4988 Driver Module & Arduino
https://lastminuteengineers.com/a4988-stepper-motor-driver-arduino-tutorial/

14 A4988 VS DRV8825[Video]: What are the differences between them?
https://www.utmel.com/components/a4988-vs-drv8825-what-are-the-differences-between-them?id=2003

17 How To Drive Stepper Motor Using A4988 IC And ESP32
https://www.makerguides.com/how-to-drive-stepper-motor-using-a4988-ic-and-esp32/

27 28 Motor Control with L9110 and ESP32 - A Beginner's Guide
https://www.espboards.dev/blog/motor-control-l9110-esp32/

36 Get the most out of your rotary encoder with an #ESP32. Interfacing ...
https://www.reddit.com/r/esp32/comments/10zlexl/get_the_most_out_of_your_rotary_encoder_with_an/

37 OneButton - Arduino Library List
https://www.arduinolibraries.info/libraries/one-button

38 Button2: Simplifying Button Control on Arduino & ESP Boards
https://openelab.io/blogs/learn/button2-library-simplify-button-control-on-arduino-and-esp-boards?srsltid=AfmBOooHaW6Fzm7OgAl0pJKjivhk1nrIX65RjCL9HfIUA8R21AIhIesp

39 esp-arduino-libs/ESP32_Button: Arduino library of driving button for ...
https://github.com/esp-arduino-libs/ESP32_Button

40 ESP32_Button - Arduino Documentation
https://docs.arduino.cc/libraries/esp32_button/

41 ESP32Servo - Arduino Library List
https://www.arduinolibraries.info/libraries/esp32-servo

42 AccelStepper Class Reference - AirSpayce
https://www.airspayce.com/mikem/arduino/AccelStepper/classAccelStepper.html

43 laurb9/StepperDriver: Arduino library for A4988, DRV8825 ... - GitHub
https://github.com/laurb9/StepperDriver

44 Valar-Systems/FastAccelStepper-ESP32 - GitHub
https://github.com/Valar-Systems/FastAccelStepper-ESP32

45 ESP32Encoder.h - GitHub
https://github.com/madhephaestus/ESP32Encoder/blob/master/src/ESP32Encoder.h

46 Rotary Encoder ESP32 in Arduino Code : KY040 Tutorial - uPesy
https://www.upesy.com/blogs/tutorials/rotary-encoder-esp32-with-arduino-code?srsltid=AfmBOoove6aod-_LlzilePxUssZ0D0alwb-jhqR1sbXOlhO2S2PQzV3q

47 ArduinoOTA: Over-the-air (OTA) updates made easy #ESP32
https://www.programmingelectronics.com/arduinoota/

48 49 50 51 52 Smooth Stepper Motor Control with ESP32 Dual Core | Medium
https://protonestiot.medium.com/smooth-stepper-motor-control-with-esp32-using-5aa3a083c11c

53 ESP32 with FreeRTOS (Arduino IDE) – Getting Started: Create Tasks
https://randomnerdtutorials.com/esp32-freertos-arduino-tasks/

54 Announcing the Arduino ESP32 Core version 3.0.0
https://developer.espressif.com/blog/announcing-the-arduino-esp32-core-version-3-0-0/

56 61 62 Flash ESP32 Firmware Like a Pro using esptool.py - Guide
https://www.espboards.dev/blog/standalone-esptool-basics/

57 What is ESP32Encoder? - ArduBadge
https://www.ardu-badge.com/ESP32Encoder

58 ESP32 core version 3.x.x has some “breaking changes”
https://community.blynk.cc/t/esp32-core-version-3-x-x-has-some-breaking-changes/68207

59 Migration from 2.x to 3.0 - - — Arduino ESP32 latest documentation
https://docs.espressif.com/projects/arduino-esp32/en/latest/migration_guides/2.x_to_3.0.html

63 Flashing Firmware - ESP32 - — esptool latest documentation
https://docs.espressif.com/projects/esptool/en/latest/esp32/esptool/flashing-firmware.html