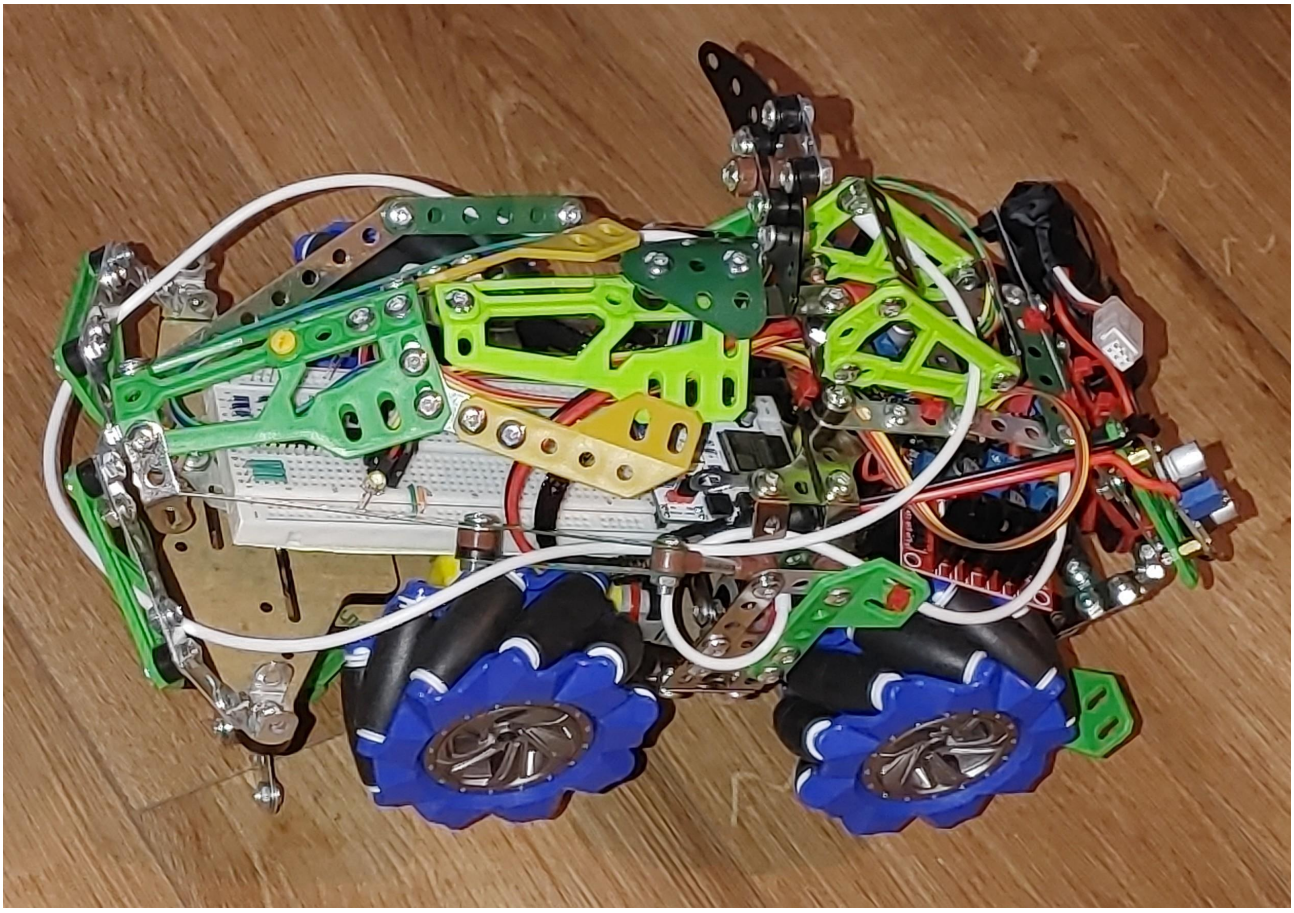
Byte Rider

Version

Table of Contents

ByteRider Highlights



OVERVIEW

At the heart of this project is ESP32-C3 IoT Link, which enables a customizable remote-controlled car to respond to real-time control inputs, handle speed adjustments, directional changes, and even extend features like measuring system telemetry values such as voltage, current, temperature, etc. The foundational setup uses ESP-NOW for transmitter and receiver devices, allowing you to control the car's behaviour wirelessly. While the design and physical appearance of the RC car can vary wildly depending on your creativity and available hardware, the control system remains the same. To facilitate wireless communication between devices, the system employs IoT Link powered by ESP-NOW, which is a lightweight and connection-free protocol ideal for fast, low-latency data transmission between ESP32 microcontrollers.

The control values are sent wirelessly to the remotely controlled car using the ESP-NOW protocol. ESP-NOW enables fast and efficient communication between ESP32 devices without the need for a Wi-Fi router, network, or pairing. The provided tutorial demonstrates a functional setup where a transmitter encapsulates and sends data to a receiver to define the car's speed and direction, forming the core communication loop. While the baseline implementation focuses on movement, additional features like lights, sensors, or telemetry can easily be integrated by expanding the source code. This modular design gives you the freedom to customize both the appearance and behaviour of your RC car, resulting in endless creative possibilities.

ABSTRACT

To enable real-time remote operation of the RC car, the system translates joystick x- and y- axis inputs into PWM (Pulse Width Modulation) signals that control the DC motors. These PWM values are stored in a predefined data structure, which is then transmitted wirelessly using ESP-NOW — a low-latency, connectionless communication protocol developed by Espressif. Both the transmitter and receiver modules are based on ESP32-C3 microcontrollers.

On the transmitter side, the joystick's X and Y coordinates are continuously monitored and converted into PWM parameters. These values are packed into the data structure and sent via ESP-NOW to the receiver.

The receiver module listens for incoming ESP-NOW packets, extracts the PWM control data, and applies it directly to the DC motors. This communication flow allows the RC car to respond instantly to user input, managing speed and direction without any physical connection between the devices.

HOW DOES IT WORK?

The bitByteRider RC car is powered by ESP32-C3 bitBoard. The Schematic and KiCAd PCB board files are available on GitHub : https://github.com/alexandrebobkov/ESP32-C3_Breadboard-Adapter

The bitByteRider RC car operates using two main units: the *transmitter* , which reads and sends the joystick's X and Y values, and the *receiver* , which interprets these values and converts them into PWM signals to control the DC motors. Both units communicate via **ESP-NOW** , a low-latency, connectionless wireless protocol that requires no Wi-Fi network or pairing.

In addition to enabling real-time control, using ESP-NOW introduces key networking concepts such as **data encapsulation** and structured communication. By using data structures to group control variables, you gain hands-on experience with how information is packaged and transmitted, laying the groundwork for understanding the fundamentals of network communication in embedded systems.

The joystick used in the bitByteRider RC car remote unit outputs analog voltages ranging from 0V to 3.3V on both the x- and y-axes, depending on the position of the joystick. These voltage levels are read by the ESP32-C3's ADC (Analog-to-Digital Converter) inputs.

When the joystick is in its neutral (centred) position, the ADC inputs on the ESP32-C3 receive approximately 1.65V on both axes. This midpoint voltage is interpreted and interpolated into a PWM (Pulse Width Modulation) value of 0, indicating no movement or motor activity.

As the joystick is pushed to its maximum positions along the x- and y-axis, the voltage increases up to 3.3V. This maximum voltage is interpolated to a PWM value of 1024, which corresponds to a 100% duty cycle on the receiver side, resulting in full-speed operation of the DC motors.

To transmit control data, the X and Y axis values are encapsulated in a C struct, along with the receiver's **MAC** address, and sent wirelessly using ESP-NOW. This protocol enables low-latency, connectionless communication between the transmitter and receiver without requiring a Wi-Fi network or pairing.

Upon reception, the RC car's receiver decapsulates the data, extracts the joystick values, and interpolates them into PWM signals. These signals are then used to control the rotation speeds of the DC motors, enabling smooth and responsive remote control.

This process not only facilitates real-time control but also introduces you to key networking concepts such as data encapsulation, data structs, and the fundamentals of wireless data transmission in embedded systems.

What is encapsulation?

Encapsulation refers to the process of organizing and packaging data into a structured format before it is transmitted between devices. This is a fundamental concept in networking and communication protocols, including those used in IoT systems.

Reserved Pins & GPIOs

The following table summarizes GPIOs and pins reserved for operations purposes.

The GPIO numbers correspond to those on the ESP32-C3 WROOM microcontroller. The Pin number corresponds to the pin on the Breadboard and Power adapter development board.

Reading the Joystick x- and y- axis

To determine the position of the Joystick, the BitRider RC car uses ADC to measure voltage on two GPIOs connected to the joystick x- and y- axis potentionometers (**GPIO0** and **GPIO1**).

Controlling the Direction and Speed

To set any desired speed of BiteRider RC car, the *ESP32-C3 Breadboard Adapter DevBoard* uses PWM to control the rotation speed of DR motors. Similarly, to set the direction of the RC car, the rotation speed of corresponding DC motors is changed as required.

Due to the design and limited number of available GPIOs, the *ESP32-C3 Breadboard DevBoard* can control rotation speed and direction of DC motors in pairs only (i.e. left and right side). Consequently, this means that the four PWM channels used for controlling the direction of the RC car.

Based on this constraint, the RC car can only move front, back, and turn/rotate left and right. Any other movements are not possible (i.e. diagonal or sideways).

PWM of DC Motors	Direction
PWM(left) = PWM(right)	Straight
PWM(left) > PWM(right)	Left

PWM(left) < PWM(right)	Right

What is PWM?

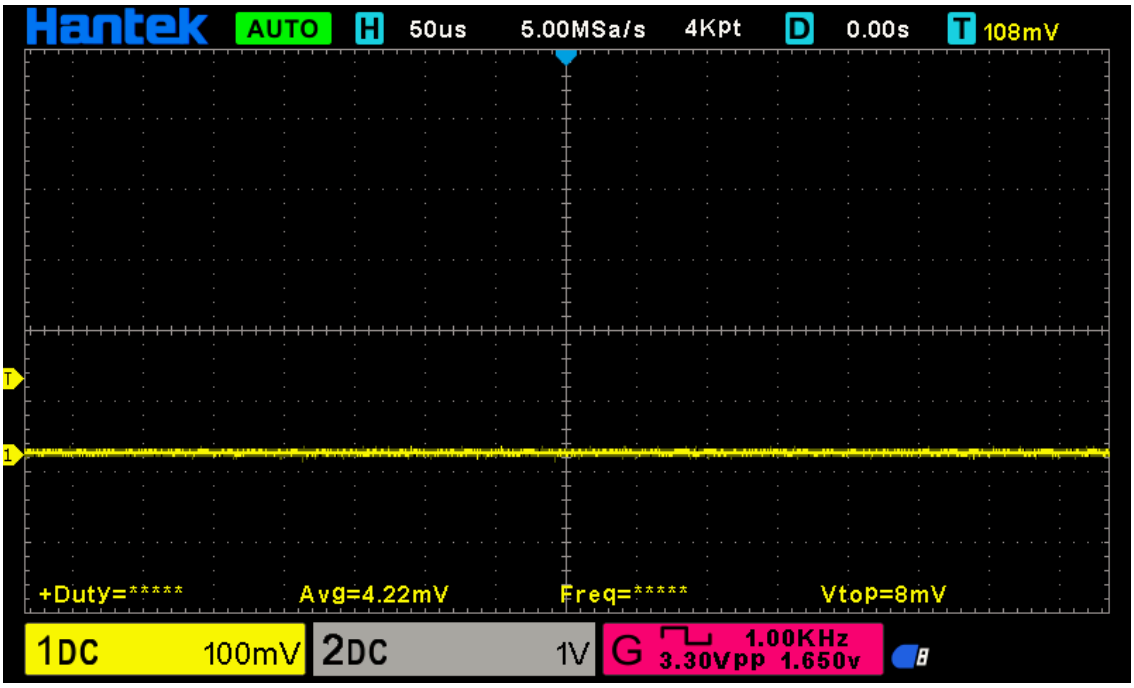
PWM stands for Pulse Width Modulation. It is a technique used to simulate analog voltage levels using discrete digital signals. It works by rapidly switching a digital GPIO pin between HIGH (on) and LOW (off) states at a fixed frequency (often, at base frequency of 5 kHz). The duty cycle—the percentage of time the signal is HIGH in one cycle determines the effective voltage delivered to a device. A higher duty cycle increases the motor speed, and a lower duty cycle decreases the motor speed. This allows for fine-grained speed control without needing analog voltage regulators.

A pair of PWM channels are used per DC motor for defining their rotation speed and direction on each side. In particular, **GPIO6** and **GPIO5** provide PWM to the left- and right- side DC motors to rotate in a **clockwise** direction. Similarly, **GPIO4** and **GPIO7** provide PWM to the left- and right- side DC motors to rotate in a **counter-clockwise** direction. Changing PWM on each channel determines the speed and direction of the RC car.

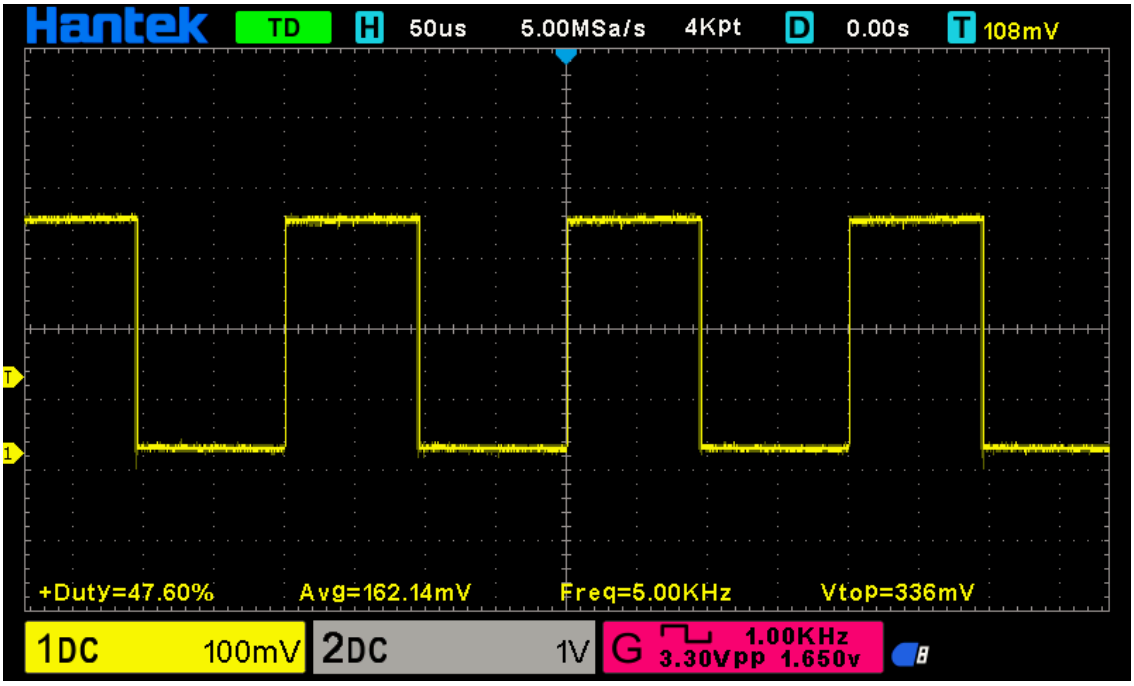
The table below summarizes the GPIO pins used for PWM to control the direction of the DC motors in the remote-controlled car.

GPIOs	State	Description	Function
GPIO6, GPIO4	PWM	Left & Right DC Motors spin clockwise	Forward
GPIO5, GPIO7	PWM	Left & Right DC Motors spin counterclockwise	Reverse
GPIO6, GPIO7	PWM	Left DC Motors spin clockwise. Right DC Motors spin counterclockwise	Left
GPIO4, GPIO5	PWM	Left DC Motors spin counterclockwise. Right DC Motors spin clockwise	Right

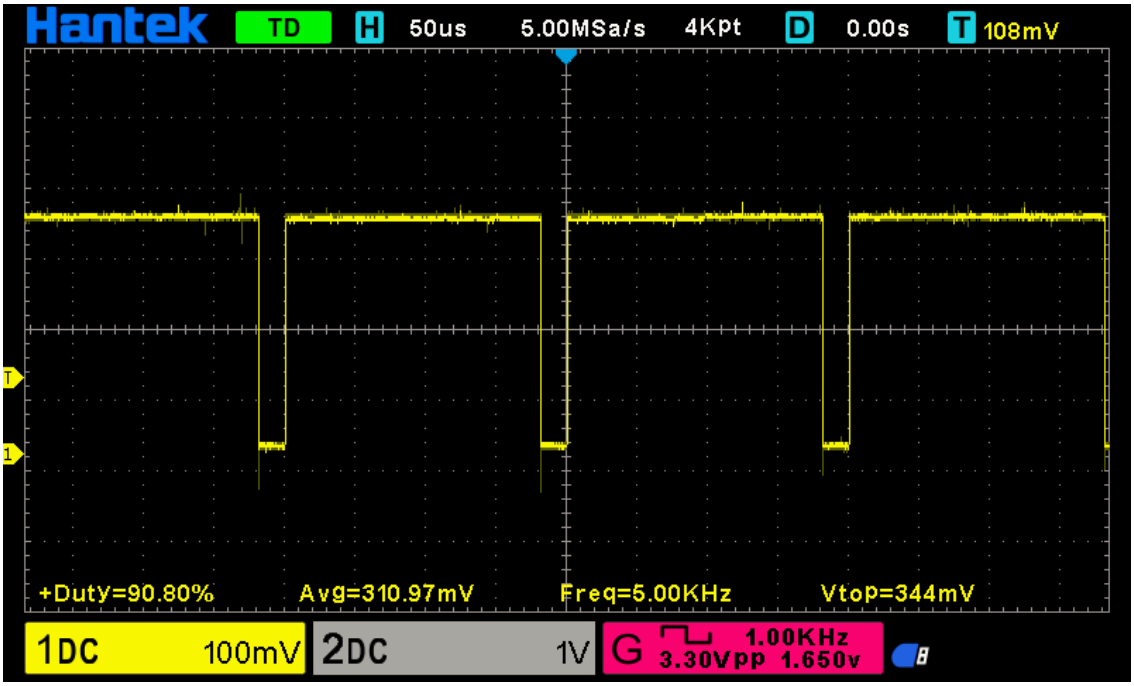
The following images illustrate various PWM duty cycles registered by oscilloscope (duty cycles 0%, 48% and 91%, resp.).



DC Motor PWM duty cycle 0%



DC Motor PWM duty cycle 47.6%



DC Motor PWM duty cycle 90.8%

Fusion of Software with Hardware

On one hand, we have the hardware designed so that the joystic x- and y- axis, and DC motors are wired to the proper GPIOs on the ESP32-C3 WROOM microcontroller. On the other hand, we have the software that reads the joystick x- and y- axis, sends the data to the receiver device, and converts that to PWM values on the receiver device.

In essense, the direction and speed of the bitByte Rider car is controlled by the two variables. On the remote controller device, the joystic x- and y- axis values are sent to the receiver device in a raw format (i.e. analog voltages, “as-is”). On the receover device, these two values are converted to the two PWM values; one for each pair of DC motors on left and right side.

When the joystick is pushed forward, the X-axis voltage remains at 1.65V (neutral), while the Y-axis voltage rises to 3.3V. The receiver on the RC car interprets this input and generates 100% PWM duty cycle signals on both sides, driving the car forward at full speed.

Similarly, when the joystick is pushed fully to the left or right, the X-axis voltage shifts while the Y-axis remains neutral. For a left turn, the receiver translates the signal into 100% PWM on the left-side motors and 0% on the right-side motors, causing the car to pivot. The opposite occurs for a right turn, with 100% PWM on the right and 0% on the left, enabling precise directional control.

The table below summarizes the reserved GPIOs. These GPIOs are hard-wired to the corresponding components, and hard-coded in the corresponding functions. For example, the GPIOs 0 and 1 are hard-wired to the joystick x- and y- axis, respectively; and, hard-coded to read analog values and store them in the corresponding x- and y- variables.

GPIO	Pin	Function	Notes
0	16	Joystick x-axis	ADC1_CH0
1	15	Joystick y-axis	ADC1_CH1
8	5	Joystick push button	NC
6	4	PWM for clockwise rotation of left-side motors	LEDC_CHANNEL_1
5	3	PWM for clockwise rotation of right-side motors	LEDC_CHANNEL_0
4	2	PWM for counter-clockwise rotation of right-side motors	LEDC_CHANNEL_2
7	6	PWM for counter-clockwise rotation of left-side motors	LEDC_CHANNEL_3

The struct used to store motor PWM values is shown below. While the bitByteRider RC car can be effectively controlled using just two PWM signals—one for each side—the structure is designed to hold four values, allowing room for future enhancements. This forward-thinking design supports potential upgrades such as improved maneuverability, individual wheel control, or advanced driving modes, making the system more adaptable and scalable for future development.

```
struct motors_rpm {
    int motor1_rpm_pwm;
    int motor2_rpm_pwm;
    int motor3_rpm_pwm;
    int motor4_rpm_pwm;
};
```

On the transmitter`` device, the PWM values for the DC motors are send to the receover using the following function. The variable **receiver_mac** stores the MAC address of the receiver device (ESP32-C3 bitBoard on the RC car).

```
// Function to send data to the receiver
void sendData (void) {
    sensors_data_t buffer;              // Declare data struct

    buffer.crc = 0;
    buffer.x_axis = 0;
    buffer.y_axis = 0;
    buffer.nav_bttn = 0;
    buffer.motor1_rpm_pwm = 0;
    buffer.motor2_rpm_pwm = 0;
    buffer.motor3_rpm_pwm = 0;
    buffer.motor4_rpm_pwm = 0;

    // Display brief summary of data being sent.
    ESP_LOGI(TAG, "Joystick (x,y) position ( 0x%04X, 0x%04X )",
(uint8_t)buffer.x_axis, (uint8_t)buffer.y_axis);
    ESP_LOGI(TAG, "pwm 1, pwm 2 [ 0x%04X, 0x%04X ]",
(uint8_t)buffer.pwm, (uint8_t)buffer.pwm);
    ESP_LOGI(TAG, "pwm 3, pwm 4 [ 0x%04X, 0x%04X ]",
(uint8_t)buffer.pwm, (uint8_t)buffer.pwm);

    // Call ESP-NOW function to send data (MAC address of receiver,
pointer to the memory holding data & data length)
    uint8_t result = esp_now_send(receiver_mac, &buffer,
sizeof(buffer));

    // If status is NOT OK, display error message and error code (in
hexadecimal).
    if (result != 0) {
        ESP_LOGE("ESP-NOW", "Error sending data! Error code:
0x%04X", result);
        deletePeer();
    }
    else
        ESP_LOGW("ESP-NOW", "Data was sent.");
}
```

This function is invoked by a dedicated FreeRTOS task every 100 milliseconds, ensuring consistent and timely transmission of control data to the receiver device. By leveraging FreeRTOS's precise task scheduling, the system maintains low-latency communication and predictable behavior—critical for real-time control in embedded applications.

```
// Continous, periodic task that sends data.
static void rc_send_data_task (void *arg) {

    while (true) {
        if (esp_now_is_peer_exist(receiver_mac))
            sendData();
        vTaskDelay (100 / portTICK_PERIOD_MS);
```

```
    }
}
```

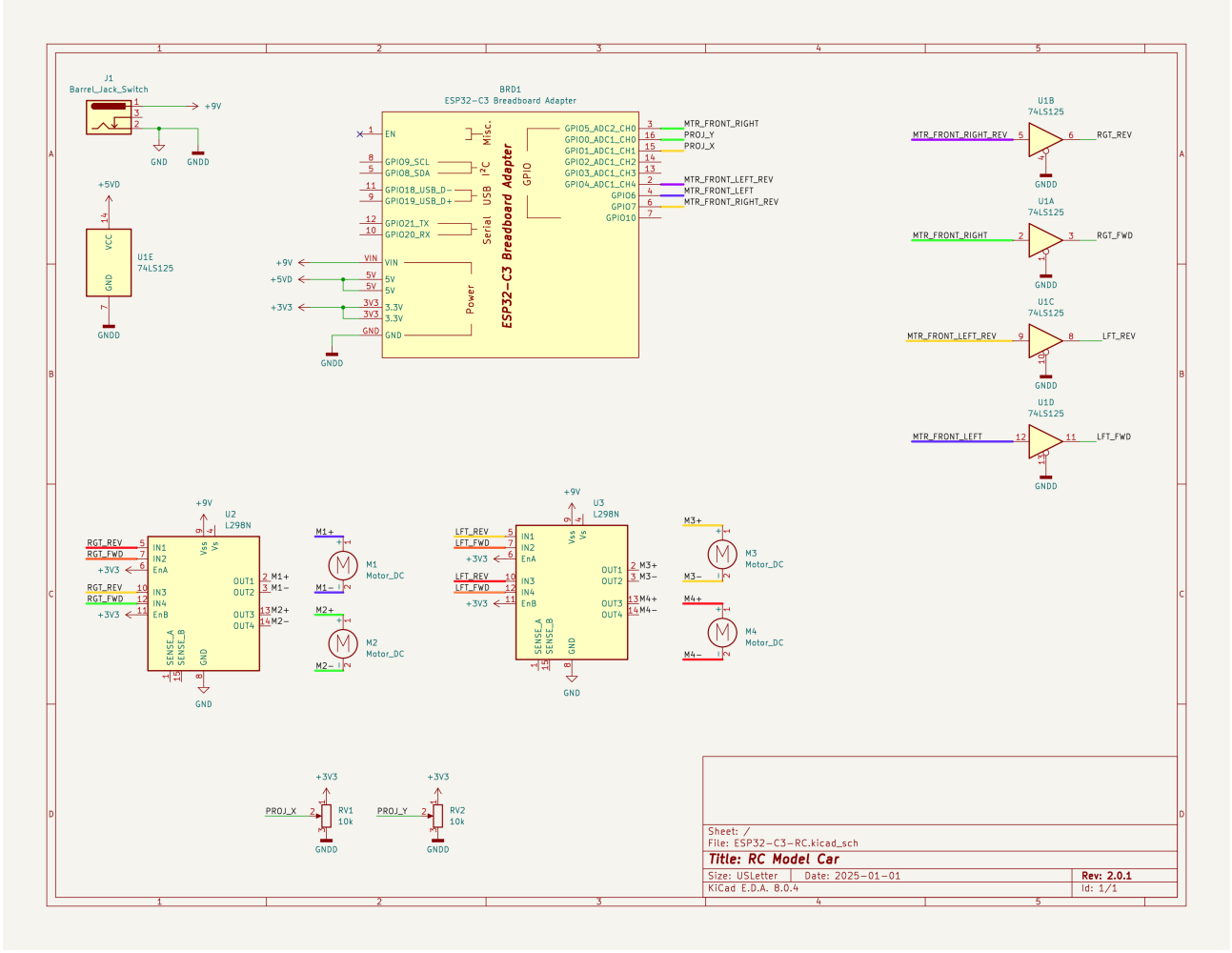
As data is being sent, the function onDataSent() is called to check & display the status of the data transmission.

```
// Call-back for the event when data is being sent
void onDataSent (uint8_t *mac_addr, esp_now_send_status_t status) {
    ESP_LOGW(TAG, "Packet send status: 0x%04X", status);
}

... ... ...
... ... ...
```

On the receiver device, the data is saved in the variables by the call-back function onDataReceived().

```
// Call-back for the event when data is being received
void onDataReceived (uint8_t *mac_addr, uint8_t *data, uint8_t
data_len) {

    buf = (sensors_data_t*)data;                              //
Allocate memory for buffer to store data being received
    ESP_LOGW(TAG, "Data was received");
    ESP_LOGI(TAG, "x-axis: 0x%04x", buf->x_axis);
    ESP_LOGI(TAG, "x-axis: 0x%04x", buf->y_axis);
    ESP_LOGI(TAG, "PWM 1: 0x%04x", buf->motor1_rpm_pwm);
}
```

Schematic


J1
Barrel_Jack_Switch
+9V
GND
GNDD
BRD1
ESP32-C3 Breadboard Adapter
ESP32-C3 Breadboard Adapter
Misc.
I²C
Serial
USB
GPIO
Power
EN
GPIO9_SCL
GPIO8_SDA
GPIO18_USB_D-
GPIO19_USB_D+
GPIO21_TX
GPIO20_RX
VIN
5V
3.3V
GPIO5_ADC2_CH0
GPIO0_ADC1_CH0
GPIO1_ADC1_CH1
GPIO2_ADC1_CH2
GPIO3_ADC1_CH3
GPIO4_ADC1_CH4
GPIO6
GPIO7
GPIO10
MTR_FRONT_RIGHT
PROJ_Y
PROJ_X
MTR_FRONT_LEFT_REV
MTR_FRONT_LEFT
MTR_FRONT_RIGHT_REV
U1E
74LS125
U1B
U1A
U1C
U1D
MTR_FRONT_RIGHT_REV
RGT_REV
MTR_FRONT_RIGHT
RGT_FWD
MTR_FRONT_LEFT_REV
LFT_REV
MTR_FRONT_LEFT
LFT_FWD
U2
L298N
U3
L298N
RGT_REV
RGT_FWD
LFT_REV
LFT_FWD
M1
Motor_DC
M2
Motor_DC
M3
Motor_DC
M4
Motor_DC
PROJ_X
PROJ_Y
RV1
10k
RV2
10k
Sheet: /
File: ESP32-C3-RC.kicad_sch
Title: RC Model Car
Size: USLetter
Date: 2025-01-01
Rev: 2.0.1
KiCad E.D.A. 8.0.4
Id: 1/1

DATA STRUCTS

The struct serves as the data payload for sending control signals from the transmitting device to the receiver using ESP-NOW. In addition, it may contain additional data such as telemetry, battery status, etc. The *sensors_data_t* struct encapsulates all control commands and sensor states relevant to the vehicle's operation. It's intended to be sent from a transmitting device (like a remote control) to a receiver (such as a microcontroller on board of the vehicle).

```
typedef struct {
    int         x_axis;             // Joystick x-position
    int         y_axis;             // Joystick y-position
    bool        nav_bttn;           // Joystick push button
    bool        led;                // LED ON/OFF state
    uint8_t     motor1_rpm_pwm;     // PWMs for 4 DC motors
    uint8_t     motor2_rpm_pwm;
    uint8_t     motor3_rpm_pwm;
    uint8_t     motor4_rpm_pwm;
} __attribute__((packed)) sensors_data_t;
```

```
struct motors_rpm {
    int motor1_rpm_pwm;
    int motor2_rpm_pwm;
    int motor3_rpm_pwm;
    int motor4_rpm_pwm;
};
```

When used with communication protocols like ESP-NOW, this struct is **encoded** into a byte stream, then **transmitted** at regular intervals or in response to user input, and finally **decoded** on the receiving end to control hardware.

What is struct?

In C programming, a struct (short for structure) is a user-defined data type that lets you group multiple variables of different types together under a single name. It's like a container that holds related information — perfect for organizing data that logically belongs together. Structs are especially powerful in systems programming, embedded projects, and when dealing with raw binary data — like parsing sensor input or transmitting control packets over ESP-NOW.

Data Payload

x_axis and *y_axis* fields capture analog input from a joystick, determining direction and speed. *nav_bttn* represents a joystick push-button.

led allows the transmitter to toggle an onboard LED and is used for status indication (e.g. pairing, battery warning, etc).

motor1_rpm_pwm to *motor4_rpm_pwm* provide individual PWM signals to four DC motors. This enables fine-grained speed control, supports differential drive configurations, and even allows for maneuvering in multi-directional platforms like omni-wheel robots.

Why use __attribute((packed))?

ESP-NOW uses fixed-size data packets (up to 250 bytes). The *__attribute__((packed))* removes compiler-added padding for precise byte alignment.

As *packed* attribute tells the compiler not to add any padding between fields in memory, this makes the struct:

- Compact
- Predictable for serialization over protocols like UART or ESP-NOW
- Ideal for low-latency transmission in embedded systems

This ensures the receiver interprets the exact byte layout you expect, minimizing bandwidth and maximizing compatibility across platforms.

TRANSMITTER

Configuration Variables

```
uint8_t receiver_mac[ESP_NOW_ETH_ALEN]  = {0xe4, 0xb0, 0x63, 0x17,
0x9e, 0x44};

typedef struct {
    int         x_axis;             // Joystick x-position
    int         y_axis;             // Joystick y-position
    bool        nav_btn;            // Joystick push button
    bool        led;                // LED ON/OFF state
    uint8_t     motor1_rpm_pwm;     // PWMs for each DC motor
    uint8_t     motor2_rpm_pwm;
    uint8_t     motor3_rpm_pwm;
    uint8_t     motor4_rpm_pwm;
} __attribute__((packed)) sensors_data_t;
```

Reading Joystick x- and y- Axis Values

Sending & Ecapsulating Data

```
void sendData (void) {

    ... ... ...
    ... ... ...

    buffer.x_axis = x_axis;
    buffer.y_axis = y_axis;

    // Call ESP-NOW function to send data (MAC address of receiver,
pointer to the memory holding data & data length)
    uint8_t result = esp_now_send((uint8_t*)receiver_mac, (uint8_t
*)&buffer, sizeof(buffer));

    ... ... ...
    ... ... ...
}
```

Main Function

```
#include "freertos/FreeRTOS.h"
#include "nvs_flash.h"
#include "esp_err.h"

... ... ...
... ... ...

void app_main(void) {

    ... ... ...
    ... ... ...

    // Initialize internal temperature sensor
    chip_sensor_init();

    // Initialize NVS
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK( nvs_flash_erase() );
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK( ret );
    wifi_init();
    joystick_adc_init();
    transmission_init();
    system_led_init();

    ... ... ...
    ... ... ...
}
```

RECEIVER

Configuration Variables

```
uint8_t transmitter_mac[ESP_NOW_ETH_ALEN] = {0x9C, 0x9E, 0x6E, 0x14,
0xB5, 0x54};

typedef struct {
    int         x_axis;             // Joystick x-position
    int         y_axis;             // Joystick y-position
    bool        nav_bttn;           // Joystick push button
    bool        led;                // LED ON/OFF state
    uint8_t     motor1_rpm_pwm;     // PWMs for 4 DC motors
    uint8_t     motor2_rpm_pwm;
    uint8_t     motor3_rpm_pwm;
    uint8_t     motor4_rpm_pwm;
} __attribute__((packed)) sensors_data_t;
```

```
struct motors_rpm {
    int motor1_rpm_pwm;
    int motor2_rpm_pwm;
    int motor3_rpm_pwm;
    int motor4_rpm_pwm;
};
```

Receiving & Extracting Data

```
void onDataReceived (const uint8_t *mac_addr, const uint8_t *data,
uint8_t data_len) {

    ... ... ...
    ... ... ...

    ESP_LOGI(TAG,
"Data received from: %02x:%02x:%02x:%02x:%02x:%02x, len=%d",
mac_addr[0], mac_addr[1], mac_addr[2], mac_addr[3], mac_addr[4],
mac_addr[5], data_len);
    memcpy(&buf, data, sizeof(buf));

    x_axis = buf.x_axis;
```

```
    y_axis = buf.y_axis;

    ... ... ...
    ... ... ...
}
```

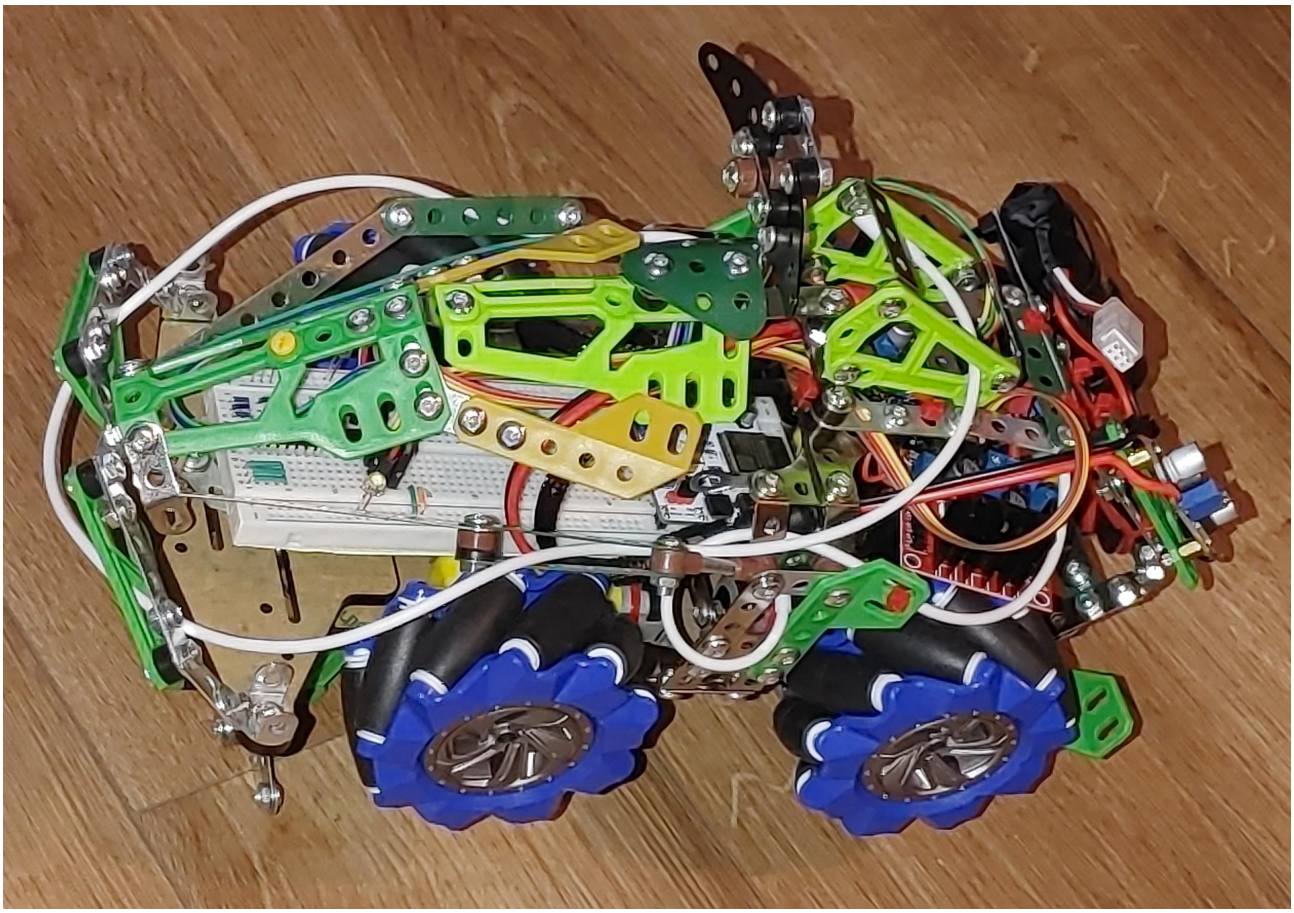
Main Function

```
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "nvs_flash.h"
#include "esp_err.h"

... ... ...
... ... ...

void app_main(void) {

    ... ... ...
    ... ... ...

    // Initialize NVS
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES ||
        ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK( nvs_flash_erase() );
        ret = nvs_flash_init();
    }

    ESP_ERROR_CHECK( ret );
    wifi_init();
    ESP_ERROR_CHECK(esp_now_init());

    esp_now_peer_info_t transmitterInfo = {0};
    memcpy(transmitterInfo.peer_addr, transmitter_mac,
ESP_NOW_ETH_ALEN);
    transmitterInfo.channel = 0; // Current WiFi channel
    transmitterInfo.ifidx = ESP_IF_WIFI_STA;
    transmitterInfo.encrypt = false;
    ESP_ERROR_CHECK(esp_now_add_peer(&transmitterInfo));

    ESP_ERROR_CHECK(esp_now_register_recv_cb((void*)onDataReceived));

    system_led_init();

    ... ... ...
```

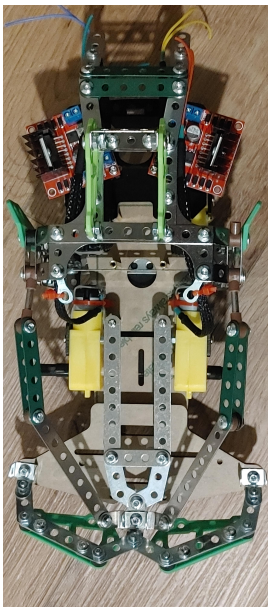
```
    ... ... ...
}
```

WORK-IN-PROGRESS WALK THROUGH

Finished Work

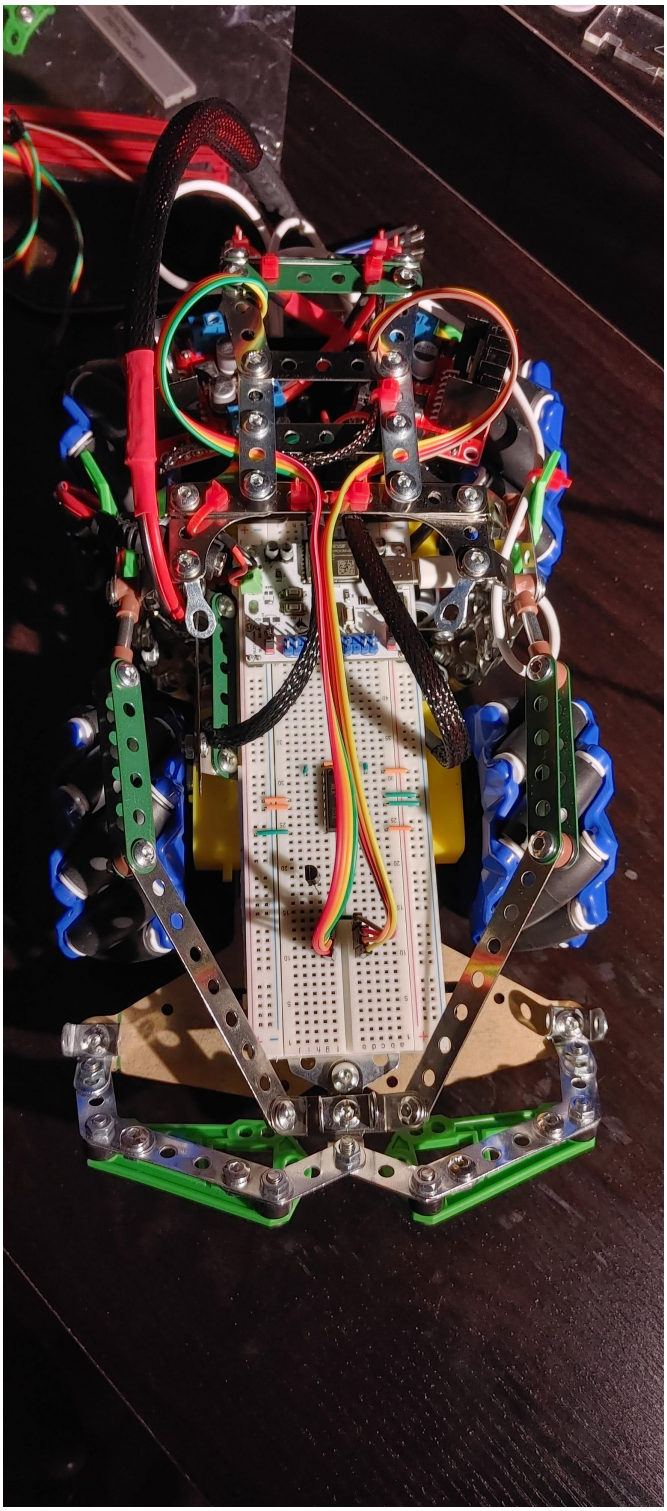


Chassis



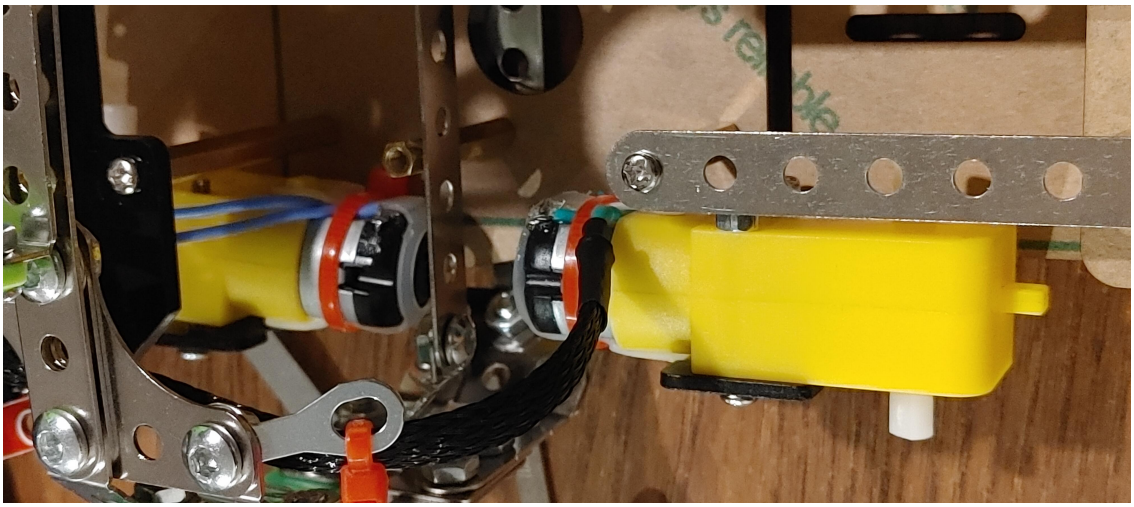
Completed chassis with only DC motor controllers installed.

Wiring



Completed wiring.

Motor Wires Harness



DC Motors wires secured inside harnes.

REFERENCES

GitHub

Complete source code with README.md file: https://github.com/alexandrebobkov/ESP-Nodes/blob/main/ESP-IDF_Robot/README.md

KiCAd Schematic and PCB design: https://github.com/alexandrebobkov/ESP32-C3_Breadboard-Adapter