Implementation Of A Remote-Controlled Model Car Using Bluetooth Low Energy

By Dominic Jacobo

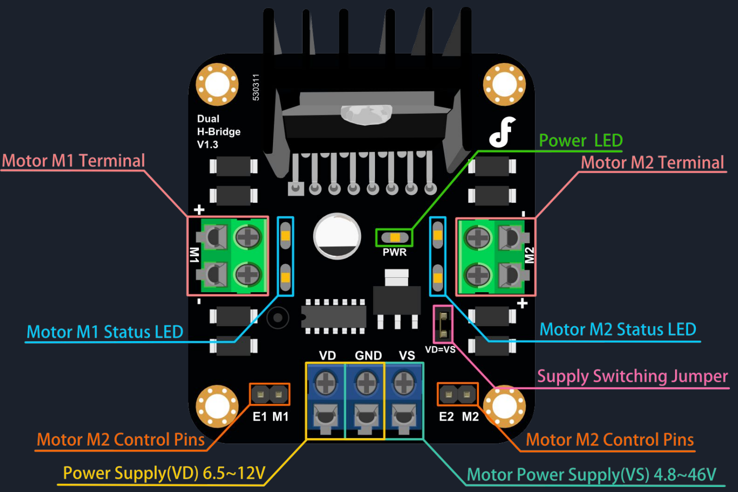
Overview

- Hardware
  - MCU: nRF5340 DK
  - Motor driver: DRI0002 (L298N)
  - Batteries: OVONIC 2S LiPo Battery 35C (Burst 70C)
  - Controls: Joysticks
- System Block diagram and Discussion
- Software
  - Bluetooth Low Energy
  - Nordic UART Service (NUS)
  - Remote Control program
  - RC car program
- Demo video
- Conclusion/Future Work

Hardware:
nRF5340 DK

- nRF5340 Soc:
  - Dual-core Bluetooth 5.4 SoC
  - 128 MHz Arm Cortex-M33 CPU with 1 MB Flash + 512 KB RAM (Application core)
  - 64 MHz Arm Cortex-M33 CPU with 256 KB Flash + 64 KB RAM (Network core)
  - 105 C extended operating temperature
  - 1.7-5.5 V supply voltage range
- nRF5340 DK:
  - Supports Bluetooth Low Energy, Bluetooth mesh, Thread, Zigbee
  - User-programmable LEDs(4) and buttons(4)
  - 2.4 GHZ antenna
  - 1.7-5.0 V supply from USB, external, Li-Po battery or CR2032 coin cell battery

Hardware:
DRI0002 (motor driver)



- Motor terminal
  - The terminals are used to connect to the motors, which labeled “+” and “-” representing motor polarity.
- Power
  - VD: Power Supply 6.5V~12V
  - VS: Motor Power Supply 4.8~46V
  - GND: The common ground of Logic Power Supply and Motor Supply
- Motor Control Pins

E	M	RUN
LOW	LOW/HIGH	STOP
HIGH	HIGH	Back Direction
HIGH	LOW	Forward Direction
PWM	LOW/HIGH	Speed

Output voltage = (on_time/off_time) * max_voltage

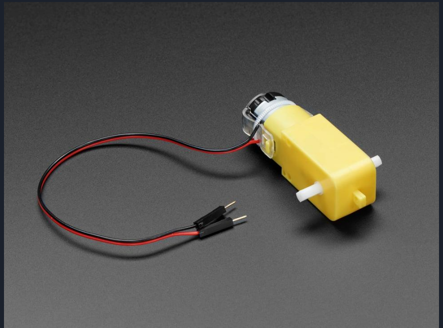
Hardware:
OVONIC 2S LiPo Battery & DC Gearbox Motor - TT Motor

- LiPo specification:
  - Voltage: 7.4v
  - Discharge Rate: 35C
  - Weight: 99g
  - Capacity: 2200mAh

Max current = Capacity X C-rating = 2.2 Ah * 35 = 77 Amps

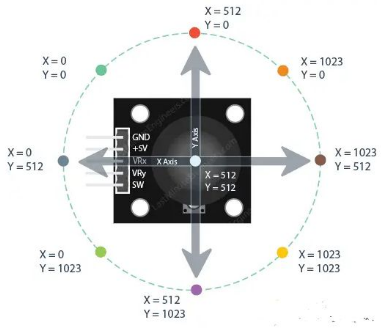
- AA alkaline (4) specification:
  - Voltage: 6v
  - Maximum current draw: 2A
  - Capacity: 2000mAh
  - Weight: 24g (4) = 96g

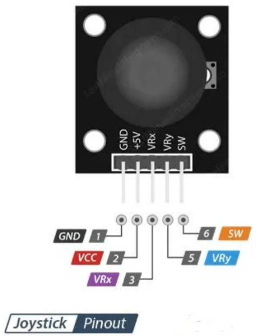
- Rate Voltage: 3~6V
- Min. Operating Speed (3V): 90+/- 10% RPM
- Min. Operating Speed (6V): 200+/- 10% RPM
- Stall Torque (3V): 0.4kg.cm
- Stall Torque (6V): 0.8kg.cm
- At 3VDC 150mA @ 120 RPM no-load, and 1.1 Amps when stalled
- At 6VDC 160mA @ 250 RPM no-load, and 1.5 Amps when stalled



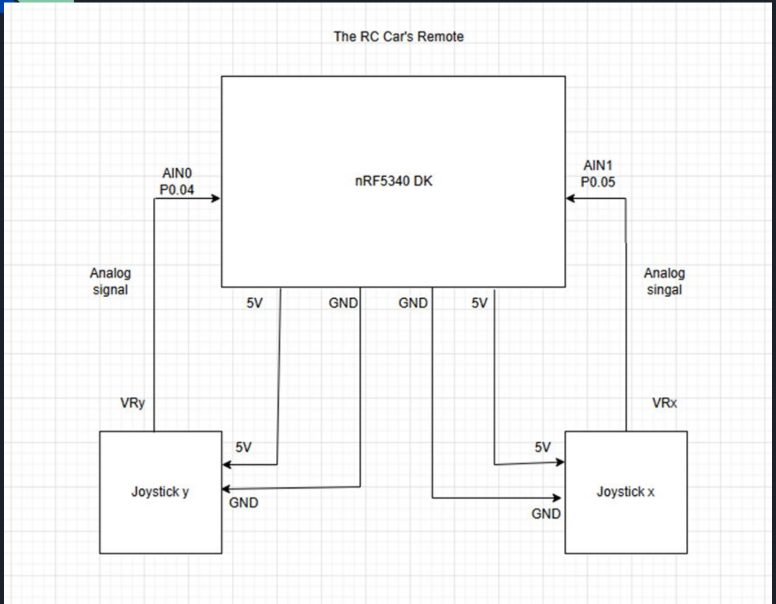
Hardware: Joystick

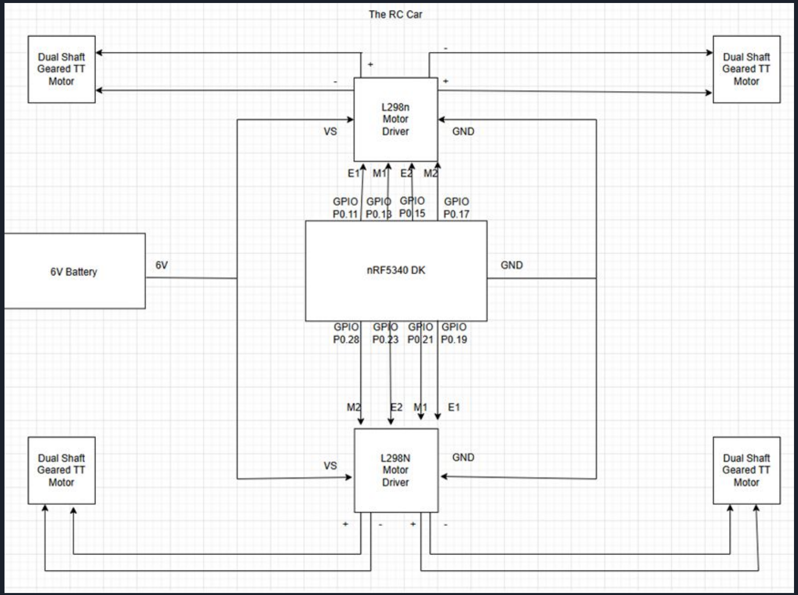
- The joysticks work by using two potentiometers. One for the x-axis and one for the y-axis
- The joystick converts the stick's position on these two axis - the X-axis (left to right) and the Y-axis (up and down)
- The output of the joysticks are analog signals that can be read using an ADC.
- The X axis potentiometer outputs to pin VRx and the y axis potentiometer outputs to pin VRy.





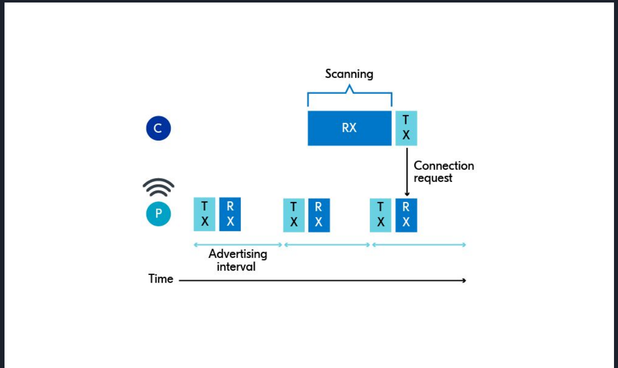
System Block Diagram

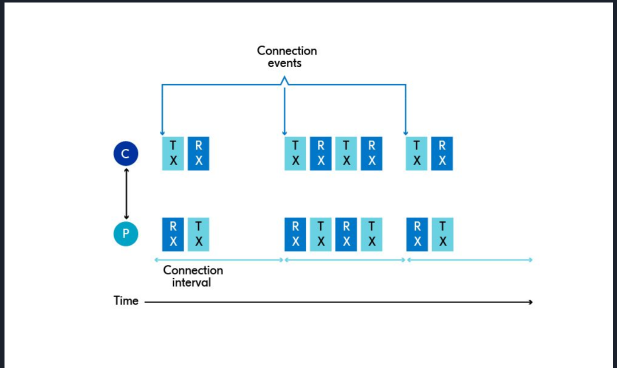




Software:
Bluetooth Low Energy (BLE)

- ❖ When Performing BLE one-to-one communication there are two roles
  - ➢ The Central (The remote control)
  - ➢ The Peripheral (The RC Car)
- ❖ The job of the Peripheral is to perform advertising, which means that it sends out packets of data in hopes that a central is listening
- ❖ The job of the Central is initiate the connection by scanning for advertising packets. After the central processes the packet it can initiate the connection by sending a connection request to the peripheral.
- ❖ When they are finally connected they will sleep until the connection interval and send data during the connection event





Software:
Nordic UART Service (NUS)

- In order to implement BLE into the project the Nordic UART Service (NUS) was used.
- This is a custom BLE GATT service that receives and writes data and serves as a bridge to the UART interface.
- This service emulates a serial port over BLE.
- NUS service is split into two parts: the NUS client and the NUS.
- NUS runs the GATT server, which is a device that stores data locally and provides access to that data to a remote GATT client. So it acts as a data source that a client can read from or write to over a BLE connection.
- The remote control was used as the NUS client and the RC Car as the NUS GATT server.
- The controller uses NUS to transfer the ADC readings to the RC car.
- The transfer of data is done through the NUS' APIs

Software:
Remote Control Program

```c
err = bt_conn_auth_cb_register(&conn_auth_callbacks);
if (err) {
    LOG_ERR("Failed to register authorization callbacks.");
    return 0;
}

err = bt_conn_auth_info_cb_register(&conn_auth_info_callbacks);
if (err) {
    printk("Failed to register authorization info callbacks.\n");
    return 0;
}

err = bt_enable(NULL);
if (err) {
    LOG_ERR("Bluetooth init failed (err %d)", err);
    return 0;
}
LOG_INF("Bluetooth initialized");

if (IS_ENABLED(CONFIG_SETTINGS)) {
    settings_load();
}

err = scan_init();
if (err != 0) {
    LOG_ERR("scan_init failed (err %d)", err);
    return 0;
}

err = nus_client_init();
if (err != 0) {
    LOG_ERR("nus_client_init failed (err %d)", err);
    return 0;
}

err = dk_buttons_init(button_handler);
if (err) {
    printk("Failed to initialize buttons (err %d)\n", err);
    return 0;
}

printk("Starting Bluetooth Central UART example\n");

err = bt_scan_start(BT_SCAN_TYPE_SCAN_ACTIVE);
if (err) {
    LOG_ERR("Scanning failed to start (err %d)", err);
    return 0;
```

```c
configure_timer();
configure_saadc();
configure_ppi();

for (;;) {
    /* Wait indefinitely for data to be sent over Bluetooth */
    struct uart_data_t *buf = k_fifo_get(&fifo_uart_rx_data,
                                         K_FOREVER);

    err = bt_nus_client_send(&nus_client, buf->data, buf->len);
    if (err) {
        LOG_WRN("Failed to send data over BLE connection"
            "(err %d)", err);
    }

    err = k_sem_take(&nus_write_sem, NUS_WRITE_TIMEOUT);
    if (err) {
        LOG_WRN("NUS send timeout");
    }

    k_free(buf);
}
```

```c
/* Define the SAADC sample interval in microseconds */
#define SAADC_SAMPLE_INTERVAL_US 50 // Was 50 (right now it was 100)

/* Define the buffer size for the SAADC */
#define SAADC_BUFFER_SIZE   8000 // Was 8000 (right now it was 3500)(10000)

/* Declaring an instance of nrfx_timer for TIMER2. */
const nrfx_timer_t timer_instance = NRFX_TIMER_INSTANCE(2);

/* Declare the buffers for the SAADC */
static int16_t saadc_sample_buffer[2][SAADC_BUFFER_SIZE];

/* STEP 4.3 - Declare variable used to keep track of which buffer was last assigned to the SAADC driver */
static uint32_t saadc_current_buffer = 0;
```

```c
case NRFX_SAADC_EVT_DONE:

    /* STEP 5.3 - Buffer has been filled. Do something with the data and proceed */
    int64_t average_AN0 = 0;
    int16_t max_AN0 = INT16_MIN;
    int16_t min_AN0 = INT16_MAX;
    int16_t current_value;

    int64_t average_AN1 = 0;
    int16_t max_AN1 = INT16_MIN;
    int16_t min_AN1 = INT16_MAX;

    for(int i=0; i < p_event->data.done.size; i++){
        current_value = ((int16_t *)(p_event->data.done.p_buffer))[i];
        average_AN0 += current_value;
        if(current_value > max_AN0){
            max_AN0 = current_value;
        }
        if(current_value < min_AN0){
            min_AN0 = current_value;
        }

        i++;
        current_value = ((int16_t *)(p_event->data.done.p_buffer))[i];
        average_AN1 += current_value;
        if(current_value > max_AN1){
            max_AN1 = current_value;
        }
        if(current_value < min_AN1){
            min_AN1 = current_value;
        }
    }
    //average = average/p_event->data.done.size;
    average_AN0 = average_AN0 / (p_event->data.done.size / 2);

    average_AN1 = average_AN1 / (p_event->data.done.size / 2);
```

```c
//LOG_INF("AVG=%d, MIN=%d, MAX=%d", (int16_t)average_AN0, min_AN0, max_AN0);
printk("AVG=%d, MIN=%d, MAX=%d\n", (int16_t)average_AN0, min_AN0, max_AN0);

//LOG_INF("AVG=%d, MIN=%d, MAX=%d", (int16_t)average_AN1, min_AN1, max_AN1);
printk("AVG=%d, MIN=%d, MAX=%d\n", (int16_t)average_AN1, min_AN1, max_AN1);


struct uart_data_t *buf;
buf = k_malloc(sizeof(*buf));
if(buf){
    buf->len = 0;
}
else{
    printk("Failed to allocate memory for buf in button_handler\n");
    return;
}

// Store the first uint16_t value into data[0] and data[1]
buf->data[0] = (uint8_t)(average_AN0 & 0xFF);
buf->data[1] = (uint8_t)((average_AN0 >> 8) & 0xFF);

// Store the second uint16_t value into data[2] and data[3]
buf->data[2] = (uint8_t)(average_AN1 & 0xFF);
buf->data[3] = (uint8_t)((average_AN1 >> 8) & 0xFF);

// Update the Length to reflect the number of bytes used
buf->len = 4; // Two uint16_t values = 4 bytes
k_fifo_put(&fifo_uart_rx_data, buf);
```

Software: RC Car Program

```c
err = bt_enable(NULL);
if (err) {
    error();
}

LOG_INF("Bluetooth initialized");

k_sem_give(&ble_init_ok);

if (IS_ENABLED(CONFIG_SETTINGS)) {
    settings_load();
}

err = bt_nus_init(&nus_cb);
if (err) {
    LOG_ERR("Failed to initialize UART service (err: %d)", err);
    return 0;
}

err = bt_le_adv_start(BT_LE_ADV_CONN, ad, ARRAY_SIZE(ad), sd,
            ARRAY_SIZE(sd));
if (err) {
    LOG_ERR("Advertising failed to start (err %d)", err);
    return 0;
}

/*New. Added to check if the dc motor device is ready */
if(!pwm_is_ready_dt(&pwm_motor)){
    LOG_ERR("Error: PWM device %s is not ready", pwm_motor.dev->name);
    return 0;
}

/*New.  Added to check if the dc motor device is ready (second channel) */
if(!pwm_is_ready_dt(&pwm_motor2)){
    LOG_ERR("Error: PWM device %s is not ready", pwm_motor2.dev->name);
    return 0;
}

if(!pwm_is_ready_dt(&pwm_motor3)){
    LOG_ERR("Error: PWM device %s is not ready", pwm_motor3.dev->name);
    return 0;
}

if(!pwm_is_ready_dt(&pwm_motor4)){
    LOG_ERR("Error: PWM device %s is not ready", pwm_motor4.dev->name);
```

```c
if(!pwm_is_ready_dt(&pwm_motor4)){
    LOG_ERR("Error: PWM device %s is not ready", pwm_motor4.dev->name);
    return 0;
}

/*New. Added to checkifthe GPIO device is ready */
if(!gpio_is_ready_dt(&M1_Pin)){
    LOG_ERR("Error: GPIO pin %d is not ready", M1_Pin.pin);
    return 0;
}

/*New. Added to check if the GPIO pin 2 device is ready */
if(!gpio_is_ready_dt(&M2_Pin)){
    LOG_ERR("Error: GPIO pin %d is not ready", M2_Pin.pin);
    return 0;
}

if(!gpio_is_ready_dt(&M3_Pin)){
    LOG_ERR("Error: GPIO pin %d is not ready", M3_Pin.pin);
    return 0;
}

if(!gpio_is_ready_dt(&M4_Pin)){
    LOG_ERR("Error: GPIO pin %d is not ready", M4_Pin.pin);
    return 0;
}

/*New. Used to configure a pin */
gpio_pin_configure_dt(&M1_Pin, GPIO_OUTPUT);
gpio_pin_configure_dt(&M2_Pin, GPIO_OUTPUT);
gpio_pin_configure_dt(&M3_Pin, GPIO_OUTPUT);
gpio_pin_configure_dt(&M4_Pin, GPIO_OUTPUT);

for (;;) {
    dk_set_led(RUN_STATUS_LED, (++blink_status) % 2);
    k_sleep(K_MSEC(RUN_LED_BLINK_INTERVAL));
}
```

```c
static void bt_receive_cb(struct bt_conn *conn, const uint8_t *const data,
              uint16_t len)
{
    int err;
    uint16_t average_AN0 = 1700;
    uint16_t average_AN1 = 1700;
    char addr[BT_ADDR_LE_STR_LEN] = {0};

    bt_addr_le_to_str(bt_conn_get_dst(conn), addr, ARRAY_SIZE(addr));

    LOG_INF("Received data from: %s", addr);

    //New code for extracting the two uint16_t values from the uint8_t data array
    if(len == 4){

    // Reconstruct the first uint16_t value from the first two bytes
    average_AN0 = (uint16_t)data[0] | ((uint16_t)data[1] << 8);

    // Reconstruct the second uint16_t value from the next two bytes
    average_AN1 = (uint16_t)data[2] | ((uint16_t)data[3] << 8);

    // Print the reconstructed values
    printf("Received uint16_t values:\n");
    printf("Value 1: %u\n", average_AN0);
    printf("Value 2: %u\n", average_AN1);
    }
    else{
        printk("Error: Received data length (%d) is too short to contain two uint16_t values.\n", len);
        return;
    }
```

```c
/*
    * pwm_motor & M1_pin are for the front left motor
    * pwm2_motor & M2_pin are for the front right motor
    * pwm3_motor & M3_pin are for the back left motor
    * pwm4_motor & M4_pin are for the back right motor
    */

    /* x joystick position also determines PWM and GPIO changes but as the outer if statement */

    /* This is done if the x joystick is in the right position */
    if(average_AN1 > 1800){
        if(average_AN0 > 3300){ // Move the motors backwards and to the right at full speed
            set_motor_speed(PWM_MOTOR_MAX_DUTY_CYCLE, &pwm_motor);
            gpio_pin_set_dt(&M1_Pin, 0);

            set_motor_speed(PWM_MOTOR_MIN_DUTY_CYCLE, &pwm_motor2);
            gpio_pin_set_dt(&M2_Pin, 0);

            set_motor_speed(PWM_MOTOR_MAX_DUTY_CYCLE, &pwm_motor3);
            gpio_pin_set_dt(&M3_Pin, 0);

            set_motor_speed(PWM_MOTOR_MIN_DUTY_CYCLE, &pwm_motor4);
            gpio_pin_set_dt(&M4_Pin, 0);
        }
        else if((average_AN0 <= 3300) && (average_AN0 > 1800)){ // Move the motos backwards and to the right at 60% speed
            set_motor_speed(PWM_50P_DUTY_CYCLE, &pwm_motor);
            gpio_pin_set_dt(&M1_Pin, 0);

            set_motor_speed(PWM_MOTOR_MIN_DUTY_CYCLE, &pwm_motor2);
            gpio_pin_set_dt(&M2_Pin, 0);

            set_motor_speed(PWM_50P_DUTY_CYCLE, &pwm_motor3);
            gpio_pin_set_dt(&M3_Pin, 0);

            set_motor_speed(PWM_MOTOR_MIN_DUTY_CYCLE, &pwm_motor4);
            gpio_pin_set_dt(&M4_Pin, 0);
        }
        else if((average_AN0 <= 1800) && (average_AN0 > 1650)){ // Don't move motors (even though x joystick is positioned right)
            set_motor_speed(PWM_MOTOR_MIN_DUTY_CYCLE, &pwm_motor);
            gpio_pin_set_dt(&M1_Pin, 1);

            set_motor_speed(PWM_MOTOR_MIN_DUTY_CYCLE, &pwm_motor2);
            gpio_pin_set_dt(&M2_Pin, 0);

            set_motor_speed(PWM_MOTOR_MIN_DUTY_CYCLE, &pwm_motor3);
            gpio_pin_set_dt(&M3_Pin, 0);
```

Demo Video

Conclusion/Future Work

- Conclusion: Implemented a working RC Car that has throttle and steering control using Bluetooth Low Energy
- Future Work:
  - Add a feature for automatic and manual headlights. Done using a light sensor
  - Add a feature for collision avoidance. Done using ultrasonic sensor or lidar sensor
  - Add a feature for detection for going uphill or downhill. Done using tilt sensor
  - Add a feature for car horn. Done using buzzer.