```
// Required App Version: 0.7.0
// ------------------------------------------------------------------------
// This Arduino Nano sketch accompanies the OpenBot Android application.
//
// The sketch has the following functionalities:
//  - receive control commands and sensor config from Android application (USB serial)
//  - produce low-level controls (PWM) for the vehicle
//  - toggle left and right indicator signals
//  - wheel odometry based on optical speed sensors
//  - estimate battery voltage via voltage divider
//  - estimate distance based on sonar sensor
//  - control LEDs for status and at front and back of vehicle
//  - send sensor readings to Android application (USB serial)
//  - display vehicle status on OLED
//
// Dependencies: Install via "Tools --> Manage Libraries" (type library name in the search field)
//  - Interrupts: PinChangeInterrupt by Nico Hood (read speed sensors and sonar)
//  - OLED: Adafruit_SSD1306 & Adafruit_GFX (display vehicle status)
//  - Servo: Built-In library by Michael Margolis (required for RC truck)
// Contributors:
//  - October 2020: OLED display support by Ingmar Stapel
//  - December 2021: RC truck support by Usman Fiaz
//  - March 2022: OpenBot-Lite support by William Tan
//  - May 2022: MTV support by Quentin Leboutet
//  - Jan 2023: BLE support by iTinker
// ------------------------------------------------------------------------

// By Matthias Mueller, 2023
// ------------------------------------------------------------------------

//---------------------------------------------------//
// DEFINITIONS - DO NOT CHANGE!
//---------------------------------------------------//

//MCUs
#define NANO 328 //Atmega328p
#define ESP32 32 //ESP32

//Robot bodies with Atmega328p as MCU --> Select Arduino Nano as board
#define DIY 0       // DIY without PCB
#define PCB_V1 1    // DIY with PCB V1
#define PCB_V2 2    // DIY with PCB V2
#define RTR_TT 3    // Ready-to-Run with TT-motors
#define RC_CAR 4    // RC truck prototypes
#define LITE 5      // Smaller DIY version for education
//Robot bodies with ESP32 as MCU --> Select ESP32 Dev Module as board
#define RTR_TT2 6   // Ready-to-Run with TT-motors
#define RTR_520 7   // Ready-to-Run with 520-motors
#define MTV 8       // Multi Terrain Vehicle
#define DIY_ESP32 9  // DIY without PCB
```

```
//------------------------------------------------------//
// SETUP - Choose your body
//------------------------------------------------------//

// Setup the OpenBot version (DIY, PCB_V1, PCB_V2, RTR_TT, RC_CAR, LITE, RTR_TT2, RTR_520,
DIY_ESP32)
#define OPENBOT DIY

//------------------------------------------------------//
// SETTINGS - Global settings
//------------------------------------------------------//

// Enable/Disable no phone mode (1,0)
// In no phone mode:
// - the motors will turn at 75% speed
// - the speed will be reduced if an obstacle is detected by the sonar sensor
// - the car will turn, if an obstacle is detected within TURN_DISTANCE
// WARNING: If the sonar sensor is not setup, the car will go full speed forward!
#define NO_PHONE_MODE 0

// Enable/Disable debug print (1,0)
#define DEBUG 0

// Enable/Disable coast mode (1,0)
// When no control is applied, the robot will either coast (1) or actively stop (0)
boolean coast_mode = 1;

//------------------------------------------------------//
// CONFIG - update if you have built the DIY version
//------------------------------------------------------//
// HAS_VOLTAGE_DIVIDER              Enable/Disable voltage divider (1,0)
// VOLTAGE_DIVIDER_FACTOR           The voltage divider factor is computed as (R1+R2)/R2
// HAS_INDICATORS                   Enable/Disable indicators (1,0)
// HAS_SONAR                        Enable/Disable sonar (1,0)
// SONAR_MEDIAN                     Enable/Disable median filter for sonar measurements (1,0)
// HAS_BUMPER                       Enable/Disable bumper (1,0)
// HAS_SPEED_SENSORS_FRONT          Enable/Disable front speed sensors (1,0)
// HAS_SPEED_SENSORS_BACK           Enable/Disable back speed sensors (1,0)
// HAS_SPEED_SENSORS_MIDDLE         Enable/Disable middle speed sensors (1,0)
// HAS_OLED                         Enable/Disable OLED display (1,0)
// HAS_LEDS_FRONT                   Enable/Disable front LEDs
// HAS_LEDS_BACK                    Enable/Disable back LEDs
// HAS_LEDS_STATUS                  Enable/Disable status LEDs
// HAS_BLUETOOTH                    Enable/Disable bluetooth connectivity (1,0)
// NOTE: HAS_BLUETOOTH will only work with the ESP32 board (RTR_TT2, RTR_520, MTV,
DIY_ESP32)

// PIN_TRIGGER                      Arduino pin tied to trigger pin on ultrasonic sensor.
// PIN_ECHO                         Arduino pin tied to echo pin on ultrasonic sensor.
// MAX_SONAR_DISTANCE               Maximum distance we want to ping for (in centimeters).
// PIN_PWM_T                        Low-level control of throttle via PWM (for OpenBot RC only)
```

```
// PIN_PWM_S                    Low-level control of steering via PWM (for OpenBot RC only)
// PIN_PWM_L1,PIN_PWM_L2        Low-level control of left DC motors via PWM
// PIN_PWM_R1,PIN_PWM_R2        Low-level control of right DC motors via PWM
// PIN_VIN                      Measure battery voltage via voltage divider
// PIN_SPEED_LB, PIN_SPEED_RB   Measure left and right back wheel speed
// PIN_SPEED_LF, PIN_SPEED_RF   Measure left and right front wheel speed
// PIN_LED_LB, PIN_LED_RB       Control left and right back LEDs (indicator signals, illumination)
// PIN_LED_LF, PIN_LED_RF       Control left and right front LEDs (illumination)
// PIN_LED_Y, PIN_LED_G, PIN_LED_B  Control yellow, green and blue status LEDs

//-------------------------DIY-------------------------//
#if (OPENBOT == DIY)
const String robot_type = "DIY";
#define MCU NANO
#define HAS_VOLTAGE_DIVIDER 0
const float VOLTAGE_DIVIDER_FACTOR = (20 + 10) / 10;
const float VOLTAGE_MIN = 2.5f;
const float VOLTAGE_LOW = 9.0f;
const float VOLTAGE_MAX = 12.6f;
const float ADC_FACTOR = 5.0 / 1023;
#define HAS_INDICATORS 0
#define HAS_SONAR 0
#define SONAR_MEDIAN 0
#define HAS_SPEED_SENSORS_FRONT 0
#define HAS_OLED 0
const int PIN_PWM_L1 = 5;
const int PIN_PWM_L2 = 6;
const int PIN_PWM_R1 = 9;
const int PIN_PWM_R2 = 10;
const int PIN_SPEED_LF = 2;
const int PIN_SPEED_RF = 3;
const int PIN_VIN = A0;
const int PIN_TRIGGER = 12;
const int PIN_ECHO = 11;
const int PIN_LED_LI = 4;
const int PIN_LED_RI = 7;

//-------------------------PCB_V1-----------------------//
#elif (OPENBOT == PCB_V1)
const String robot_type = "PCB_V1";
#define MCU NANO
#define HAS_VOLTAGE_DIVIDER 1
const float VOLTAGE_DIVIDER_FACTOR = (100 + 33) / 33;
const float VOLTAGE_MIN = 2.5f;
const float VOLTAGE_LOW = 9.0f;
const float VOLTAGE_MAX = 12.6f;
const float ADC_FACTOR = 5.0 / 1023;
#define HAS_INDICATORS 1
#define HAS_SONAR 1
#define SONAR_MEDIAN 0
#define HAS_SPEED_SENSORS_FRONT 1
```

```
#define HAS_OLED 0
const int PIN_PWM_L1 = 9;
const int PIN_PWM_L2 = 10;
const int PIN_PWM_R1 = 5;
const int PIN_PWM_R2 = 6;
const int PIN_SPEED_LF = 2;
const int PIN_SPEED_RF = 4;
const int PIN_VIN = A7;
const int PIN_TRIGGER = 3;
const int PIN_ECHO = 3;
const int PIN_LED_LI = 7;
const int PIN_LED_RI = 8;

//-------------------------PCB_V2-----------------------//
#elif (OPENBOT == PCB_V2)
const String robot_type = "PCB_V2";
#define MCU NANO
#define HAS_VOLTAGE_DIVIDER 1
const float VOLTAGE_DIVIDER_FACTOR = (20 + 10) / 10;
const float VOLTAGE_MIN = 2.5f;
const float VOLTAGE_LOW = 9.0f;
const float VOLTAGE_MAX = 12.6f;
const float ADC_FACTOR = 5.0 / 1023;
#define HAS_INDICATORS 1
#define HAS_SONAR 1
#define SONAR_MEDIAN 0
#define HAS_SPEED_SENSORS_FRONT 1
#define HAS_OLED 0
const int PIN_PWM_L1 = 9;
const int PIN_PWM_L2 = 10;
const int PIN_PWM_R1 = 5;
const int PIN_PWM_R2 = 6;
const int PIN_SPEED_LF = 2;
const int PIN_SPEED_RF = 3;
const int PIN_VIN = A7;
const int PIN_TRIGGER = 4;
const int PIN_ECHO = 4;
const int PIN_LED_LI = 7;
const int PIN_LED_RI = 8;

//-------------------------RTR_TT-----------------------//
#elif (OPENBOT == RTR_TT)
const String robot_type = "RTR_TT";
#define MCU NANO
#define HAS_VOLTAGE_DIVIDER 1
const float VOLTAGE_DIVIDER_FACTOR = (30 + 10) / 10;
const float VOLTAGE_MIN = 2.5f;
const float VOLTAGE_LOW = 9.0f;
const float VOLTAGE_MAX = 12.6f;
const float ADC_FACTOR = 5.0 / 1023;
#define HAS_INDICATORS 1
```

```
#define HAS_SONAR 1
#define SONAR_MEDIAN 0
#define HAS_BUMPER 1
#define HAS_SPEED_SENSORS_FRONT 1
#define HAS_SPEED_SENSORS_BACK 1
#define HAS_LEDS_FRONT 1
#define HAS_LEDS_BACK 1
#define HAS_LEDS_STATUS 1
const int PIN_PWM_L1 = 10;
const int PIN_PWM_L2 = 9;
const int PIN_PWM_R1 = 6;
const int PIN_PWM_R2 = 5;
const int PIN_SPEED_LF = A3;
const int PIN_SPEED_RF = 7;
const int PIN_SPEED_LB = A4;
const int PIN_SPEED_RB = 8;
const int PIN_VIN = A6;
const int PIN_TRIGGER = 4;
const int PIN_ECHO = 2;
const int PIN_LED_LI = A5;
const int PIN_LED_RI = 12;
const int PIN_LED_LB = A5;
const int PIN_LED_RB = 12;
const int PIN_LED_LF = 3;
const int PIN_LED_RF = 11;
const int PIN_LED_Y = 13;
const int PIN_LED_G = A0;
const int PIN_LED_B = A1;
const int PIN_BUMPER = A2;
const int BUMPER_NOISE = 512;
const int BUMPER_EPS = 10;
const int BUMPER_AF = 951;
const int BUMPER_BF = 903;
const int BUMPER_CF = 867;
const int BUMPER_LF = 825;
const int BUMPER_RF = 786;
const int BUMPER_BB = 745;
const int BUMPER_LB = 607;
const int BUMPER_RB = 561;

//-------------------------RC_CAR-----------------------//
#elif (OPENBOT == RC_CAR)
const String robot_type = "RC_CAR";
#define MCU NANO
#include <Servo.h>
Servo ESC;
Servo SERVO;
#define HAS_VOLTAGE_DIVIDER 0
const float VOLTAGE_DIVIDER_FACTOR = (20 + 10) / 10;
const float VOLTAGE_MIN = 0.0f;
const float VOLTAGE_LOW = 6.4f;
```

```
const float VOLTAGE_MAX = 8.4f;
const float ADC_FACTOR = 5.0 / 1023;
#define HAS_INDICATORS 0
#define HAS_SONAR 0
#define SONAR_MEDIAN 0
const int PIN_PWM_T = A0;
const int PIN_PWM_S = A1;
const int PIN_VIN = A7;
const int PIN_TRIGGER = 4;
const int PIN_ECHO = 4;
const int PIN_LED_LI = 7;
const int PIN_LED_RI = 8;

//-------------------------LITE-------------------------//
#elif (OPENBOT == LITE)
const String robot_type = "LITE";
#define MCU NANO
const float VOLTAGE_MIN = 2.5f;
const float VOLTAGE_LOW = 4.5f;
const float VOLTAGE_MAX = 5.0f;
#define HAS_INDICATORS 1
const int PIN_PWM_L1 = 5;
const int PIN_PWM_L2 = 6;
const int PIN_PWM_R1 = 9;
const int PIN_PWM_R2 = 10;
const int PIN_LED_LI = 4;
const int PIN_LED_RI = 7;

//-------------------------RTR_TT2----------------------//
#elif (OPENBOT == RTR_TT2)
const String robot_type = "RTR_TT2";
#define MCU ESP32
#include <esp_wifi.h>
#define HAS_BLUETOOTH 1
#define analogWrite ledcWrite
#define attachPinChangeInterrupt attachInterrupt
#define detachPinChangeInterrupt detachInterrupt
#define digitalPinToPinChangeInterrupt digitalPinToInterrupt
#define PIN_PWM_L1 CH_PWM_L1
#define PIN_PWM_L2 CH_PWM_L2
#define PIN_PWM_R1 CH_PWM_R1
#define PIN_PWM_R2 CH_PWM_R2
#define HAS_VOLTAGE_DIVIDER 1
const float VOLTAGE_DIVIDER_FACTOR = (30 + 10) / 10;
const float VOLTAGE_MIN = 6.0f;
const float VOLTAGE_LOW = 9.0f;
const float VOLTAGE_MAX = 12.6f;
const float ADC_FACTOR = 3.3 / 4095;
#define HAS_INDICATORS 1
#define HAS_SONAR 1
#define SONAR_MEDIAN 0
```

```
#define HAS_BUMPER 1
#define HAS_SPEED_SENSORS_FRONT 1
#define HAS_SPEED_SENSORS_BACK 1
#define HAS_LEDS_FRONT 1
#define HAS_LEDS_BACK 1
#define HAS_LEDS_STATUS 1
//PWM properties
const int FREQ = 5000;
const int RES = 8;
const int CH_PWM_L1 = 0;
const int CH_PWM_L2 = 1;
const int CH_PWM_R1 = 2;
const int CH_PWM_R2 = 3;
const int CH_LED_LF = 4;
const int CH_LED_RF = 5;
const int CH_LED_LB = 6;
const int CH_LED_RB = 7;
const int PIN_PWM_LF1 = 16;
const int PIN_PWM_LF2 = 17;
const int PIN_PWM_LB1 = 19;
const int PIN_PWM_LB2 = 18;
const int PIN_PWM_RF1 = 25;
const int PIN_PWM_RF2 = 26;
const int PIN_PWM_RB1 = 32;
const int PIN_PWM_RB2 = 33;
const int PIN_SPEED_LF = 21;
const int PIN_SPEED_RF = 35;
const int PIN_SPEED_LB = 23;
const int PIN_SPEED_RB = 36;
const int PIN_VIN = 39;
const int PIN_TRIGGER = 12;
const int PIN_ECHO = 14;
const int PIN_LED_LI = 22;
const int PIN_LED_RI = 27;
const int PIN_LED_LB = 22;
const int PIN_LED_RB = 27;
const int PIN_LED_LF = 4;
const int PIN_LED_RF = 13;
const int PIN_LED_Y = 0;
const int PIN_LED_G = 2;
const int PIN_LED_B = 15;
const int PIN_BUMPER = 34;
const int BUMPER_NOISE = 512;
const int BUMPER_EPS = 50;
const int BUMPER_AF = 3890;
const int BUMPER_BF = 3550;
const int BUMPER_CF = 3330;
const int BUMPER_LF = 3100;
const int BUMPER_RF = 2930;
const int BUMPER_BB = 2750;
const int BUMPER_LB = 2180;
```

```
const int BUMPER_RB = 2000;

//-------------------------RTR_520----------------------//
#elif (OPENBOT == RTR_520)
const String robot_type = "RTR_520";
#define MCU ESP32
#include <esp_wifi.h>
#define HAS_BLUETOOTH 1
#define analogWrite ledcWrite
#define attachPinChangeInterrupt attachInterrupt
#define detachPinChangeInterrupt detachInterrupt
#define digitalPinToPinChangeInterrupt digitalPinToInterrupt
#define PIN_PWM_L1 CH_PWM_L1
#define PIN_PWM_L2 CH_PWM_L2
#define PIN_PWM_R1 CH_PWM_R1
#define PIN_PWM_R2 CH_PWM_R2
#define HAS_VOLTAGE_DIVIDER 1
const float VOLTAGE_DIVIDER_FACTOR = (30 + 10) / 10;
const float VOLTAGE_MIN = 6.0f;
const float VOLTAGE_LOW = 9.0f;
const float VOLTAGE_MAX = 12.6f;
const float ADC_FACTOR = 3.3 / 4095;
#define HAS_INDICATORS 1
#define HAS_SONAR 1
#define SONAR_MEDIAN 0
#define HAS_BUMPER 1
#define HAS_SPEED_SENSORS_FRONT 1
#define HAS_SPEED_SENSORS_BACK 1
#define HAS_LEDS_FRONT 1
#define HAS_LEDS_BACK 1
#define HAS_LEDS_STATUS 1
//PWM properties
const int FREQ = 5000;
const int RES = 8;
const int CH_PWM_L1 = 0;
const int CH_PWM_L2 = 1;
const int CH_PWM_R1 = 2;
const int CH_PWM_R2 = 3;
const int CH_LED_LF = 4;
const int CH_LED_RF = 5;
const int CH_LED_LB = 6;
const int CH_LED_RB = 7;
const int PIN_PWM_LF1 = 16;
const int PIN_PWM_LF2 = 17;
const int PIN_PWM_LB1 = 19;
const int PIN_PWM_LB2 = 18;
const int PIN_PWM_RF1 = 26;
const int PIN_PWM_RF2 = 25;
const int PIN_PWM_RB1 = 33;
const int PIN_PWM_RB2 = 32;
const int PIN_SPEED_LF = 21;
```

```
const int PIN_SPEED_RF = 35;
const int PIN_SPEED_LB = 23;
const int PIN_SPEED_RB = 36;
const int PIN_VIN = 39;
const int PIN_TRIGGER = 12;
const int PIN_ECHO = 14;
const int PIN_LED_LI = 22;
const int PIN_LED_RI = 27;
const int PIN_LED_LB = 22;
const int PIN_LED_RB = 27;
const int PIN_LED_LF = 4;
const int PIN_LED_RF = 13;
const int PIN_LED_Y = 0;
const int PIN_LED_G = 2;
const int PIN_LED_B = 15;
const int PIN_BUMPER = 34;
const int BUMPER_NOISE = 512;
const int BUMPER_EPS = 50;
const int BUMPER_AF = 3890;
const int BUMPER_BF = 3550;
const int BUMPER_CF = 3330;
const int BUMPER_LF = 3100;
const int BUMPER_RF = 2930;
const int BUMPER_BB = 2750;
const int BUMPER_LB = 2180;
const int BUMPER_RB = 2000;

//---------------------------MTV-------------------------//
#elif (OPENBOT == MTV)
const String robot_type = "MTV";
#define MCU ESP32
#include <esp_wifi.h>
#define HAS_BLUETOOTH 1
#define analogWrite ledcWrite
#define attachPinChangeInterrupt attachInterrupt
#define detachPinChangeInterrupt detachInterrupt
#define digitalPinToPinChangeInterrupt digitalPinToInterrupt
#define HAS_VOLTAGE_DIVIDER 0
const float VOLTAGE_MIN = 17.0f;
const float VOLTAGE_LOW = 20.0f;
const float VOLTAGE_MAX = 24.0f;
#define HAS_SPEED_SENSORS_FRONT 1
#define HAS_SPEED_SENSORS_BACK 1
#define HAS_SPEED_SENSORS_MIDDLE 1
#define HAS_INDICATORS 0
#define HAS_SONAR 0
#define SONAR_MEDIAN 0
#define HAS_BUMPER 0
#define HAS_LEDS_FRONT 0
#define HAS_LEDS_BACK 0
#define HAS_LEDS_STATUS 0
```

```
const int PIN_PWM_R = 19;
const int PIN_DIR_R = 18;
const int PIN_PWM_L = 33;
const int PIN_DIR_L = 32;

// Encoder setup:
const int PIN_SPEED_LF = 17;  // PIN_SPEED_LF_A = 17, PIN_SPEED_LF_B = 5
const int PIN_SPEED_RF = 14;  // PIN_SPEED_RF_A = 14, PIN_SPEED_RF_B = 13
const int PIN_SPEED_LM = 4;   // PIN_SPEED_LM_A = 4, PIN_SPEED_LM_B = 16
const int PIN_SPEED_RM = 26;  // PIN_SPEED_RM_A = 26, PIN_SPEED_RM_B = 27
const int PIN_SPEED_LB = 15;  // PIN_SPEED_LB_A = 15, PIN_SPEED_LB_B = 2
const int PIN_SPEED_RB = 35;  // PIN_SPEED_RB_A = 35, PIN_SPEED_RB_B = 25

// PWM properties:
const int FREQ = 5000;
const int RES = 8;
const int LHS_PWM_OUT = 0;
const int RHS_PWM_OUT = 1;

//------------------------DIY_ESP32----------------------//
#elif (OPENBOT == DIY_ESP32)
const String robot_type = "DIY_ESP32";
#define MCU ESP32
#include <esp_wifi.h>
#define HAS_BLUETOOTH 1
#define analogWrite ledcWrite
#define attachPinChangeInterrupt attachInterrupt
#define detachPinChangeInterrupt detachInterrupt
#define digitalPinToPinChangeInterrupt digitalPinToInterrupt
#define PIN_PWM_L1 CH_PWM_L1
#define PIN_PWM_L2 CH_PWM_L2
#define PIN_PWM_R1 CH_PWM_R1
#define PIN_PWM_R2 CH_PWM_R2
#define HAS_VOLTAGE_DIVIDER 1
const float VOLTAGE_DIVIDER_FACTOR = (30 + 10) / 10;
const float VOLTAGE_MIN = 6.0f;
const float VOLTAGE_LOW = 9.0f;
const float VOLTAGE_MAX = 12.6f;
const float ADC_FACTOR = 3.3 / 4095;
#define HAS_INDICATORS 1
#define HAS_SONAR 1
#define SONAR_MEDIAN 0
#define HAS_SPEED_SENSORS_FRONT 1
//PWM properties
const int FREQ = 5000;
const int RES = 8;
const int CH_PWM_L1 = 0;
const int CH_PWM_L2 = 1;
const int CH_PWM_R1 = 2;
const int CH_PWM_R2 = 3;
const int PIN_PWM_LF1 = 13;
```

```
const int PIN_PWM_LF2 = 12;
const int PIN_PWM_LB1 = 13;
const int PIN_PWM_LB2 = 12;
const int PIN_PWM_RF1 = 27;
const int PIN_PWM_RF2 = 33;
const int PIN_PWM_RB1 = 27;
const int PIN_PWM_RB2 = 33;
const int PIN_SPEED_LF = 5;
const int PIN_SPEED_RF = 18;
const int PIN_VIN = 39;
const int PIN_TRIGGER = 25;
const int PIN_ECHO = 26;
const int PIN_LED_LI = 22;
const int PIN_LED_RI = 16;
#endif
//-----------------------------------------------------//

#if (HAS_BLUETOOTH)
#include <BLEDevice.h>
#include <BLEServer.h>
#include <BLEUtils.h>
#include <BLE2902.h>

BLEServer *bleServer = NULL;
BLECharacteristic *pTxCharacteristic;
BLECharacteristic *pRxCharacteristic;
bool deviceConnected = false;
bool oldDeviceConnected = false;
const char *SERVICE_UUID = "61653dc3-4021-4d1e-ba83-8b4eec61d613";  // UART service UUID
const char *CHARACTERISTIC_UUID_RX = "06386c14-86ea-4d71-811c-48f97c58f8c9";
const char *CHARACTERISTIC_UUID_TX = "9bf1103b-834c-47cf-b149-c9e4bcf778a7";
#endif

enum msgParts {
  HEADER,
  BODY
};

msgParts msgPart = HEADER;
char header;
char endChar = '\n';
const char MAX_MSG_SZ = 60;
char msg_buf[MAX_MSG_SZ] = "";
int msg_idx = 0;

#if (HAS_BLUETOOTH)
void on_ble_rx(char inChar) {
  if (inChar != endChar) {
    switch (msgPart) {
      case HEADER:
        process_header(inChar);
```

```
      return;
    case BODY:
      process_body(inChar);
      return;
   }
  } else {
   msg_buf[msg_idx] = '\0';  // end of message
   parse_msg();
  }
}

//Initialization of classes for bluetooth
class MyServerCallbacks : public BLEServerCallbacks {
  void onConnect(BLEServer *bleServer, esp_ble_gatts_cb_param_t *param) {
   deviceConnected = true;

   // // Set the preferred connection parameters
   // uint16_t minInterval = 0; // Minimum connection interval in 1.25 ms units (50 ms)
   // uint16_t maxInterval = 800; // Maximum connection interval in 1.25 ms units (1000 ms)
   // uint16_t latency = 0;       // Slave latency
   // uint16_t timeout = 5000;     // Supervision timeout in 10 ms units (50 seconds)

   // bleServer->updateConnParams(param->connect.remote_bda, minInterval, maxInterval, latency,
timeout);

   Serial.println("BT Connected");
  };

  void onDisconnect(BLEServer *bleServer) {
   deviceConnected = false;
   Serial.println("BT Disconnected");
  }
};

class MyCallbacks : public BLECharacteristicCallbacks {
  void onWrite(BLECharacteristic *pCharacteristic) {
   std::string bleReceiver = pCharacteristic->getValue();
   if (bleReceiver.length() > 0) {
    for (int i = 0; i < bleReceiver.length(); i++) {
     on_ble_rx(bleReceiver[i]);
    }
   }
  }
};
#endif

//-------------------------------------------------------//
// INITIALIZATION
//-------------------------------------------------------//
#if (NO_PHONE_MODE)
unsigned long turn_direction_time = 0;
```

```
unsigned long turn_direction_interval = 5000;
unsigned int turn_direction = 0;
int ctrl_max = 192;
int ctrl_slow = 96;
int ctrl_min = (int)255.0 * VOLTAGE_MIN / VOLTAGE_MAX;
#endif

#if (HAS_SONAR)
#if (MCU == NANO)
#include "PinChangeInterrupt.h"
#endif
// Sonar sensor
const float US_TO_CM = 0.01715;             //cm/uS -> (343 * 100 / 1000000) / 2;
const unsigned int MAX_SONAR_DISTANCE = 300;  //cm
const unsigned long MAX_SONAR_TIME = (long)MAX_SONAR_DISTANCE * 2 * 10 / 343 + 1;
const unsigned int STOP_DISTANCE = 10;  //cm
#if (NO_PHONE_MODE)
const unsigned int TURN_DISTANCE = 50;
unsigned long sonar_interval = 100;
#else
unsigned long sonar_interval = 1000;
#endif
unsigned long sonar_time = 0;
boolean sonar_sent = false;
boolean ping_success = false;
unsigned int distance = -1;          //cm
unsigned int distance_estimate = -1;  //cm
unsigned long start_time;
unsigned long echo_time = 0;
#if (SONAR_MEDIAN)
const unsigned int distance_array_sz = 3;
unsigned int distance_array[distance_array_sz] = {};
unsigned int distance_counter = 0;
#endif
#else
const unsigned int TURN_DISTANCE = -1;  //cm
const unsigned int STOP_DISTANCE = 0;   //cm
unsigned int distance_estimate = -1;    //cm
#endif

#if (HAS_OLED)
#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

const int OLED_RESET = -1;  // not used
Adafruit_SSD1306 display(OLED_RESET);

// OLED Display SSD1306
const unsigned int SCREEN_WIDTH = 128;  // OLED display width, in pixels
```

```
const unsigned int SCREEN_HEIGHT = 32;  // OLED display height, in pixels
#endif

//Vehicle Control
int ctrl_left = 0;
int ctrl_right = 0;

#if (HAS_VOLTAGE_DIVIDER)
// Voltage measurement
unsigned int vin_counter = 0;
const unsigned int vin_array_sz = 10;
int vin_array[vin_array_sz] = { 0 };
unsigned long voltage_interval = 1000;  //Interval for sending voltage measurements
unsigned long voltage_time = 0;
#endif

#if (HAS_SPEED_SENSORS_FRONT or HAS_SPEED_SENSORS_BACK or HAS_SPEED_SENSORS_MIDDLE)
#if (OPENBOT == RTR_520)
// Speed sensor
// 530rpm motor - reduction ratio 19, ticks per motor rotation 11
// One revolution = 209 ticks
const unsigned int TICKS_PER_REV = 209;
#elif (OPENBOT == MTV)
// Speed sensor
// 178rpm motor - reduction ratio 56, ticks per motor rotation 11
// One revolution = 616 ticks
const unsigned int TICKS_PER_REV = 616;
#else
// Speed Sensor
// Optical encoder - disk with 20 holes
const unsigned int TICKS_PER_REV = 20;
#endif
// Speed sensor
const unsigned long SPEED_TRIGGER_THRESHOLD = 1;  // Triggers within this time will be ignored
(ms)

volatile int counter_lf = 0;
volatile int counter_rf = 0;
volatile int counter_lb = 0;
volatile int counter_rb = 0;
volatile int counter_lm = 0;
volatile int counter_rm = 0;
float rpm_left = 0;
float rpm_right = 0;
unsigned long wheel_interval = 1000;  // Interval for sending wheel odometry
unsigned long wheel_time = 0;
#endif

#if (HAS_INDICATORS)
// Indicator Signal
unsigned long indicator_interval = 500;  // Blinking rate of the indicator signal (ms).
```

```
unsigned long indicator_time = 0;
bool indicator_left = 0;
bool indicator_right = 0;
#endif

#if (HAS_LEDS_FRONT || HAS_LEDS_BACK)
unsigned int light_front = 0;
unsigned int light_back = 0;
#endif

// Bumper
#if HAS_BUMPER
bool bumper_event = 0;
bool collision_lf = 0;
bool collision_rf = 0;
bool collision_cf = 0;
bool collision_lb = 0;
bool collision_rb = 0;
unsigned long bumper_interval = 750;
unsigned long bumper_time = 0;
const int bumper_array_sz = 5;
int bumper_array[bumper_array_sz] = { 0 };
int bumper_reading = 0;
#endif

//Heartbeat
unsigned long heartbeat_interval = -1;
unsigned long heartbeat_time = 0;

#if (HAS_OLED || DEBUG)
// Display (via Serial)
unsigned long display_interval = 1000;  // How frequently vehicle data is displayed (ms).
unsigned long display_time = 0;
#endif

//-------------------------------------------------------//
// SETUP
//-------------------------------------------------------//
void setup() {
#if (OPENBOT == LITE)
  coast_mode = !coast_mode;
#endif
  // Outputs
#if (OPENBOT == RC_CAR)
  pinMode(PIN_PWM_T, OUTPUT);
  pinMode(PIN_PWM_S, OUTPUT);
  // Attach the ESC and SERVO
  ESC.attach(PIN_PWM_T, 1000, 2000);    // (pin, min pulse width, max pulse width in microseconds)
  SERVO.attach(PIN_PWM_S, 1000, 2000);  // (pin, min pulse width, max pulse width in
microseconds)
#endif
```

```
#if (MCU == NANO)
  pinMode(PIN_PWM_L1, OUTPUT);
  pinMode(PIN_PWM_L2, OUTPUT);
  pinMode(PIN_PWM_R1, OUTPUT);
  pinMode(PIN_PWM_R2, OUTPUT);
#endif
  // Initialize with the I2C addr 0x3C
#if (HAS_OLED)
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
#endif
#if (HAS_INDICATORS)
  pinMode(PIN_LED_LI, OUTPUT);
  pinMode(PIN_LED_RI, OUTPUT);
#endif
#if (HAS_LEDS_BACK)
  pinMode(PIN_LED_LB, OUTPUT);
  pinMode(PIN_LED_RB, OUTPUT);
#endif
#if (HAS_LEDS_FRONT)
  pinMode(PIN_LED_LF, OUTPUT);
  pinMode(PIN_LED_RF, OUTPUT);
#endif
#if (HAS_LEDS_STATUS)
  pinMode(PIN_LED_Y, OUTPUT);
  pinMode(PIN_LED_G, OUTPUT);
  pinMode(PIN_LED_B, OUTPUT);
#endif
  // Test sequence for indicator LEDs
#if HAS_INDICATORS
  digitalWrite(PIN_LED_LI, LOW);
  digitalWrite(PIN_LED_RI, LOW);
  delay(500);
  digitalWrite(PIN_LED_LI, HIGH);
  delay(500);
  digitalWrite(PIN_LED_LI, LOW);
  digitalWrite(PIN_LED_RI, HIGH);
  delay(500);
  digitalWrite(PIN_LED_RI, LOW);
#endif
#if (HAS_SONAR)
  pinMode(PIN_ECHO, INPUT);
  pinMode(PIN_TRIGGER, OUTPUT);
#endif
#if (HAS_VOLTAGE_DIVIDER)
  pinMode(PIN_VIN, INPUT);
#endif
#if (HAS_BUMPER)
  pinMode(PIN_BUMPER, INPUT);
#endif

#if (HAS_SPEED_SENSORS_BACK)
```

```
  pinMode(PIN_SPEED_LB, INPUT_PULLUP);
  pinMode(PIN_SPEED_RB, INPUT_PULLUP);
  attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(PIN_SPEED_LB), update_speed_lb,
RISING);
  attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(PIN_SPEED_RB), update_speed_rb,
RISING);
#endif
#if (HAS_SPEED_SENSORS_FRONT)
  pinMode(PIN_SPEED_LF, INPUT_PULLUP);
  pinMode(PIN_SPEED_RF, INPUT_PULLUP);
  attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(PIN_SPEED_LF), update_speed_lf,
RISING);
  attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(PIN_SPEED_RF), update_speed_rf,
RISING);
#endif
#if (HAS_SPEED_SENSORS_MIDDLE)
  pinMode(PIN_SPEED_LM, INPUT_PULLUP);
  pinMode(PIN_SPEED_RM, INPUT_PULLUP);
  attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(PIN_SPEED_LM), update_speed_lm,
RISING);
  attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(PIN_SPEED_RM), update_speed_rm,
RISING);
#endif

#if (MCU == ESP32)
  esp_wifi_deinit();
#endif

#if (MCU == ESP32 && OPENBOT != MTV)
  // PWMs
  // Configure PWM functionalitites
  ledcSetup(CH_PWM_L1, FREQ, RES);
  ledcSetup(CH_PWM_L2, FREQ, RES);
  ledcSetup(CH_PWM_R1, FREQ, RES);
  ledcSetup(CH_PWM_R2, FREQ, RES);

  // Attach the channel to the GPIO to be controlled
  ledcAttachPin(PIN_PWM_LF1, CH_PWM_L1);
  ledcAttachPin(PIN_PWM_LB1, CH_PWM_L1);
  ledcAttachPin(PIN_PWM_LF2, CH_PWM_L2);
  ledcAttachPin(PIN_PWM_LB2, CH_PWM_L2);
  ledcAttachPin(PIN_PWM_RF1, CH_PWM_R1);
  ledcAttachPin(PIN_PWM_RB1, CH_PWM_R1);
  ledcAttachPin(PIN_PWM_RF2, CH_PWM_R2);
  ledcAttachPin(PIN_PWM_RB2, CH_PWM_R2);

#if (HAS_LEDS_BACK)
  ledcSetup(CH_LED_LB, FREQ, RES);
  ledcSetup(CH_LED_RB, FREQ, RES);
  ledcAttachPin(PIN_LED_RB, CH_LED_RB);
  ledcAttachPin(PIN_LED_LB, CH_LED_LB);
```

```
#endif

#if (HAS_LEDS_FRONT)
  ledcSetup(CH_LED_LF, FREQ, RES);
  ledcSetup(CH_LED_RF, FREQ, RES);
  ledcAttachPin(PIN_LED_LF, CH_LED_LF);
  ledcAttachPin(PIN_LED_RF, CH_LED_RF);
#endif

#endif

#if (OPENBOT == MTV)
  // PWMs
  // PWM signal configuration using the ESP32 API
  ledcSetup(LHS_PWM_OUT, FREQ, RES);
  ledcSetup(RHS_PWM_OUT, FREQ, RES);

  // Attach the channel to the GPIO to be controlled
  ledcAttachPin(PIN_PWM_L, LHS_PWM_OUT);
  ledcAttachPin(PIN_PWM_R, RHS_PWM_OUT);

  pinMode(PIN_DIR_L, OUTPUT);
  pinMode(PIN_DIR_R, OUTPUT);
  pinMode(PIN_DIR_L, LOW);
  pinMode(PIN_DIR_R, LOW);
#endif

#if (OPENBOT == DIY_ESP32)
  // PWMs
  // Configure PWM functionalitites
  ledcSetup(CH_PWM_L1, FREQ, RES);
  ledcSetup(CH_PWM_L2, FREQ, RES);
  ledcSetup(CH_PWM_R1, FREQ, RES);
  ledcSetup(CH_PWM_R2, FREQ, RES);

  // Attach the channel to the GPIO to be controlled
  ledcAttachPin(PIN_PWM_L1, CH_PWM_L1);
  ledcAttachPin(PIN_PWM_L2, CH_PWM_L2);
  ledcAttachPin(PIN_PWM_R1, CH_PWM_R1);
  ledcAttachPin(PIN_PWM_R2, CH_PWM_R2);
#endif

  Serial.begin(115200, SERIAL_8N1);
  // SERIAL_8E1 - 8 data bits, even parity, 1 stop bit
  // SERIAL_8O1 - 8 data bits, odd parity, 1 stop bit
  // SERIAL_8N1 - 8 data bits, no parity, 1 stop bit
  // Serial.setTimeout(10);
  Serial.println('r');

#if (HAS_BLUETOOTH)
  String ble_name = "OpenBot: " + robot_type;
```

```
  BLEDevice::init(ble_name.c_str());
  bleServer = BLEDevice::createServer();
  bleServer->setCallbacks(new MyServerCallbacks());
  BLEService *pService = bleServer->createService(BLEUUID(SERVICE_UUID));

  pTxCharacteristic = pService->createCharacteristic(BLEUUID(CHARACTERISTIC_UUID_TX),
BLECharacteristic::PROPERTY_NOTIFY);
  pTxCharacteristic->addDescriptor(new BLE2902());

  pRxCharacteristic = pService->createCharacteristic(BLEUUID(CHARACTERISTIC_UUID_RX),
BLECharacteristic::PROPERTY_WRITE_NR);
  pRxCharacteristic->setCallbacks(new MyCallbacks());
  pRxCharacteristic->addDescriptor(new BLE2902());

  pService->start();

  // Start advertising
  BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
  pAdvertising->addServiceUUID(BLEUUID(SERVICE_UUID));
  bleServer->getAdvertising()->start();
  Serial.println("Waiting a client connection to notify...");
#endif
}

//----------------------------------------------------//
//LOOP
//----------------------------------------------------//
void loop() {

#if (HAS_BLUETOOTH)
  // disconnecting
  if (!deviceConnected && oldDeviceConnected) {
    delay(500);                 // give the bluetooth stack the chance to get things ready
    bleServer->startAdvertising();  // restart advertising
    Serial.println("Waiting a client connection to notify...");
    oldDeviceConnected = deviceConnected;
  }
  // connecting
  if (deviceConnected && !oldDeviceConnected) {
    oldDeviceConnected = deviceConnected;
  }
#endif

#if (NO_PHONE_MODE)
  if ((millis() - turn_direction_time) >= turn_direction_interval) {
    turn_direction_time = millis();
    turn_direction = random(2);  //Generate random number in the range [0,1]
  }
  // Drive forward
  if (distance_estimate > 3 * TURN_DISTANCE) {
    ctrl_left = distance_estimate;
```

```
  ctrl_right = ctrl_left;
  digitalWrite(PIN_LED_LI, LOW);
  digitalWrite(PIN_LED_RI, LOW);
 }
 // Turn slightly
 else if (distance_estimate > 2 * TURN_DISTANCE) {
  ctrl_left = distance_estimate;
  ctrl_right = ctrl_left / 2;
 }
 // Turn strongly
 else if (distance_estimate > TURN_DISTANCE) {
  ctrl_left = ctrl_max;
  ctrl_right = -ctrl_max;
 }
 // Drive backward slowly
 else {
  ctrl_left = -ctrl_slow;
  ctrl_right = -ctrl_slow;
  digitalWrite(PIN_LED_LI, HIGH);
  digitalWrite(PIN_LED_RI, HIGH);
 }
 // Flip controls if needed and set indicator light
 if (ctrl_left != ctrl_right) {
  if (turn_direction > 0) {
   int temp = ctrl_left;
   ctrl_left = ctrl_right;
   ctrl_right = temp;
   digitalWrite(PIN_LED_LI, HIGH);
   digitalWrite(PIN_LED_RI, LOW);
  } else {
   digitalWrite(PIN_LED_LI, LOW);
   digitalWrite(PIN_LED_RI, HIGH);
  }
 }

 // Enforce limits
 ctrl_left = ctrl_left > 0 ? max(ctrl_min, min(ctrl_left, ctrl_max)) : min(-ctrl_min, max(ctrl_left, -
ctrl_max));
 ctrl_right = ctrl_right > 0 ? max(ctrl_min, min(ctrl_right, ctrl_max)) : min(-ctrl_min, max(ctrl_right, -
ctrl_max));
#else  // Check for messages from the phone
 if (Serial.available() > 0) {
  on_serial_rx();
 }
 if (distance_estimate <= STOP_DISTANCE && ctrl_left > 0 && ctrl_right > 0) {
  ctrl_left = 0;
  ctrl_right = 0;
 }
 if ((millis() - heartbeat_time) >= heartbeat_interval) {
  ctrl_left = 0;
  ctrl_right = 0;
```

```
  }
#endif

#if HAS_BUMPER
  if (analogRead(PIN_BUMPER) > BUMPER_NOISE && !bumper_event) {
    delayMicroseconds(500);
    for (unsigned int i = 0; i < bumper_array_sz; i++) {
      bumper_array[i] = analogRead(PIN_BUMPER);
    }
    bumper_reading = get_median(bumper_array, bumper_array_sz);
    if (bumper_reading > BUMPER_NOISE)
      emergency_stop();
  }

  bool collison_front = collision_lf || collision_rf || collision_cf;
  bool collision_back = collision_lb || collision_rb;
  bool control_front = ctrl_left > 0 && ctrl_right > 0;
  bool control_back = ctrl_left < 0 && ctrl_right < 0;

  if (!bumper_event || (control_back && collison_front) || (control_front && collision_back)) {
    update_vehicle();
  }
#else
  update_vehicle();
#endif

#if HAS_VOLTAGE_DIVIDER
  // Measure voltage
  vin_array[vin_counter % vin_array_sz] = analogRead(PIN_VIN);
  vin_counter++;
#endif

#if HAS_SONAR
  // Check for successful sonar reading
  if (!sonar_sent && ping_success) {
    distance = echo_time * US_TO_CM;
    update_distance_estimate();
    send_sonar_reading();
    sonar_sent = true;
  }
  // Measure distance every sonar_interval
  if ((millis() - sonar_time) >= max(sonar_interval, MAX_SONAR_TIME)) {
    if (!sonar_sent && !ping_success) {  // Send max val if last ping was not returned
      distance = MAX_SONAR_DISTANCE;
      update_distance_estimate();
      send_sonar_reading();
      sonar_sent = true;
    }
    sonar_time = millis();
    sonar_sent = false;
    send_ping();
```

```
  }
#endif

#if HAS_INDICATORS
  // Check indicator signal every indicator_interval
  if ((millis() - indicator_time) >= indicator_interval) {
    update_indicator();
    indicator_time = millis();
  }
#endif

#if HAS_BUMPER
  // Check bumper signal every bumper_interval
  if ((millis() - bumper_time) >= bumper_interval && bumper_event) {
    reset_bumper();
    bumper_time = millis();
  }
#endif
#if HAS_VOLTAGE_DIVIDER
  // Send voltage reading via serial
  if ((millis() - voltage_time) >= voltage_interval) {
    send_voltage_reading();
    voltage_time = millis();
  }
#endif
#if (HAS_SPEED_SENSORS_FRONT or HAS_SPEED_SENSORS_BACK or HAS_SPEED_SENSORS_MIDDLE)
  // Send wheel odometry reading via serial
  if ((millis() - wheel_time) >= wheel_interval) {
    send_wheel_reading(millis() - wheel_time);
    wheel_time = millis();
  }
#endif
#if (HAS_OLED || DEBUG)
  // Display vehicle measurments for via serial every display_interval
  if ((millis() - display_time) >= display_interval) {
    display_vehicle_data();
    display_time = millis();
  }
#endif
}

//-------------------------------------------------------//
// FUNCTIONS
//-------------------------------------------------------//
#if HAS_VOLTAGE_DIVIDER

float get_voltage() {
  unsigned long array_sum = 0;
  unsigned int array_size = min(vin_array_sz, vin_counter);
  for (unsigned int index = 0; index < array_size; index++) {
    array_sum += vin_array[index];
```

```
  }
  return float(array_sum) / array_size * ADC_FACTOR * VOLTAGE_DIVIDER_FACTOR;
}

#endif

void update_vehicle() {
#if (OPENBOT == RC_CAR)
  update_throttle();
  update_steering();
#elif (OPENBOT == MTV)
  update_left_motors_mtv();
  update_right_motors_mtv();
#else
  update_left_motors();
  update_right_motors();
#endif
}

#if (OPENBOT == RC_CAR)
void update_throttle() {
  if (ctrl_left == 0 || ctrl_right == 0) {
    ESC.write(90);  //set throttle to zero
  } else {
    int throttle = map(ctrl_left + ctrl_right, -510, 510, 0, 180);
    ESC.write(throttle);
  }
}

void update_steering() {
  int steering = map(ctrl_left - ctrl_right, -510, 510, 0, 180);
  if (ctrl_left + ctrl_right < 0) {
    SERVO.write(steering);
  } else {
    SERVO.write(180 - steering);
  }
}

#elif (OPENBOT == MTV)
void update_left_motors_mtv() {
  if (ctrl_left < 0) {
    ledcWrite(LHS_PWM_OUT, -ctrl_left);
    digitalWrite(PIN_DIR_L, HIGH);
  } else if (ctrl_left > 0) {
    ledcWrite(LHS_PWM_OUT, ctrl_left);
    digitalWrite(PIN_DIR_L, LOW);
  } else {
    if (coast_mode) {
      coast_left_motors_mtv();
    } else {
      stop_left_motors_mtv();
```

```
    }
  }
}

void stop_left_motors_mtv() {
  ledcWrite(LHS_PWM_OUT, 0);
  digitalWrite(PIN_DIR_L, LOW);
}

void coast_left_motors_mtv() {
  ledcWrite(LHS_PWM_OUT, 0);
  digitalWrite(PIN_DIR_L, LOW);
}

void update_right_motors_mtv() {
  if (ctrl_right < 0) {
    ledcWrite(RHS_PWM_OUT, -ctrl_right);
    digitalWrite(PIN_DIR_R, HIGH);
  } else if (ctrl_right > 0) {
    ledcWrite(RHS_PWM_OUT, ctrl_right);
    digitalWrite(PIN_DIR_R, LOW);
  } else {
    if (coast_mode) {
      coast_right_motors_mtv();
    } else {
      stop_right_motors_mtv();
    }
  }
}

void stop_right_motors_mtv() {
  ledcWrite(RHS_PWM_OUT, 0);
  digitalWrite(PIN_DIR_R, LOW);
}

void coast_right_motors_mtv() {
  ledcWrite(RHS_PWM_OUT, 0);
  digitalWrite(PIN_DIR_R, LOW);
}

#else

void update_left_motors() {
  if (ctrl_left < 0) {
    analogWrite(PIN_PWM_L1, -ctrl_left);
    analogWrite(PIN_PWM_L2, 0);
  } else if (ctrl_left > 0) {
    analogWrite(PIN_PWM_L1, 0);
    analogWrite(PIN_PWM_L2, ctrl_left);
  } else {
    if (coast_mode) {
```

```
    coast_left_motors();
   } else {
    stop_left_motors();
   }
  }
}

void stop_left_motors() {
  analogWrite(PIN_PWM_L1, 255);
  analogWrite(PIN_PWM_L2, 255);
}

void coast_left_motors() {
  analogWrite(PIN_PWM_L1, 0);
  analogWrite(PIN_PWM_L2, 0);
}

void update_right_motors() {
  if (ctrl_right < 0) {
   analogWrite(PIN_PWM_R1, -ctrl_right);
   analogWrite(PIN_PWM_R2, 0);
  } else if (ctrl_right > 0) {
   analogWrite(PIN_PWM_R1, 0);
   analogWrite(PIN_PWM_R2, ctrl_right);
  } else {
   if (coast_mode) {
    coast_right_motors();
   } else {
    stop_right_motors();
   }
  }
}

void stop_right_motors() {
  analogWrite(PIN_PWM_R1, 255);
  analogWrite(PIN_PWM_R2, 255);
}

void coast_right_motors() {
  analogWrite(PIN_PWM_R1, 0);
  analogWrite(PIN_PWM_R2, 0);
}

#endif

boolean almost_equal(int a, int b, int eps) {
  return abs(a - b) <= eps;
}

#if HAS_BUMPER
void emergency_stop() {
```

```
  bumper_event = true;
  stop_left_motors();
  stop_right_motors();
  ctrl_left = 0;
  ctrl_right = 0;
#if HAS_INDICATORS
  indicator_left = 1;
  indicator_right = 1;
  indicator_time = millis() - indicator_interval;  // update indicators
#endif
  bumper_time = millis();
  char bumper_id[2];
  if (almost_equal(bumper_reading, BUMPER_AF, BUMPER_EPS)) {
    collision_cf = 1;
    collision_lf = 1;
    collision_rf = 1;
    strncpy(bumper_id, "af", sizeof(bumper_id));
#if DEBUG
    Serial.print("All Front: ");
#endif
  } else if (almost_equal(bumper_reading, BUMPER_BF, BUMPER_EPS)) {
    collision_lf = 1;
    collision_rf = 1;
    strncpy(bumper_id, "bf", sizeof(bumper_id));
#if DEBUG
    Serial.print("Both Front: ");
#endif
  } else if (almost_equal(bumper_reading, BUMPER_CF, BUMPER_EPS)) {
    collision_cf = 1;
    strncpy(bumper_id, "cf", sizeof(bumper_id));
#if DEBUG
    Serial.print("Camera Front: ");
#endif
  } else if (almost_equal(bumper_reading, BUMPER_LF, BUMPER_EPS)) {
    collision_lf = 1;
    strncpy(bumper_id, "lf", sizeof(bumper_id));
#if DEBUG
    Serial.print("Left Front: ");
#endif
  } else if (almost_equal(bumper_reading, BUMPER_RF, BUMPER_EPS)) {
    collision_rf = 1;
    strncpy(bumper_id, "rf", sizeof(bumper_id));
#if DEBUG
    Serial.print("Right Front: ");
#endif
  } else if (almost_equal(bumper_reading, BUMPER_BB, BUMPER_EPS)) {
    collision_lb = 1;
    collision_rb = 1;
    strncpy(bumper_id, "bb", sizeof(bumper_id));
#if DEBUG
    Serial.print("Both Back: ");
```

```
#endif
  } else if (almost_equal(bumper_reading, BUMPER_LB, BUMPER_EPS)) {
    collision_lb = 1;
    strncpy(bumper_id, "lb", sizeof(bumper_id));
#if DEBUG
    Serial.print("Left Back: ");
#endif
  } else if (almost_equal(bumper_reading, BUMPER_RB, BUMPER_EPS)) {
    collision_rb = 1;
    strncpy(bumper_id, "rb", sizeof(bumper_id));
#if DEBUG
    Serial.print("Right Back: ");
#endif
  } else {
    strncpy(bumper_id, "??", sizeof(bumper_id));
#if DEBUG
    Serial.print("Unknown: ");
#endif
  }
#if DEBUG
  Serial.println(bumper_reading);
#endif
  send_bumper_reading(bumper_id);
}

void reset_bumper() {
#if HAS_INDICATORS
  indicator_left = 0;
  indicator_right = 0;
#endif
  collision_lf = 0;
  collision_rf = 0;
  collision_cf = 0;
  collision_lb = 0;
  collision_rb = 0;
  bumper_reading = 0;
  bumper_event = false;
}

void send_bumper_reading(char bumper_id[]) {
  sendData("b" + String(bumper_id));
}
#endif

void process_ctrl_msg() {
  char *tmp;                  // this is used by strtok() as an index
  tmp = strtok(msg_buf, ",:");  // replace delimiter with \0
  ctrl_left = atoi(tmp);       // convert to int
  tmp = strtok(NULL, ",:");     // continues where the previous call left off
  ctrl_right = atoi(tmp);      // convert to int
#if DEBUG
```

```
  Serial.print("Control: ");
  Serial.print(ctrl_left);
  Serial.print(",");
  Serial.println(ctrl_right);
#endif
}

#if (HAS_LEDS_FRONT || HAS_LEDS_BACK)
void process_light_msg() {
  char *tmp;                // this is used by strtok() as an index
  tmp = strtok(msg_buf, ",:");  // replace delimiter with \0
  light_front = atoi(tmp);      // convert to int
  tmp = strtok(NULL, ",:");     // continues where the previous call left off
  light_back = atoi(tmp);       // convert to int
#if DEBUG
  Serial.print("Light: ");
  Serial.print(light_front);
  Serial.print(",");
  Serial.println(light_back);
#endif
  update_light();
}
#endif

void process_heartbeat_msg() {
  heartbeat_interval = atol(msg_buf);  // convert to long
  heartbeat_time = millis();
#if DEBUG
  Serial.print("Heartbeat Interval: ");
  Serial.println(heartbeat_interval);
#endif
}

#if HAS_INDICATORS

void process_indicator_msg() {
  char *tmp;                // this is used by strtok() as an index
  tmp = strtok(msg_buf, ",:");  // replace delimiter with \0
  indicator_left = atoi(tmp);   // convert to int
  tmp = strtok(NULL, ",:");     // continues where the previous call left off
  indicator_right = atoi(tmp);  // convert to int
#if DEBUG
  Serial.print("Indicator: ");
  Serial.print(indicator_left);
  Serial.print(",");
  Serial.println(indicator_right);
#endif
}

#endif
```

```
#if HAS_LEDS_STATUS
void process_notification_msg() {
  char *tmp;                  // this is used by strtok() as an index
  tmp = strtok(msg_buf, ",:");  // replace delimiter with \0
  char led = tmp[0];
  tmp = strtok(NULL, ",:");  // continues where the previous call left off
  int state = atoi(tmp);     // convert to int
  switch (led) {
    case 'y':
      digitalWrite(PIN_LED_Y, state);
      break;
    case 'g':
      digitalWrite(PIN_LED_G, state);
      break;
    case 'b':
      digitalWrite(PIN_LED_B, state);
      break;
  }
#if DEBUG
  Serial.print("Notification: ");
  Serial.print(led);
  Serial.println(state);
#endif
}
#endif

#if HAS_BUMPER
void process_bumper_msg() {
  bumper_interval = atol(msg_buf);  // convert to long
}
#endif
#if HAS_SONAR

void process_sonar_msg() {
  sonar_interval = atol(msg_buf);  // convert to long
}

#endif

void process_voltage_msg() {
#if HAS_VOLTAGE_DIVIDER
  voltage_interval = atol(msg_buf);  // convert to long
#endif
  Serial.println(String("vmin:") + String(VOLTAGE_MIN, 2));
  Serial.println(String("vlow:") + String(VOLTAGE_LOW, 2));
  Serial.println(String("vmax:") + String(VOLTAGE_MAX, 2));
}

#if (HAS_SPEED_SENSORS_FRONT or HAS_SPEED_SENSORS_BACK or HAS_SPEED_SENSORS_MIDDLE)

void process_wheel_msg() {
```

```
  wheel_interval = atol(msg_buf);  // convert to long
}

#endif

void process_feature_msg() {
  String msg = "f" + robot_type + ":";
#if HAS_VOLTAGE_DIVIDER
  msg += "v:";
#endif
#if HAS_INDICATORS
  msg += "i:";
#endif
#if HAS_SONAR
  msg += "s:";
#endif
#if HAS_BUMPER
  msg += "b:";
#endif
#if HAS_SPEED_SENSORS_FRONT
  msg += "wf:";
#endif
#if HAS_SPEED_SENSORS_BACK
  msg += "wb:";
#endif
#if HAS_SPEED_SENSORS_MIDDLE
  msg += "wm:";
#endif
#if HAS_LEDS_FRONT
  msg += "lf:";
#endif
#if HAS_LEDS_BACK
  msg += "lb:";
#endif
#if HAS_LEDS_STATUS
  msg += "ls:";
#endif
  sendData(msg);
}

void on_serial_rx() {
  char inChar = Serial.read();
  if (inChar != endChar) {
    switch (msgPart) {
      case HEADER:
        process_header(inChar);
        return;
      case BODY:
        process_body(inChar);
        return;
    }
```

```
  } else {
    msg_buf[msg_idx] = '\0';  // end of message
    parse_msg();
  }
}

void process_header(char inChar) {
  header = inChar;
  msgPart = BODY;
}

void process_body(char inChar) {
  // Add the incoming byte to the buffer
  msg_buf[msg_idx] = inChar;
  msg_idx++;
}

void parse_msg() {
  switch (header) {
#if HAS_BUMPER
    case 'b':
      process_bumper_msg();
      break;
#endif
    case 'c':
      process_ctrl_msg();
      break;
    case 'f':
      process_feature_msg();
      break;
    case 'h':
      process_heartbeat_msg();
      break;
#if HAS_INDICATORS
    case 'i':
      process_indicator_msg();
      break;
#endif
#if (HAS_LEDS_FRONT || HAS_LEDS_BACK)
    case 'l':
      process_light_msg();
      break;
#endif
#if HAS_LEDS_STATUS
    case 'n':
      process_notification_msg();
      break;
#endif
#if HAS_SONAR
    case 's':
      process_sonar_msg();
```

```
      break;
#endif
#if HAS_VOLTAGE_DIVIDER
    case 'v':
      process_voltage_msg();
      break;
#endif
#if (HAS_SPEED_SENSORS_FRONT or HAS_SPEED_SENSORS_BACK or HAS_SPEED_SENSORS_MIDDLE)
    case 'w':
      process_wheel_msg();
      break;
#endif
  }
  msg_idx = 0;
  msgPart = HEADER;
  header = '\0';
}

#if HAS_OLED
// Function for drawing a string on the OLED display
void drawString(String line1, String line2, String line3, String line4) {
  display.clearDisplay();
  // set text color
  display.setTextColor(WHITE);
  // set text size
  display.setTextSize(1);
  // set text cursor position
  display.setCursor(1, 0);
  // show text
  display.println(line1);
  display.setCursor(1, 8);
  // show text
  display.println(line2);
  display.setCursor(1, 16);
  // show text
  display.println(line3);
  display.setCursor(1, 24);
  // show text
  display.println(line4);
  display.display();
}
#endif

#if (HAS_OLED || DEBUG)

void display_vehicle_data() {
#if HAS_VOLTAGE_DIVIDER
  float voltage_value = get_voltage();
  String voltage_str = String("Voltage:    ") + String(voltage_value, 2);
#else
  String voltage_str = String("Voltage:    ") + String("N/A");
```

```
#endif
#if (HAS_SPEED_SENSORS_FRONT or HAS_SPEED_SENSORS_BACK or HAS_SPEED_SENSORS_MIDDLE)
  String left_rpm_str = String("Left RPM:  ") + String(rpm_left, 0);
  String right_rpm_str = String("Right RPM:  ") + String(rpm_left, 0);
#else
  String left_rpm_str = String("Left RPM:  ") + String("N/A");
  String right_rpm_str = String("Right RPM:  ") + String("N/A");
#endif
#if HAS_SONAR
  String distance_str = String("Distance:   ") + String(distance_estimate);
#else
  String distance_str = String("Distance:   ") + String("N/A");
#endif
#if DEBUG
  Serial.println("------------------");
  Serial.println(voltage_str);
  Serial.println(left_rpm_str);
  Serial.println(right_rpm_str);
  Serial.println(distance_str);
  Serial.println("------------------");
#endif
#if HAS_OLED
  // Set display information
  drawString(
    voltage_str,
    left_rpm_str,
    right_rpm_str,
    distance_str);
#endif
}

#endif

#if (HAS_VOLTAGE_DIVIDER)

void send_voltage_reading() {
  sendData("v" + String(get_voltage(), 2));
}

#endif

#if (HAS_SPEED_SENSORS_FRONT or HAS_SPEED_SENSORS_BACK or HAS_SPEED_SENSORS_MIDDLE)

void send_wheel_reading(long duration) {
  float rpm_factor = 60.0 * 1000.0 / duration / TICKS_PER_REV;
  rpm_left = (counter_lf + counter_lb + counter_lm) * rpm_factor;
  rpm_right = (counter_rf + counter_rb + counter_rm) * rpm_factor;
  counter_lf = 0;
  counter_rf = 0;
  counter_lb = 0;
  counter_rb = 0;
```

```
  counter_lm = 0;
  counter_rm = 0;
#if (HAS_SPEED_SENSORS_FRONT and HAS_SPEED_SENSORS_BACK and
HAS_SPEED_SENSORS_MIDDLE)
  sendData("w" + String(rpm_left / 3) + "," + String(rpm_right / 3));
#elif ((HAS_SPEED_SENSORS_FRONT and HAS_SPEED_SENSORS_BACK) or
(HAS_SPEED_SENSORS_FRONT and HAS_SPEED_SENSORS_MIDDLE) or
(HAS_SPEED_SENSORS_MIDDLE and HAS_SPEED_SENSORS_BACK))
  sendData("w" + String(rpm_left / 2) + "," + String(rpm_right / 2));
#elif (HAS_SPEED_SENSORS_FRONT or HAS_SPEED_SENSORS_BACK or
HAS_SPEED_SENSORS_MIDDLE)
  sendData("w" + String(rpm_left) + "," + String(rpm_right));
#endif
}

#endif

#if (HAS_INDICATORS)

void update_indicator() {
  if (indicator_left > 0) {
#if ((OPENBOT == RTR_TT2 || OPENBOT == RTR_520) && PIN_LED_LI == PIN_LED_LB)
    ledcDetachPin(PIN_LED_LB);
#endif
    digitalWrite(PIN_LED_LI, !digitalRead(PIN_LED_LI));
  } else {
#if (HAS_LEDS_BACK)
    digitalWrite(PIN_LED_LI, PIN_LED_LI == PIN_LED_LB ? light_back : LOW);
#else
    digitalWrite(PIN_LED_LI, LOW);
#endif
#if ((OPENBOT == RTR_TT2 || OPENBOT == RTR_520) && PIN_LED_LI == PIN_LED_LB)
    ledcAttachPin(PIN_LED_LB, CH_LED_LB);
#endif
  }
  if (indicator_right > 0) {
#if ((OPENBOT == RTR_TT2 || OPENBOT == RTR_520) && PIN_LED_RI == PIN_LED_RB)
    ledcDetachPin(PIN_LED_RB);
#endif
    digitalWrite(PIN_LED_RI, !digitalRead(PIN_LED_RI));
  } else {
#if (HAS_LEDS_BACK)
    digitalWrite(PIN_LED_RI, PIN_LED_RI == PIN_LED_RB ? light_back : LOW);
#else
    digitalWrite(PIN_LED_RI, LOW);
#endif
#if ((OPENBOT == RTR_TT2 || OPENBOT == RTR_520) && PIN_LED_RI == PIN_LED_RB)
    ledcAttachPin(PIN_LED_RB, CH_LED_RB);
#endif
  }
}
```

```
#endif

#if (HAS_LEDS_FRONT || HAS_LEDS_BACK)
void update_light() {
#if (HAS_LEDS_FRONT)
#if (OPENBOT == RTR_TT2 || OPENBOT == RTR_520)
  analogWrite(CH_LED_LF, light_front);
  analogWrite(CH_LED_RF, light_front);
#else
  analogWrite(PIN_LED_LF, light_front);
  analogWrite(PIN_LED_RF, light_front);
#endif
#endif

#if (HAS_LEDS_BACK)
#if (OPENBOT == RTR_TT2 || OPENBOT == RTR_520)
  analogWrite(CH_LED_LB, light_back);
  analogWrite(CH_LED_RB, light_back);
#else
  analogWrite(PIN_LED_LB, light_back);
  analogWrite(PIN_LED_RB, light_back);
#endif
#endif
}
#endif

int get_median(int a[], int sz) {
  //bubble sort
  for (int i = 0; i < (sz - 1); i++) {
    for (int j = 0; j < (sz - (i + 1)); j++) {
      if (a[j] > a[j + 1]) {
        int t = a[j];
        a[j] = a[j + 1];
        a[j + 1] = t;
      }
    }
  }
  return a[sz / 2];
}

#if HAS_SONAR

void send_sonar_reading() {
  sendData("s" + String(distance_estimate));
}

// Send pulse by toggling trigger pin
void send_ping() {
  echo_time = 0;
  ping_success = false;
```

```
  if (PIN_TRIGGER == PIN_ECHO)
    pinMode(PIN_TRIGGER, OUTPUT);
  digitalWrite(PIN_TRIGGER, LOW);
  delayMicroseconds(5);
  digitalWrite(PIN_TRIGGER, HIGH);
  delayMicroseconds(10);
  digitalWrite(PIN_TRIGGER, LOW);
  if (PIN_TRIGGER == PIN_ECHO)
    pinMode(PIN_ECHO, INPUT);
  attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(PIN_ECHO), start_timer, RISING);
}

void update_distance_estimate() {
#if SONAR_MEDIAN
  distance_array[distance_counter % distance_array_sz] = distance;
  distance_counter++;
  distance_estimate = get_median(distance_array, distance_array_sz);
#else
  distance_estimate = distance;
#endif
}

#endif

void sendData(String data) {
Serial.print(data);
Serial.println();
#if (HAS_BLUETOOTH)
  if (deviceConnected) {
    char outData[MAX_MSG_SZ] = "";
    for (int i = 0; i < data.length(); i++) {
      outData[i] = data[i];
    }
    pTxCharacteristic->setValue(outData);
    pTxCharacteristic->notify();
  }
#endif
}

//-------------------------------------------------------//
// INTERRUPT SERVICE ROUTINES (ISR)
//-------------------------------------------------------//
#if HAS_SONAR

// ISR: Start timer to measure the time it takes for the pulse to return
void start_timer() {
  start_time = micros();
  attachPinChangeInterrupt(digitalPinToPinChangeInterrupt(PIN_ECHO), stop_timer, FALLING);
}

// ISR: Stop timer and record the time
```

```
void stop_timer() {
  echo_time = micros() - start_time;
  detachPinChangeInterrupt(digitalPinToPinChangeInterrupt(PIN_ECHO));
  ping_success = true;
}

#endif

#if (HAS_SPEED_SENSORS_FRONT)

// ISR: Increment speed sensor counter (left front)
void update_speed_lf() {
  if (ctrl_left < 0) {
    counter_lf--;
  } else if (ctrl_left > 0) {
    counter_lf++;
  }
}

// ISR: Increment speed sensor counter (right front)
void update_speed_rf() {
  if (ctrl_right < 0) {
    counter_rf--;
  } else if (ctrl_right > 0) {
    counter_rf++;
  }
}

#endif

#if (HAS_SPEED_SENSORS_BACK)
// ISR: Increment speed sensor counter (left back)
void update_speed_lb() {
  if (ctrl_left < 0) {
    counter_lb--;
  } else if (ctrl_left > 0) {
    counter_lb++;
  }
}

// ISR: Increment speed sensor counter (right back)
void update_speed_rb() {
  if (ctrl_right < 0) {
    counter_rb--;
  } else if (ctrl_right > 0) {
    counter_rb++;
  }
}
#endif

#if (HAS_SPEED_SENSORS_MIDDLE)
```

```
// ISR: Increment speed sensor counter (left mid)
void update_speed_lm() {
  if (ctrl_left < 0) {
    counter_lm--;
  } else if (ctrl_left > 0) {
    counter_lm++;
  }
}

// ISR: Increment speed sensor counter (right mid)
void update_speed_rm() {
  if (ctrl_right < 0) {
    counter_rm--;
  } else if (ctrl_right > 0) {
    counter_rm++;
  }
}
#endif
```