



**Wydział Elektroniki
i Technik Informatycznych**

POLITECHNIKA WARSZAWSKA

28 stycznia 2025

Sprawozdanie PMiK

Zdalnie sterowany samochodzik

Marcin Krawiec 325027

Tymoteusz Krasnowski 325026



Spis treści

1.	Wstęp.....	3
1.1.	Temat projektu	3
1.2.	Wykaz realizowanych funkcji	3
1.3.	Ogólny opis algorytmu	4
1.4.	Wykaz użytych elementów	4
2.	Prace projektowe.....	5
2.1.	Schemat blokowy	5
2.2.	Schemat elektryczny.....	6
2.3.	Komunikacja	8
2.4.	ADC.....	10
2.4.1	Obsługa joysticka – plik konfiguracyjny i wykonawczy:.....	11
2.5.	Odczyt danych i filtracja:	12
2.6.	Sterowanie silnikami <i>DC 5V</i>	13
2.6.1	Sterownik DRV8835:.....	13
2.6.2	Funkcje sterujące pracą drivera:	16
	Rys. 16: drv8835.h interfejs do sterowania mostkiem H DRV8835	16
2.7	Czujnik odległości HC SR-04	18
2.7.1	Opis czujnika.....	18
2.7.2	Funkcje sterujące czujnikiem	19
2.8	Pętla główna:	21
2.8.1	Struktura danych zmiennych w pętli main STM32(1):	21
2.8.2	Algorytm obliczania mocy silników w pętli głównej STM32(1):	22
2.8.3	Praca układu:	24
3	Napotkane problemy i ich rozwiązania	25
3.1	Błędne odczyty ADC	25
3.2	Transmisja danych STM32 – ESP32.....	26
3.3	Transmisja danych za pomocą WiFi.....	26
3.4	Niedziałanie programu przy zewnętrznym zasilaniu	27
3.5	Zaszumione dane z wifi:	27
3.6	Silniki nie chciały się kręcić w momencie wychylenia joysticka.	27
3.7	Sygnał PWM nie uruchamiał drugiego silnika	28

3.8	Zbyt mały moment obrotowy silników	28
3.9	Tragiczny sygnał zasilający	28
4	Druk 3D:	28

1. Wstęp

1.1. Temat projektu

Tematem projektu jest zbudowanie od podstaw zdalnie sterowanego samochodu z napędem na cztery koła, z wykorzystaniem mikrokontrolera STM32 Nucleo. Sterowanie pojazdem będzie odbywać się bezprzewodowo, za pomocą joysticka komunikującego się z pojazdem przez moduły Wi-Fi oparte na ESP8266. Pojazd będzie zasilany napięciem ok. 6 V z połączonych ze sobą szeregowo baterii. Napęd składa się z 4 silników *DC*, sterowanych parami z *drivera PWM*. Dodatkowo, pojazd będzie posiadał czujnik odległości, który będzie ostrzegał użytkownika przed zbyt małą odległością od przeszkody za pomocą zapalenia czerwonej diody LED.

1.2. Wykaz realizowanych funkcji

1. Zastosowanie mikrokontrolerów STM32 Nucleo F303RE:
 - Wykorzystanie dwóch STM32 jako jednostek centralnych sterujących pracą silników, odczytem danych z joysticka, komunikacją przez Wi-Fi oraz dodatkowymi czujnikami i peryferiami
2. Sterowanie napędem pojazdu:
 - Regulacja pracy silników, za pomocą zewnętrznego *drivera DRV8835*,
 - Regulacja prędkości za pomocą sygnału *PWM* generowanego przez mikrokontroler *STM32*
 - Wybór kierunku obrotu silników za pomocą odpowiedniego stanu logicznego 2 portów *GPIO*
3. Sterowanie pojazdem za pomocą joysticka:
 - Odczyt wychylenia gałki joysticka za pomocą napięć generowanych z dwóch potencjometrów oraz przetworzenie ich na wartość cyfrową za pomocą przetwornika *ADC*
 - Komunikacja bezprzewodowa przez Wi-Fi
4. Obsługa czujnika odległości:
 - Czujnik odległości odczytuje dystans od przeszkody i generuje przerwanie w momencie, gdy odległość jest mniejsza niż 30 cm. W obsłudze przerwania mikrokontroler zaświeca czerwoną diodę LED

1.3. Ogólny opis algorytmu

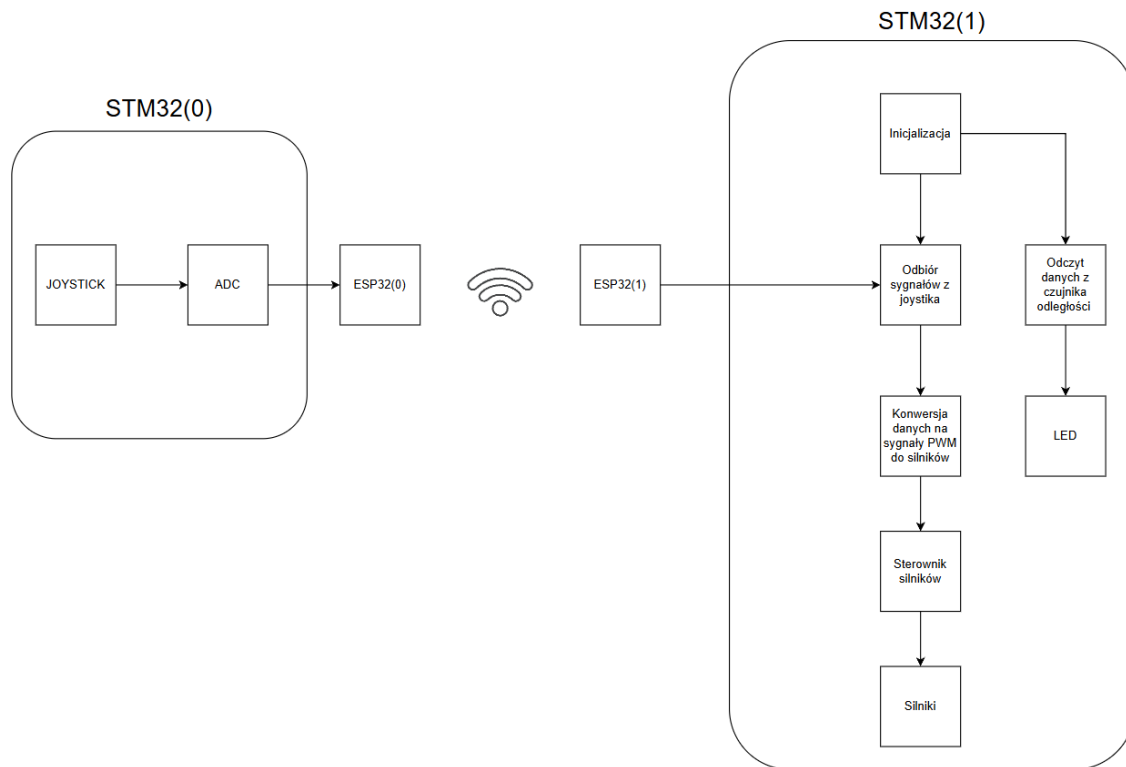
1. Inicjalizacja:
 - a. Konfigurowanie wszystkich wymaganych komponentów i urządzeń w tym pinów dla drivera silnika, modułów *ESP8266*, czujnika odległości i diod *LED*
 - b. Inicjalizacja liczników generujących sygnał *PWM*
 - c. Inicjalizacja modułów komunikacyjnych *Wi-Fi* do odbioru danych z joysticka
2. Odbiór sygnałów z joysticka:
 - a. Odbiór informacji o wartości wychylenia joysticka w kierunku x i y
 - b. Przetworzenie danych przez *ADC* na wartości cyfrowe
3. Przekazywanie sygnałów poprzez *Wi-Fi*
 - a. Przesyłanie informacji o pozycji joysticka w osi x i y poprzez interfejs komunikacyjny *UART* do modułu *ESP8266*
 - b. Przesłanie danych między modułami *ESP8266* i przekazanie ich do drugiego mikrokontrolera *STM32* za pomocą *UART*
4. Sterowanie pracą silników:
 - a. *STM32* odczytuje nadesłane dane i za pomocą algorytmu generuje 2 sygnały *PWM* o odpowiednim wypełnieniu, sterujące pracą silników. W tym momencie również wystawia linie *GPIO* w odpowiedni stan logiczny
5. Monitorowanie odległości od przeszkód:
 - a. Odczyt danych z czujnika odległości w pętli z wykorzystaniem timera
 - b. Jeżeli odległość jest mniejsza niż progowa, zapala się dioda *LED* ostrzegająca o możliwej kolizji
6. Aktualizacja stanu pojazdu:
 - a. System odbiera dane z joysticka i na bieżąco dostosowuje sygnał *PWM* do sterowania kierunkiem i prędkością

1.4. Wykaz użytych elementów

1. Zestaw ewaluacyjny *STM32 Nucleo F303RE* - 2x
2. Zestaw ewaluacyjny *ESP32-WROOM-32*
3. Moduł *Wi-Fi ESP8266*
4. Silnik szczotkowy *DC FC-130SA* - 4x
5. Driver *DRV-8835* - 2x
6. Powerbank *Xiaomi Redmi 10000 mAh*
7. Baterie *AAA* - 6x
8. Koszyk na baterie *AAA* - 2x

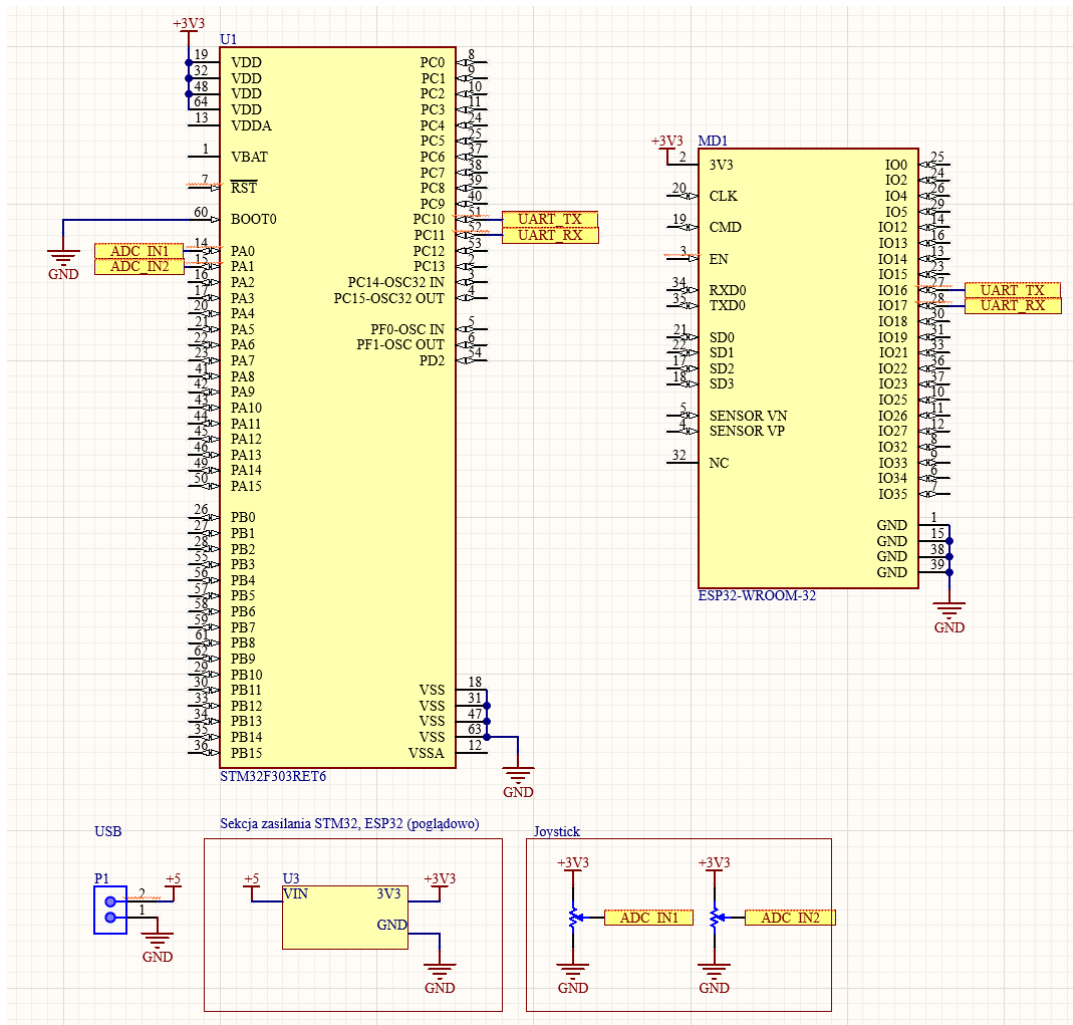
2. Prace projektowe

2.1. Schemat blokowy



Rys 1. Schemat blokowy przedstawiający działanie samochodu i kontrolera

2.2. Schemat elektryczny

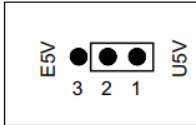
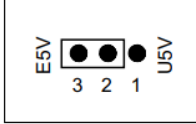


Rys 2. Schemat elektryczny kontrolera

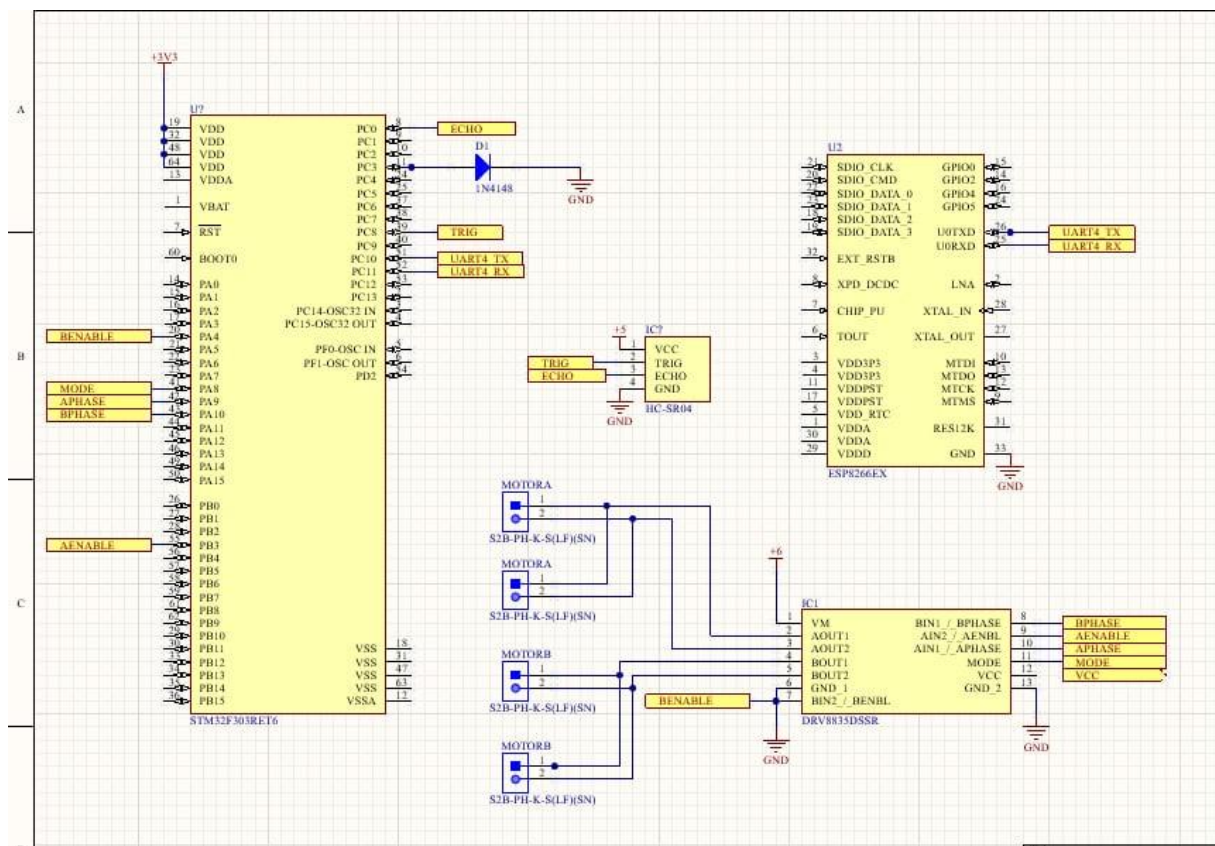
Z racji na pusty (bez wyprowadzeń) prostokąt w Altium Designer zestawów ewaluacyjnych, na schemacie umieszczone są *MCU*. Pomińto w nich połączenia takie jak: *debugger*, diody, przyciski, kwarce itd. Natomiast dla przejrzystości zasilania układu, gdyż nie odbywa się przez port *USB debuggera*, widoczny jest blok sekcji zasilania (poglądowy).

Zasilanie kontrolera odbywa się przez *USB-C* z powerbanka. W *Nucleo F303RE* za zasilanie MCU odpowiada zworka w wyprowadzeniu J5, która wpinana jest na 2 sposoby: zasilanie z *USB* przez debugger oraz zewnętrznie. Dla naszych potrzeb zasilamy zewnętrznie. ESP32 nie posiada zworek, zasilanie odbywa się bezpośrednio wpinając się do pinu stabilizatora.

Table 8. Power-related jumper

Jumper	Description
JP5	U5V (ST-LINK VBUS) is used as a power source when JP5 is set as shown below (Default setting)
	
JP5	VIN or E5V is used as a power source when JP5 is set as shown below.
	

Rys 3. Tabela z noty katalogowej przedstawiająca ułożenie zworki na piny J5



Rys 4. Schemat elektryczny samochodu

2.3. Komunikacja



Rys 5. Podgląd komunikacji na oscyloskopie.

Komunikacja odbywa się za pomocą interfejsu *UART* i protokołu *ESP-NOW* między *ESP32*. *STM32* po stronie kontrolera wysyła 5 bajtów: ramka i 4 wartości z *ADC*, odpowiednio: starszy i młodszy bajt wychylenia X joysticka oraz starszy i młodszy bajt wychylenia Y joysticka.

```
void data_prepare(Communication *communication){  
    communication->txBuffer[0] = 0xA0; // Znacznik początku ramki  
    communication->txBuffer[1] = (joystick.adc_value[0] >> 8) & 0xFF; // starszy bajt X  
    communication->txBuffer[2] = joystick.adc_value[0] & 0xFF; // młodszy bajt X  
    communication->txBuffer[3] = (joystick.adc_value[1] >> 8) & 0xFF; // starszy bajt Y  
    communication->txBuffer[4] = joystick.adc_value[1] & 0xFF; // młodszy bajt Y
```

Rys 6. Bufor danych do wysłania, z ramką i rozdzieleniem wartości 16 bitowych na dwie 8-bitowe.

Następnie dane są wysyłane do *ESP32* z prędkością 115200 b/s przez *DMA*. Tam następuje ich odbiór i wysyłka za pomocą protokołu *ESP-NOW*, o ile zostanie znaleziony adres *MAC* odbiorcy.


```

void loop() {
  if (Serial2.available() >= 5) {
    Serial2.readBytesUntil('\n', buffer, 5);
  }

  // Wysłanie danych za pomocą ESP-NOW
  esp_err_t result = esp_now_send(receiverMacAddress, buffer, sizeof(buffer));

  if (result == ESP_OK) {
    Serial.println("Data sent via ESP-NOW");
  } else {
    Serial.println("Error sending the data");
  }
}

```

Rys 7. Pętla główna ESP32, w której dane są odczytywane z UART (Serial2) i wysyłane (esp_now_send).

Protokół ESP-NOW odbiera dane tylko w przerwaniu, zatem za każdym razem, gdy przyjdą nowe dane, wywołuje się przerwanie, w którym dane są kopiowane i wysyłane za pomocą *UART* do drugiej *STM32*.

```

void onReceive(uint8_t *mac, uint8_t *incomingData, uint8_t len) {
  memcpy(&buffer, incomingData, sizeof(buffer));
  Serial.write(buffer, len);
}

```

Rys 8. Callback do odbioru danych, pętla while jest pusta.

Warto tutaj zaznaczyć, że domyślnie *ESP32* jak i moduł *ESP8266* sygnalizują status komunikacji niebieską diodą, dzięki czemu, gdy wystąpi jakiś problem można od razu wiedzieć, czy komunikacja przebiega prawidłowo – ułatwiony *debugging*.

Następnie, dane muszą zostać z powrotem przekonwertowane na swoje 16-bitowe wartości. Służy do tego funkcja *convertwifitada*.

```

void convertwifidata(Wifidataconv *wifidataconv) {
    for (uint8_t i = 0; i < 5; i++) {
        if (dataread.rxbuffer[i] == 0xA0) {
            switch (i) {
                case 0:
                    wifidataconv->posX = dataread.rxbuffer[1] << 8 | dataread.rxbuffer[2];
                    wifidataconv->posY = dataread.rxbuffer[3] << 8 | dataread.rxbuffer[4];
                    break;
                case 1:
                    wifidataconv->posX = dataread.rxbuffer[2] << 8 | dataread.rxbuffer[3];
                    wifidataconv->posY = dataread.rxbuffer[4] << 8 | dataread.rxbuffer[0];
                    break;
                case 2:
                    wifidataconv->posX = dataread.rxbuffer[3] << 8 | dataread.rxbuffer[4];
                    wifidataconv->posY = dataread.rxbuffer[0] << 8 | dataread.rxbuffer[1];
                    break;
                case 3:
                    wifidataconv->posX = dataread.rxbuffer[4] << 8 | dataread.rxbuffer[0];
                    wifidataconv->posY = dataread.rxbuffer[1] << 8 | dataread.rxbuffer[2];
                    break;
                case 4:
                    wifidataconv->posX = dataread.rxbuffer[0] << 8 | dataread.rxbuffer[1];
                    wifidataconv->posY = dataread.rxbuffer[2] << 8 | dataread.rxbuffer[3];
                    break;
            }

            printf("%d, %d\n", wifidataconv->posX, wifidataconv->posY);
            HAL_Delay(50);
            break;
        }
    }
}

```

Rys 9. Funkcja convertwifidata

Ta funkcja nie tylko konwertuje 2 8-bitowe wartości na jedną 16-bitową, ale też wyszukuje ramki, czyli początku transmisji. Po odbiorze danych przez *ESP8266* okazało się, że ramki nie przychodzą po kolei – są przesunięte w czasie. Musiał powstać mechanizm, który zapewni poprawny odczyt danych.

Jednak to nie koniec. Wartości z *ADC* potrafią być zaszumione, za sprawą beznadziejnej jakości joysticka z zestawu ewaluacyjnego arduino (klon, chińczyk), a podczas transmisji po *Wi-Fi* danych również może dojść do uszkodzenia jej zawartości. W związku z tym, po odbiorze danych i konwersji ich do docelowych wartości, powstał filtr medianowy. Dzięki temu prostemu rozwiązaniu problem zniknął, a sama transmisja i jej dekodowanie pozostało niezmienione (mam tu na myśli bardziej zaawansowany mechanizm, na przykład z sumą kontrolną).

2.4. ADC

Joystick to nic innego jak dwa potencjometry, których rezystancję reguluje się odchyleniem gałki analogowej. Zatem obsługa joysticka wymaga użycia przetwornika analogowo-cyfrowego. Próbkuje on wartości z dwóch kanałów – X i Y gałki analogowej, o rozdzielczości 12 bitów i wysyła bezpośrednio do *DMA*.

ADC odczytuje wartości odchylenia joysticka z prędkością 61.5 cykli.

▼ ADC_Regular_ConversionMode	
Enable Regular Conversions	Enable
Number Of Conversion	2
External Trigger Conversion Source	Regular Conversion launched by software
External Trigger Conversion Edge	None
SequencerNbRanks	1
▼ Rank	1
Channel	Channel 1
Sampling Time	61.5 Cycles
Offset Number	No offset
Offset	0
▼ Rank	2
Channel	Channel 2
Sampling Time	61.5 Cycles
Offset Number	No offset
Offset	0

Rys 10. Fragment konfiguracji ADC z IOC.

2.4.1 Obsługa joysticka – plik konfiguracyjny i wykonawczy:

```

14 #ifndef SRC_JOYSTICKINIT_H_
15 #define SRC_JOYSTICKINIT_H_
16
17 //include "stm32f3xx_hal_msp.c"
18
19 #include <stdio.h>
20 #include <string.h>
21
22 extern ADC_HandleTypeDef hadc1; // Obsługa ADC dla joysticka
23 extern TIM_HandleTypeDef htim2;
24
25 #define ADC_MAX 4096 // Maksymalna wartość ADC (12-bitowa rozdzielczość)
26 #define SCALE_FACTOR 4.095 // Skalowanie ADC do zakresu 0-1000
27 #define BUFFER_SIZE 5 // Liczba próbek w buforze
28
29 typedef struct Joystick{ //Struktura przechowująca dane joysticka
30
31     volatile uint16_t parameter[2]; //volatile to było ze tylko dma może to zmieniać - użytkownikowi nie pozwili na zmianę
32     uint32_t convertedparameter[2];
33
34 }Joystick;
35
36 void joystick_run(Joystick *joystick);
37 void joystick_printconv(Joystick *joystick);
38 void joystick_conv(Joystick *joystick);
39
40
41 #endif /* SRC_JOYSTICKINIT_H_
42

```

Rys 11. Plik konfiguracyjny joystickinit.h

Plik joystickinit.h jest stworzoną przez nas biblioteką obsługi joysticka. Znajdują się w niej definicje przetwornika, odpowiednie stałe pozwalające na konwersje danych przekazywanych z ADC na takie bardziej przystępne dla naszych wymagań. Następnie plik zawiera strukturę danych, ze zmiennymi w postaci tablicy, do której przypisywane są aktualne wartości wychYLENIA joysticka. Plik zakończony jest deklaracjami funkcji wykorzystywanych w pliku implementacyjnym.

```

21 void joystick_run(Joystick *joystick) //Funkcja uruchamia kalibrację ADC oraz startuje DMA do odczytu danych z joysticka
22 {
23     HAL_ADCEX_Calibration_Start(&hadc1, ADC_SINGLE_ENDED);
24     HAL_ADC_Start_DMA(&hadc1, (uint32_t *)joystick->parameter, 2);
25 }
26

```

Rys 12. Plik implementacyjny joystickstart.c – funkcja obsługi ADC z DMA

Funkcja *joystick_run* zawiera fragment inicjalizacji ADC oraz jego kalibracji, wraz z ze startem DMA. Pozwala na rozpoczęcie odczytywania danych z joysticka i zapisywanie ich w tablicy *parameter[]*. Następnie te dane zostają wysłane do drugiego STM32 zgodnie z procedurą opisaną powyżej.

2.5. Odczyt danych i filtracja:

Wysyłanie danych poprzez Wi-Fi ma tę niewygodną cechę, że wartości są dość podatne na szumy występujące w przestrzeni. W naszym projekcie sam fakt, że moduł ESP odbierający dane był bardzo blisko silników prądu stałego, które mogą powodować zakłamania odbieranych wartości. Problem występuje co kilka, kilkanaście próbek dlatego podjęliśmy działania mające na celu eliminację tego problemu.

Pierwszym z nich było zastosowanie filtra medianowego:

```
64 /** Funkcja sortuje bufor i zwraca jego medianę. Służy do filtrowania szumu w danych ADC */
65
66 uint16_t get_median(uint16_t *buffer)
67 {
68     uint16_t temp[BUFFER_SIZE];
69     memcpy(temp, buffer, sizeof(temp)); // Skopiuj dane, aby nie zmieniać bufora
70
71     // Sortowanie tablicy (metoda bąbelkowa)
72     for (int i = 0; i < BUFFER_SIZE - 1; i++) {
73         for (int j = i + 1; j < BUFFER_SIZE; j++) {
74             if (temp[i] > temp[j]) {
75                 uint16_t t = temp[i];
76                 temp[i] = temp[j];
77                 temp[j] = t;
78             }
79         }
80     }
81     return temp[BUFFER_SIZE / 2];
82 }
83
84 // Funkcja filtruje dane z joysticka za pomocą mediany, a następnie skaluje je do zakresu 0-1000
85 void joystick_conv(Wifidataconv *wifidataconv)
86 {
87     static uint16_t buffer_x[BUFFER_SIZE] = {0}; // Bufor dla osi X
88     static uint16_t buffer_y[BUFFER_SIZE] = {0}; // Bufor dla osi Y
89     static uint8_t buffer_index = 0; // Indeks cykliczny dla bufora
90
91     // Dodawanie nowych danych do buforów
92     buffer_x[buffer_index] = wifidataconv->posX;
93     buffer_y[buffer_index] = wifidataconv->posY;
94     buffer_index = (buffer_index + 1) % BUFFER_SIZE; // Cykliczny indeks
95
96     // Przetwarzanie danych filtrem medianowym
97     wifidataconv->posX = (get_median(buffer_x) * 1000) / 4096; // Skalowanie na 0-1000
98     wifidataconv->posY = 1000 - ((get_median(buffer_y) * 1000) / 4096);
99 }
100
```

Rys 13. Plik wifidataconvert.c – funkcja filtrująca dane i skalująca do odpowiednich wartości

Pierwsza funkcja, `get_median`, jest odpowiedzialna za filtrowanie danych. Implementuje metodę mediany, polegającą na posortowaniu kopii tablicy danych i wybraniu wartości środkowej. Dane są kopiowane do tymczasowej tablicy `temp`, aby nie zmieniać oryginalnych wartości. Następnie funkcja stosuje algorytm sortowania bąbelkowego, który porównuje i zamienia miejscami elementy, aż cała tablica zostanie uporządkowana. Po zakończeniu zwracana jest mediana wartości. Mediana jest używana, ponieważ skutecznie eliminuje wpływ pojedynczych, znacząco odchylonych wartości (szumów) na dane wejściowe.

Druga funkcja, *joystick_conv*, wykorzystuje funkcję *get_median* do przetwarzania danych z joysticka. Dane te są reprezentowane przez współrzędne osi X i Y, które są odczytywane z obiektu *Wifidataconv*. Funkcja przechowuje ostatnie wartości z joysticka w dwóch oddzielnych buforach, jednym dla osi X i jednym dla osi Y. Bufory te mają ustaloną wielkość (zdefiniowaną przez *BUFFER_SIZE*), a nowe wartości są dodawane w sposób cykliczny, nadpisując najstarsze dane.

Dla każdej nowej wartości dane w buforze są przetwarzane za pomocą funkcji *get_median*, co pozwala na odfiltrowanie szumów. Po zastosowaniu mediany wartości są skalowane do zakresu 0–1000.

Drugą metodą, którą szerzej opiszę w pętli głównej jest zastosowanie ogranicznika wartości. W momencie, gdy po skalowaniu wartości, wychylenia joysticka są większe niż 1000 (czyli większe niż maksymalne wypełnienie PWM) to wykorzystywana jest funkcja ograniczająca ten zakres do wartości maksymalnej – 1000.

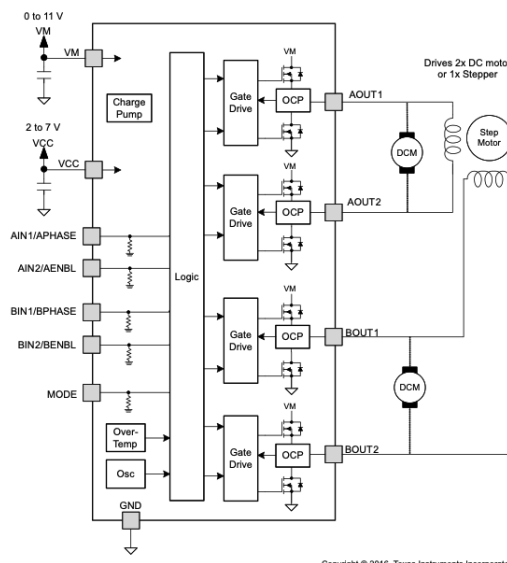
Wszystkie stałe opisane są w bibliotece *wifidataconvert.h*:

```
30 //extern ADC_HandleTypeDef hadc1;      // Obsługa ADC dla joysticka
31 extern TIM_HandleTypeDef htim2;
32
33 #define ADC_MAX 4096                    // Maksymalna wartość ADC (12-bitowa rozdzielczość)
34 #define SCALE_FACTOR 4.095              // Skalowanie ADC do zakresu 0-1000
35 #define BUFFER_SIZE 5                  // Liczba próbek w buforze
36
37 typedef struct wifidataconv{
38     uint16_t posX;
39     uint16_t posY;
40 }Wifidataconv;
```

Rys. 14 Struktury danych do obsługi wartości z joysticka

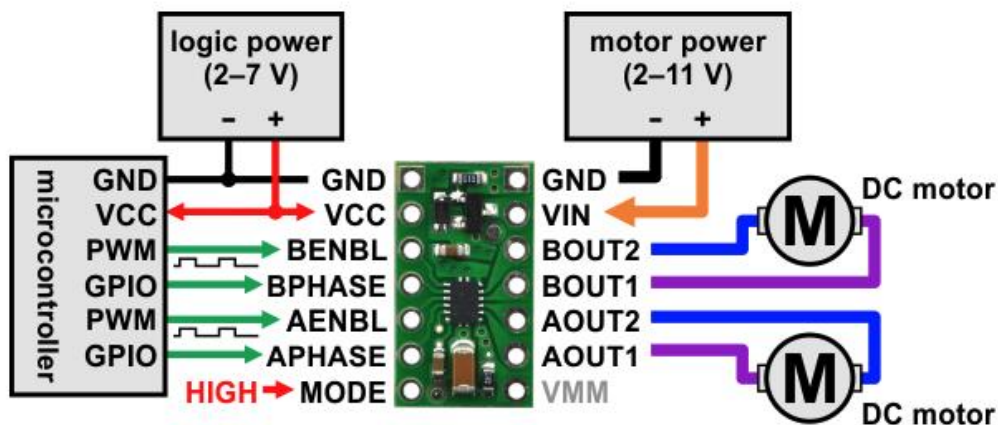
2.6. Sterowanie silnikami DC 5V

2.6.1 Sterownik DRV8835:



Rys. 15 Schemat blokowy drivera

DRV8835 to zintegrowany układ sterownika silnika przeznaczony do sterowania silnikami szczotkowymi. Urządzenie integruje dwa mostki H, co umożliwia sterowanie dwoma silnikami DC lub jednym silnikiem krokowym. Blok wyjściowy każdego mostka H składa się z tranzystorów mocy MOSFET typu N. Wewnętrzna pompa ładunkowa generuje napięcia sterujące bramkami tranzystorów. Funkcje ochronne obejmują ochronę przed przeciążeniem prądowym, zwarcie, blokadę podnapięciową oraz ochronę przed przegrzaniem.



Rys. 12 Schemat połączeń drivera z silnikami i STM32

Sterownik ma dwa tryby pracy, które możemy wybrać za pomocą stanu niskiego lub wysokiego na wejściu MODE:

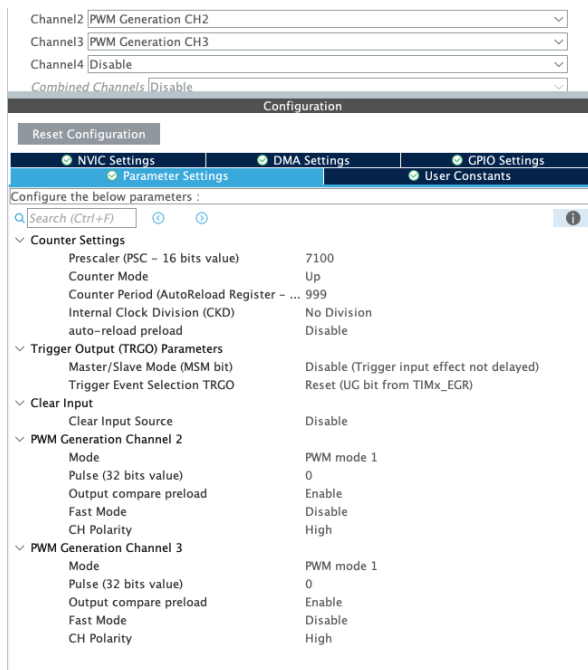
- Tryb PHASE/ENABLE to prostszy tryb sterowania – do uruchomienia silnika potrzebujemy jednego sygnału PWM oraz jednego sygnału cyfrowego GPIO. W trybie tym możemy sterować zarówno kierunkiem, jak i prędkością obrotu silnika oraz wywołać hamowanie. W naszym programie wykorzystaliśmy tryb PHASE/ENABLE
- Tryb IN/IN pozwala na bardziej zaawansowane sterowanie silnikiem. Do uruchomienia tego trybu wymagane są dwa sygnały PWM. Poza sterowaniem kierunkiem i prędkością oraz hamowaniem, mamy też możliwość wyłączenia wyjść sterownika (bieg jałowy).

PHASE	ENABLE	OUT1	OUT2	Opis
0	PWM	PWM	L	ruch w jednym kierunku z prędkością PWM
1	PWM	L	PWM	ruch w drugim kierunku z prędkością PWM
X	0	L	L	hamowanie (wyjścia zwarte do GND)

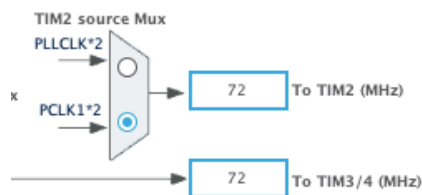
Rys. 13 Opis funkcjonalności drivera dla PWM

Częstotliwość sygnałów *PWM* z obydwu kanałów timera została ustawiona na 10 Hz, co pozwala na pracę silników z pełną mocą. Zgodnie ze wzorem:

$$PWM_Freq = Timer_Freq / (Prescaler * Counter Period)$$



Rys. 14 Konfiguracja timera dla sygnału *PWM*



Rys. 15 Częstotliwość zegara dla timera 2

2.6.2 Funkcje sterujące pracą drivera:

```
13 #ifndef INC_DRV8835_H_
14 #define INC_DRV8835_H_
15
16 #include <stdint.h>
17
18 //-----
19 typedef enum
20 {
21     In_In_Mode = 0,
22     Phase_Enable_Mode = 1
23 }DRV8835_Mode;
24
25 //-----
26 typedef enum
27 {
28     CWA = 0,
29     CCWA = 1
30 }DRV8835_DirectionA;
31 //-----
32
33 typedef enum
34 {
35     CWB = 0,
36     CCWB = 1
37 }DRV8835_DirectionB;
38 //-----
39
40 void drv8835_init();
41 void drv8835_mode_control(DRV8835_Mode);
42 void drv8835_mode2_control(DRV8835_Mode);
43 void drv8835_set_motorA_direction(DRV8835_DirectionA);
44 void drv8835_set_motorB_direction(DRV8835_DirectionB);
45 void drv8835_set_motorA_speed(uint16_t speedA);
46 void drv8835_set_motorB_speed(uint16_t speedB);
47 uint16_t drv8835_limit_motor_speed(int16_t speed);
48
49 #endif /* INC_DRV8835_H_ */
51 |
```

Rys. 16: drv8835.h interfejs do sterowania mostkiem H DRV8835

Plik drv8835.h definiuje interfejs do sterowania mostkiem H DRV8835. Znajdują się w nim:

1. Typy wyliczeniowe:

- **DRV8835_Mode:** Tryby pracy sterownika:
 - In_In_Mode – bezpośrednie sterowanie sygnałami kierunkowymi.
 - Phase_Enable_Mode – sterowanie za pomocą sygnału fazowego i PWM.
- **DRV8835_DirectionA i DRV8835_DirectionB:** Kierunki obrotów dla silników A i B (zgodny lub przeciwny do ruchu wskazówek zegara).

2. Implementacje funkcji

W pliku drv8835.c znajdują się funkcje obsługujące sterownik:

```
18 //-----  
19 void drv8835_mode_control(DRV8835_Mode mode) //Funkcja wyboru trybow pracy drivera,  
20 { //korzystamy z Phase_Enable_Mode  
21     if(mode == Phase_Enable_Mode)  
22         HAL_GPIO_WritePin(MODE_GPIO_Port, MODE_Pin, SET);  
23     else if(mode == In_In_Mode)  
24         HAL_GPIO_WritePin(MODE_GPIO_Port, MODE_Pin, RESET);  
25 }  
26 //-----
```

Rys.17 Funkcja wyboru trybu pracy

Funkcja wyboru trybu pracy ustawia port GPIO PA8 (nazwa MODE) w odpowiedni stan w zależności od trybu pracy. W naszym przypadku jest to stan wysoki (tryb PHASE/ENABLE)

```
--  
37 void drv8835_set_motorA_direction(DRV8835_DirectionA dir) //Funkcja wybor kierunku obrotu prawego silnika  
38 {  
39     if(dir == CCWA)  
40         HAL_GPIO_WritePin(APHASE_GPIO_Port, APHASE_Pin, SET);  
41     else if(dir == CWA)  
42         HAL_GPIO_WritePin(APHASE_GPIO_Port, APHASE_Pin, RESET);  
43 }  
44 //-----  
45 void drv8835_set_motorB_direction(DRV8835_DirectionB dir) //Funkcja wybor kierunku obrotu lewego silnika  
46 {  
47     if(dir == CCWB)  
48         HAL_GPIO_WritePin(BPHASE_GPIO_Port, BPHASE_Pin, SET);  
49     else if(dir == CWB)  
50         HAL_GPIO_WritePin(BPHASE_GPIO_Port, BPHASE_Pin, RESET);  
51 }  
--
```

Rys.18 Funkcja wyboru kierunku obrotów

Funkcja ta wystawia na porty GPIO PA9 i PA10 odpowiednie stany logiczne umożliwiające wybór kierunku obrotów silnika. Są 2 możliwości CW – „clockwise” – zgodnie z ruchem wskazówek zegara i CCW – „counter clockwise” – przeciwnie do ruchu wskazówek zegara.

```
52 //-----  
53 void drv8835_set_motorA_speed(uint16_t speedA) //Ustawianie predkosci prawego silnika  
54 { //PWM max 1000, ustawienie wybranego wypelnienia  
55     TIM2->CCR2 = speedA;  
56 }  
57 //-----  
58 void drv8835_set_motorB_speed(uint16_t speedB) //Ustawianie predkosci lewego silnika  
59 { //PWM max 1000, ustawienie wybranego wypelnienia  
60     TIM3->CCR2 = speedB;  
61 }  
62 }  
63 //-----  
64 void drv8835_init() //Inicjalizacja drivera: tryb phase/enable, obrot zgodnie z ruchem
```

Rys.19 Funkcja przypisujące prędkość

Te dwie funkcje przypisują bezpośrednio wartość do rejestru CCR2 licznika 2 i CCR3 licznika 3. W ten sposób odbywa się modulowanie wartości wypełnienia sygnału PWM z odpowiednich liczników. W związku z tym, że wartość speedA i speedB jest bezpośrednio przypisywana do rejestru, a maksymalne wypełnienie współczynnika PWM to 1000 to musi jeszcze być stworzona funkcja ograniczająca maksymalny zakres zmiennych speed.

```

73 //
74 uint16_t drv8835_limit_motor_speed(uint16_t speed) // Funkcja ograniczająca maks. prędkość silnika do maks. wypełnienia PWM
75 {
76     if (speed > 1000) return 1000; // Maksymalna wartość PWM
77     if (speed < 0) return 0; // Minimalna wartość PWM
78     return (uint16_t)speed;
79 }

```

Rys.20 Funkcja ograniczająca wartości zmiennych speed

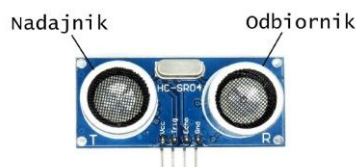
Funkcja *drv8835_limit_motor_speed* zapewnia, że do rejestrów CCR2 TIM2 i CCR3 TIM3 nie zostanie przypisana wartość spoza zakresu 0-1000.

2.7 Czujnik odległości HC SR-04

2.7.1 Opis czujnika

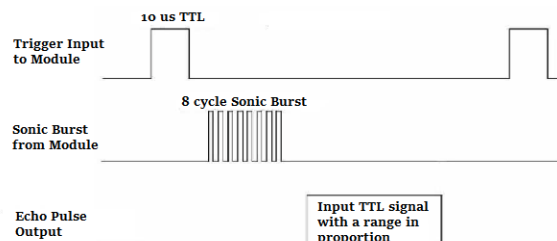
HC-SR04 to ultradźwiękowy czujnik odległości, czyli czujnik, który do pomiaru wykorzystuje falę dźwiękową o częstotliwości niesłyszalnej dla człowieka. Taką częstotliwość wykorzystują często zwierzęta np. pies, delfin czy nietoperz. Ultradźwięki dzięki małej długości fali wykorzystywane są przez człowieka m.in. w sonarach okrętów podwodnych (do detekcji obiektów) oraz medycynie.

Czujnik HC-SR04 wykorzystuje falę dźwiękową o częstotliwości 40 kHz. Zbudowany jest z nadajnika i odbiornika oraz zestawu układów elektronicznych, które wstępnie przetwarzają sygnał generowany przez nadajnik oraz przychodzący do odbiornika w taki sposób, aby wyprowadzić nam sygnał możliwie łatwy do odczytu. Czujnik standardowo ma wyprowadzone cztery piny: VCC(zasilanie), GND (masa), TRIG (wejście wyzwalające pomiar) oraz ECHO (wyjście zwracające wartość pomiaru).



Rys. 21 Czujnik HC-SR04

Interfejs czujnika (wejście TRIG i wyjście ECHO) bazuje na sygnałach czasowych. Aby wywołać pomiar, należy podać na pin TRIG stan wysoki o czasie trwania minimum 10 μ s. Sygnał ten zamieniany jest na osiem impulsów o częstotliwości 40 kHz i generowany przez sondę nadawczą. Jeżeli fala dźwiękowa napotka przeszkodę na swojej drodze, odbija się od niej i wraca do odbiornika. Czujnik na wyjściu ECHO generuje impuls o szerokości proporcjonalnej do zmierzonej odległości.

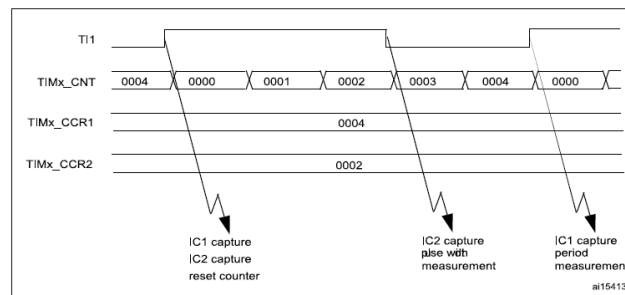


Rys. 22 Praca czujnika

Pomiar czasu trwania impulsu na wyjściu ECHO pozwala nam na odczytanie, w jakiej odległości od czujnika znajduje się przeszkoda. Zamiana czasu trwania impulsu na odległość podaną w centymetrach sprowadza się do prostych obliczeń matematycznych. Czas impulsu jest bowiem ściśle związany z czasem wędrowania fali dźwiękowej do przeszkody i z powrotem. Wiedząc, że fala dźwiękowa porusza się z prędkością 340 m/s, możemy obliczyć odległość, jaką przebyła. Ponieważ interesuje nas odległość do przedmiotu, musimy podzielić drogę przebytą przez falę dźwiękową przez 2.

$$\text{odległość_m} = (340 \text{ m/s} * \text{czas_impulsu_s}) / 2$$

Czujnik może pracować w 3 trybach, jednak my wykorzystaliśmy tryb *PWM Input*. Jest on swego rodzaju odmianą trybu Input Capture – wykorzystuje dwa sygnały Input Capture zmapowane na ten sam pin. Każdy z sygnałów jest aktywny na inne zbocze – pierwszy na zbocze narastające, a drugi na opadające. W momencie wywołania pierwszego zbocza licznik jest resetowany. Gdy nastąpi zbocze opadające pobierana jest informacja o liczniku i zapisywana w rejestrze CCR2 (kanał 2). W momencie wystąpienia kolejnego zbocza narastającego, w rejestrze CCR1 zapisywana jest informacja o okresie sygnału.



Rys. 23 Tryb PWM Input

2.7.2 Funkcje sterujące czujnikiem

```

14 struct us_sensor_str
15 {
16     TIM_HandleTypeDef *htim_echo;
17     TIM_HandleTypeDef *htim_trig;
18     uint32_t trig_channel;
19
20     volatile uint32_t distance_cm; //zmienna przechowująca odległość w cm
21     volatile uint8_t led_state;    // Zmienna przechowująca stan LEDa
22 };
23
24 void hc_sr_04_init(struct us_sensor_str *us_sensor, TIM_HandleTypeDef *htim_echo, TIM_HandleTypeDef *htim_trig, uint32_t trig_channel);
25
26 void hc_sr_04_convert_us_to_cm(uint32_t distance_us);
27
28 #endif /* INC_HC_SR_04_H */

```

Rys. 24 Plik konfiguracyjny hc_sr_04.h

W pliku hc_sr_04.h stworzymy strukturę, która będzie przechowywała informację o używanych timerach oraz zmierzoną odległość. Dodajemy również deklaracje funkcji i typedef-a.

```

13 void hc_sr_04_init(struct us_sensor_str *us_sensor, TIM_HandleTypeDef *htim_echo, TIM_HandleTypeDef *htim_trig, uint32_t trig_channel)
14 {
15     us_sensor->htim_echo = htim_echo;           // Przypisz timer dla echa
16     us_sensor->htim_trig = htim_trig;           // Przypisz timer dla triggera
17     us_sensor->trig_channel = trig_channel;      // Przypisz kanał triggera
18
19     // Rozpocznij działanie kanałów Input Capture oraz PWM
20     HAL_TIM_IC_Start_IT(us_sensor->htim_echo, TIM_CHANNEL_1 | TIM_CHANNEL_2);
21     HAL_TIM_PWM_Start(us_sensor->htim_trig, us_sensor->trig_channel);
22 }

```

Rys. 25 Funkcja inicjalizująca pracę czujnika

Funkcja `hc_sr_04_init` Służy do inicjalizacji czujnika *HC-SR04*. Przyjmuje wskaźnik na strukturę *us_sensor_str* reprezentującą stan czujnika oraz wskaźniki na uchwytach timerów STM32 używanych do obsługi trybów *Echo* i *Trigger*.

1. Przypisanie timerów:

- *htim_echo*: Timer używany do pomiaru czasu trwania echa (Input Capture).
- *htim_trig*: Timer generujący sygnał *Trigger* (PWM).
- *trig_channel*: Kanał timera przypisany do wyjścia *Trigger*.

2. Uruchomienie timerów:

- Aktywuje tryb **Input Capture** dla pomiaru czasu trwania sygnału Echo. Działa na kanałach *TIM_CHANNEL_1* i *TIM_CHANNEL_2*.
- Rozpoczyna generowanie sygnału PWM na kanale odpowiedzialnym za *Trigger*.

```

25 uint32_t hc_sr_04_convert_us_to_cm(uint32_t distance_us)
26 {
27     return (distance_us / HC_SR04_US_TO_CM_CONVERTER);
28 }
29

```

Rys. 25 Funkcja konwersji czasu na odległość

Służy do konwersji zmierzonego czasu (w mikrosekundach) na odległość (w centymetrach). Stała konwersji *HC_SR04_US_TO_CM_CONVERTER* wynosi 58 i odpowiada przelicznikowi: $1\text{ cm} = 58\text{ mikrosekund}$.

Czujnik odczytuje odległość w pętli. Jeżeli jednak zmierzona odległość będzie mniejsza niż 30 cm to czujnik zgłosi przerwanie, a w jego obsłudze mikrokontroler wystawia stan wysoki na pin PC3 (*Led_Warning*), do którego podłączona jest zewnętrzna dioda LED.

```

276 //-----
277 void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim) //callback z czujnika odleglosci gdy dystans bedzie
278 //bedzie mniejszy niz 30 cm
279 {
280     if(TIM1 == htim->Instance)
281     {
282         uint32_t echo_us;
283
284         echo_us = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2);
285         distance_sensor.distance_cm = hc_sr_04_convert_us_to_cm(echo_us);
286
287         if (distance_sensor.distance_cm <= 30)
288         {
289             distance_sensor.led_state = 1; // Zapal diode
290         }
291         else
292         {
293             distance_sensor.led_state = 0; // Zgaś diode
294         }
295     }
296 }
297 //-----

```

Rys. 26 Callback od przerwania zgłoszonego przez czujnik

2.8 Pętla główna:

W pętli głównej STM32 wykorzystywanej do odczytywania wartości z joysticka jest jedynie zaimplementowana funkcja wysyłania danych do ESP ponieważ sam odczyt danych z joysticka odbywa się przez DMA więc nie musimy już nic więcej inicjować

```

joystick_run(&joystick);

Communication communication;
send(&communication);

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart);

while (1)
{
    data_prepare(&communication);

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
}

```

Rys. 27 Pętla główna STM32(0)

2.8.1 Struktura danych zmiennych w pętli main STM32(1):

```

44 typedef struct {
45     struct {
46         uint16_t forward_backward; // Oryginalne dane osi Y (0-1000)
47         uint16_t left_right; // Oryginalne dane osi X (0-1000)
48         int16_t normalized_x; // Normalizowane dane osi X (-500 do 500)
49         int16_t normalized_y; // Normalizowane dane osi Y (-500 do 500)
50         int16_t base_speed;
51         int16_t turn_adjust;
52
53         uint16_t speedA; // Predkość silnika A (0-1000)
54         uint16_t speedB; // Predkość silnika B (0-1000)
55         // uint8_t directionA; // Kierunek silnika A (CW lub CCW)
56         // uint8_t directionB; // Kierunek silnika B (CW lub CCW)
57     } motors;
58 } RobotState;
59

```

Rys. 27 Struktura danych zmiennych w pętli

2.8.2 Algorytm obliczania mocy silników w pętli głównej STM32(1):

```
138 while (1)
139 {
140 //----- Sterowanie silnikami
141     convertwifidata(&wifidataconv);
142 //     buffer_sum();
143 //     check_sum();
144
145     //connectioncheck();
146
147     //Konwersja danych z joysticka (0-1000 dla obu osi)
148     joystick_conv(&wifidataconv);
149
150     robot.motors.forward_backward = wifidataconv.posY; // Qś Y
151     robot.motors.left_right = wifidataconv.posX; // Qś X
152
153     // Normalizacja wartości do zakresu -500 do 500 (środek = 0)
154     robot.motors.normalized_y = robot.motors.forward_backward - 500;
155     robot.motors.normalized_x = robot.motors.left_right - 500;
156
157     // Obliczanie prędkości bazowej (przód/tył) i różnicy dla skretu
158     robot.motors.base_speed = abs( robot.motors.normalized_y ) * 2; // Prędkość w zakresie 0-1000
159     robot.motors.turn_adjust = robot.motors.normalized_x * 2; // Dostosowanie prędkości dla skretu
160
161     // Ograniczenie prędkości w zakresie 0-1000
162     if (robot.motors.base_speed > 1000) robot.motors.base_speed = 1000;
163     if (robot.motors.turn_adjust > 1000) robot.motors.turn_adjust = 1000;
164     if (robot.motors.turn_adjust < -1000) robot.motors.turn_adjust = -1000;
165
166     // Obliczanie prędkości dla każdego silnika
167     robot.motors.speedA = robot.motors.base_speed - robot.motors.turn_adjust; // Silnik A (lewy)
168     robot.motors.speedB = robot.motors.base_speed + robot.motors.turn_adjust; // Silnik B (prawy)
169
170     // Ograniczenie prędkości silników w zakresie 0-1000
171     robot.motors.speedA = drv8835_limit_motor_speed(robot.motors.speedA);
172     robot.motors.speedB = drv8835_limit_motor_speed(robot.motors.speedB);
173
174     // Ustawianie kierunku obrotów silników
175     if (robot.motors.normalized_y <= -50) { // Tył
176         drv8835_set_motorA_direction(CCW);
177         drv8835_set_motorB_direction(CW);
178     } else { // Przód
179         drv8835_set_motorA_direction(CW);
180         drv8835_set_motorB_direction(CCW);
181     }
182
183     // Dodanie tolerancji dla pozycji neutralnej (martwa strefa)
184     if (abs(robot.motors.normalized_x) <= 50 && abs(robot.motors.normalized_y) <= 50) {
185         drv8835_set_motorA_speed(0);
186         drv8835_set_motorB_speed(0);
187
188         //continue; // Przejdź do następnej iteracji
189     }
190     else{
191         drv8835_set_motorA_speed(robot.motors.speedA);
192         drv8835_set_motorB_speed(robot.motors.speedB);
193     }
194
195 //----- Sensor odległości
196 if (distance_sensor.led_state)
197 {
198     // Zapal diode, gdy odległość <= 30 cm
199     HAL_GPIO_WritePin(LED_Warning_GPIO_Port, LED_Warning_Pin, GPIO_PIN_SET);
200 }
201 else
202 {
203     // Zgaś diode, gdy odległość > 30 cm
204     HAL_GPIO_WritePin(LED_Warning_GPIO_Port, LED_Warning_Pin, GPIO_PIN_RESET);
205 }
206 }
```

Rys. 28 Pętla główna

Opis algorytmu:

1. Pobranie danych wejściowych:

- Dane z joysticka (w zakresie 0–1000 dla osi X i Y) są przetwarzane przez funkcję joystick_conv, a następnie zapisane w strukturze wifidataconv.
- Dane z osi Y (posY) reprezentują ruch przód-tył.
- Dane z osi X (posX) reprezentują ruch lewo-prawo.

2. Normalizacja danych:

- Wartości z joysticka są przesuwane, aby środek zakresu znajdował się w punkcie 0:
 - $\text{normalized_y} = \text{posY} - 500$ – określa ruch przód-tył (wartości dodatnie dla przodu, ujemne dla tyłu).
 - $\text{normalized_x} = \text{posX} - 500$ – określa ruch lewo-prawo (wartości dodatnie dla skrętu w prawo, ujemne dla skrętu w lewo).

3. Obliczenie prędkości:

- `base_speed`: prędkość bazowa silników zależy od wartości na osi Y i jest skalowana do zakresu 0–1000.
- `turn_adjust`: korekta prędkości zależy od osi X i jest proporcjonalna do odchylenia joysticka od środka (lewo-prawo).

4. Ograniczenie wartości prędkości:

- Prędkość bazowa oraz korekta są ograniczane do zakresu 0–1000, aby zapobiec przekroczeniu maksymalnych wartości PWM.

5. Obliczenie prędkości dla każdego silnika:

- Prędkości dla lewego i prawego silnika są obliczane w oparciu o:
 - $\text{speedA} = \text{base_speed} - \text{turn_adjust}$ – prawy silnik zwalnia podczas skrętu w prawo.
 - $\text{speedB} = \text{base_speed} + \text{turn_adjust}$ – lewy silnik przyspiesza podczas skrętu w prawo.
- Prędkości są dodatkowo ograniczane przez funkcję `drv8835_limit_motor_speed`.

6. Ustawienie kierunków obrotów:

- Kierunki obrotu silników są ustawiane na podstawie wartości `normalized_y`:
 - Jeśli joystick wskazuje wartość poniżej -50, samochód porusza się do tyłu (CCWA dla lewego i CWB dla prawego silnika).
 - W przeciwnym wypadku samochód porusza się do przodu (CWA dla lewego i CCWB dla prawego silnika).

7. Martwa strefa (neutralna):

- Jeśli joystick znajduje się w zakresie neutralnym (wartości X i Y bliskie środka, tj. ± 50), silniki są zatrzymywane ($\text{speedA} = \text{speedB} = 0$).

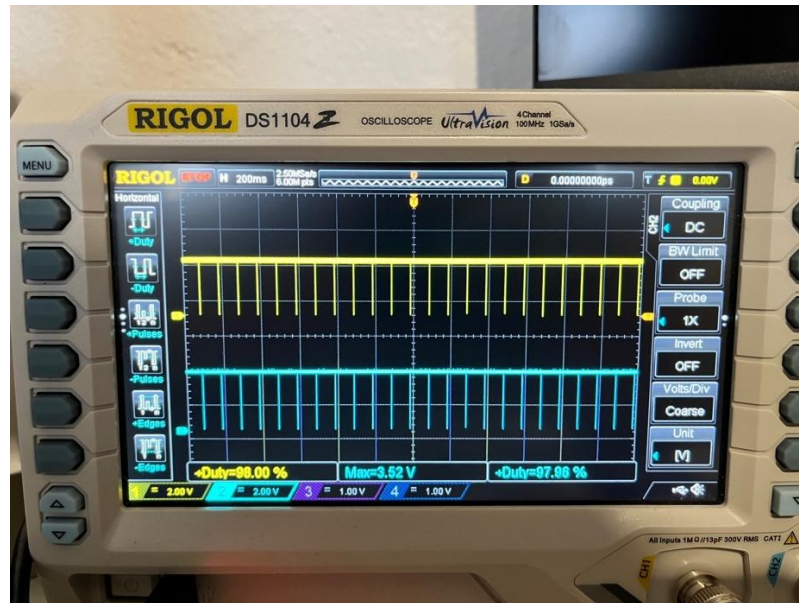
8. Przypisanie prędkości do silników:

- Jeśli joystick wyjdzie poza martwą strefę, prędkości i kierunki są przypisywane do sterownika silników za pomocą funkcji `drv8835_set_motorA_speed` i `drv8835_set_motorB_speed`.

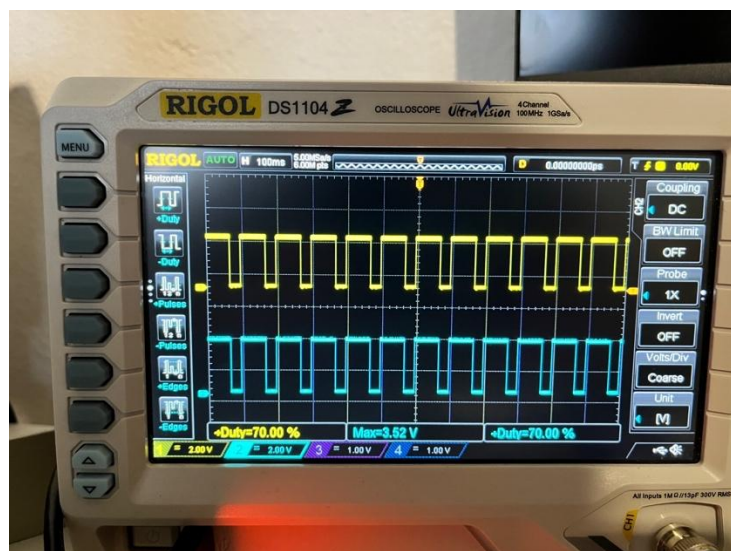
9. Sprawdzanie stanu diody wartości z czujnika:

- Jeśli `led.state` ma wartość 1 to znaczy, że zgłoszono przerwanie, odległość mniejsza od 30cm mikrokontroler zapala diodę led.

2.8.3 Praca układu:

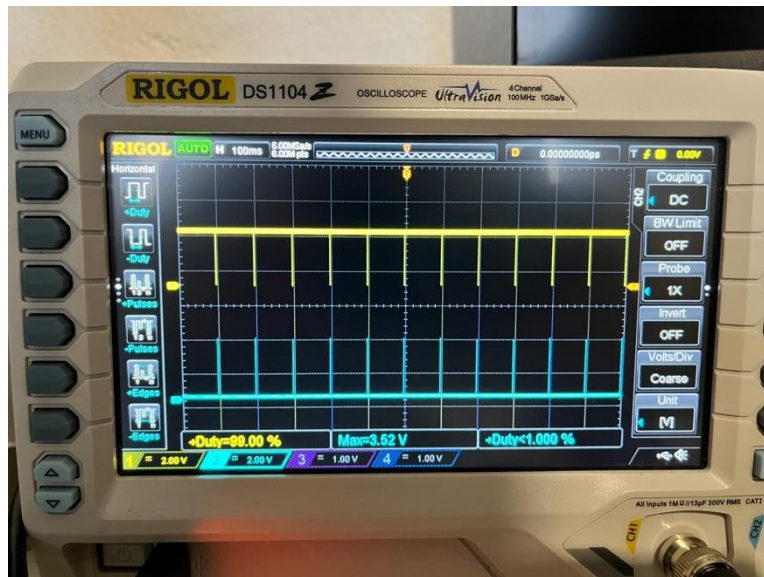


Rys. 29 Maksymalne wychylenie joysticka w przód – silniki pracują z maksymalną mocą (100% wypełnienia)

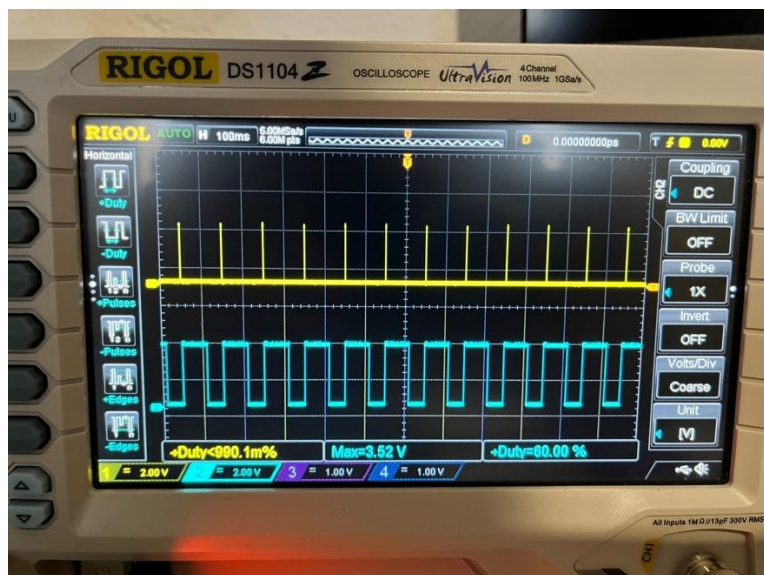


Rys. 30 Lekkie wychylenie joysticka w przód, powyżej strefy martwej

Obydwa silniki pracują z taką samą mocą, jednak wychylenie joysticka nie jest liniowe i bardzo szybko po wyjściu ze strefy martwej, pojazd nabiera prędkości



Rys. 31 Maksymalny skręt w prawo, silnik prawy nie pracuje, natomiast lewy pracuje z maksymalną mocą



Rys. 32 Lekki skręt w lewo, silnik lewy nie pracuje a prawy pracuje z mocą proporcjonalną do wychylenia

3 Napotkane problemy i ich rozwiązania

3.1 Błędne odczyty ADC

Z racji na 12-bitową rozdzielczość przetwornika analogowo-cyfrowego, zakres wartości jakie może odczytać jest w zakresie 0-4096. Joystick w spoczynku jest w połowie rezystancji. Przy szybszym próbkowaniu wartości były bardzo podatne na rozrzut, jak i sam joystick pozostawia wiele do życzenia. Przy delikatnym ruszaniu można stwierdzić, że nie jest do końca liniowy – ma zakres, gdzie zmiany są marginalne. Wyjaśnieniem tego jest fakt, że pochodzi on z zestawu ewaluacyjnego dla klona *Arduino*, zakupionego na

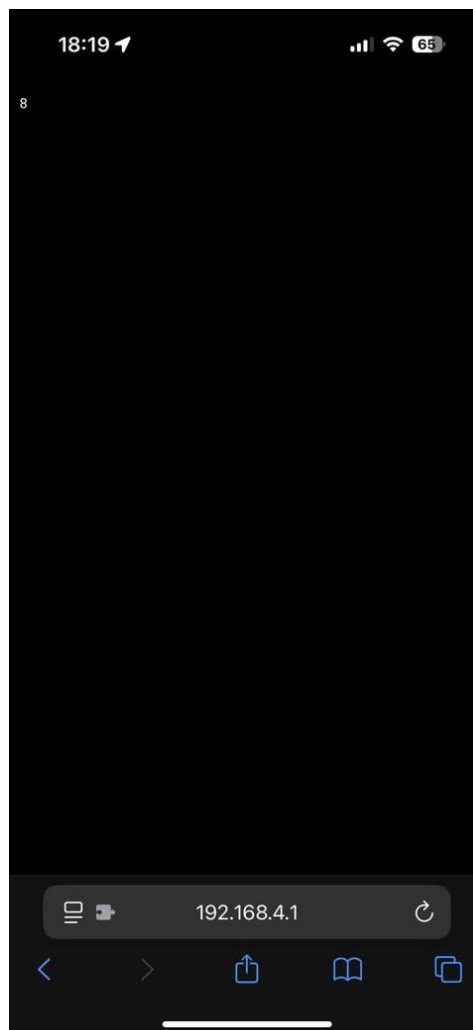
znanej chińskiej platformie internetowej. Natomiast błędne pomiary zostały wyeliminowane zwiększając czas odczytu danych.

3.2 Transmisja danych STM32 – ESP32

Początkowo transmisja miała odbywać się za pomocą *SPI*. Niestety, domyślnie *ESP32* nie wspiera odbioru danych (bycia *slavem*) z tego interfejsu, więc konieczne było użycie zewnętrznych bibliotek. Niestety nie działały. Zatem po długiej walce postanowiliśmy użyć do tego celu *UART*. Z tego poziomu już wszystko działało, *ESP32* prawidłowo odczytywał dane.

3.3 Transmisja danych za pomocą WiFi

Zanim znalazłem protokół *ESP-NOW* dane otrzymywane przez *ESP32* trafiały na serwer http. Mikrokontroler działał jako *access point*, z nazwą *SSID* i hasłem. Po przyłączeniu się do jego sieci można było wejść na stronę z poziomu przeglądarki, i odświeżając stronę dane się aktualizowały – był to serwer http asynchroniczny. Zostały do tego użyte biblioteki *EPAAsyncWebServer* z *githuba*.



Rys 33. Przesyłanie danych na serwer http, z widoczną cyfrą 8 – odebraną przez *UART*.

Na szczęście, istnieje protokół firmy *Espressif ESP-NOW*, umożliwiający bezpośrednią komunikację mikrokontrolerów ESP32 między sobą, za pomocą adresów *MAC*.

W tym momencie pojawił się duży problem – przesuwanie się ramki początku transmisji. Po odebraniu danych przez *ESP8266* co transmisje bajty przesuwały się względem siebie. Musiał więc zostać uwzględniony mechanizm szukania początku ramki i dopiero wtedy składania podzielonych danych.

Gdy zostało to zrealizowane, pojawił się kolejny problem – naszą ramką była wartość *0xFF*. Gdy wychylenie joysticka było maksymalne, dostawaliśmy starszy bajt o wartości właśnie 255. Dane wychodziły błędne. Próbując zmodyfikować ramkę, by była na więcej niż 1 bajcie, dane przestawały być poprawnie odbierane – już zabrakło czasu na debugowanie, więc nie wiemy w którym miejscu był błąd. Protokół *ESP-NOW* pozwala na jednorazowe wysłanie 200 bajtów, więc zwiększenie nie powinno (w teorii) pogorszyć niczego. Natomiast została wykorzystana wcześniej omówiona nieliniowość joysticka. Są pewne wartości rezystancji, których prawdopodobieństwo wystąpienia jest na tyle niskie, że nie przeszkadza to w poprawnej pracy. Zatem nowa wartość ramki to *0xA0*.

3.4 Niedziałanie programu przy zewnętrznym zasilaniu

Po przetestowaniu programu i stwierdzeniu jego poprawności, STM32 została zasilona zewnętrznym. Program nie uruchamiał się. Po przejrzeniu forów internetowych wydawało się, że nie zapisuje programu do pamięci *flash* tylko do *ramu*. W programie *STlink Utility* wszystko wyglądało na poprawnie, dane były zapisane do pamięci *flash*. Jednak program nie działał w dalszym ciągu. Został wygenerowany też blik binarny z programem i był wgrywany z poziomu *STlink Utility*. Dalej nie działało. Powodem okazał się być włączony *semihosting*. Wymaga on działającego *debuggera*, w przeciwnym razie spowoduje to uruchomienie wyjątku, powodującego zatrzymanie pracy *MCU*. Po jego wyłączeniu program działał bez podpiętego przewodu USB, z pinu *E5V*.

3.5 Zaszumione dane z wifi:

Po odbiorze danych przez ESP i przekazanie ich do STM32(1) zauważyliśmy, że co kilka kilkanaście próbek odczyt jest mocno zawyżony. Po z debugowaniu problemu zdecydowaliśmy się na filtrację wyników za pomocą filtra medianowego.

3.6 Silniki nie chciały się kręcić w momencie wychylenia joysticka.

Gdy udało się wysyłać i odbierać dane to zauważyliśmy dziwny problem. Mimo, że wartości obliczonej prędkości silników były prawidłowo przekazywane do rejestrów licznika, generujących PWM (sprawdziliśmy to podglądając wartości na bieżąco w rejestrach oraz printując wyniki na konsoli) to okazało się, że silniki albo się nie kręcą,

albo nie przestają kręcić, gdy joystick nie jest wychylony. W tym momencie wpadliśmy na pomysł, żeby wysterować linię odpowiedzialną za przekazywanie sygnału PWM do drivera w tryb Pull-down, czyli podciągania do masy. Faktycznie przy braku tej opcji nie wiedzieliśmy co się dzieje na linii w momencie, gdy sygnał PWM przestaje być generowany, a dzięki temu rozwiązaniu zapewniliśmy sobie logiczne 0 na linii. Również inne linie wysterowaliśmy w podobny sposób.

3.7 Sygnał PWM nie uruchamiał drugiego silnika

Gdy podłączyliśmy cały układ, na wydrukowanym przez nas samochodzie to zauważyliśmy, że silnik lewy nie chce się kręcić. Wydawało się to dziwne, ponieważ zarówno w programie, jak i podłączając się sondom do pinu na stm32 widzieliśmy generowany sygnał PWM. Multimetrem zbadaliśmy czy nie są uszkodzone kable łączące driver z silnikami. W tym momencie zauważyliśmy, że jeden z polutowanych pinów na driverze nie do końca styka. Gdy go docisnęliśmy to silnik zaczął się kręcić

3.8 Zbyt mały moment obrotowy silników

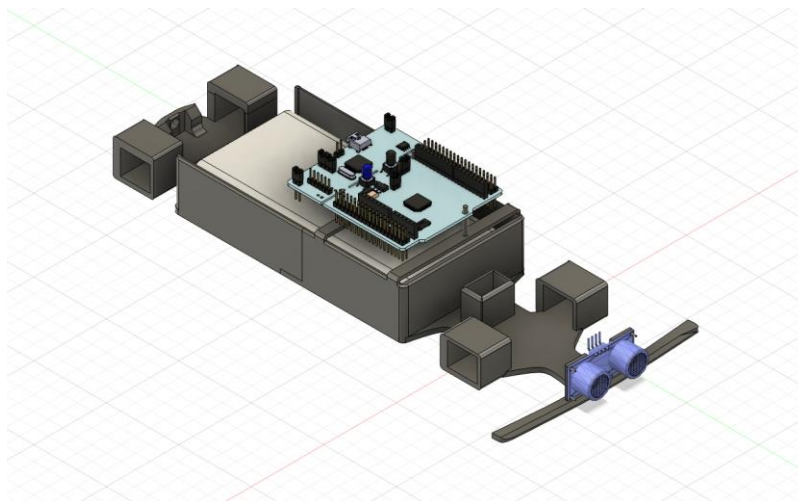
Niestety nasze silniki miały bardzo dużą prędkość obrotową, ale zbyt mały moment obrotowy. Niestety na ten problem nie udało się nic poradzić. W momencie, gdy samochód był trzymany w powietrzu to koła kręciły się prawidłowo. Jednak, gdy kładliśmy go na powierzchni, to siła tarcia statycznego była zbyt duża, a moment obrotowy silników zbyt mały i pojazd nie był w stanie się poruszyć. Rozwiązaniem byłoby wymiana silników na takie z przekładnią pozwalającą zwiększyć moment obrotowy kosztem prędkości obrotowej.

3.9 Tragiczny sygnał zasilający

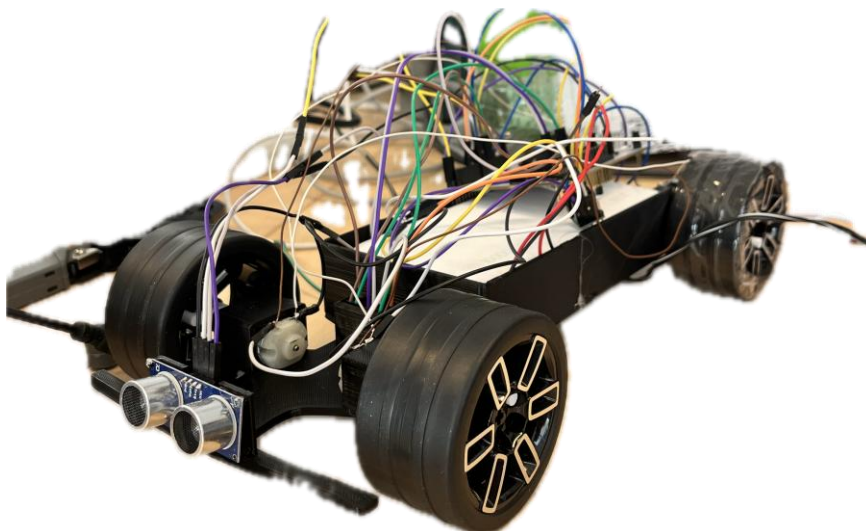
Zasilaniem naszego układu było 6 baterii AAA połączone na 2 koszyczkach zapewniających w sumie 6V napięcia zasilania dla silników i STM32. Jednak napięcie zasilania z baterii było tak bardzo niestabilne, że podglądając wartość na oscyloskopie obserwowaliśmy nawet kilkaset mV tętnień. Powodowało to, że STM32 miała tak niestabilne napięcie, że po prostu się resetowała. Postanowiliśmy więc zasilić STM32 z powerbanka, który miał napięcie dużo stabilniejsze, a sam driver pozostawić na zasilaniu z baterii (nie wymaga on bardzo stabilnego zasilania).

4 Druk 3D:

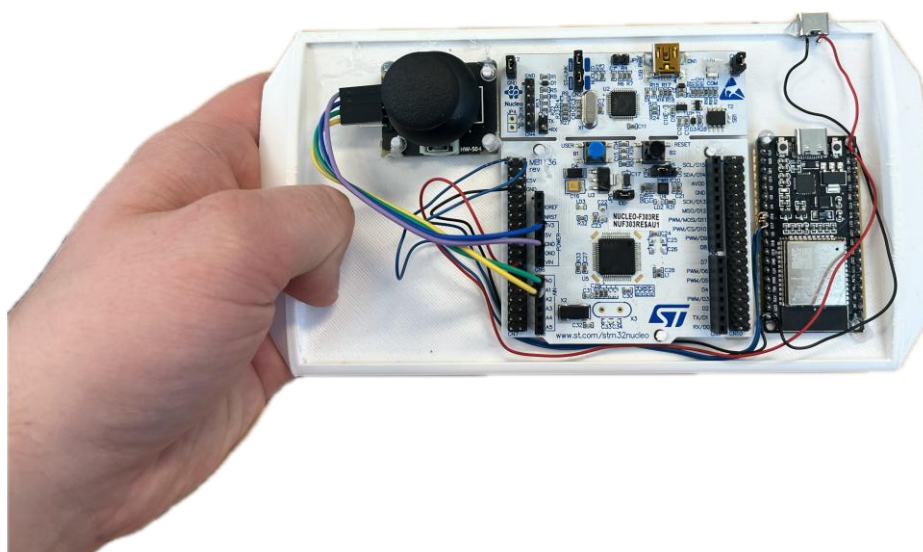
Cały samochód zaprojektowaliśmy w programie Fusion 360 umożliwiającym stworzenie plików .stl umożliwiających ich wydrukowanie.



Rys. 34 Zaprojektowany samochód w programie do grafiki 3D



Rys. 35 Wykonany model samochodu



Rys 36. Zaprojektowany kontroler jazdy