Pojazd sterowany przez WiFi zbudowany przy użyciu STM32L4, Adafruit Motor Shield oraz Magician Chassis

AUTORZY

PIOTR KARDAŚ • TOMASZ ZDYBEL

pkardas.it@gmail.com • tomek.zdybel@gmail.com

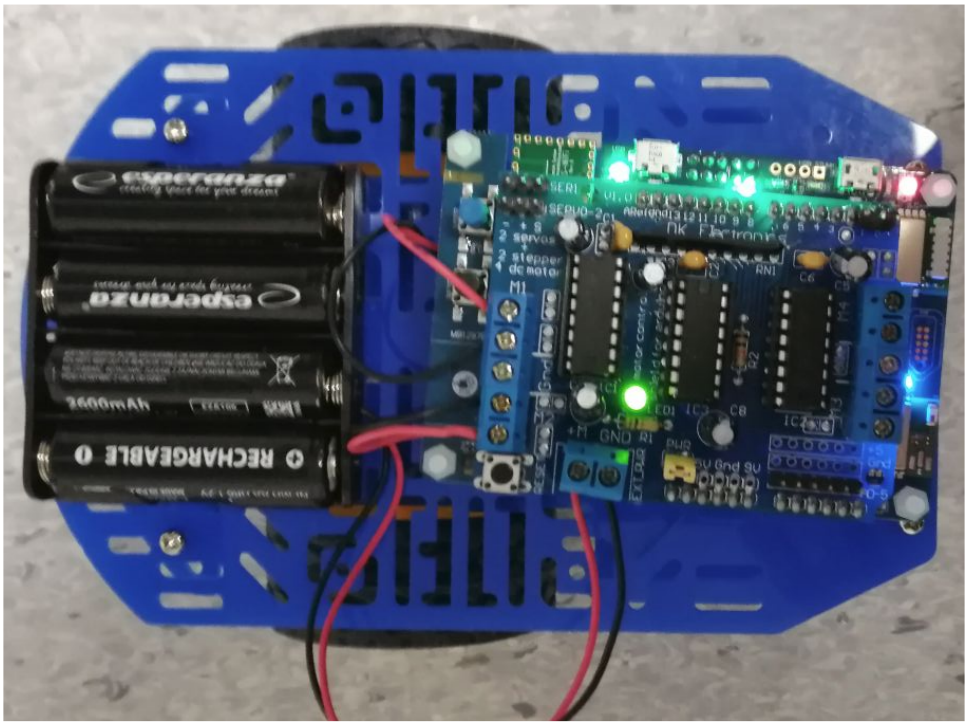
0. Spis treści

1. Wstęp

Celem naszego projektu było stworzenie pojazdu sterowanego zdalnie przez WiFi z poziomu innego urządzenia.

Link do repozytorium: **github.com/logx/smart-vehicle**

Złożone urządzenie prezentuje się następująco:

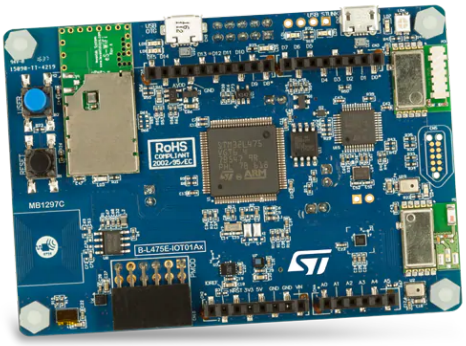


Złożone urządzenie

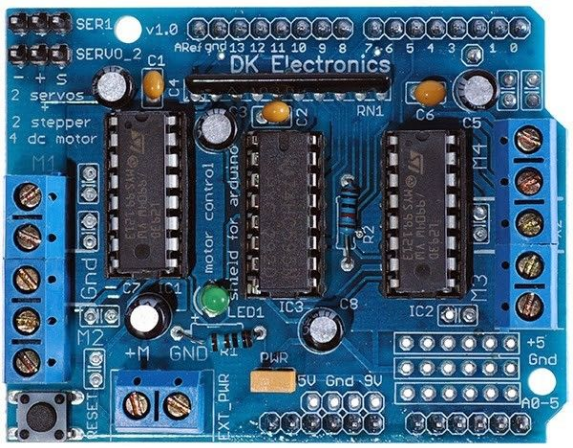
W kolejnych punktach dokumentu zostaną omówione szczegóły techniczne oraz napotkane problemy.

2. Użyty sprzęt

Do zbudowania urządzenia wykorzystaliśmy płytkę **STM32 B-L475E-IOT01A** oraz **Adafruit Motor Shield** w wersji 1.



STM32 B-L475E-IOT01A



Adafruit Motor Shield

Płytka STM32 służyła do udostępniania interfejsu WiFi oraz zarządzania sygnałami PWM podawanymi do płytki Adafruit. W ten sposób przez płytkę Motor Shield uzyskaliśmy kontrolę nad silnikami i mogliśmy sterować prędkością obrotu kół.

2.1 STM32 B-L475E-IOT01A

Na dzień dzisiejszy (styczeń 2019) płytka ta jest jedną z najnowocześniejszych płytek o niskim poborze mocy, jej głównym przeznaczeniem jest *Internet of Things*, co oznacza, że została zaopatrzona w szereg interfejsów bezprzewodowych - NFC, Bluetooth oraz WiFi. Jak się przekonaliśmy i omówimy to w dalszej części niektóre interfejsy nie są jeszcze dokładnie opisane przez twórców przez co użycie niekiedy sprawiało problem ale to tylko dowodzi, że pracowaliśmy na najnowocześniejszym sprzęcie.

Ponadto, płytka jest wyposażona w mikrokontroler z rdzeniem ARM Cortex M4, 1MB pamięci Flash, 128KB SRAM, magnetometr, żyroskop, mikrofon, czujnik wilgotności, termometr czy barometr.

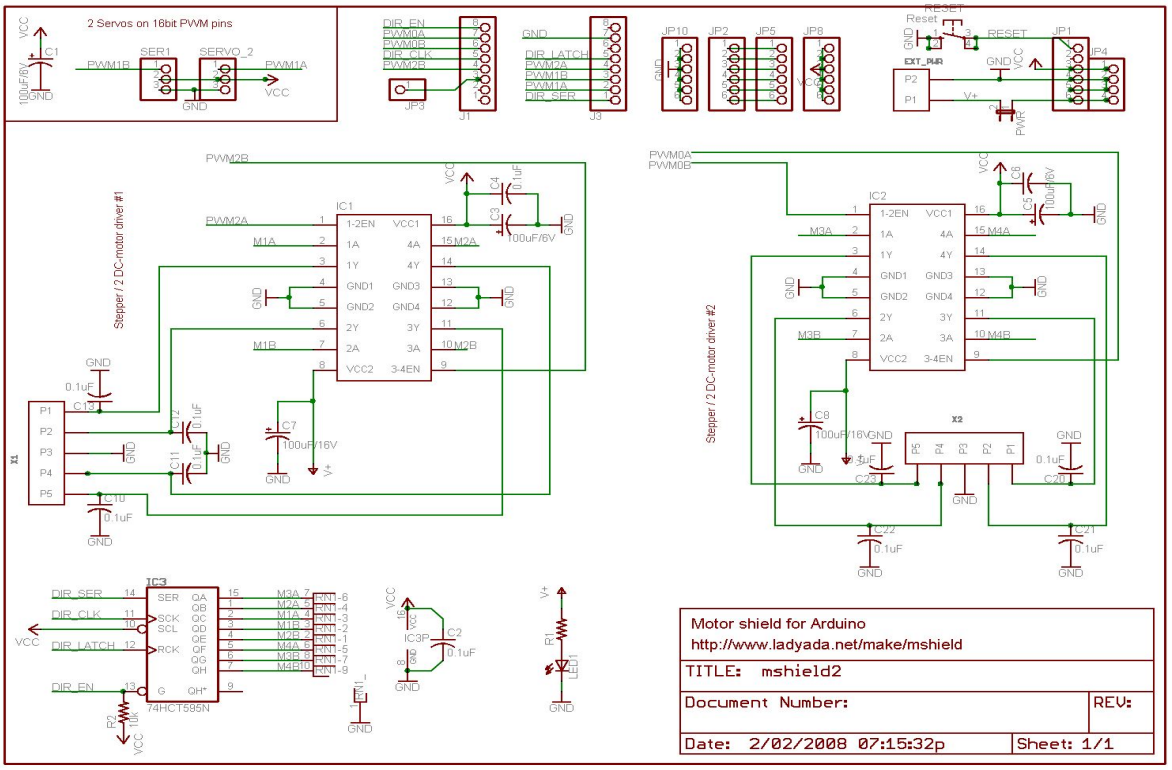
Z racji ograniczonej ilości czasu nie udało nam się wykorzystać pełnego potencjału drzemiącego w układzie.

2.2 Adafruit Motor Shield

Z kolei ta płytka służy do kontroli działania silników elektrycznych. Tutaj kluczowymi elementami były dla nas układy scalone o nazwie 74HCT595N.

74HCT595N

Pozwalają one na kontrolę nad sygnałami PWM. Ich umieszczenie na schemacie płytki wygląda następująco:



Schemat Adafruit Motor Shield

Mamy tutaj dwa egzemplarze (na schemacie oznaczone jako IC1 i IC2) tego układu scalonego, co oznacza, że możemy obsługiwać do 4 silników (po 2 na jeden układ).

3. Oprogramowanie

Implementacja oprogramowania łazika wymagała od nas pisania kodu w **C**. Aplikacja do zdalnej kontroli została napisana w **Pythonie**.

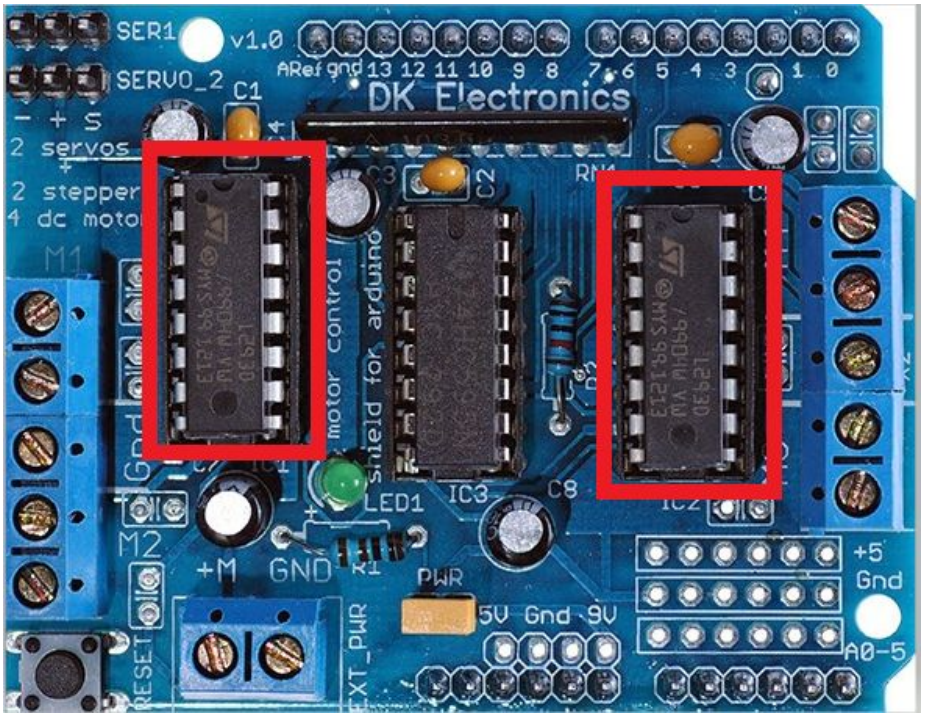
3.0 Wstępne czynności

Zaczęliśmy wykonywanie projektu od uruchomienia STM32CubeMX, wybraniu naszej płytki i ustawieniu podstawowych parametrów. Program wygenerował nam wstępny projekt, który następnie otworzyliśmy z pomocą Atollic TrueStudio.

3.1 Sterowanie silnikami

3.1.1 Omówienie problemu.

Po wygenerowaniu podstawowego kodu źródłowego za pomocą STM32CubeMX, stanęliśmy przed problemem sterowania silnikami.
Problem nie sprowadzał się jedynie do napisania kilku linijek kodu typu "*StartMotor*, *StopMotor"* - musieliśmy sami rozpracować jak przeprowadzić komunikację między naszą płytką IoT Node a Arduino Motor Shield, które piny ustawiać i w jakim celu.
Oprócz tego do poprawnego działania silników potrzebowaliśmy wiedzy na temat modulacji PWM (*Pulse-Width Modulation*).



Zaznaczone kolorem czerwonym elementy powyższej płytki są odpowiedzialne za sterowanie silnikami (a raczej za tworzenie i wysyłanie odpowiednich sygnałów na wyjścia M1, M2, M3 i M4). Odpowiadają one elementom ze schematu oznaczonym jako IC1 i IC2.

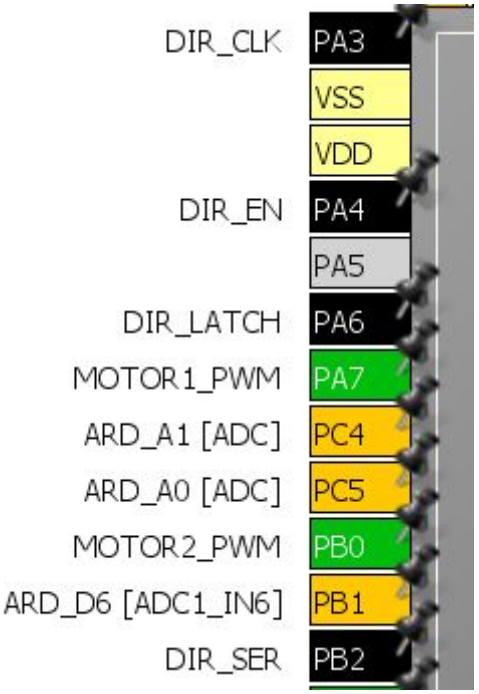
Nimi właśnie (i nie tylko nimi) musieliśmy się nauczyć poprawnie posługiwać.

3.1.2 Rejestr SIPO

W celu odpowiedniego ustawienia wejść M1A, M1B, M2A, M2B itd. zauważyliśmy, że potrzebny będzie do tego rejest SIPO (*Serial in parallel out)* oznaczony na schemacie jako IC3. Jest on odpowiedzialny za wymienione wyżej parametry - nie są one ustawiane bezpośrednio przez piny między płytką IoT a Arduino, lecz przez ten właśnie rejestr.
Natomiast sam rejestr posiada wejścia takie jak: DIR_SER, DIR_LATCH, DIR_EN, DIR_CLK, które z kolei są już ustawiane z pomocą pinów GPIO mikrokontrolera.

3.1.3 Odpowiedniki pinów Arduino na STM32L4

Aby ustawić odpowiednio rejestr SIPO musieliśmy dojść, które z pinów płytki IoT Node są podłączone do interesujących nas pinów Arduino Motor Shielda.
W tym celu, z pomocą schematów dostępnych w internecie, sporządziliśmy rysunek przyporządkowujący do siebie piny tych dwóch płytek, a następnie na jego podstawie ustawiliśmy w STM32CubeMX odpowiednie wartości.



Nasze piny oznaczone kolorem czarnym są pinami z STMa, które odpowiadają tym opisanym obok.

3.1.4 Konfiguracja PWM w STM32CubeMX

Jednocześnie z konfiguracją pinów w celu ustawienia rejestru, szukaliśmy też które z nich mogą działać jako nasze PWMy. Są one wyszczególnione również na powyższym wycinku ekranu jako MOTOR1_PWM i MOTOR2_PWM.

Są to jednakże tylko piny, a do podawania odpowiednich wartości na nie musieliśmy jeszcze odpowiednio skonfigurować zegar ogólnego przeznaczenia i dla niego dwa kanały, odpowiednio jeden dla jednego PWMa.

Oprócz tego trzeba było ustawić jak ma zachowywać się licznik: czy ma zliczać w górę, do ilu, jaki preskaler.
Nasze ustawienia zamieszczamy poniżej:

Counter Settings	
Prescaler (PSC - 16 bits value)	20
Counter Mode	Up
Counter Period (AutoReload Register - 16 bits val...	0x100
Internal Clock Division (CKD)	No Division
auto-reload preload	Disable

Są to ustawienia ogólne do obydwu z PWMów, te które są specyficzne dla każdego, czyli puls startowy (*Pulse*) i kilka innych w które się nie zagłębialiśmy, ustawione są tak samo.

3.1.5 Ustawianie rejestru sterującego

W końcu, po pokonaniu tych powyższych problemów, mogliśmy przystąpić do napisania kodu umożliwiającego ustawianie interesujących nas wartości na odpowiednich polach w rejestrze. W tym celu napisaliśmy następujący fragment:

```
void setIC3(int new_register_state) {
    HAL_GPIO_WritePin(DIR_CLK_GPIO_Port, DIR_CLK_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(DIR_EN_GPIO_Port, DIR_EN_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(DIR_SER_GPIO_Port, DIR_SER_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(DIR_LATCH_GPIO_Port, DIR_LATCH_Pin, GPIO_PIN_RESET);

    shiftOut(new_register_state);
    osDelay(2);
    HAL_GPIO_WritePin(DIR_LATCH_GPIO_Port, DIR_LATCH_Pin, GPIO_PIN_SET);
    osDelay(2);
    HAL_GPIO_WritePin(DIR_LATCH_GPIO_Port, DIR_LATCH_Pin, GPIO_PIN_RESET);
}
```

Powyższy kod odpowiada jedynie za ustawienie pinów na odpowiednie stany startowe, tak jak było to zapisane w dokumentacji rejestru. To ustawienie zapewnia, że wraz z kolejnymi zboczami rosnącymi sygnału zegara DIR_CLK stan spod DIR_SER jest zapisywany na pierwsze miejsce, reszta w rejestrze jest przesuwana o jedno dalej.
Jeśli chodzi o ustawienie GPIO_PIN_SET i RESET w DIR_LATCH, jest to potrzebne w celu "zatrzaśnięcia" wyniku - tak jakby potwierdzenia, że operacja wpisywania została zakończona.

```
void shiftOut(uint8_t value) {
    for (int i = 7; i >= 0; i--) {
        HAL_GPIO_WritePin(DIR_SER_GPIO_Port, DIR_SER_Pin,
(GPIO_PinState)!!(value & (1 << (7 - i))));
        HAL_GPIO_WritePin(DIR_CLK_GPIO_Port, DIR_CLK_Pin, GPIO_PIN_SET);
        HAL_GPIO_WritePin(DIR_CLK_GPIO_Port, DIR_CLK_Pin, GPIO_PIN_RESET);
    }
}
```

Ten kod natomiast jest sercem całej operacji. Wykonuje on 8 iteracji, wpisując kolejne bity wejściowej wartości, od najmłodszego do najstarszego, do DIR_SER, następnie przez wykonanie jednego okresu DIR_CLK wartość ta zapisuje się w rejestrze i przesuwa o jedno miejsce w prawo.
Po wykonaniu całej pętli for w rejestrze mamy wpisaną dokładnie tę samą wartość *value***,** tylko że binarnie.

3.1.6 Wydawanie poleceń silnikom

Teraz wreszcie możemy zająć się czynnościami jakie trzeba wykonać, aby uruchomić nasze silniki. Po pierwsze, trzeba było ustalić jakie parametry podać do rejestru, aby silniki działały według naszego uznania.
Stworzyliśmy funkcję tworzącą odpowiednią wartość, którą następnie podawaliśmy jako parametr do ***setIC3(int new_register_state).***

```
uint8_t createStorageRegisterValue(int t[],  int n) {
   uint8_t value = 0;
   for (int i = 0; i < n; ++i) value = (value | (1 << (7 - t[i])));
   return value;
}
```

Dostaje on na wejściu tablicę n-elementową, w której zapisane są numery od 0 do 7. Określają one, które bity mają zostać ustawione w parametrze value. Funkcja tworzy odpowiednią wartość i zwraca ją.
Przykład jej użycia znajduje się później.

Po drugie, trzeba było ustalić w jaki sposób sterować szybkością pojazdu. Stworzyliśmy w tym celu funkcję ***setSpeed(int speed, int channel).*** Ustawia ona prędkość modyfikując współczynnik wypełnienia dla odpowiedniego sygnału PWM.

```
void setSpeed(int speed, int channel) {
      if (channel == 3) {
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_3, speed);
      } else if (channel == 2) {
            __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2, speed);
      }
}
```

Po trzecie, trzeba było ustalić zasady skręcania: czy w miejscu (wtedy jeden silnik porusza się do przodu, drugi do tyłu) czy może jako jazda do przodu (jeden silnik wolniejszy od drugiego). Zdecydowaliśmy się na drugą opcję.

```
void MotorCommand(int fwd_bck, int left_right, int brake, int
speed) {
    int motors[2];

    if (brake) {
      speed = max(motor1_speed, motor2_speed);
      for(int i = 0x00; i <= speed; i+=10) {
            setSpeed(speed-i, TIM_CHANNEL_MOTOR1);
            setSpeed(speed-i, TIM_CHANNEL_MOTOR2);
            osDelay(10);
      }
      motor1_speed = 0x00;
      motor2_speed = 0x00;
      return;
    }

      switch(fwd_bck) {
         case MTBACKWARD:
            motors[0] = MOTOR1_A;
```

```
            motors[1] = MOTOR2_A;
            setIC3(createStorageRegisterValue(motors, 2));
             break;
        case MTFORWARD:
            motors[0] = MOTOR1_B;
            motors[1] = MOTOR2_B;
            setIC3(createStorageRegisterValue(motors, 2));
             break;
        default:
             break;
    }


    switch(left_right) {
          case MTLEFT:
                setSpeed(speed, TIM_CHANNEL_MOTOR1);
                motor1_speed = speed;
                setSpeed(speed - 0x20, TIM_CHANNEL_MOTOR2);
                motor2_speed = speed - 0x20;
                break;
          case MTRIGHT:
                setSpeed(speed - 0x20, TIM_CHANNEL_MOTOR1);
                motor1_speed = speed - 0x20;
                setSpeed(speed, TIM_CHANNEL_MOTOR2);
                motor2_speed = speed;
                break;
          default:
                setSpeed(speed, TIM_CHANNEL_MOTOR1);
                motor1_speed = speed;
                setSpeed(speed, TIM_CHANNEL_MOTOR2);
                motor2_speed = speed;
                break;
    }
}
```

Powyższa funkcja jest główną funkcją, który wydaje polecenia silnikom. Kolejne parametry odpowiadają: pierwszy - jazda do przodu/tyłu, drugi - w lewo/prawo, trzeci - hamulec, czwarty - prędkość.

Samo działanie jest dość proste:

- jeśli wartość *brake* jest ustawiona - wtedy pojazd hamuje z pomocą prostej pętli spowalniającej, mającej na celu zapobieganie zbyt gwałtownego zatrzymania
- jeśli *brake* jest 0 ustawiamy na podstawie pierwszego z parametrów jak mają kręcić się silniki. W tym celu tworzymy tablicą dwuelementową i ustawiamy odpowiednie wartości. **Ważne jest, żeby użyć tylko po jednym z parametrów A i B jednego silnika. W przypadku kiedy ustawimy obydwie wartości w rejestrze silnik nie będzie w ogóle pracował, lub w przypadku kiedy płytka nie posiada zabezpieczeń dojdzie do spięcia.** W związku z tym ustalamy tylko po jednym parametrze dla każdego z silników: czy ma się kręcić do przodu czy do tyłu. Odpowiednie wartości są zdefiniowane przy pomocy makr w pliku main.h. Ustawiane są na tej podstawie potem parametry w rejestrze. Samo skręcanie czy jazda bez skrętu polega już jedynie na dobraniu parametrów aby jeden silnik chodził wolniej od drugiego.

3.1.7 Stworzone makra

Poniżej zamieszczam listę makr stworzonych przez nas w pliku main.h

```
#define max(a, b) ((a)>(b)) ? (a) : (b)

//A - JEDZIE DO TYŁU, B - DO PRZODU
#define MOTOR1_A 2          //MOTOR 1 - PRAWE KÓŁKO
#define MOTOR1_B 3
#define MOTOR2_A 1          //MOTOR 2 - LEWE KÓŁKO
#define MOTOR2_B 4

#define MTFORWARD 1
#define MTBACKWARD 2
#define MTLEFT 3
#define MTRIGHT 4
#define MAXSPEED 0xFE
#define NORMALSPEED 0xE0

#define MTBRAKE 1

#define TIM_CHANNEL_MOTOR1 3
#define TIM_CHANNEL_MOTOR2 2

#define WIFI_READ_TIMEOUT  10
```

3.2 Dostęp przez WiFi

Po początkowych próbach stworzenia dostępu przez Bluetooth przeszliśmy do rozwiązania opartego na WiFi, powodem takiej decyzji było zbyt duże skomplikowanie interfejsu programistycznego udostępnionego dla technologii Bluetooth.

Początkowo użycie WiFi również stanowiło pewien problem. Nasz pierwotny zamysł zakładał stworzenie Access Pointa na pojeździe i łączenie się z nim z poziomu aplikacji do sterowania:

Jednak dostarczone sterowniki do tworzenia Access Pointa na płytce STM nie zostały opisane na tyle dokładnie aby można było ich w miarę łatwo użyć w naszym projekcie.

Dlatego wykorzystaliśmy inne podejście, zarówno pojazd jak i klient muszą podłączyć się do wcześniej stworzonej sieci:

Proces ustanawiania połączenia polega na zainicjalizowaniu WiFi:

```
uint32_t socket = 0;


if (WIFI_Init() == WIFI_STATUS_OK) {
      printf("WiFi initialized");
} else printf("WiFi not initialized");
```

a następnie otwarciu połączenia pod danym SSID:

```
const char* MYSSID = "vehicle";
const char* PASSWORD = "12345678";

if (WIFI_Connect((__uint8_t *) MYSSID, (__uint8_t *) PASSWORD,
    WIFI_ECN_WPA2_PSK) == WIFI_STATUS_OK) {
      printf("WiFi connection established");
} else printf("Cannot establish WiFi connection");
```

Aby zdalny kontroler mógł sterować urządzeniem konieczne jest otwarcie połączenia do niego po stronie naszego urządzenia:

```
uint8_t ipaddr[4] = {192, 168, 43, 120};
if (WIFI_OpenClientConnection(socket, WIFI_UDP_PROTOCOL, "ala123",
    ipaddr, 2137, 0) == WIFI_STATUS_OK) {
      printf("Opened Client Connection");
} else printf("Client connection cannot be opened");
```

Łączymy się z kontrolerem o IP *192.168.42.120*, jako protokół warstwy 4 wybraliśmy UDP ze względu na niewielki rozmiar przesyłanych danych oraz ze względu na to, że nie potrzebujemy potwierdzenia przesłania komunikatu bo jak się okaże w dalszej części dokumentu przesyłamy wiele (często jednakowych) komunikatów do pojazdu.

Dane od kontrolera odbieramy w nieskończonej pętli wywołując:

```
WIFI_ReceiveData(socket, buff, sizeof(buff), &datalen, WIFI_READ_TIMEOUT)
```

Dane przyjmowane w buforze mają postać ciągu 6 znaków, które mogą mieć wartość ‘0’ (fałsz) lub ‘1’ (prawda) i ułożone są w buforze według poniższej tabeli:

buff[0]	buff[1]	buff[2]	buff[3]	buff[4]	buff[5]
Hamuj	Do przodu	Do tyłu	W lewo	W prawo	Przyspiesz

```
int fwd_bck = MTFORWARD, left_right = 0, brake = 0, speed =
    NORMALSPEED;
if ((char)buff[0] == '1') brake = MTBRAKE;
else {
    if ((char)buff[1] == '1') fwd_bck = MTFORWARD;
        else if ((char)buff[2] == '1') fwd_bck = MTBACKWARD;
        if ((char)buff[3] == '1') left_right = MTLEFT;
        else if ((char)buff[4] == '1') left_right = MTRIGHT;
        if ((char)buff[5] == '1') speed = MAXSPEED;
    }
MotorCommand(fwd_bck, left_right, brake, speed);
```

W zależności od wartości logicznych w poszczególnych komórkach wybierane są flagi: *MTFORWARD*, *MTBACKWARD*, *MTLEFT*, *MTRIGHT*, które następnie podawane są do funkcji **MotorCommand** i w zależności od tego nasz pojazd jeździ tak jak chcemy.

3.3 Zdalna kontrola urządzenia

Do zdalnej kontroli urządzenia wykorzystywane jest narzędzie napisane w języku Python. Sterowanie odbywa się za pomocą strzałek na klawiaturze (kierunek jazdy) oraz Caps Lock’a (przyśpieszanie).

Po połączeniu się i zaakceptowaniu połączenia przez pojazd, program przesyła dane zakodowane według tabeli powyżej.

Kontrola poprawności logiki odbywa się po stronie kontrolera, tj. dba on o to, aby niemożliwe było np. wysłanie komunikatu w którym pojazd ma jechać jednocześnie w lewo i prawo.

Komunikaty wysyła w regularnych odstępach czasu tak aby nie przeciążać pojazdu. Jeśli narzędzie wykryje zmianę kierunku jazdy wysyła komunikat o zmianie natychmiast. Co oznacza, że jeśli trzymamy strzałkę w górę program wysyła komunikaty *010000, 010000, 010000, ...* Jeśli użytkownik zwolni klawisz strzałki, kontroler wykryje zmianę i natychmiast zmieni sekwencję wysyłanych wiadomości na *10000, 10000, 10000, ...*

3.4 Problemy w komunikacji przez WiFi

Niestety komunikacja z urządzeniem nie odbywa się bez problemów, po pewnym czasie pojazd traci połączenie z kontrolerem. Próbowaliśmy rozwiązać ten problem wykorzystując udostępnioną możliwość wysyłania komunikatów PING, jednak nie dawało to oczekiwanych rezultatów - pojazd nadal tracił łączność z kontrolerem.
Spróbowaliśmy wtedy odświeżać połączenie zaraz po tym, jak zostało utracone, lecz ten pomysł także spełzł na niczym. Nie byliśmy w stanie wykryć momentu utraty połączenia, a sam program na płytce zacinał się i nie był w stanie wykonać żadnych kroków w celu odnowy połączenia.

4. Podsumowanie

Stworzenie projektu na płytce STM32 dało nam możliwość poszerzenia swojej wiedzy na temat systemów wbudowanych. Przeszliśmy drogę od zrozumienia modulacji szerokości impulsów (PWM) do odkrycia niedoskonałości w interfejsie WiFi dostarczonym do płytki. Mimo że poświęciliśmy na ten projekt zdecydowanie więcej czasu niż było przewidziane, staliśmy się bogatsi o doświadczenia, które na dalszych etapach nauki, a także pracy, mogą się okazać bardzo przydatne. Jednak nic z tych rzeczy nie może się równać z radością, jaką po wielu godzinach pracy odczuliśmy, kiedy nasz pojazd wreszcie zaczął pracować.