

Telemetria pojazdów mechanicznych
Projekt Zespołowy

Konrad Maciąłek

28 października 2021

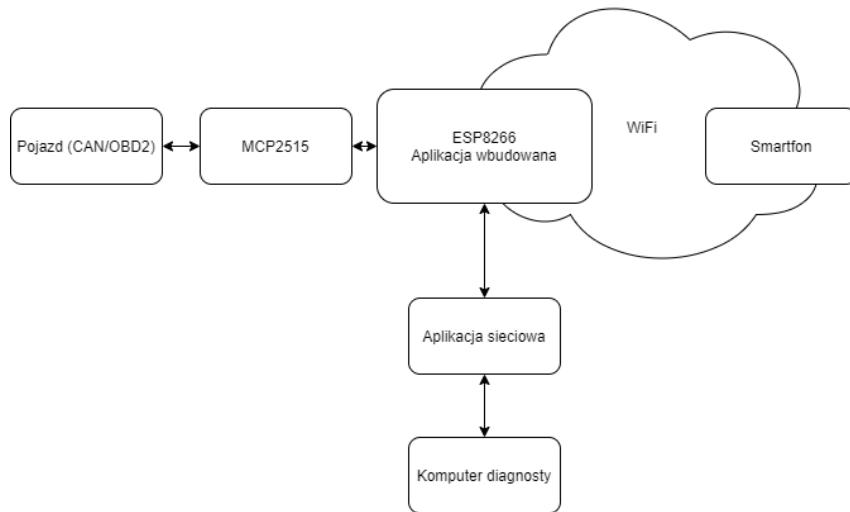
1 Zadanie projektowe

Zaprojektowanie rozwiązania, służącego do zdalnej diagnostyki pojazdów mechanicznych. Powstałe urządzenie wraz z aplikacją powinno umożliwiać analizę danych, odczytanych z podstawowych czujników wbudowanych we współczesne samochody.

Dane powinny być dostępne poprzez stronę www.

2 Opis projektu

2.1 Ogólny opis rozwiązania problemu



Rysunek 1: Diagram ogólny

Finałowy produkt składa się z dwóch głównych części: pierwszej, odczytującej zadane dane oraz drugiej - aplikacji sieciowej, umożliwiającej podgląd i sterowanie danymi.

2.1.1 Odczyt danych (VagCan)

Głównym elementem sterującym jest mikro-kontroler ESP8266. Działająca w nim aplikacja komunikuje się z systemami diagnostyki OBD pojazdu za pomocą protokołu CAN poprzez kontroler MCP2515. Pobrane informacje wysyła następnie poprzez sieć WiFi udostępnioną ze smartfona, do aplikacji sieciowej, gdzie są one zapisywane i udostępniane użytkownikowi końcowemu.

2.1.2 Aplikacja sieciowa (RemoteCarDiagz)

Aplikacja sieciowa składa się z kilku komponentów: serwisu aplikacyjnego typu REST API, publikującego interfejs umożliwiający odczyt i zapis danych oraz aplikacji klienckiej z graficznym interfejsem użytkownika, pozwalającym na podgląd oraz manipulację danymi. Dodatkowo wykorzystano serwisy wspomagające, takie jak baza danych Prometheus, zoptymalizowana do zapisu i odczytu danych zaszergowanych czasowo.

2.2 Wybór technologii

Wszystkie technologie zostały wybrane mając na uwadze ograniczenia licencyjne, tj. narzędzia, biblioteki, środowiska programistyczne są oparte na otwartym kodzie, przy wykorzystaniu licencji GPL lub podobnych. Cały projekt został stworzony pod kontrolą systemu operacyjnego Linux/Ubuntu.

2.2.1 Część odczytująca dane

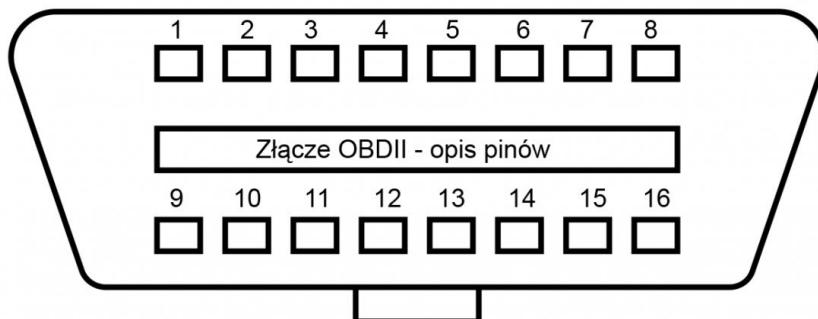
Wybór mikrokontrolera ESP8266 podyktowany został łatwością implementacji obsługi transmisji danych w sieci WiFi. MCP2515 jest natomiast najpopularniejszym kontrolerem służącym do obsługi magistrali CAN. Z uwagi na powyższe, oraz dostępność bibliotek programistycznych, wybór technologii został ograniczony środowiskiem- język C++ pod kontrolą Visual Studio Code.

2.2.2 Aplikacja sieciowa

Z uwagi na wcześniejsze doświadczenie w języku C# oraz chęć poznania nowych technologii, zdecydowano się na wybór .Net5. Wybór ten pozwolił na zaimplementowanie aplikacji klienckiej przy wykorzystaniu nowoczesnego framework'a Blazor WASM.

2.3 Protokół OBD i CAN

On-board diagnostics (OBD) jest terminem opisującym systemy autodiagnostyki pojazdów samochodowych. Po raz pierwszy został wprowadzony w latach 80. XX wieku w Stanach Zjednoczonych, mając na celu umożliwienie szybkiej detekcji nieprawidłowości związanych z emisją zanieczyszczeń do atmosfery. Nowoczesne implementacje dają możliwość wglądu do danych diagnostycznych zbieranych przez elektroniczne moduły kontroli (ECU) zainstalowane w pojazdach mechanicznych, poprzez standaryzowane złącze OBD-II:



Rysunek 2: Złącze OBD-II- opis pinów

Pin	Opis
4	masa
16	+12V z akumulatora
6	CAN High
14	CAN Low

Na powyższym diagramie opisano tylko piny wykorzystane w projekcie, z czego najważniejszymi są 6 i 14. Są to piny umożliwiające komunikację poprzez protokół magistrali CAN (sformułowany w normach ISO 15765-4 i SAE J2284).

CAN to szeregowa magistrala komunikacyjna, mająca głównie zastosowanie w przemyśle samochodowym, powstała w latach 80. XX wieku. Wykorzystuje ona dwuprzewodową skrętkę, co pozwala uzyskać prędkość przesyłu na poziomie 1Mb/s przy dystansie do 40 metrów. Z uwagi na brak jednostki nadzornej, komunikacja ma charakter rozgłoszeniowy, tzn. komunikaty nadawane na magistralę, odbierane są przez wszystkie urządzenia do niej podpięte. W przypadku pojazdu samochodowego, urządzeniami używającymi tej magistrali są ECU, odpowiedzialne np. za działanie silnika lub systemów bezpieczeństwa, takich jak ABS.

Najważniejsze cechy:

- 8 bitów danych w komunikacie

- rozpoznawanie komunikatów poprzez identyfikatory
- automatyczna obsługa dostępu do magistrali
- sprzętowa obsługa błędów

2.3.1 PID

W standardzie OBD-II poszczególne parametry diagnostyczne dostępne są poprzez identyfikatory PID (parameter identification number). Żądanie wysyłane za pomocą linii CAN zawiera identyfikator PID, co pozwala odpowiednim modułom pojazdu na nie odpowiedzieć. Przykładowe parametry podano w tabeli poniżej.

PID	Opis
0x03	Status układu paliwowego
0x04	Obliczone obciążenie silnika
0x05	Temperatura czynnika chłodzącego
0x0	Aktualna prędkość samochodu

Norma SAE J1979 definiuje również 10 serwisów, grupujących dane. W projekcie wykorzystano tylko serwis 01, podający aktualne dane diagnostyczne. Przykładowo, serwis 02 podaje te same dane, ale zebrane podczas ostatniego wystąpienia błędu diagnostycznego.

2.3.2 Format komunikacji

Czytnik diagnostyczny rozpoczyna komunikację poprzez wysłanie żądania pod adres 0x7DFh, który pełni funkcję adresu rozgłoszeniowego. ECU, które potrafią odpowiedzieć na żądania OBD nasłuchują zarówno pod adresem 0x7DFh, jak i pod jednym ze specyficznych adresów w zakresie 0x7E0h do 0x7E7h. Moduły obsługujące żądanie podają w odpowiedzi adres swojego ID, zwiększone o 8, tj. od 0x7E8h do 0x7EFh.

Żądanie W poniższej tabeli przedstawiono przykładowy format żądania, wysyłanego przez urządzenie diagnostyczne.

Bajt							
0	1	2	3	4	5	6	7
Liczba dodatkowych bajtów danych: 0x02	ID Serwisu: 0x01	PID (np. 0x05)	nieużywane, 0xCC				

Tablica 1: Format żądania

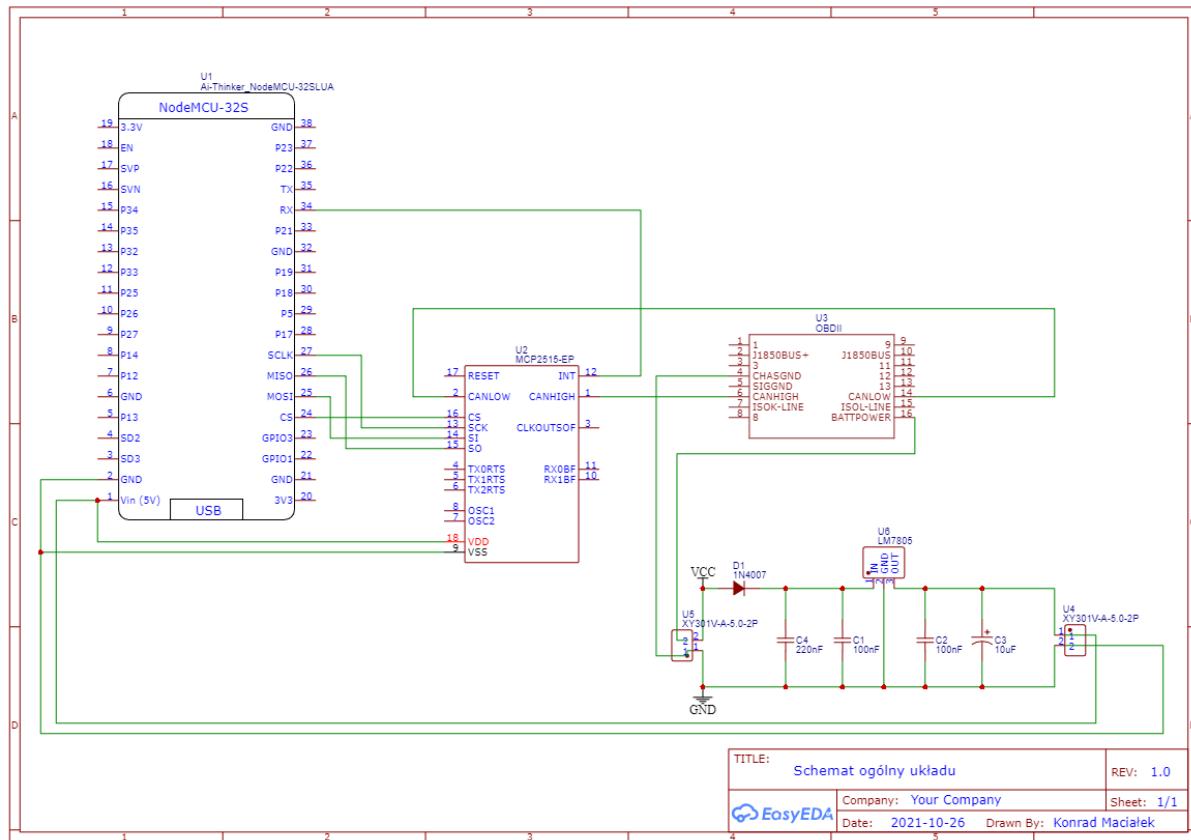
Odpowiedź Pojazd odpowiada na żądanie na magistrali CAN z identyfikatorem wiadomości zależnym od tego, który ECU to żądanie przetworzyło. W projekcie wykorzystano tylko wiadomości przetworzone przez główny moduł o identyfikatorze odpowiedzi 0x7E8h.

Bajt							
0	1	2	3	4	5	6	7
Liczba dodatkowych bajtów danych: od 0x3 do 0x6	ID serwisu zwiększone o 40 0x41	PID (np. 0x05)	wartość A	wartość B	wartość C	wartość D	nieużywane 0x55

Tablica 2: Format odpowiedzi

3 Implementacja sprzętowa

3.1 Schemat ogólny



Rysunek 3: Ogólny schemat układu

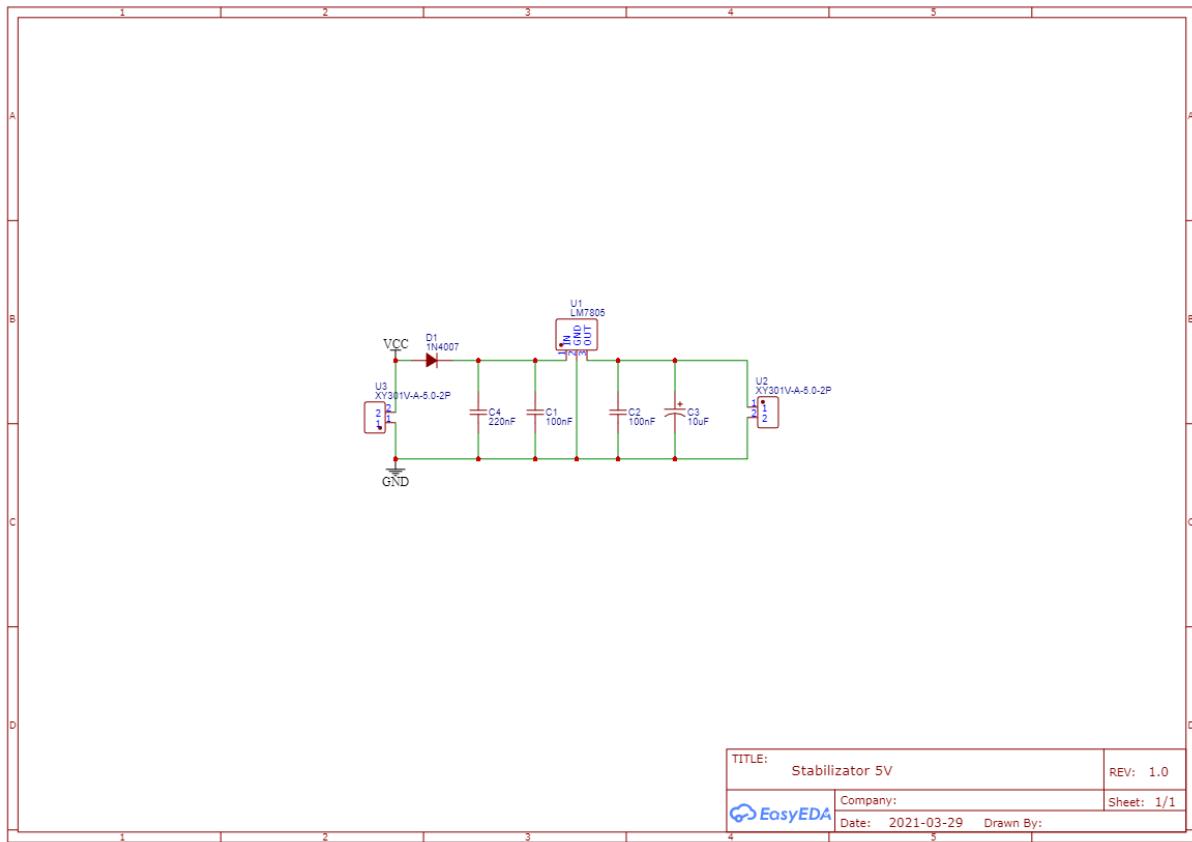
Na diagramie pokazany jest schemat ogólny układu. Głównym elementem jest mikrokontroler ESP8266, w postaci gotowego modułu NodeMCU. Moduł ten za pomocą interfejsu SPI komunikuje się z kontrolerem CAN opartym na MCP2515, który z kolei poprzez niewidoczny na schemacie interfejs pośredniczący TJA1050 podłączony jest do magistrali CAN przez gniazdo OBD-II. Zasilanie całego układu realizowane jest przez stabilizator oparty na układzie LM7805, przekształcający napięcie +12V akumulatora na +5V akceptowalne przez obydwa podłączone równolegle moduły NodeMCU i MCP2515. Warto zauważyć, że napięcie pobierane jest z tego samego gniazda, na którym zrealizowana jest komunikacja.

3.2 Stabilizator napięcia

Występujące między pinami 16 i 4 gniazda OBD-II napięcie +12V pochodzi z akumulatora diagnostycznego samochodu. Z uwagi na to, że zarówno NodeMCU, jak i MCP251 nie tolerują tak wysokich wartości, zaistniała potrzeba wykonania stabilizatora napięcia. Zdecydowano się na implementację opartą o układ LM7805, z uwagi na prostotę, dostępność oraz cenę. Projekt oraz produkcja płyt drukowanych została wykonana w całości sposobem własnym. Do pracy wykorzystano środowisko EasyEDA, umożliwiające kompleksowe wspomaganie projektowania, od etapu schematu, do wydruku ścieżek transferowalnych na płytce drukowanej.

3.2.1 Schemat

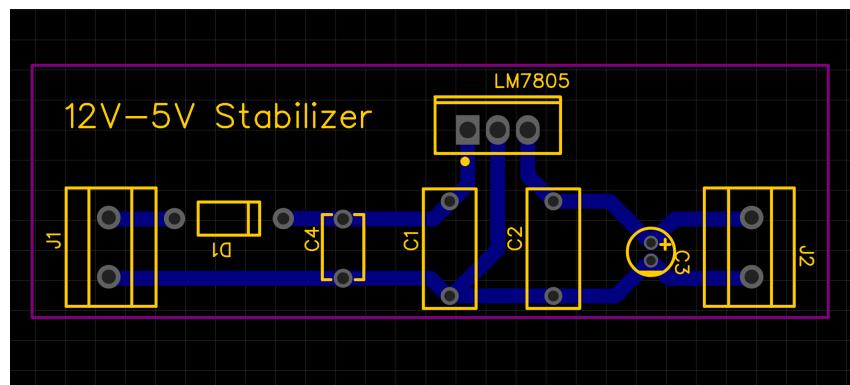
Schemat, na którym oparto stabilizator opisany jest w nocy katalogowej LM7805. Uzupełniono go natomiast o złącza śrubowe celem łatwiejszego podłączania kablowania. Dioda 1N4007 zabezpiecza przed odwrotnym podłączeniem przewodów.



Rysunek 4: Schemat stabilizatora napięcia

3.2.2 Widok PCB

Zastosowanie montażu przewlekanego pozwoliło w łatwy sposób wykonać płytę w warunkach domowych.



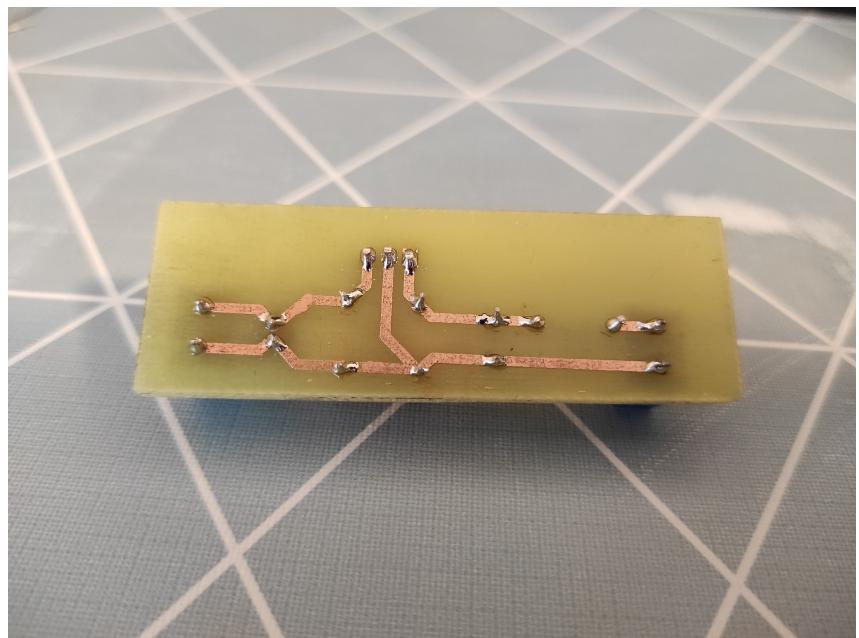
Rysunek 5: Widok PCB stabilizatora w programie EasyEDA

3.2.3 Produkcja PCB

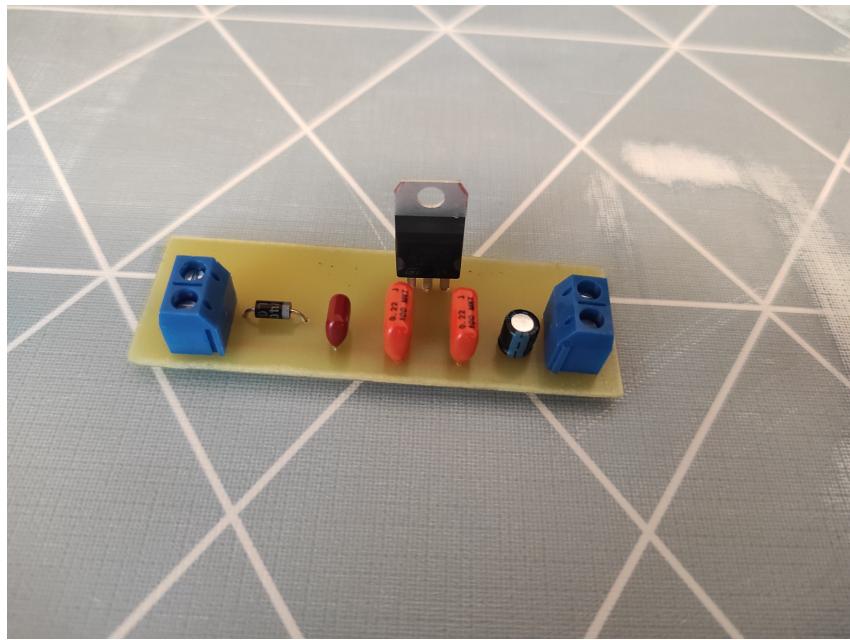
Ścieżki, wydrukowane laserowo, przeniesiono na miedź metodą transferu acetonowego. Podczas wielokrotnych prób, najlepszym nośnikiem okazały się materiały marketingowe sklepów wielkopowierzchniowych, tzw. gazetki. Wytrawianie przebiegło pod kontrolą czynnika B-327 (nadsiarczanu sodowego). Gotowa płyta pokryta została izopropylowym roztworem kalafonii celem zapobiegania utlenianiu miedzi.



Rysunek 6: Wytrawianie płytka stabilizatora



Rysunek 7: Wytrawione ścieżki stabilizatora



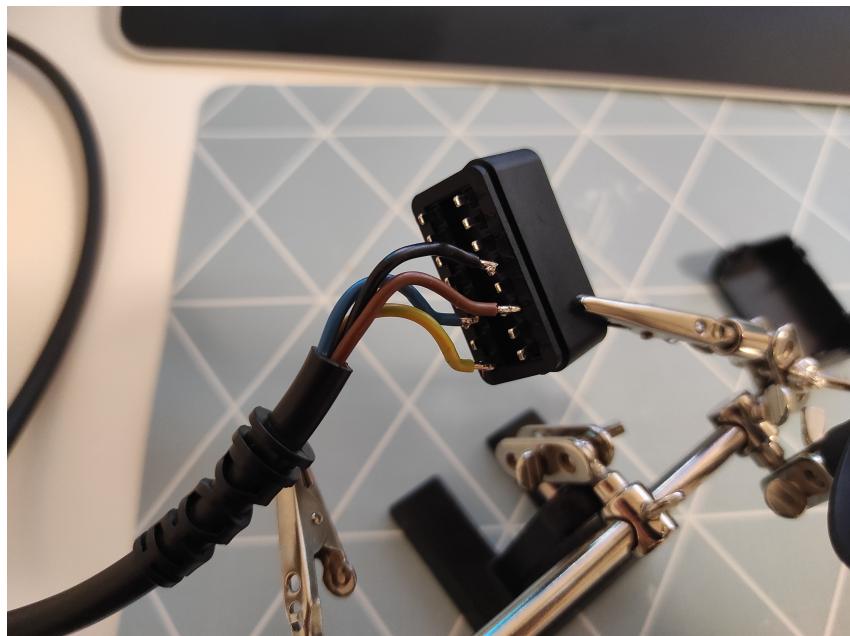
Rysunek 8: Zmontowany układ stabilizatora napięcia

3.2.4 Sprawdzenie poprawności działania

Testy układu wykonane multimetrem uniwersalnym pokazały, że po podłączeniu źródła napięcia w postaci akumulatora samochodowego, zarówno z obciążeniem modułami jak i bez niego, układ stabilizatora utrzymuje stałe napięcie bliskie +5V. Układ zatem zadziałał prawidłowo i mógł zostać wykorzystany w projekcie.

3.3 Kabel OBD-II

Do działania układu niezbędne było wykonanie kabla pozwalającego dostarczyć napięcie oraz sygnały CANH i CANL z gniazda OBD. Kabel wykonano z czterożyłowego przewodu, zakończonego wtykiem męskim OBD. Na zdjęciu widoczne przewody:



Rysunek 9: Kabel z wtyczką OBD-II

Kolor	Pin	Opis
czarny	4	masa
żółty	16	+12V z akumulatora
brązowy	6	CAN High
niebieski	14	CAN Low

Tablica 3: Kolory przewodów i znaczenie

4 Implementacja programowa

Programowo, rozwiązanie możemy podzielić na dwie grupy:

- aplikacja działająca po stronie sprzętowej, w samochodzie - VagCan
- zestaw aplikacji działających po stronie sieciowej i użytkownika - RemoteCarDiagz

4.1 VagCan

VagCan napisany jest w języku C++ i działa na platformie NodeMCU.

Do jego zadań należy:

- wysyłanie żądań PID na magistralę CAN
- odczyt odpowiedzi z magistrali
- wysłanie danych pomiarowych do aplikacji serwerowej z pakietu RemoteCarDiagz
- pobieranie aktywnych PID do przeprowadzenia pomiarów z aplikacji serwerowej RemoteCarDiagz
- obsługa połączenia WiFi

4.1.1 Wykorzystane biblioteki

Aplikacja korzysta z kilku bibliotek:

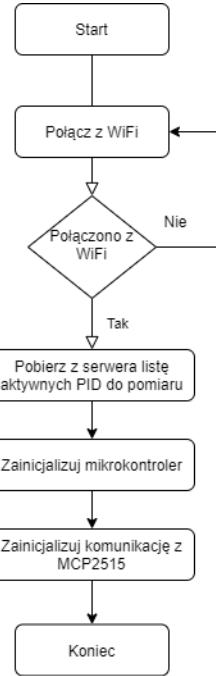
- MCPCAN.h autorstwa Cory J. Fowler - do obsługi protokołu CAN przez MCP2515
- ESP8266WiFi.h - obsługa WiFi
- ArduinoJson.h - obsługa serializacji i deserializacji danych

4.1.2 Algorytm działania

Cały program zawarty jest w pliku main.cpp. Z uwagi na prototypowy charakter, a także na niewielki stopień skomplikowania, zaniechano wydzielania poszczególnych metod od osobnych plików. Podobnie jak każdy program na platformie kontrolera NodeMCU, także ten zawiera dwie główne funkcje:

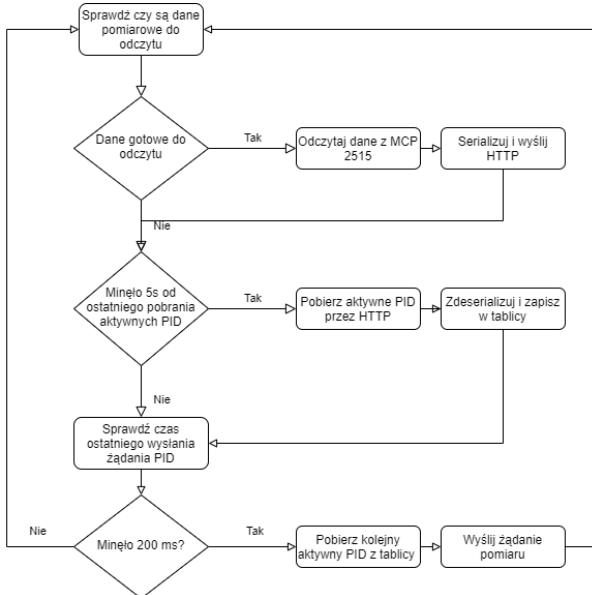
- setup() - uruchamiana tylko raz po resetie mikrokontrolera,
- loop() - działająca nieprzerwanie, aż do momentu wyłączenia lub resetu, główna pętla programu

Przebieg działania metody setup() ilustruje poniższy diagram:



Rysunek 10: Algorytm działania metody setup()

W metodzie loop() działa cała główna logika programu. W ogólności, co 5 sekund wysyłany jest request do aplikacji serwerowej, pobierający aktywne PID, które będą wysyłane w żądaniach diagnostycznych. Odpowiedź z serwera jest deserializowana i zapisywana w tablicy aktywnych PID. Co 200 ms wysyłane jest żądanie pomiaru kolejnego aktywnego PID na magistralę CAN. W każdym przebiegu pętli programu sprawdzany jest stan portu GPIO2 mikrokontrolera ESP8266. Stan niski na tymże, oznacza, że w rejestrach MCP2515 są gotowe do odczytu dane diagnostyczne. W takim wypadku, dane zostają odczytane przez ESP8266, zserializowane do postaci JSON i wyslane do aplikacji serwerowej.



Rysunek 11: Algorytm działania metody loop()

Warto od razu zauważyć wadę takiego rozwiązania. MCP2515 dysponuje dwoma buforami odczytu, działającymi równolegle. Zatem można przekształcić tak działanie programu, aby żądać, odczytywać i wysyłać do serwera dane pomiarowe parami, zamiast pojedynczo. Pozwoliłoby to na pewno bardziej efektywnie wykorzystać czas działania programu, a także zmniejszyć narzut opóźnienia związany ze zbyt

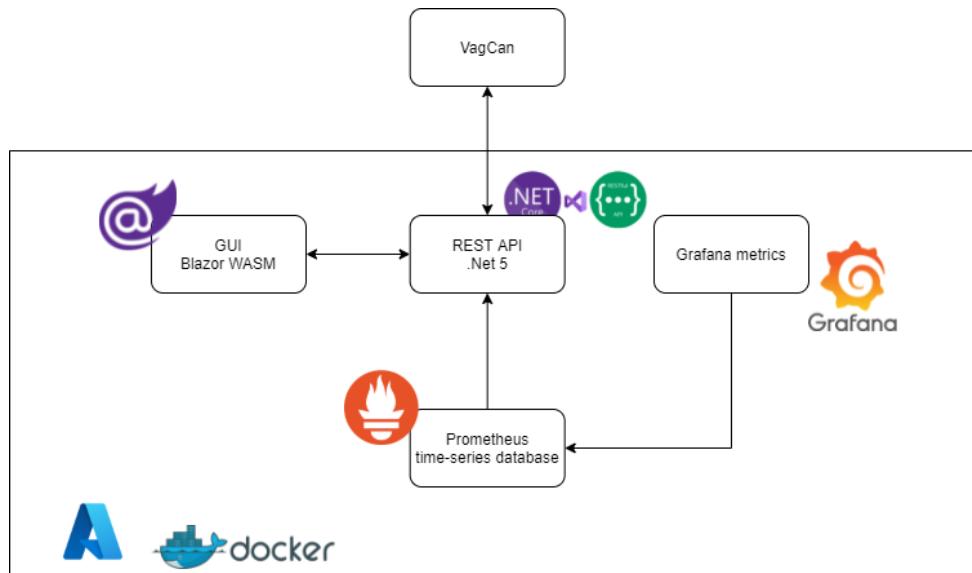
częstym wysyłaniem danych przez protokół HTTP.

Kolejną rzucającą się w oczy wadą, jest wpisany w kod programu adres serwera, a także hasło i SSID sieci WiFi. W prototypowym rozwiążaniu jest to dopuszczalne, w przypadku wersji produkcyjnej należy ten problem rozwiązać inaczej- dobrym pomysłem wydaje się dodanie obsługi komunikacji Bluetooth i przesyłanie tych danych w momencie konfiguracji aplikacji poprzez sparowany z mikrokontrolerem smartfon. Ciekawym rozwinięciem projektu byłoby dodanie wyświetlacza, lub zestawu LEDów informujących o statusie urządzenia.

4.2 RemoteCarDiagz

Rozwiązanie działające po stronie sieciowej, składa się z kilku serwisów:

- Server - REST API przyjmujące, przetwarzające i zapisujące dane z VagCan oraz GUI; udostępnia dane dla bazy danych Prometheus
- Client - GUI - umożliwia użytkownikowi ustawienie aktywnych PID, komunikuje się z Server
- aplikacje pomocnicze - baza danych pomiarowych Prometheus oraz Grafana odpowiedzialna za ich wizualizację



Rysunek 12: RemoteCarDiagz - schemat ogólny

Całość napisana została w języku C# dla .Net 5 i umieszczona w pojedynczej solucji.

4.2.1 Server (REST API)

Standardowe WEB API, upubliczniające dwa endpointy: /configuration i /measurements a także specjalny, wewnętrzny endpoint /metrics służący do pobierania danych pomiarowych przez aplikację Prometheus.

/configuration korzysta z niego graficzny interfejs użytkownika (GUI) oraz VagCan. Metoda GET pozwala na pobranie aktualnie aktywnych PID do pomiaru (lub wyświetlenia na GUI), podczas gdy POST zapisuje aktywne PID do bazy danych SQLite.

/measurements używany jest jedynie przez VagCan - metodą POST przesyłane są wartości pomiarowe, które po obliczeniach trafiają do bazy danych Prometheus

4.3 Server - przykładowe obliczenia

Poniżej pokazano przetwarzanie danych dla PID 0x05, czyli temperatury płynu chłodzącego. Dane do serwera trafiają w postaci payloadu JSON:

```
{  
    "PIDCode": 5,  
    "A": 86,  
    "B": 0,  
    "C": 0,  
    "D": 0  
}
```

PIDCode jest identyfikatorem PID, natomiast A, B, C, D to wartości poszczególnych bajtów odczytanych przez VagCan z odpowiedzi na żądanego pomiaru PID. Przetwarzanie takiego requestu HTTP polega na przejściu przez łańcuch kolejnych handlerów, z których każdy definiuje sposób wyliczenia odpowiedniej wartości pomiarowej dla danego PID. W przypadku PID 0x05 jest to $A - 40$, zatem zmierzona temperatura płynu chłodzącego wynosi 46 stopni Celsjusza. Tak obliczona wartość jest zapisywana do bazy Prometheus, skąd następnie może zostać zwizualizowana przez Grafanę, w postaci wykresu czasowego.

4.3.1 Client - GUI

Aplikacja kliencka napisana została z wykorzystaniem frameworka Blazor WASM. Jest to dość szczątkowy interfejs użytkownika, pokazujący jednak możliwość zdalnego sterowania urządzeniem zamontowanym w samochodzie przez stronę www. Dodatkowo, przy wykorzystaniu pakietu Grafana, użytkownik ma możliwość rozbudowanej wizualizacji i analizy danych pomiarowych otrzymanych z pojazdu.



Rysunek 13: Zrzut ekranu aplikacji klienckiej - konfiguracja aktywnych pomiarów

Na ekranie widoczna jest lista PID, wspieranych przez aplikację. Spośród wszystkich dostępnych identyfikatorów, zaimplementowano tylko kilka najważniejszych:

Nazwa	Kod	Opis	Jednostka
PID_ENGINE_LOAD	0x04	Obciążenie silnika	%
PID_COOLANT_TEMP	0x05	Temperatura czynnika chłodzącego	°C
PID_FUEL_PRESSURE	0x0A	Ciśnienie paliwa	kPa
PID_INTAKE_MAP	0x0A	Ciśnienie bezwzględne w kolektorze dolotowym	kPa
PID_ENGINE_RPM	0x0B	Obroty silnika	RPM
PID_VEHICLE_SPEED	0x0D	Prędkość samochodu	km/h
PID_INTAKE_TEMP	0x0F	Temperatura powietrza w kolektorze dolotowym	°C
PID_MAF_FLOW	0x10	Przepływ w czujniku masowego przepływu powietrza	grams/sec
PID_THROTTLE	0x10	Stopień otwarcia przepustnicy	%

Tablica 4: Wybrane PID, zaimplementowane w aplikacji

Interfejs użytkownika pozwala na zmianę stanu danego identyfikatora. Stan aktywny oznacza, że identyfikator będzie cyklicznie wysyłany na magistralę CAN celem zebrania danych pomiarowych wielkości

odpowiadających identyfikatorowi. W stanie nieaktywnym, nie będą zbierane dane pomiarowe tego identyfikatora.

4.3.2 Prometheus

4.3.3 Grafana

5 Praca zespołowa

6 Podsumowanie