

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI, FOTONIKI I
MIKROSYSTEMÓW

KIERUNEK: Automatyka i Robotyka (AIR)
SPECJALNOŚĆ: Robotyka (ARR)

**PRACA DYPLOMOWA
INŻYNIERSKA**

Projekt systemu sensorycznego bazującego na
protokole MQTT

Design a sensor system based on MQTT protocol

AUTOR:
Marcin Bober

PROWADZĄCY PRACĘ:
Dr inż., Mateusz Cholewiński, K29W12ND02

Spis treści

1 Wprowadzenie	2
1.1 Cel pracy	2
2 Schemat działania projektu	3
2.1 Założenia	3
2.2 Architektura systemowa	3
2.2.1 Aplikacja dostępową	4
2.2.2 Urządzenie wykonawcze	4
2.2.3 Serwer	4
2.3 Wymiana informacji	4
2.3.1 Wykorzystywane tematy	5
3 Szczegółowy opis brokera MQTT	7
3.1 Protokół	7
3.2 Broker	7
3.3 Typy wiadomości	7
3.4 Instalacja i konfiguracja	9
3.4.1 Tworzenie kontenera	9
3.4.2 Konfiguracja	9
3.4.3 Dodawanie użytkowników	10
4 Szczegółowy opis sterownika	11
4.1 Mikrokontroler	11
4.2 System operacyjny	11
4.2.1 Szeregowanie zadań	13
4.3 Framework	13
4.4 Kolejki i wątki	14
4.4.1 Opis	14
4.4.2 Implementacja menadżera wątków	15
4.4.3 Implementacja menadżera kolejek	17
4.5 Mostek H	19
4.6 Silnik	20
4.7 Płytką PCB	20
4.7.1 Opis	20
4.7.2 Wygląd	21
4.7.3 Schemat	21
4.8 Licznik impulsów	26
4.8.1 Opis	26
4.8.2 Konfiguracja	27

4.9	Modulacja szerokości impulsu - PWM	29
4.9.1	Opis	29
4.9.2	Implementacja	29
4.10	Regulator PID	32
4.10.1	Implementacja	32
4.10.2	Proces PID	33
4.11	Pomiar napięcia	37
4.11.1	Konfiguracja ADC	37
4.11.2	Dzielnik napięcia	38
4.11.3	Wyznaczanie współczynnika konwersji	39
4.11.4	Rozdzielcość pomiaru	40
4.11.5	Wykonywanie pomiarów	40
4.12	Łączenie z brokerem MQTT	42
4.12.1	Konfiguracja	42
4.12.2	Obsługa zdarzeń	42
4.12.3	Wysyłanie danych	45
4.13	Gotowe urządzenie	46
4.13.1	Symulacja	46
4.13.2	Prototyp	46
5	Szczegółowy opis aplikacji dostępowej	48
5.1	Grupa docelowa	48
5.2	Wykorzystana technologia	48
5.3	System sygnałów i slotów	49
5.4	Implementacja komunikacji	51
5.4.1	Łączenie z brokerem	51
5.4.2	Obsługa zdarzeń	52
5.4.3	Subskrybowanie tematów	52
5.4.4	Wysyłanie danych	53
5.5	Wykresy	55
5.5.1	Definicja klasy	55
5.5.2	Dodawanie punktów do wykresu	55
5.5.3	Czyszczenie wykresu	56
5.5.4	Ukrywanie serii	56
5.6	Abstrakcja silnika	58
5.6.1	Definicja klasy silnika	58
5.6.2	Przeliczanie wartości obrotów	58
5.7	Interfejs użytkownika	60
5.7.1	Górná belka	60
5.7.2	Panel parametrów	60
5.7.3	Graf	61
6	Podsumowane	62
Literatura		63
Spis listingów		66

Rozdział 1

Wprowadzenie

Rozwój technologii powoduje zmiany w każdej dziedzinie życia. Dotyka to nie tylko nasze codzienne otoczenie, ale i przede wszystkim gałęzie przemysłu. To właśnie między innymi potrzeby przemysłu napędzają innowacje poprzez wciąż rosnące zapotrzebowanie na nowe, lepsze i wydajniejsze rozwiązania.

Dzisiejsza technologia pozwala na bardzo precyzyjne sterowanie takim silnikami prądu stałego, a same sterowniki nie zajmują już połowy pokoju. Wręcz przeciwnie - technika analogowa coraz to częściej musi ustępować tej mikroprocesorowej. Powód takiego stanu rzeczy jest bardzo wiele. Od większej uniwersalności na łatwość poprawy ewentualnych błędów kończąc.

Jeżeli chodzi jednak o przemysł i produkcję urządzeń na wielką skalę to jest jeden parametr który przyćmiewa wszystkie inne. Są to oczywiście koszty produkcji. Koszty zaprojektowania i wdrożenia produktu na rynek są również ważne, ale to koszty produkcji są powodem dla którego księgowi rwą po nocach włosy z głowy szukając oszczędności na każdym drobnym elemencie. W tym miejscu pojawiają się mikroprocesory. Układy o bardzo dużej wszechstronności, które można w każdej chwili przeprogramować całkowicie zmieniając ich działanie. W rękach sprawnego programisty są bardzo wydajne, a jednocześnie niezwykle energooszczędnne. Jednakże w mojej pracy energooszczędność nie jest najważniejszym celem.

1.1 Cel pracy

Myślą przewodnią stojącą za powstaniem tego projektu jest zaprojektowanie systemu zdolnego do sterowania pracą silnika prądu stałego. Co więcej, sterowanie odbywać się będzie w sposób zdalny. Jeden sygnał z komputera i sterownik umieszczony nawet na innej półkuli wysteruje silnik tak, jak sobie tego zażyczymy. Połączenie będzie zrealizowane z wykorzystaniem, bardzo popularnego w robotyce i rozwiązaniach internetu rzeczy, protokołu MQTT. Projekt dopełniać będzie miła dla oka aplikacja okienkowa, która pozwoli na łatwą obsługę i przejrzysty wgląd w najważniejsze parametry pracy silnika.

Dodatkowym celem dla projektu jest zachowanie jak najniższej ceny, co będzie później warunkowało wybór konkretnych elementów systemu. Ma to również wpływ na poziom trudności projektu, w szczególności regulatora napięcia elementu wykonawczego, ponieważ tanie elementy cechują znacznie gorsze parametry aniżeli drogie, markowe produkty.

Rozdział 2

Schemat działania projektu

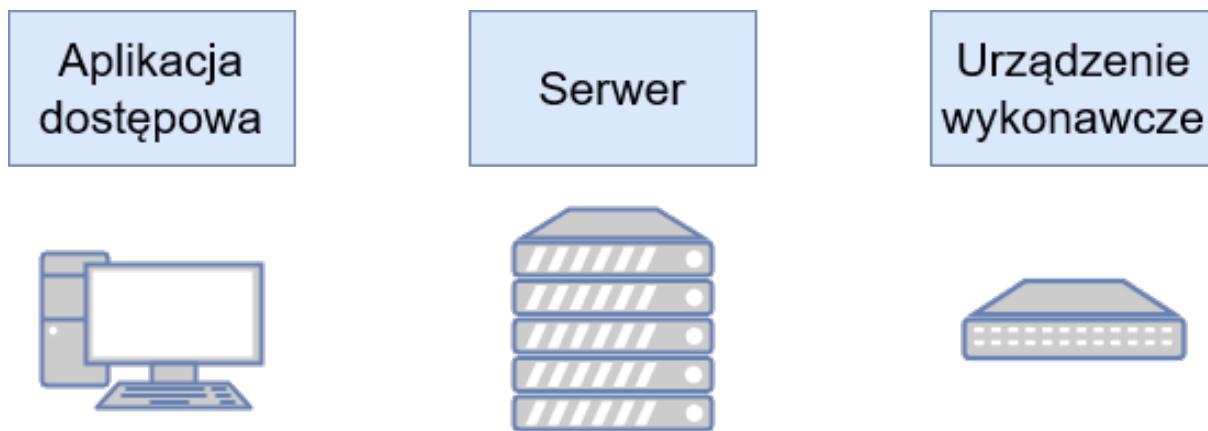
2.1 Założenia

Zostały zdefiniowane ogólne założenia dotyczące każdego z elementów, których spełnienie definiuje kryterium sukcesu.

- System składa się z trzech głównych elementów, połączonych ze sobą:
 - urządzenia wykonawczego,
 - brokera MQTT,
 - aplikacji dostępowej.
- Komunikacja pomiędzy częściami składowymi odbywa się z poprzez sieć bezprzewodową wykonaną w technologii WiFi z wykorzystaniem protokołu MQTT.
- Urządzenie wykonawcze ma zadawać napięcia na silnik prądu stałego w taki sposób, aby uzyskać parametry jak najbardziej zbliżone do zadanych przez aplikację dostępową.
- Aplikacja dostępowa ma za zadanie stanowić prosty i przejrzysty interfejs do sterowania urządzeniem. Umożliwia ona:
 - zadawania prędkości obrotowej silnika,
 - odczytu aktualnej prędkości silnika,
 - zmiany nastaw regulatora PID,
 - odczytu napięcia zasilania urządzenia,
 - odczytu wypełnienia sygnału PWM.
- Broker MQTT on za zadanie być lekkim i szybkim pośrednikiem w komunikacji między aplikacją dostępową, a urządzeniem wykonawczym.

2.2 Architektura systemowa

Projekt opiera się o współpracę trzech kluczowych elementów ciągle komunikujących się ze sobą. Komponenty wchodzące w skład systemu zostały umieszczone na rysunku 2.1.



Rysunek 2.1: Elementy systemu

2.2.1 Aplikacja dostępowa

W celu komfortowej obsługi sterownika silnika, została stworzona aplikacja dostępowa w języku C++, która będzie komunikowała się w protokole MQTT. W tym celu wykorzystano otwarto źródłową wersję narzędzia programistycznego jakim jest QT [1]. Biblioteki QT posiadają dobrze zdefiniowane warstwy abstrakcji oraz bardzo wylewną dokumentację, co sprawia że tworzenie zaawansowanych, przenośnych aplikacji okienkowych jest niezwykle łatwe i przyjemne.

2.2.2 Urządzenie wykonawcze

Urządzenie końcowe jest dedykowanym rozwiązańem przygotowanym specjalnie na poczet tego projektu. Bazuje ono na nowoczesnym SoC z rodziny ESP32, który łączy się z brokerem MQTT. Odbierane z niego dane wykorzystane są w procesie sterowania silnikiem. Poza odczytem informacji z brokera, urządzenie publikuje również aktualny stan silnika.

2.2.3 Serwer

Ostatnim elementem projektu jest Broker MQTT. Jego zadaniem jest bycie łącznikiem pomiędzy urządzeniami. Pozwala on na łatwe subskrybowanie jak i publikowanie informacji. Ta rola przypadła dla otwarto źródłowego brokera Mosquitto wydanego przez fundację Eclipse [19]. Jest to jeden z popularniejszych programów tego typu, a zawdzięcza to małym wymaganiom sprzętowym, skalowalności oraz dostępności na wielu architekturach sprzętowych. Co więcej, w celu wdrożenia przenośności projektu, oprogramowanie to zostało poddane konteneryzacji, dzięki czemu można łatwo transportować go i uruchamiać na innych komputerach wraz z całą konfiguracją przy użyciu ekosystemu Docker [20].

2.3 Wymiana informacji

W protokole MQTT kluczowy aspekt mają tematy, z angielskiego "Topics". Pozwalają one na wymianę i porządkowanie informacji. Każda nadawana wiadomość jest kierowana do konkretnego tematu na brokerze. W celu odbioru wiadomości niezbędna jest subskrypcja danego tematu.

2.3.1 Wykorzystywane tematy

Tematy tworzone i zarządzane przez aplikację okienkową to:

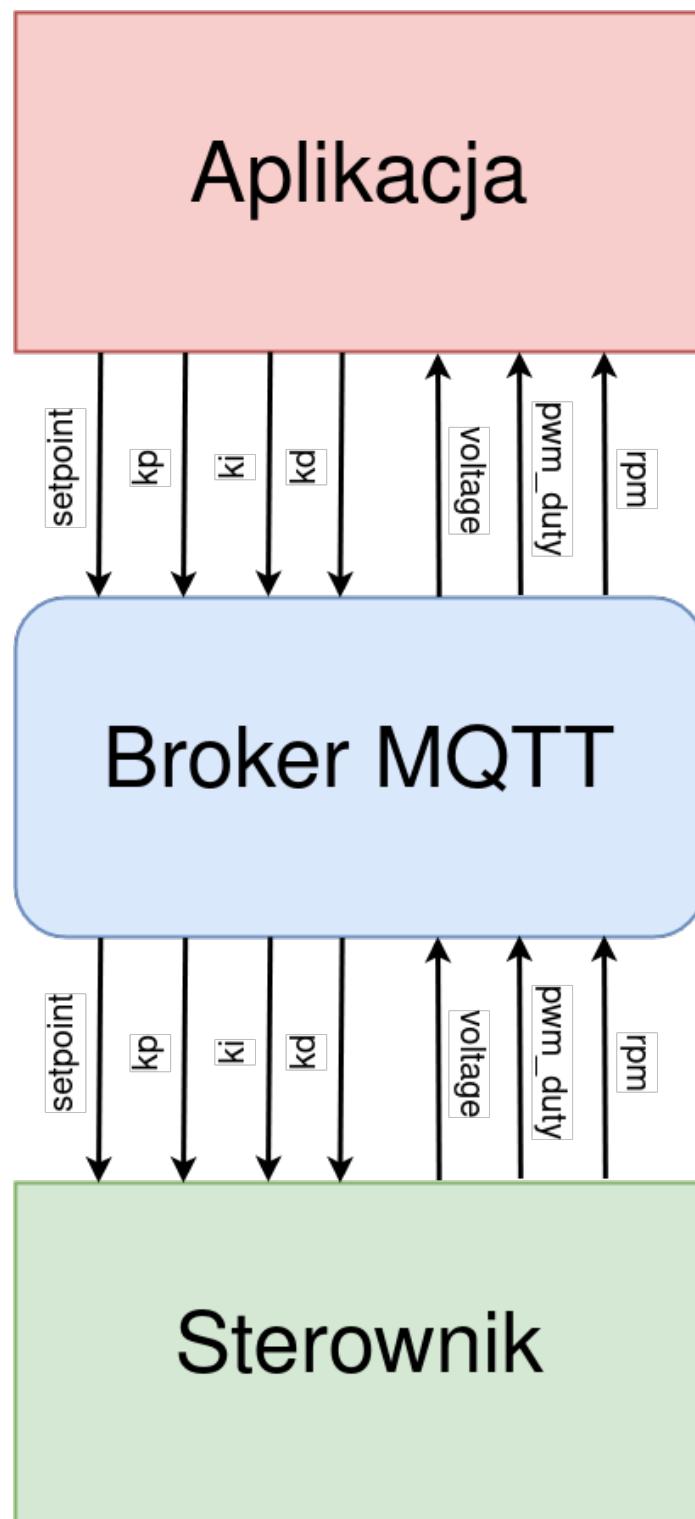
- kp - Wartość członu proporcjonalnego. Parametr sterujący pracą regulatora PID,
- ki - Wartość członu całkującego. Parametr sterujący pracą regulatora PID,
- kd - Wartość członu różniczkującego. Parametr sterujący pracą regulatora PID,
- setpoint - Wartość zadana dla regulatora. Wyrażona w obrotach na minutę.

Należy zauważyc że są to tylko i wyłącznie wartości zarządzające pracę regulatora PID. Dzięki udostępnieniu ich poza obszar programu istnieje możliwość dynamicznej zmiany parametrów regulatora, co sprawia że nawet osoba nie mająca dużego doświadczenia z regulatorami PID potrafi empirycznie wyznaczyć zadowalające wartości.

Poza tematami utworzonymi przez aplikację dostępową, potrzebne są jeszcze trzy dodatkowe tematy:

- rpm - ilość obrotów odczytana z silnika przy pomocy enkodera. Wyrażona w obrotach na minutę. Służy jedynie w celach kontrolnych. Daje możliwość zorientować się z jaką prędkością obraca się silnik w rzeczywistości i porównać wynik z wartością zadaną.
- pwm_duty - wartość wypełnienia PWM, wyrażona w procentach. Obrazuje stopień wykorzystania dostępnej mocy silnika.
- voltage - Napięcie zasilania urządzenia.

Obaj klienci brokera MQTT subskrybują nawzajem swoje tematy. Schemat wymiany danych został przedstawiony na rysunku 2.2. Obrazuje on sposób komunikacji i zależności.



Rysunek 2.2: Schemat wymiany danych w systemie

Rozdział 3

Szczegółowy opis brokera MQTT

3.1 Protokół

MQTT jest skrótem od MQ Telemetry Transport. Jego głównym założeniem jest niesamowita prostota implementacji jednocześnie zachowując wybitnie skromne wymagania sprzętowe. Jego zalety szybko zostały zauważone czego najlepszym przykładem jest fakt że znalazł on szerokie zastosowanie w takich branżach jak Automotive, logistyka, czy produkcja. Jednak najczęściej kojarzony jest on z tematami Internetu Rzeczy oraz Inteligentnych Domów. Świeśnie nadaje się on do łączenia w jedną sieć małych energooszczędnich urządzeń.

Wiadomości są zorganizowane w hierarchii tematów. Każda z wiadomości przypisana jest do jakiegoś tematu. W przypadku, gdy temat nie istnieje, zostaje automatycznie utworzony wraz z napływem pierwszej wiadomości. Broker po odebraniu wiadomości informuje wszystkich klientów, którzy zapisali się do listy subskrypcji danego tematu. W przypadku, gdy dany temat już istnieje następuje nadpisanie jego wartości nowymi danymi. Dzieje się tak dlatego że broker przechowuje tylko i wyłącznie ostatnią wiadomość z każdego tematu.

Ciekawą opcją jest ustawienie tak zwanego testamentu. Jest to wiadomość publikowana, gdy klient który sobie taką opcję zażyczył, nieoczekiwane utraci połączenie z brokerem.

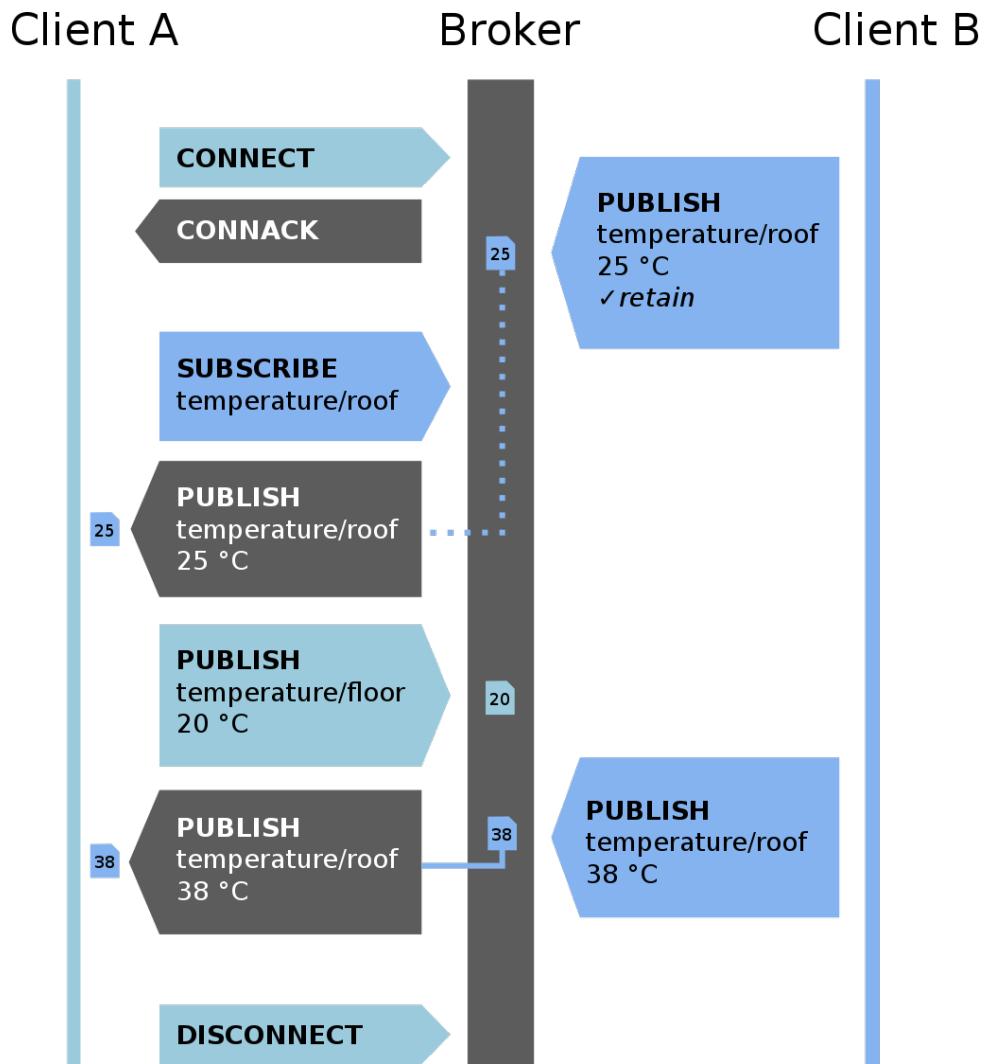
Klienci podłączeni do jednego brokera nie znają się nawzajem. Nie są bowiem udostępniane żadne dane bezpośrednio pomiędzy klientami. Przedstawia to schemat umieszczony na rys. 3.1. Wszystkie przekazywane wiadomości muszą przejść przez broker.

3.2 Broker

Broker MQTT jest to pewnego rodzaju serwer który zbiera wiadomości od wszystkich klientów, a następnie przekierowuje je klientów docelowych zgodnie z ich subskrypcjami. Jego zadaniem jest więc być pośrednikiem w wymianie informacji pomiędzy klientami oraz dbanie o odpowiedni przepływ wiadomości.

3.3 Typy wiadomości

W dokumentacji protokołu MQTT [5] istnieje aż 15 obsługiwanych typów wiadomości. Odpowiadają one za:



Rysunek 3.1: Schemat działania protokołu MQTT [27]

- CONNECT - Nawiązanie połączenia,
- CONNACK - Potwierdzenie nawiązania połączenia,
- PUBLISH - Publikacja wiadomości,
- PUBACK - Potwierdzenie publikacji wiadomości
- PUBREC - Potwierdzenie otrzymania wiadomości,
- PUBREL - Potwierdzenie wysłania wiadomości,
- PUBCOMP - Potwierdzenie końca publikowania wiadomości,
- SUBSCRIBE - Subskrypcja tematu,
- SUBACK - Potwierdzenie subskrypcji,
- UNSUBSCRIBE - Anulowanie subskrypcji,
- UNSUBACK - Potwierdzenie anulowania subskrypcji,

- PINGREQ - Żądanie PINGU,
- PINGRESP - Odpowiedź na żądanie PINGU,
- DISCONNECT - Zakończenie połączenia,
- AUTH - Uwierzytelnienie.

3.4 Instalacja i konfiguracja

3.4.1 Tworzenie kontenera

Instalacja brokera Mosquitto w kontenerze jest niezwykle prosta. Do uruchomienia instancji brokera wystarczy jedna komenda 3.4.1 wpisana w konsoli. W przypadku, gdy Docker nie znajdzie lokalnie danego kontenera, automatycznie pobierze on niezbędne pliki, rozpakuje je, a następnie kontynuuje uruchomienie. Jedynym wymogiem jest poprawnie przeprowadzona instalacja programu Docker.

```
docker run -it --name mosquitto -p 1883:1883  
-v $(pwd)/mosquitto:/mosquitto/ eclipse-mosquitto
```

Listing 3.4.1: Utworzenie instancji brokera MQTT w kontenerze

Do kompleksowego zrozumienia użytego polecania warto pochylić się nad sensem wykorzystanych opcji:

- *run* uruchamia nowy kontener.
- *-it* włącza interaktywność kontenera, przekierowuje standardowy strumień wejściowy do kontenera.
- *-name* przypisuje kontenerowi przyjazną nazwę.
- *-p* to opcja pozwalająca przekierować port poza kontener. Domyślny port wykorzystywany przy komunikacji MQTT to 1883.
- *-v* montowanie woluminu. W tym przypadku montowany jest on w folderze użytkownika. Taka opcja znacząco ułatwia ingerencję w pliki konfiguracyjne.

Na końcu polecenia znajduje się nazwa obrazu z biblioteki, który ma zostać uruchomiony.

3.4.2 Konfiguracja

Po pomyślnej instalacji wymagana jest minimalna konfiguracja programu. Plik "mosquitto.conf" jest głównym plikiem konfiguracyjnym, w którym należy zawrzeć opcje wymienione w listingu 3.4.2. Przede wszystkim należy ustawić wykorzystywany port na identyczny z podanym przy tworzeniu kontenera. Kolejno należy wybrać możliwość zapisywania stanów oraz ścieżkę zapisu. Ważnym aspektem jest także wymuszenie autoryzacji podczas połączenia wraz ze wskazaniem na listę dopuszczanych klientów i haseł.

```
allow_anonymous false
listener 1883
persistence true
persistence_location /mosquitto/data/
password_file /mosquitto/pass
```

Listing 3.4.2: Minimalna konfiguracja brokera

3.4.3 Dodawanie użytkowników

Ostatnim krokiem w przygotowywaniu brokera do przyjęcia klientów jest zdefiniowanie listy akceptowanych loginów wraz z przypisanymi im hasłami. Najprostszym sposobem jest wykorzystanie dedykowanej temu celu komendy widocznej w listingu 3.4.3.

```
mosquitto_passwd -c passwordfile user
```

Listing 3.4.3: Dodawanie pierwszego użytkownika

Tworzy ona nowy plik z hasłami o nazwie *passwordfile* jednocześnie wyzwalając inicjalizację pierwszego użytkownika o loginie *user*. Po wykonaniu polecenia zostaje wystosowane również dwukrotne zapytanie o hasło dla nowego konta.

Rozdział 4

Szczegółowy opis sterownika

Urządzenie wykonawcze jest głównym elementem całego systemu. Jest ono także najbardziej skomplikowanym fragmentem, a wynika to z wielu części składowych niezbędnych do przygotowania kompleksowego produktu. Składa się na niego między innymi projekt płytka drukowanej oraz projekt oprogramowania. Warto zaznaczyć że do skonstruowania w pełni autorskiego rozwiązania niezbędna jest szczegółowa wiedza z wielu dziedzin. Poczynając od elektroniki na programowaniu systemów wbudowanych kończąc.

4.1 Mikrokontroler

Jednostka obliczeniowa wybrana do przeprowadzania obliczeń mieści się w popularnym SoC (z angielskiego System on a chip) o oznaczeniu ESP32-WROOM-32UE [4]. SoC oznacza że na jednej małej płytce drukowanej znajduje się kilka sprzężonych ze sobą modułów. Takie rozwiązanie znaczaco upraszcza konstrukcję nowych urządzeń ponieważ wszystkie krytyczne elementy są już rozmieszczone przez producenta SoC, który gwarantuje ich poprawne działanie.

ESP32 jest to układ chińskiej firmy Espressif Systems [14]. Został on wybrany ze względu na zintegrowany moduł WiFi oraz bardzo niską cenę wynoszącą poniżej 2\$ za sztukę, co perfekcyjnie wpisuje się w założenia projektu. Ponadto znajduje się w nim wiele ciekawych komponentów takich jak:

- dwurdzeniowy procesor o taktowaniu do 240MHz,
- 520 KiB RAM, 448 KiB ROM, 4 MB FLASH
- Bluetooth w standardzie 4.2 oraz BLE,
- 34 wyprowadzenia GPIO.

Użycie gotowego SoC znacznie upraszcza konstrukcję i gwarantuje poprawną współpracę zawartych w nim układów. Zainstalowany w nim mikrokontroler to ESP32-D0WD-V3.

4.2 System operacyjny

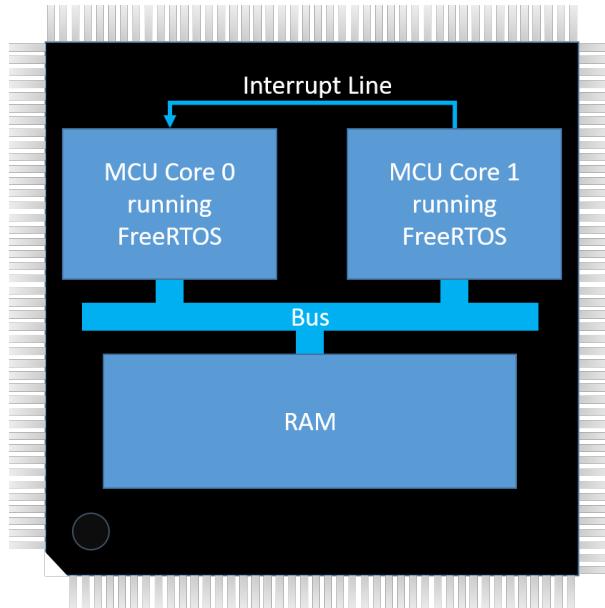
Jednym z wielu wyzwań stawianych przed programistą systemów wbudowanych jest stworzenie rozbudowanego programu wykonującego wiele zadań na pojedynczej jednostce obliczeniowej. Powoduje to konieczność wykorzystywania rozbudowanych maszyn stanów i



Rysunek 4.1: Wykorzystana jednostka centralna [29]

bardzo skomplikowanej logiki. Z pomocą przychodzą systemy operacyjne, które umożliwiają tworzenie programów wielowątkowych i uprzyjemniają korzystanie z procesorów wielordzeniowych. Ze względu na to że użyty w projekcie procesor ma dwa rdzenie logiczne oraz że program ten powinien wykonywać wiele czynności jednocześnie, zdecydowano się skorzystać z funkcjonalności oferowanej przez systemy operacyjne.

Program wykonuje się więc pod kontrolą otwarto źródłowego systemu czasu rzeczywistego FreeRTOS [9]. Jest on znany z dużej szybkości działania. Implementuje on wszystkie elementy niezbędne do pracy z wieloma procesami takie jak mutexy, semafory czy kolejki priorytetowe. Bez niego obsługa WiFi czy protokołu MQTT byłaby zdecydowanie bardziej skomplikowana i czasochłonna.



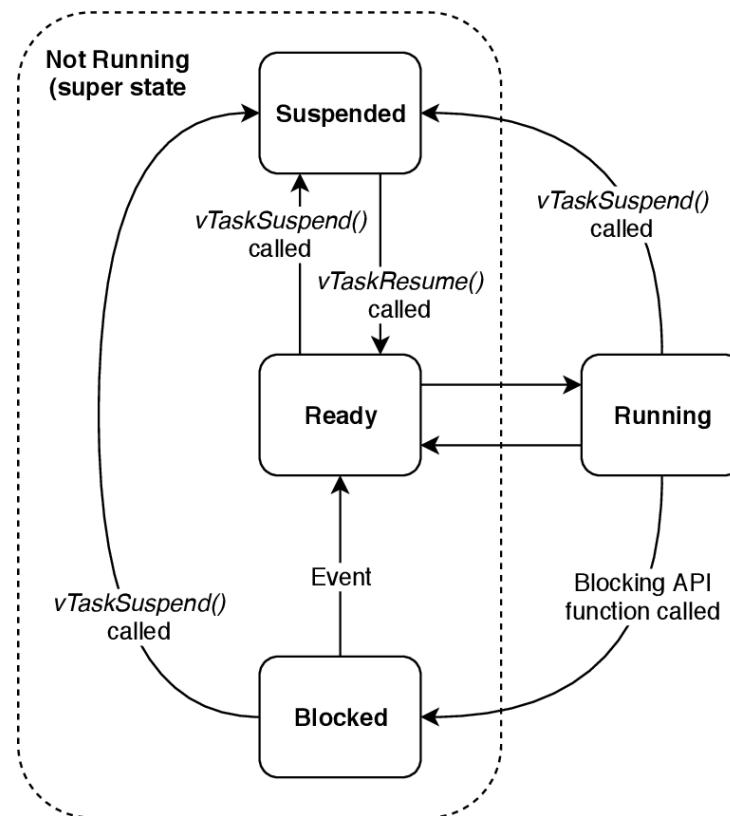
Rysunek 4.2: Topologia procesora z użyciem FreeRTOS [23]

Mówiąc o zaletach systemów operacyjnych warto też wspomnieć o ich wadach. Najważniejszą z nich jest narzut obliczeniowy wynikający między innymi z pracy planisty oraz przełączania kontekstów. Z tego powodu część czasu procesora poświęczana jest na pracę samego systemu i nie bierze udziału w wykonywaniu obliczeń na rzecz programu. Należy zaznaczyć że nie są to duże wartości i są one zależne od indywidualnej konfiguracji systemu operacyjnego. W przytoczonym przez dokumentację FreeRTOS przykładzie [17] przełączenie kontekstu zajmuje 84 cykle procesora, co jest bardzo małą wartością biorąc pod uwagę taktowanie na poziomie 240MHz. Z tego powodu systemy operacyjne znajdują-

ja szerokie zastosowanie nie tylko w zastosowaniach profesjonalnych, bo dzięki prostocie FreeRTOS, jest on coraz częściej wykorzystywany przez amatorów i hobbistów.

4.2.1 Szeregowanie zadań

Ważnym aspektem funkcjonowania systemu operacyjnego jest współbieżność procesów. Każdy z nich posiada swój priorytet oraz aktualny stan. Możliwe stany w systemie FreeRTOS zostały przedstawione na rysunku 4.3.



Rysunek 4.3: Możliwe stany procesów w FreeRTOS [31]

Zadania w stanie gotowości szeregowane są względem ustawionego priorytetu. W przypadku znalezienia się kilku zadań o identycznym priorytecie, są one szeregowane według algorytmu karuzelowego (z ang. Round Robin [18]).

4.3 Framework

Przy opracowywaniu kodu źródłowego sterownika został użyty framework ESP-IDF [8] przygotowany przez producenta procesora. Na stronie dostawcy tego rozwiązania można znaleźć także obszerną dokumentację tłumaczącą wykorzystanie frameworka w praktyce wraz z wieloma przykładami użycia. Zastosowanie gotowej abstrakcji zdejmuje z dewelopera obowiązek implementacji wielu skomplikowanych aspektów programu. W tym projekcie szczególnie przydatna jest implementacja stosu TCP/IP i protokołu MQTT.

4.4 Kolejki i wątki

4.4.1 Opis

Największą zaletą posiadania systemu operacyjnego jest możliwość tworzenia osobnego procesu dla każdego zadania. To znowu pozwala dokładnie kontrolować czasy wywołań i częstotliwości odpowiednich wydarzeń. Główne zadania stawiane temu sterownikowi to:

- obsługa silnika wraz z liczeniem PID,
- pomiary napięcia zasilania,
- wysyłanie danych przez MQTT,
- obsługa WiFi, odbieranie danych z MQTT, planista i obsługa przerwań

W ten sposób kształtują się cztery główne procesy, które biorą udział w pracy systemu:

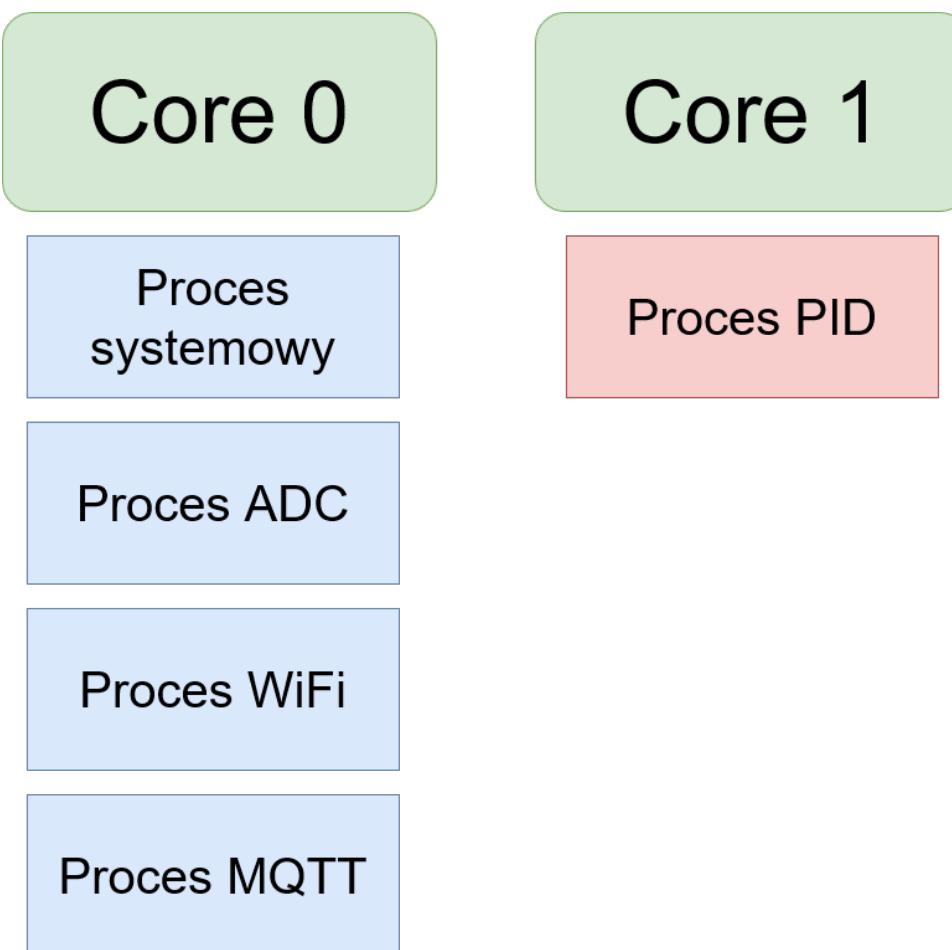
Główny Jest głównym procesem w systemie. Odpowiada on za początkową konfigurację peryferiów przy starcie systemu. Z niego powstaje także reszta procesów. Po poprawnym starcie zajmuje się on obsługą połączenia z siecią, klientem MQTT, przerwaniami oraz planistą systemu.

PID Proces PID wykonuje się dokładnie co 10ms. Każde odchylenie w czasie tego procesu powodowałoby niedokładną pracę regulatora PID, co skutkowałoby niestabilnym zachowaniem się silnika. Z tego też powodu, ten proces został na stałe powiązany do drugiego rdzenia procesora oraz otrzymał bardzo wysoki priorytet. Jego schemat działania został zamieszczony na rys. 4.16

ADC Proces ADC odpowiedzialny jest za pomiary zasilania. Wykonuje się co około 20ms, ale ze względu na ustawiony bardzo niski priorytet ustępuje każdemu innemu zadaniu. Wykonuje się więc tylko gdy procesor nic innego nie robi. Nie stanowi to żadnego problemu, ponieważ opóźnienia w wykonywaniu tego procesu nie wpływają z znacznym stopniu na prawidłową pracę systemu. Informacja przez niego generowana trafia wyłącznie na panel kontrolny użytkownika i jest wyświetlana w aplikacji dostępowej, gdzie aktualizowana jest z częstotliwością około 5Hz. Drobne odchyłki w czasie aktualizacji są więc niewykrywalne. Proces ten został przypisany do rdzenia pierwszego.

WiFi Proces WiFi. Rola tego procesu jest bardzo skromna i ogranicza się jedynie do inicjalizacji połączenia z WiFi. Po wykonaniu tego zadania proces ten się kończy, a rolę obsługi połączenia bezprzewodowego przejmuje proces systemowy.

MQTT Zadaniem procesu MQTT jest odbieranie danych z kolejek priorytetowych FIFO i wysyłanie ich do brokera. Otrzymał on średni priorytet. Wykonuje się w momencie gdy, w którejś z kolejek pojawią się jakieś nowe dane. On również został przypisany do rdzenia pierwszego.



Rysunek 4.4: Diagram wątków

4.4.2 Implementacja menadżera wątków

Został stworzony specjalny moduł odpowiedzialny za zarządzanie wątkami. Pozwala to na łatwe dodawanie i usuwanie funkcjonalności z systemu. Dodanie nowego procesu ogranicza się do wypełnienia struktury przedstawionej na listingu 4.4.1, a następnie dopisania kolejnej pozycji w liście z listingu 4.4.2.

```
12  /*----- TASK STRUCTS -----*/
13  struct Task_config_t
14  {
15      TaskFunction_t pvTaskCode;
16      const char *pcName;
17      uint32_t usStackDepth;
18      void *pvParameters;
19      UBaseType_t uxPriority;
20      TaskHandle_t pvCreatedTask;
21      BaseType_t xCoreID;
22  };
```

Listing 4.4.1: Struktura konfiguracyjna wątku

```

25 enum Task_id
26 {
27     MOTOR_TASK_ID,
28     WIFI_TASK_ID,
29     MQTT_TASK_ID,
30     BATTERY_TASK_ID,
31     TASKS_MAX_ID,
32 };

```

Listing 4.4.2: Lista wątków

Implementacja wątków zawartych w systemie przedstawiona została na listingu 4.4.3. Jest to tablica wcześniej wspomnianych struktur. Pierwsze pole struktury jest uchwytem do wybranej funkcji. Następnie należy podać nazwę procesu. Jest to niezwykle przydatne przy szukaniu błędów, ponieważ nazwa ta jest używana podczas wypisywania informacji diagnostycznych. Kolejne pola to wielkość stosu, przekazywane parametry, priorytet wykonywania, uchwyt do procesu oraz numer rdzenia który ma zajmować się obsługą konfigurowanego zadania.

```

19 Task_config_t tasks[TASKS_MAX_ID] =
20 {
21     {
22         .pvTaskCode = motorTask,
23         .pcName = "motorTask",
24         .usStackDepth = 4096,
25         .pvParameters = nullptr,
26         .uxPriority = configMAX_PRIORITIES-1,
27         .pvCreatedTask = NULL,
28         .xCoreID = 1,
29     },
30
31     {
32         .pvTaskCode = batteryTask,
33         .pcName = "batteryTask",
34         .usStackDepth = 4096,
35         .pvParameters = nullptr,
36         .uxPriority = 3,
37         .pvCreatedTask = NULL,
38         .xCoreID = 0,
39     },
40
41     {
42         .pvTaskCode = wifiTask,
43         .pcName = "wifiTask",
44         .usStackDepth = 4096,
45         .pvParameters = nullptr,
46         .uxPriority = 4,
47         .pvCreatedTask = NULL,
48         .xCoreID = 0,
49     },
50
51     {
52         .pvTaskCode = mqttTask,
53         .pcName = "mqttTask",
54         .usStackDepth = 4096,

```

```

55     .pvParameters = nullptr,
56     .uxPriority = 5,
57     .pvCreatedTask = NULL,
58     .xCoreID = 0,
59 },
60 };

```

Listing 4.4.3: Konfiguracja wątków

4.4.3 Implementacja menadżera kolejek

Zastosowanie kolejek priorytetowych umożliwia komunikację między procesami. Można je deklarować i inicjalizować w modułach, które polegają na przesyłanych danych, ale sprowadza to znaczny bałagan w kodzie. Dużo lepszym rozwiązaniem jest umieszczenie ich w jednym module, który zadba o ich konfigurację i inicjalizację. Zostało to zaprojektowane w sposób bliźniaczy do menadżera wątków. Podstawą jest struktura pokazana w listingu 4.4.4.

```

40 ----- QUEUES STRUCTS -----*/
41 struct Queue_config_t
42 {
43     QueueHandle_t handle;
44     uint32_t uxItemSize;
45     uint32_t uxQueueLength;
46     const char *name;
47 };

```

Listing 4.4.4: Struktura konfiguracyjna kolejki

Struktura ta zawiera uchwyt do kolejki, wielkość transportowanego elementu, dopuszczalną ilość elementów w kolejce oraz nazwę. Przykład wykorzystania został przedstawiony w listingu 4.4.5.

```

63 Queue_config_t queue[QUEUES_MAX_ID] =
64 {
65     {
66         .handle = NULL,
67         .uxItemSize = sizeof(Pid_config_t),
68         .uxQueueLength = 10,
69         .name = "PID Queue",
70     },
71     {
72         .handle = NULL,
73         .uxItemSize = sizeof(int16_t),
74         .uxQueueLength = 10,
75         .name = "Power Queue",
76     },
77     {
78
79     }

```

```

80     .handle = NULL,
81     .uxItemSize = sizeof(int16_t),
82     .uxQueueLength = 10,
83     .name = "Setpoint Queue",
84 },
85
86 {
87     .handle = NULL,
88     .uxItemSize = sizeof(float),
89     .uxQueueLength = 10,
90     .name = "Voltage Queue",
91 },
92
93 {
94     .handle = NULL,
95     .uxItemSize = sizeof(int),
96     .uxQueueLength = 10,
97     .name = "Pulses Queue",
98 },
99 };

```

Listing 4.4.5: Konfiguracja kolejek FIFO

Uchwyty do kolejek są przypisywane podczas inicjalizacji kolejek. Kod dotyczący inicjalizacji umieszczony jest pod listingiem 4.4.6.

```

235 void Task_manager::init_queues()
236 {
237     for (size_t id = 0; id < QUEUES_MAX_ID; id++)
238     {
239         queue[id].handle = xQueueCreate(queue[id].uxQueueLength,
240                                         → queue[id].uxItemSize);
241         ESP_LOGI(TASK_MNGM_TAG, "Queue %s' initialized", queue[id].name);
242     }

```

Listing 4.4.6: Inicjalizacja kolejek FIFO

Wykorzystanie kolejek poza modułem byłoby mocno utrudnione bez zdefiniowania odpowiednich makr, które odnoszą się do przypisanych uchwytów. Kod znajduje się w listingu 4.4.7.

```

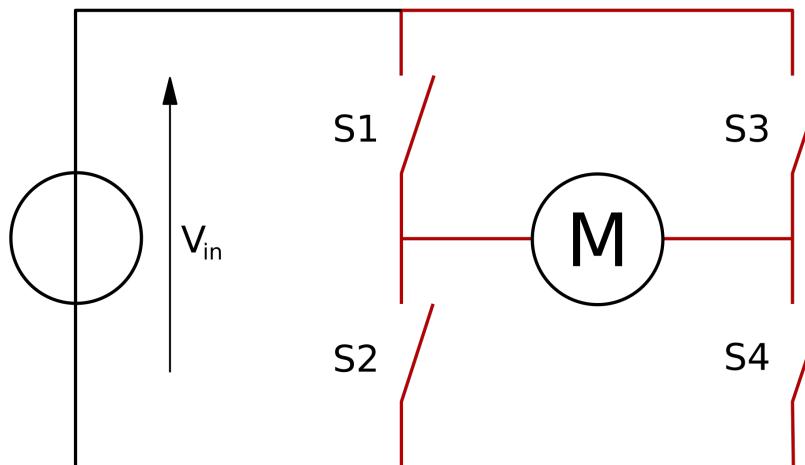
63 ----- QUEUES -----*/
64 extern Queue_config_t queue[QUEUES_MAX_ID];
65
66 #define pidQueue      queue[PID_QUEUE_ID].handle
67 #define powerQueue    queue[POWER_QUEUE_ID].handle
68 #define setpointQueue queue[SETPOINT_QUEUE_ID].handle
69 #define voltageQueue  queue[VOLTAGE_QUEUE_ID].handle
70 #define pulsesQueue   queue[PULSES_QUEUE_ID].handle
71 -----*/

```

Listing 4.4.7: Makra do uchwytów kolejek FIFO

4.5 Mostek H

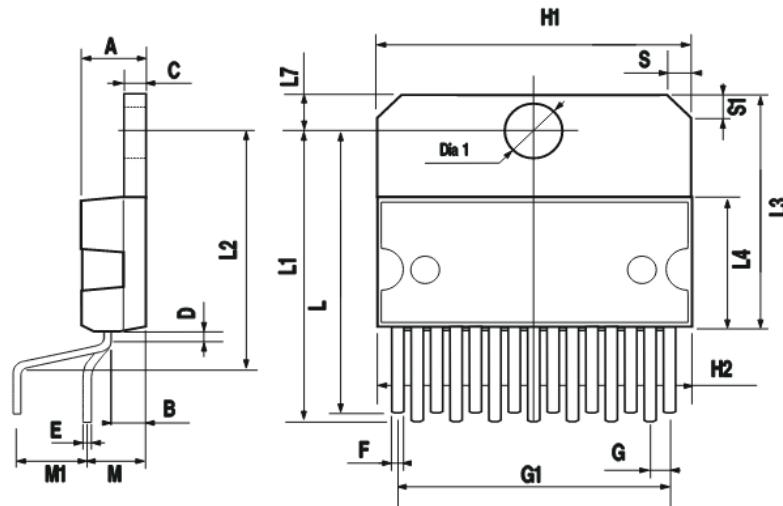
Główym zadaniem mikrokontrolerów jest zbieranie danych z czujników, wykonywanie obliczeń i podejmowanie decyzji. Nie zostały jednak stworzone do sterowania elementami o wyższym napięciu czy dużych prądach, ponieważ takie wymagania znacznie utrudniają miniaturyzację. Z tego powodu wyjścia mikrokontrolerów potrafią dostarczyć zazwyczaj około kilkudziesięciu miliamperów prądu. W przypadku mikrokontrolera użytego w projekcie dokumentacja [3] informuje o bardzo wysokim prądzie wyjściowym dochodzącym nawet do 40mA w sprzyjających warunkach. Jest to wystarczające natężenie prądu do zasilenia diody LED (ok. 20mA) czy sterowania tranzystorem. Z drugiej strony jest to wielokrotnie za mało, aby uruchomić silnik prądu stałego. Takie silniki zazwyczaj nie dość że działają na znaczenie wyższych napięciach niż 3.3V to często potrafią także zużyć ponad 2A prądu podczas obciążenia. Jednym ze sposobów kontrolowania prędkości obrotowej takiego jest zastosowanie wcześniej wspomnianego tranzystora lub przekaźnika, ale minusem takiego rozwiązania jest brak możliwości zmiany kierunku obrotów. Z tego powodu najczęściej stosowanym rozwiązaniem, które pozwoli na sterowanie silnikiem w obu kierunkach jest użycie mostka H. Jego schemat przedstawiono na rys. 4.5.



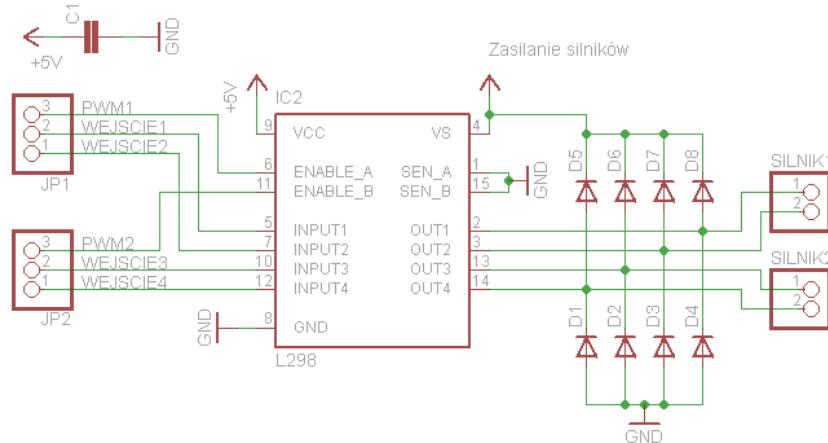
Rysunek 4.5: Ogólny schemat mostka H [24]

W roli mostka H został wyselekcjonowany bardzo popularny układ scalony L298N w obudowie Multiwatt15. Nie jest to nowy układ scalony, ale jest on przetestowany i zdecydowanie wystarczający dla tego zastosowania. Może się on poszczycić napięciem zasilania silników do 46V i prądem szczytowym na poziomie 2A. W rzeczywistości posiada on dwa osobne kanały, co sprawia że z jego pomocą możliwe jest jednoczesne sterowanie dwoma silnikami. Jednakże projekt ten zakłada wykorzystanie tylko jednego silnika, więc drugi kanał pozostanie nieaktywny. Może to spowodować pozytywny wpływ na wolniejsze nagrzewanie się układu, co z pewnością przełoży się na jego stabilniejszą pracę.

Układ scalony zastosowany w projekcie został przedstawiony na rys. 4.6. Redukuje on znacząco poziom skomplikowania urządzenia ponieważ w celu uruchomienia go wystarczą cztery diody redukujące prądy indukowane na cewkach silników podczas ich pracy oraz kondensator filtrujący zasilanie [2]. Przykładowe podłączenie jest zaprezentowane na rys. 4.7.



Rysunek 4.6: Schemat wykorzystanego mostka H [26]



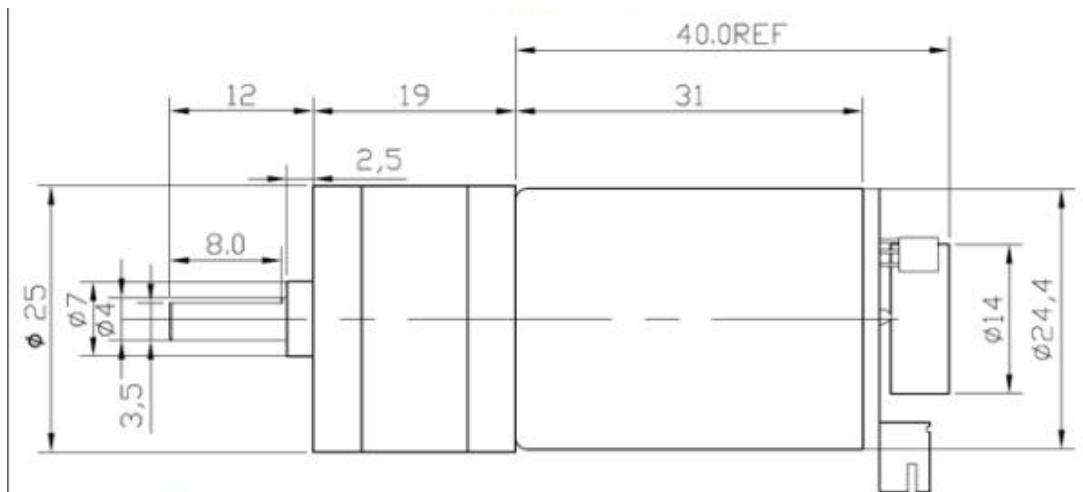
4.6 Silnik

Silnik jest to bardzo konkurencyjnie wyceniany produkt firmy DFROBOT. Został on wybrany w głównej mierze ze względu na swoją niską cenę i parametry, które w zupełności wystarczą do zrealizowania tego projektu. Jest on wyposażony w metalową przekładnię zapewniającą przełożenie napędu w proporcji 20:1. Ponadto, na końcu silnika znajduje się fabrycznie zamontowany enkoder kwadraturowy, który produkuje 11 impulsów na każdy obrót wałka głównego. Schemat silnika wraz z enkoderem umieszczony został na rys. 4.8

4.7 Płytki PCB

4.7.1 Opis

Urządzenie wykonawcze zawiera mikrokontroler, mostek H oraz element wykonawczy w postaci silnika prądu stałego wyposażonego w enkoder kwadraturowy. Poza możliwością zadawania napięcia na silnik, urządzenie dysponuje funkcją pomiaru napięcia zasilania. Poza wymienionymi jednostkami na płytce drukowanej znajdują się także wszystkie ele-



Rysunek 4.8: Schemat wykorzystanego silnika [25]

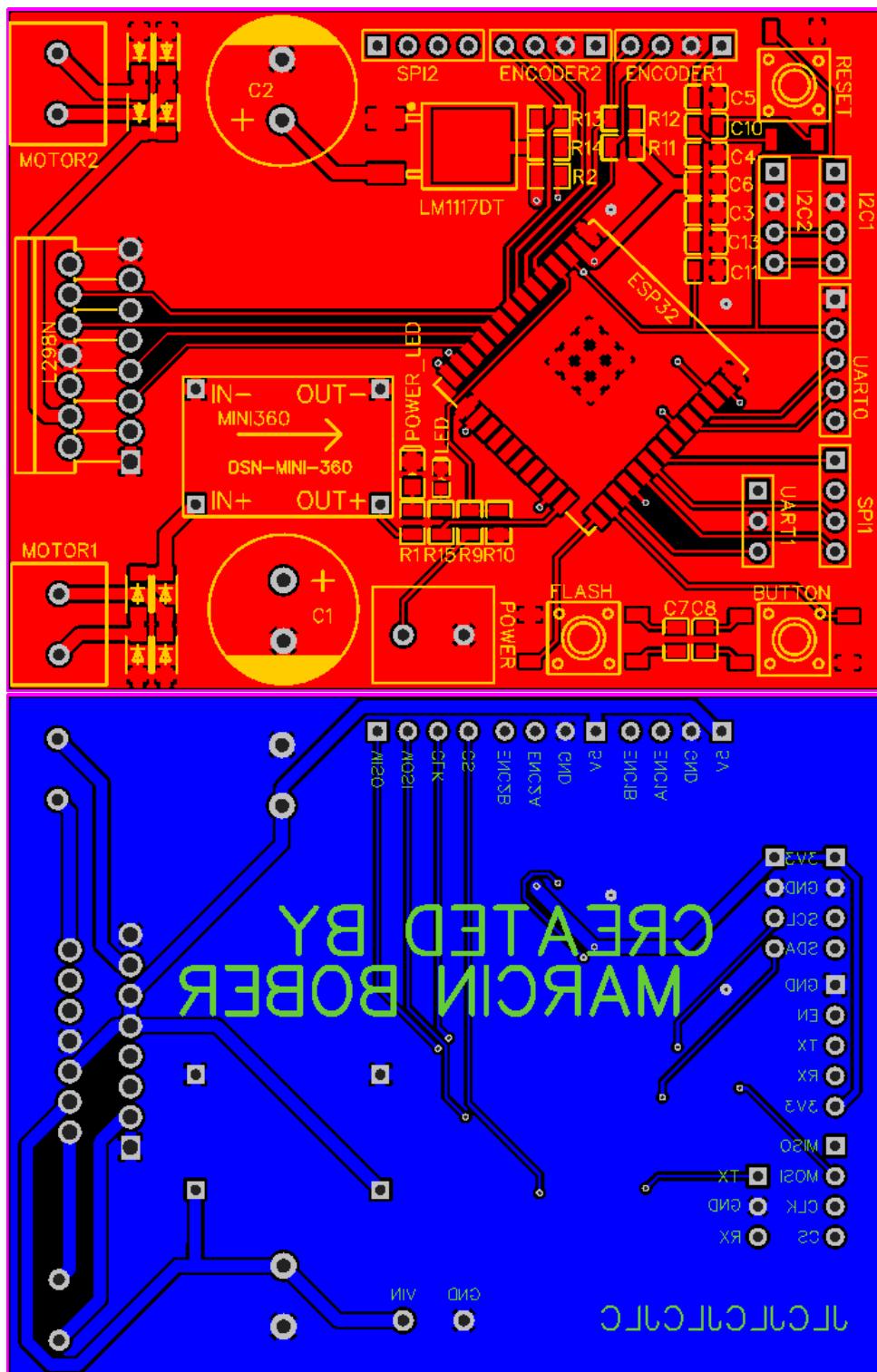
menty niezbędne do prawidłowej pracy procesora takie jak kondensatory i rezystory.

4.7.2 Wygląd

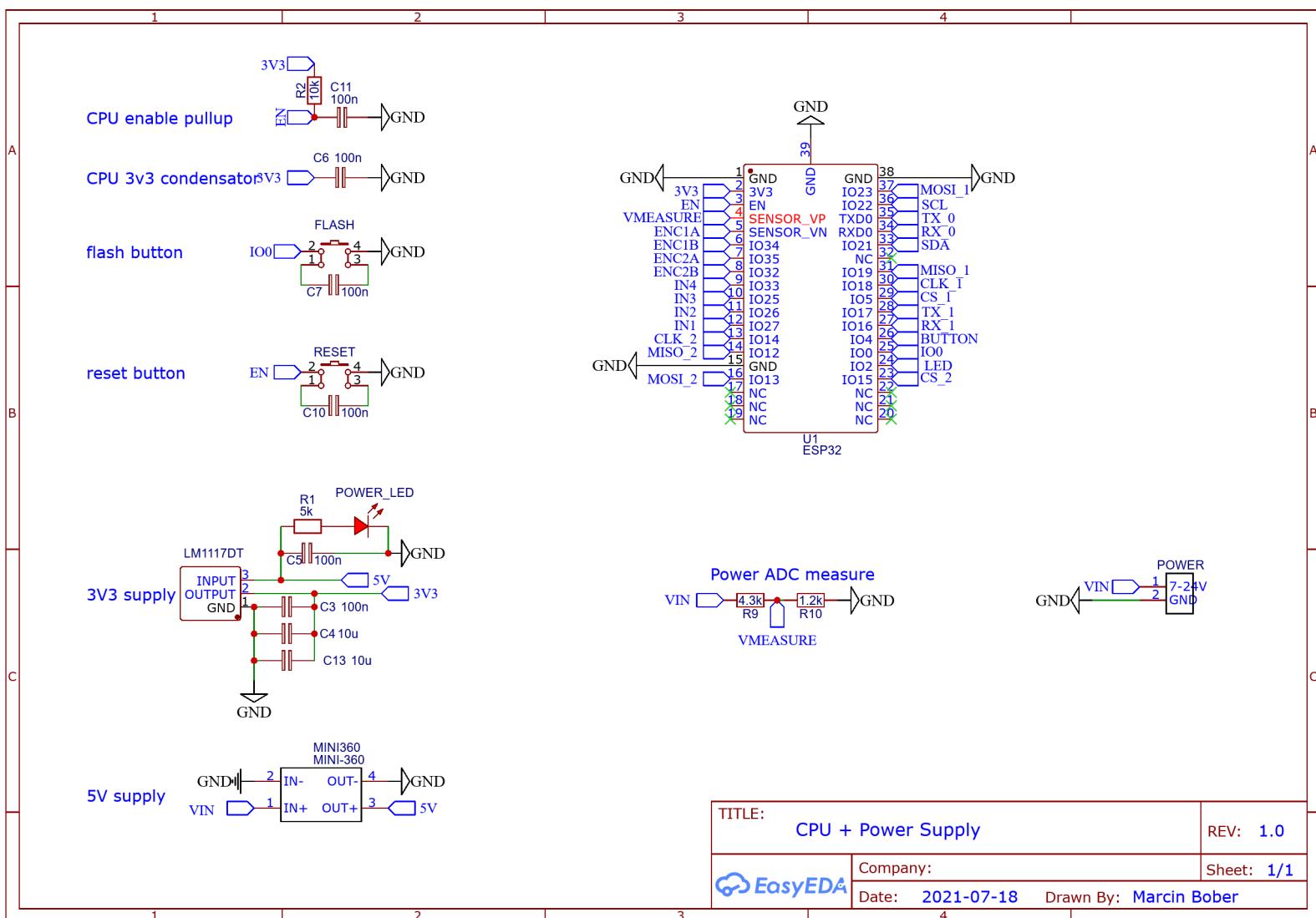
Przygotowany wzór płytki drukowanej z rozlokowanymi elementami i poprowadzonymi ścieżkami został złączony do tego dokumentu. Obie strony płytki zarówno górną jak i dolną dostępne są jako rysunek 4.9.

4.7.3 Schemat

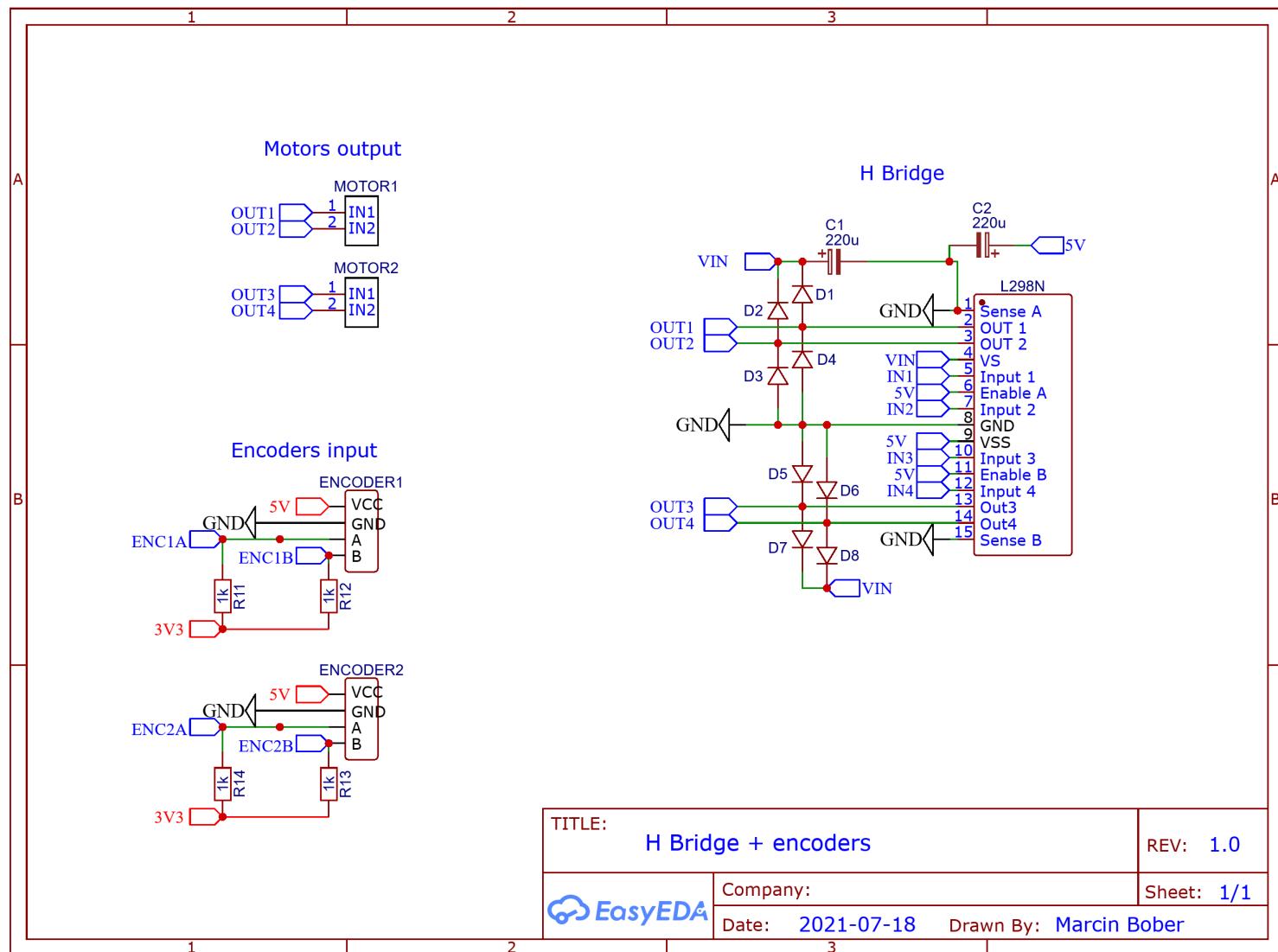
Schemat składa się z trzech stron. Pierwsza strona (rys. 4.10) dotyczy sekcji zasilania oraz połączeń z SoC wraz z elementami niezbędnymi do jego prawidłowej pracy. Druga strona (rys. 4.11) opisuje połączenia enkoderów oraz mostka H. Trzecia strona (rys. 4.12) są to jednie wyprowadzenia interfejsów.



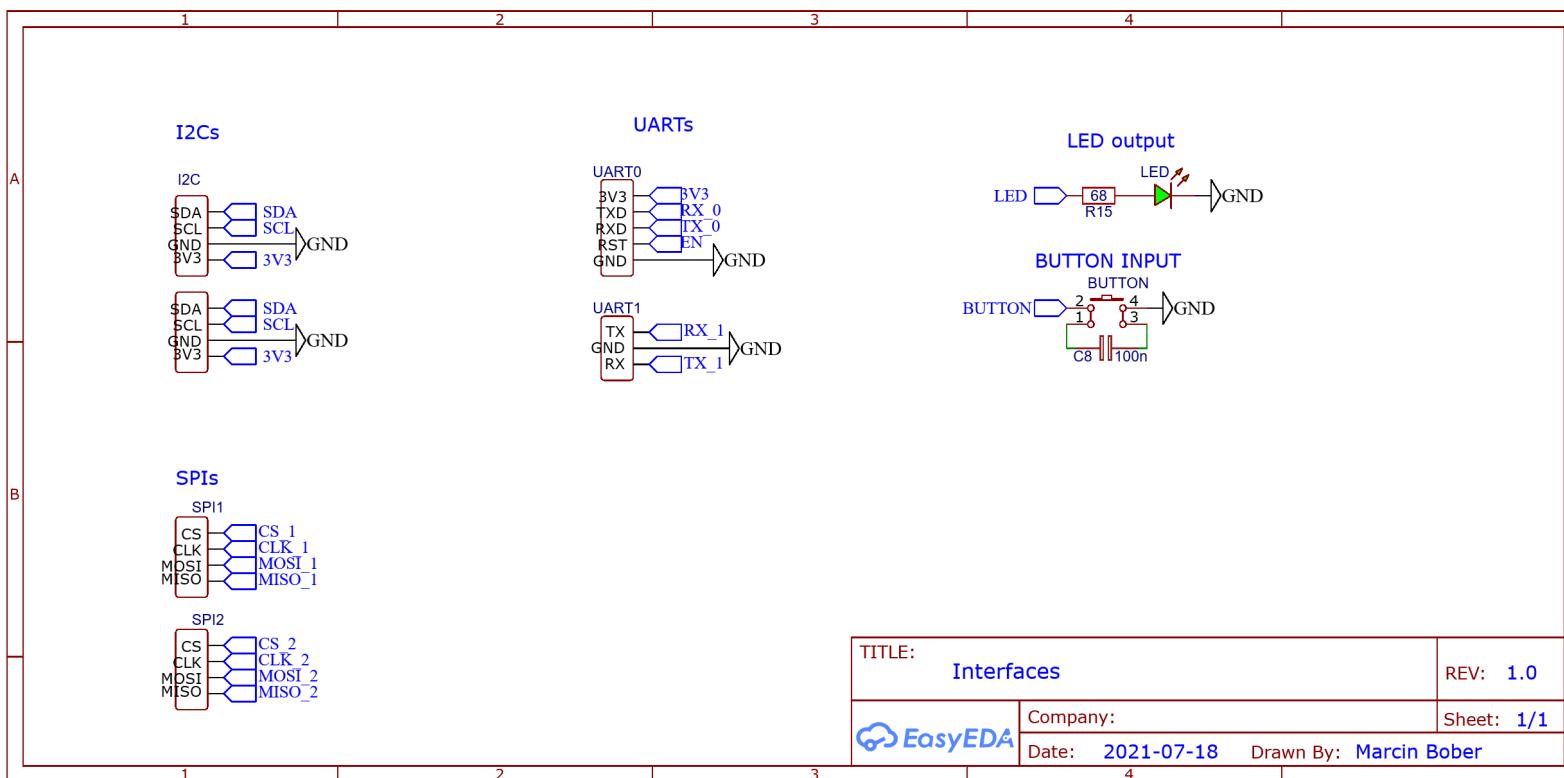
Rysunek 4.9: Wygląd PCB



Rysunek 4.10: Schemat płytka PCB (CPU+PSU)



Rysunek 4.11: Schemat płytki PCB (mostek + enkodery)



Rysunek 4.12: Schemat płytki PCB (interfejsy)

4.8 Licznik impulsów

4.8.1 Opis

Licznik impulsów (Pulse Counter) jest peryferium, które ułatwia obsługę enkoderów. Procesory bez takich udogodnień są skazane na odczytywanie przerwania za każdym razem jak enkoder wyśle impuls, aby następnie za pomocą kodu Graya rozpoznawać kierunek obrotu [12]. W takim zastosowaniu użycie enkodera jako pokrętła jest zrozumiałe. Zmieniając enkoder ręczny na taki zamontowany na silniku znacznie zwiększymyczęstość generowania impulsów, a co za tym idzie również przerwań, odbieranych w sekundzie pracy procesora. W przypadku użytego w tym projekcie silnika prędkość obrotowa silnika przed przekładnią równa jest:

$$V = \frac{n \cdot k}{t_n} = \frac{300RPM \cdot 20.4}{60s} = 102RPS$$

Gdzie:

- n - ilość obrotów silnika za przekładnią w ciągu minuty pracy,
- k - współczynnik redukcji przekładni,
- t_n - okres czasu z którego brane są impulsy.

Wiedząc że zintegrowany enkoder generuje 11 impulsów na każdy obrót, zatem procesor zostanie będzie musiał obsłużyć 1122 przerwań w ciągu zaledwie jednej sekundy pracy. Taka kolej rzeczy z pewnością odbiłaby się na wydajności urządzenia. W ekstremalnym przypadku użycie większego silnika lub dokładniejszego enkodera mogłoby doprowadzić do sytuacji, gdzie procesor nie byłby w stanie obsłużyć tak dużej ilości przerwań. Rozwiązaniem tego problemu jest użycie liczników impulsów, które odciążają mikrokontrolery z takich wyzwań. Posiadają one swoją pamięć w której gromadzą dane na temat zliczonych sygnałów. W takiej sytuacji zadaniem procesora jest jedynie czytanie z tej pamięci. ESP32 posiada aż 8 takich liczników, co stwarza ogromne możliwość wykorzystania tego układu.

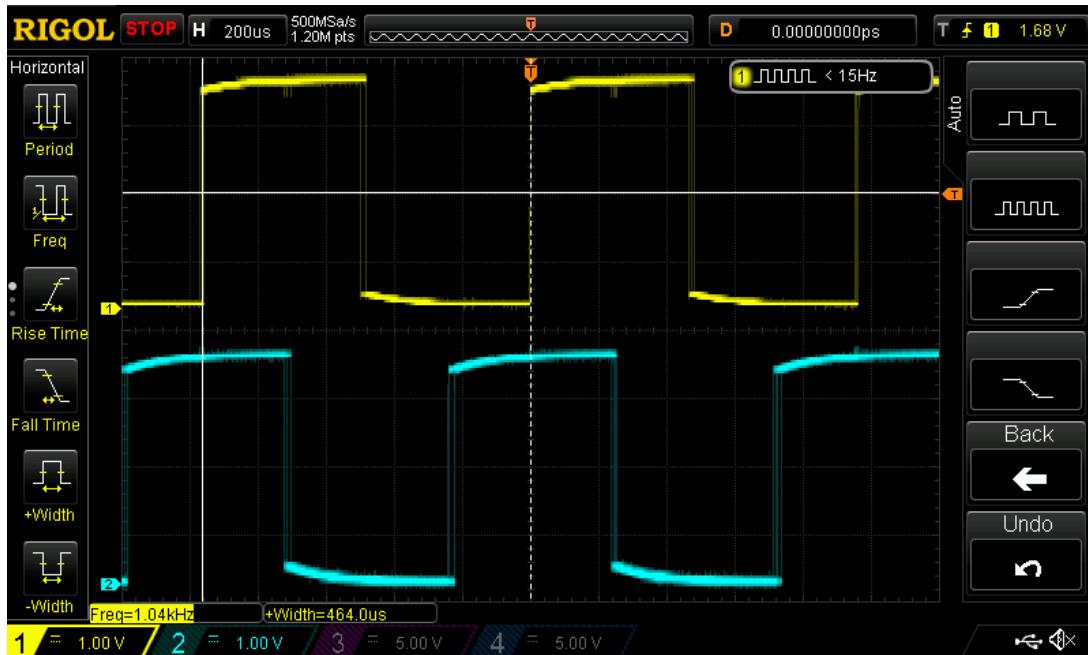
Producent układu w dokumentacji procesora informuje że minimalny okres pomiędzy impulsami, który gwarantuje nie pominięcia żadnego odczytu nie może być mniejszy jak $12.5ns$. Wykorzystany w projekcie silnik obraca się z prędkością 100 obrotów na sekundę, a przy każdym obrocie enkoder produkuje 11 impulsów. Częstotliwość występowania impulsów jest następująca:

$$f = \frac{n}{t_n} = \frac{100 \cdot 11}{1s} = 1100Hz$$

Gdzie n oznacza generowaną ilość impulsów przez enkoder w okresie czasu t_n . Okres pomiędzy impulsami jest odwrotnością częstotliwości:

$$T = \frac{1}{f} = \frac{1}{1100Hz} = 909.09us$$

Obliczenia zostały poparte pomiarami z oscyloskopu. Pomiar został umieszczony na rys. 4.13. Widać na nim że częstotliwość wahą się w okolicy 1040Hz. Delikatne zwiększenie napięcia zasilania spowodowałoby minimalny wzrost obrotów, więc uzyskanie wyliczonej częstotliwości nie jest problemem. Niemniej jednak, okres pomiędzy impulsami enkodera jest o rzędy wielkości większy niż okres graniczny dla tego licznika. Oznacza to że można wypełnić wskazaniom licznika ponieważ że żadne impulsy nie są pomijane. Abstrahując od okresów pomiędzy impulsami obraz z oscyloskopu wspaniale prezentuje przesunięcie przebiegów pomiędzy kanałami enkodera.



Rysunek 4.13: Pomiar impulsów z oscyloskopu

4.8.2 Konfiguracja

W przypadku tego układu, konfiguracja jest bardzo prosta i sprowadza się do wypełnienia zaledwie dwóch struktur i wysłania ich do peryferium.

```

5  /**
6   * @brief Encoder class constructor.
7   * @param encoderA encoder A channel GPIO pin.
8   * @param encoderB encoder B channel GPIO pin.
9   * @param pcnt_unit PCNT unit.
10 */
11 Encoder::Encoder(gpio_num_t encoderA, gpio_num_t encoderB, pcnt_unit_t pcnt_unit) :
12     pcnt_unit(pcnt_unit)
13 {
14     pcnt_config_t pcnt_config = {};
15     pcnt_config.pulse_gpio_num = encoderA;
16     pcnt_config.ctrl_gpio_num = encoderB;
17     pcnt_config.channel = PCNT_CHANNEL_0;
18     pcnt_config.unit = pcnt_unit;
19     pcnt_config.pos_mode = PCNT_COUNT_DEC;
20     pcnt_config.neg_mode = PCNT_COUNT_INC;
21     pcnt_config.lctrl_mode = PCNT_MODE_REVERSE;
22     pcnt_config.hctrl_mode = PCNT_MODE_KEEP;
23     pcnt_config.counter_h_lim = PCNT_COUNT_LIMIT;
24     pcnt_config.counter_l_lim = -PCNT_COUNT_LIMIT;
25
26     pcnt_config_t pcnt_config2 = {};
27     pcnt_config2.pulse_gpio_num = encoderB;
28     pcnt_config2.ctrl_gpio_num = encoderA;
29     pcnt_config2.channel = PCNT_CHANNEL_1;
30     pcnt_config2.unit = pcnt_unit;
31     pcnt_config2.pos_mode = PCNT_COUNT_INC;
32     pcnt_config2.neg_mode = PCNT_COUNT_DEC;
33     pcnt_config2.lctrl_mode = PCNT_MODE_REVERSE;

```

```
34     pcnt_config2.hctrl_mode = PCNT_MODE_KEEP;
35     pcnt_config2.counter_h_lim = PCNT_COUNT_LIMIT;
36     pcnt_config2.counter_l_lim = -PCNT_COUNT_LIMIT;
37
38     ESP_ERROR_CHECK(pcnt_unit_config(&pcnt_config));
39     ESP_ERROR_CHECK(pcnt_unit_config(&pcnt_config2));
40
41     ESP_ERROR_CHECK(pcnt_counter_pause(pcnt_unit));
42     ESP_ERROR_CHECK(pcnt_counter_clear(pcnt_unit));
43     ESP_ERROR_CHECK(pcnt_filter_disable(pcnt_unit));
44     ESP_ERROR_CHECK(pcnt_intr_disable(pcnt_unit));
45     ESP_ERROR_CHECK(pcnt_counter_resume(pcnt_unit));
46
47     ESP_LOGI(ENCODER_TAG, "Encoder %d initialized", pcnt_unit);
48 }
```

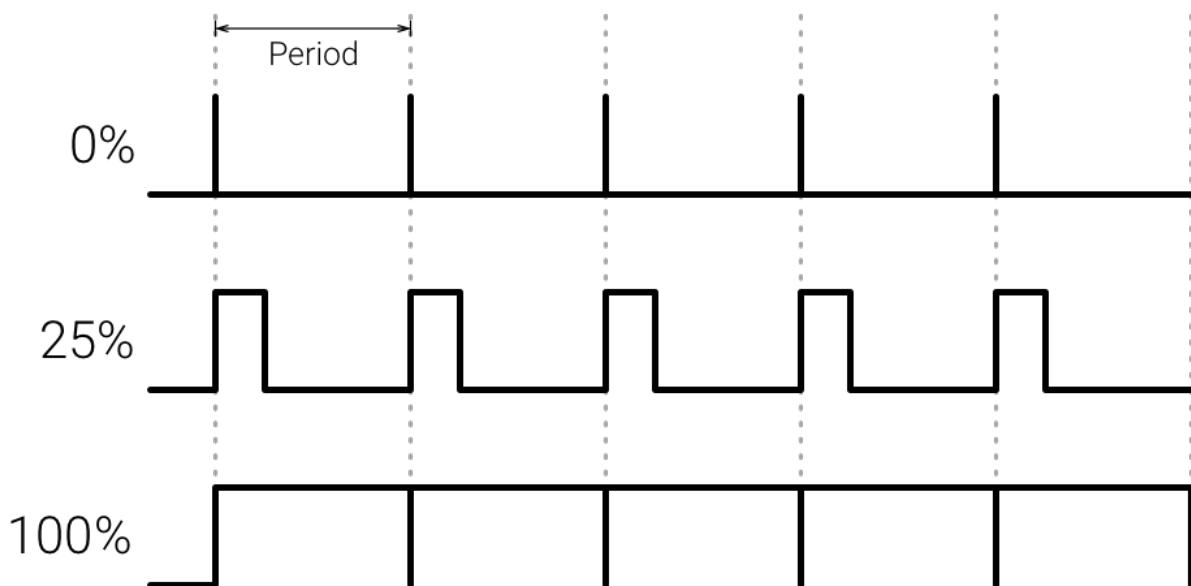
Listing 4.8.1: Konfiguracja licznika impulsów

Użycie jednej struktury powoduje włączenie jednego z dwóch kanałów w liczniku. Licznik automatycznie ewaluuje zbocza narastające na obu wyjściach enkodera, co podwaja wykrywaną ilość impulsów. Poprawne skonfigurowanie obu kanałów w liczniku w sposób pokazany na listingu 4.8.1 sprawia że licznik poprzez analizę zbocz narastających jak i opadających zwiększa dokładność enkodera aż czterokrotnie. Opisane rozwiązanie sprawiło że użyty w projekcie tani enkoder dostarcza aż 897 impulsów na każdy obrót wału wyjściowego z przekładni. Skutkuje to impusem co około 0.4° . Proces ten został opisany w publikacji [13]. Poza konfiguracją licznika wystarczy jedynie dbać o odczyt zliczonych impulsów oraz zerowanie pamięci po wykonanym odczycie. Funkcje odpowiedzialne za te czynności zostały przedstawione w listingu 4.10.1.

4.9 Modulacja szerokości impulsu - PWM

4.9.1 Opis

Najpopularniejszym sposobem sterowania silnikami prądu stałego jest wykorzystanie metody PWM [15] czyli modulacji szerokości impulsu. Cechuje się ona tym że nie wymaga użycia skomplikowanych przetworników cyfrowo-analogowych, jednocześnie umożliwiając płynne sterowanie mocą elementów takich jak diody czy silniki. Jej działanie polega na bardzo szybkim przełączaniu pomiędzy stanami wysokim i niskim. Im dłuższy okres czasu zajmuje stan wysoki w cyklu pracy regulatora, tym więcej energii dostarczane jest do sterowanego elementu. Stosunek tego okresu do okresu pracy PWM nazywany jest wypełnieniem. Przebiegi przy trzech różnych wypełnieniach zostały przedstawione na obrazku 4.14.

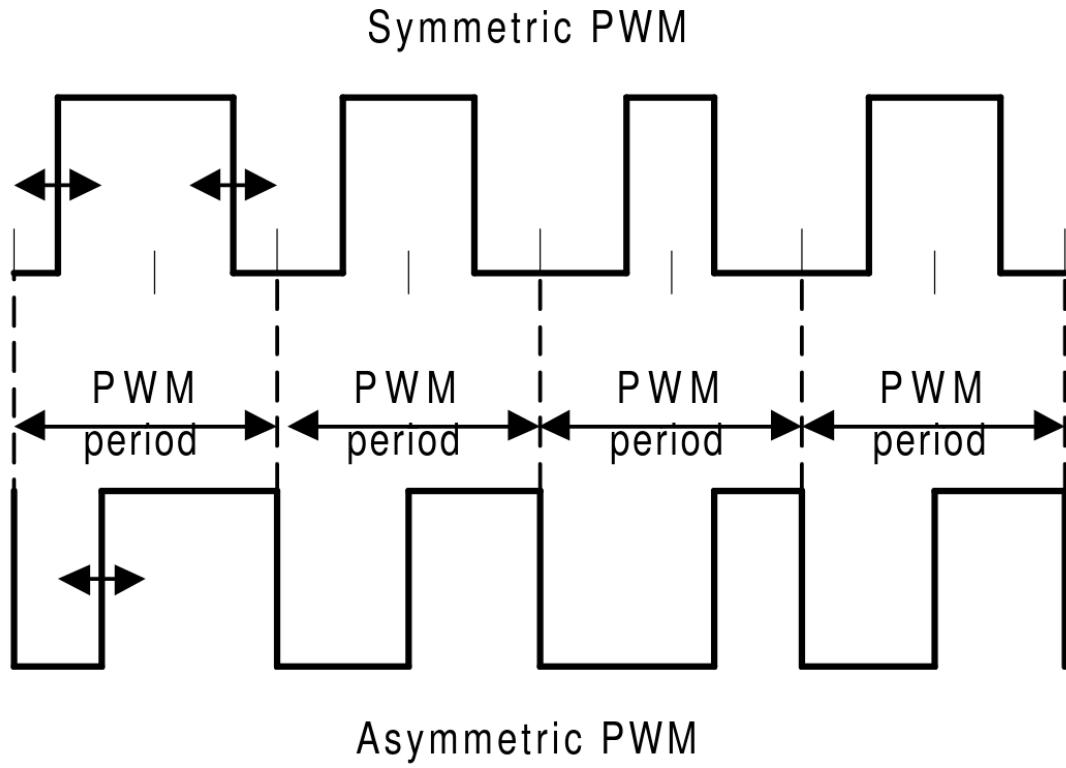


Rysunek 4.14: Przebieg modulacji PWM [30]

4.9.2 Implementacja

Za obsługę peryferium PWM odpowiada klasa "Motor" przedstawiona w listingu 4.9.1. Utworzenie obiektu tej klasy automatycznie inicjalizuje peryferium odpowiedzialne za obsługę PWM. Wybór częstotliwości pracy był podyktowany minimalizacją hałasu generowanego podczas pracy silnika. Z tego powodu zdecydowano się na wartość 22kHz, czyli granice ludzkiego słuchu. Jest to także typowa wartość proponowana przez producenta użytego mostka H [2].

Peryferium znajdujące się w wykorzystanym układzie pozwala na generowanie symetrycznego PWM. Różni się ono od tradycyjnej asymetrycznej regulacji tym że generowane impulsy są zawsze symetryczne względem środka. Zaletą takiego rozwiązania jest generowanie mniejszych harmonicznych w napięciach i prądach wyjściowych oraz lepiej nadaje się ono do sterowania silnikami DC [16]. Porównanie przebiegów symetrycznego i asymetrycznego PWM zostało umieszczone na obrazku 4.15



Rysunek 4.15: Przebieg modulacji PWM [16]

```

3  /**
4   * @brief Motor class constructor.
5   * @param mcpwm_unit MC_PWM unit.
6   * @param in1 input1 GPIO pin.
7   * @param in2 input2 GPIO pin.
8   * @param timer_num PWM timer.
9   */
10 Motor::Motor(mcpwm_unit_t mcpwm_unit, gpio_num_t in1, gpio_num_t in2, mcpwm_timer_t
11   ↪ timer_num) :
12     mcpwm_unit(mcpwm_unit),
13     timer_num(timer_num)
14 {
15   ESP_ERROR_CHECK(mcpwm_gpio_init(mcpwm_unit, MCPWM0A, in1));
16   ESP_ERROR_CHECK(mcpwm_gpio_init(mcpwm_unit, MCPWM0B, in2));
17
18   mcpwm_config_t mcpwm_config = {};
19   mcpwm_config.frequency = MOTOR_PWM_FREQUENCY;
20   mcpwm_config.counter_mode = MCPWM_UP_DOWN_COUNTER;
21   mcpwm_config.duty_mode = MCPWM_DUTY_MODE_0;
22
23   ESP_ERROR_CHECK(mcpwm_init(mcpwm_unit, timer_num, &mcpwm_config));
24   ESP_LOGI(MOTOR_TAG, "Motor %d initialized", mcpwm_unit);
25 }
```

Listing 4.9.1: Inicjalizacja PWM

Jedyna metoda tej klasy wykorzystywana podczas typowej pracy programu jest przedstawiona na listingu 4.9.2. Jej zadaniem jest zmiana ustawień wypełnienia PWM oraz kierunku obrotu silnika w zależności od przekazanej wartości.

```
27  /**
28   * @brief Set motor duty cycle.
29   * @param duty_cycle - float - Duty cycle to set. Range from -100 to 100.
30   */
31 void Motor::set_duty(float duty_cycle)
32 {
33     /* Motor moves in forward direction. */
34     if (duty_cycle > 0)
35     {
36         mcpwm_set_signal_low(this->mcpwm_unit, this->timer_num, MCPWM_GEN_A);
37         mcpwm_set_duty(this->mcpwm_unit, this->timer_num, MCPWM_GEN_B, duty_cycle);
38         mcpwm_set_duty_type(this->mcpwm_unit, this->timer_num, MCPWM_GEN_B,
39                             MCPWM_DUTY_MODE_0);
40     }
41     /* Motor moves in backward direction */
42     else
43     {
44         mcpwm_set_signal_low(this->mcpwm_unit, this->timer_num, MCPWM_GEN_B);
45         mcpwm_set_duty(this->mcpwm_unit, this->timer_num, MCPWM_GEN_A, -duty_cycle);
46         mcpwm_set_duty_type(this->mcpwm_unit, this->timer_num, MCPWM_GEN_A,
47                             MCPWM_DUTY_MODE_0);
48     }
49 }
```

Listing 4.9.2: Zmiana wypełnienia PWM

4.10 Regulator PID

4.10.1 Implementacja

Najważniejszą częścią programu jest implementacja regulatora PID. Odpowiednie zaprojektowanie jego działania jest kluczowe w kwestii wydajności oraz poprawności funkcjonowania całego systemu [21]. Schemat funkcjonowania algorytmu jest prosty i został on przedstawiony na listingu 4.10.2.

Pierwszym krokiem jest odczytanie z rejestrów licznika ilości zliczonych impulsów. Następnie należy wyzerować ten rejestr, aby mógł on ponownie zliczać kolejne impulsy zaczynając od zera. Operacja ta musi zostać wykonana bezpośrednio po dokonaniu odczytu, aby nie pominąć żadnych impulsów, które będą pojawiać się w trakcie wykonywania dalszych obliczeń. Realizacja tego krótkiego procesu przesłonięta przez marko zawarte w 4.10.1 w celu zwiększenia czytelności kodu.

```
10 #define GET_ENCODER_VALUE(pcnt_unit, input)      \
11     pcnt_get_counter_value(pcnt_unit, input);    \
12     pcnt_counter_clear(pcnt_unit);
```

Listing 4.10.1: Pobieranie wartości i czyszczenie rejestrów licznika

Warto zaznaczyć że nie jest zapisywany moment pobrania danych. Musi więc być zagwarantowane że funkcja wykonująca odczyt będzie wykonywać się cyklicznie w ścisłe określonym przedziale czasowym. Każde odstępstwo od tego będzie powodowało błędy w poprawnym działaniu algorytmu.

```
17 /**
18 * @brief Compute PID result.
19 * @param setpoint Setpoint.
20 */
21 int16_t Pid::compute(const int &setpoint) {
22     int16_t input;
23     GET_ENCODER_VALUE(PCNT_UNIT_0, &input);
24
25     int epsilon = setpoint - input;
26
27     // calculate integral error
28     this->integral_error+= epsilon;
29     NORMALIZE_I(this->integral_error);
30
31     // calculate derivative error
32     this->derivative_error = epsilon - last_epsilon;
33
34     // save last epsilon
35     this->last_epsilon = epsilon;
36
37     // calculate PID
38     int16_t p = this->config.kp * epsilon;
39     int16_t i = this->config.ki * this->integral_error;
40     int16_t d = this->config.kd * this->derivative_error;
41     int16_t pid = p + i + d;
42     NORMALIZE_PID(pid);
43 }
```

```
44     return pid;
45 }
```

Listing 4.10.2: Pętla regulatora PID

Kolejnym krokiem jest wyliczenie uchybu regulacji i jest to tradycyjnie różnica wartości ustalonej i zliczonych impulsów. Następnie liczymy wartości dla każdego z członów regulacji, jednocześnie gwarantując że mieszczą się one we wcześniej zdefiniowanych przedziałach. To również ukryte zostało pod makrem dostępnym w listingu 4.10.3.

Wartości każdego z członów mnożymy poprzez wybrane przez użytkownika współczynniki. Są one zapisane w zmiennych dzięki czemu istnieje możliwość zmiany ich wartości dynamicznie podczas pracy programu. Teraz wystarczy już zsumować wszystkie wartości oraz zapewnić że uzyskana wartość nie przekroczy wartości granicznych możliwych do ustawienia dla PWM. W tym momencie część obliczeniowa jest gotowa. Istnieje jednak spore ryzyko że uzyskana w ten sposób wartość nie będzie mieściła się w zakresie przyjmowanym przez peryferium odpowiedzialnym za modulowanie napięcia. Niezbędne jest więc wykonanie kolejnej normalizacji do akceptowalnych wartości 4.10.3.

```
10 #define NORMALIZE_I(integral_error) ({\
11     if(integral_error > INTEGRAL_LIMIT) integral_error = INTEGRAL_LIMIT; \
12     else if(integral_error < -INTEGRAL_LIMIT) integral_error = -INTEGRAL_LIMIT; \
13 }
14 #define NORMALIZE_PID(pid) ({\
15     if(pid > PID_LIMIT) pid = PID_LIMIT; \
16     else if(pid < -PID_LIMIT) pid = -PID_LIMIT; \
17 })
```

Listing 4.10.3: Normalizacja wartości regulatora

4.10.2 Proces PID

Pętla regulatora PID zaimplementowana jest w swoim własnym procesie. Ułatwia to kontrolę częstości wykonywania obliczeń i umożliwia zatrzymanie procesu gdy jest on niepotrzebny. Na przykład przy braku połączenia z serwerem MQTT.

Implementacja procesu zawarta jest w listingu 4.10.4. Pierwsze linie kodu rozpoczynają inicjalizację niezbędnych obiektów. W skład nich wchodzi:

- config - struktura zawierająca konfigurację parametrów regulatora. Niezbędna do dynamicznej zmiany parametrów PID.
- engine - instancja sterująca peryferium PWM,
- encoder - instancja do obsługi enkodera,
- pid - instancja regualtora PID,
- pid_result - zmienna zawierająca wyliczoną wartość regulacji,

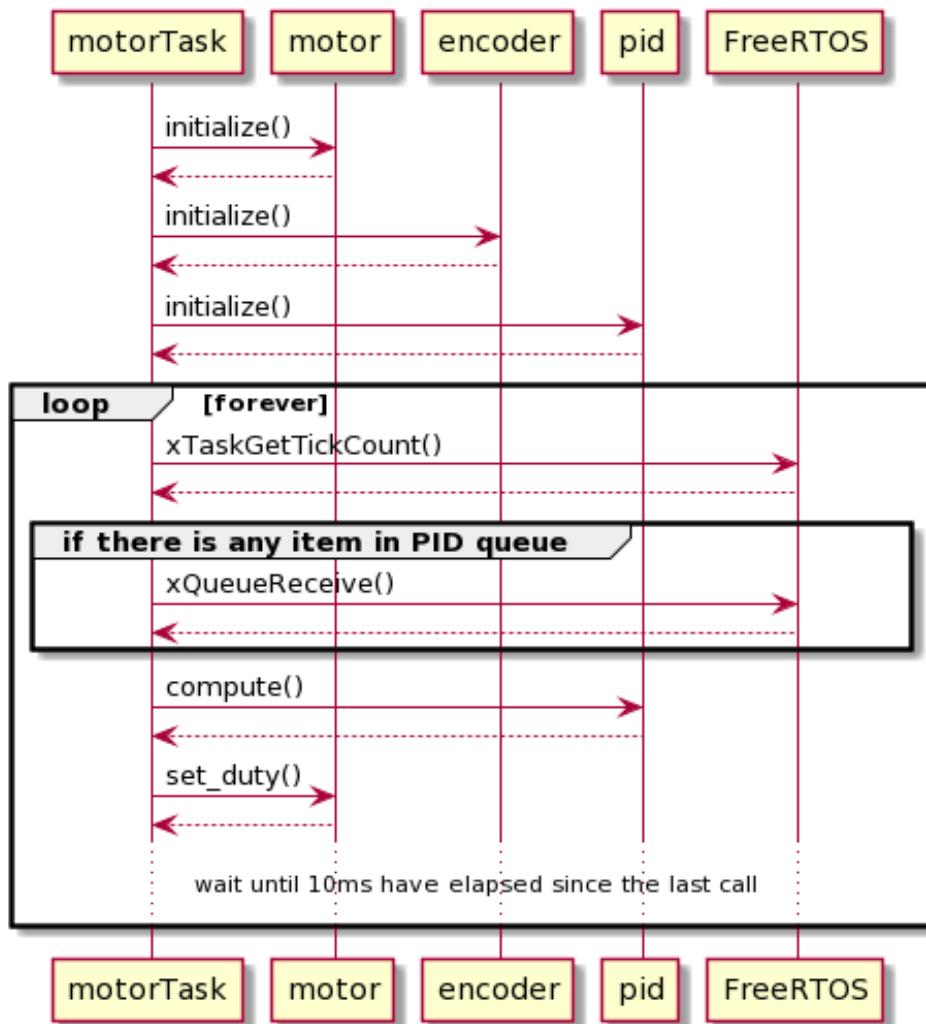
Kolejne linie kodu wykonują się w nieskończonej pętli. Następuje w niej odczyt aktualnego stanu licznika systemowego. Jest on wykorzystywany do wyliczenia jak długo wykonują się operacje w pętli. Następnie sprawdzana jest kolejka FIFO. W przypadku znalezienia się w niej nowej konfiguracji dla regulatora, jest ona natychmiast zastosowana.

```
58  /**
59   * @brief Motor control task.
60   */
61 void motorTask(void*)
62 {
63     Pid_message_t mess;
64     Pid_config_t config =           // create PID config object
65     {
66         .kp = DEFAULT_PID_KP,
67         .ki = DEFAULT_PID_KI,
68         .kd = DEFAULT_PID_KD,
69     };
70
71     // init engine object
72     Motor engine(MOTOR_UNIT, MOTOR_IN1, MOTOR_IN2, MOTOR_TIMER);
73
74     // init encoder object
75     Encoder encoder(ENC_A, ENC_B, ENC_PCNT_UNIT);
76
77     // init PID object
78     Pid pid(encoder, &config);
79
80     TickType_t xLastWakeTime;
81     int16_t pid_result;
82
83     int setpoint = 0;
84
85     while (true)
86     {
87         xLastWakeTime = xTaskGetTickCount();
88
89         if (pdTRUE == xQueueReceive(pidQueue, &mess, 0L))
90         {
91             switch (mess.parameter)
92             {
93                 case SETPOINT_PARAMETER:
94                     setpoint = RPMtoTick(mess.value);
95                     break;
96
97                 case KP_PARAMETER:
98                     config.kp = mess.value;
99                     break;
100
101                case KI_PARAMETER:
102                    config.ki = mess.value;
103                    break;
104
105                case KD_PARAMETER:
106                    config.kd = mess.value;
107                    break;
108            }
109        }
110    }
111}
```

```
109     default:  
110         break;  
111     }  
112  
113     pid.set_parameters(&config);  
114 }  
115  
116     pid_result = pid.compute(setpoint);  
117     engine.set_duty(pid_result);  
118  
119     // set PID loop to 10ms  
120     vTaskDelayUntil(&xLastWakeTime, PID_LOOP_PERIOD);  
121 }  
122  
123 vTaskDelete(NULL);  
124 }
```

Listing 4.10.4: Proces wykonywanie PID

Po opuszczeniu instrukcji warunkowej następuje obliczenie wartości regulatora i ustalenie modulacji szerokości impulsu zgodnie z uzyskanym wynikiem. Ostatnim krokiem jest zatrzymanie procesu. Czas zatrzymania jest zależny od prędkości wykonywania pętli i warunkuje częstotliwość wyzwalania PID na 100Hz. Na obrazku 4.16 znajduje się diagram aktywności opisywanego procesu.



Rysunek 4.16: Diagram aktywności procesu PID

4.11 Pomiar napięcia

Pomiar napięcia dokonywany jest przy użyciu przetwornika analogowo-cyfrowego wbudowanego w mikroprocesor [3]. Posiada on bowiem dwa 12 bitowe przetworniki, których wejścia są multipleksowane na aż 18 pinów. Po wczytaniu się w dokumentację okazuje się jednak że użyteczność tych przetworników ma wiele wykluczeń. Najważniejszym z nich jest interferencja przetwornika ADC2 z peryferium odpowiedzialnym za połączenie WiFi. Oznacza to że w projektach od których wymagamy stałego połączenia z siecią jesteśmy ograniczeni jedynie do 8 pinów obsługujących odczyty analogowe.

Kolejnym ciekawym aspektem tego układu jest możliwość programowego wyboru tłumienia sygnału wejściowego. Producent umożliwia nam wybór od 0dB poprzez 2.5dB oraz 6dB aż do 11dB tłumienia. Im wyższy parametr zostanie wybrany tym wyższe napięcia jest w stanie obsługiwać przetwornik. Niemniej jednak wybór wyższych oczek znacznie degraduje precyzję pomiaru.

4.11.1 Konfiguracja ADC

Po dokładnym przestudiowaniu dokumentacji przyjęte zostało ustawienie tłumienia na poziomie 2,5dB 4.11.1. Wyższe ustawienia wprowadzają znaczną niedokładność pomiarów. Według dokumentacji [3] takie ustawienie sprawia że efektywny zakres pomiarowy kształtuje się w zakresie od 100mV do 1250mV na nóżce mikrokontrolera. Błędy pomiaru zadeklarowane przez producenta nie powinny przekroczyć $\pm 30mV$. Jednakże, zasilanie płytki jest zaprojektowane do obsługi dużo większych napięć, tak aby móc wykorzystywać szeroki zakres silników. Podanie napięcia wyższego jak zasilanie procesora, które w tym wypadku wynosi 3.3V, spowodowałoby trwałe uszkodzenia układu. Do bezpiecznego pomiaru napięcia zasilania niezbędne jest więc zastosowanie dzielnika napięcia.

Rozdzielcość peryferium została ustawiona na 12 bitów. Wynika z tego że istnieje $2^{12} = 4096$ poziomów kwantyzacji. Zaprojektowanie płytki drukowanej, która będzie w stanie w pełni wykorzystać tak dużą rozdzielcość jest bardzo trudne. Obecność przetwornicy impulsowej oraz niewielki dystans pomiędzy procesorem, a mostkiem H sprawia że poziom szumu kwantyzacji będzie znaczący. W przypadku wymogu bardzo szybkich pomiarów lepszym wyborem okazałaby się zmiana rozdzielcości na 10 bitów, co przełożyłoby się na krótszy czas operacji. W tym przypadku czas pomiaru nie jest kluczowy, więc przyjętą strategią uzyskiwania dokładnych wyników jest wykonywanie serii następujących po sobie pomiarów, z którym następnie wyliczana jest średnia.

```

8  /**
9   * Constructs ADC object and set it up.
10  */
11 myADC::myADC()
12 {
13     adc1_config_width(ADC_WIDTH_BIT_12);
14     adc1_config_channel_atten(ADC1_CHANNEL_0, ADC_ATTEN_DB_6);
15
16     for (int i = 0; i < ADC_DATA_SIZE; i++) nextMeasurement();
17 }
```

Listing 4.11.1: Konfiguracja przetwornika ADC

4.11.2 Dzielnik napięcia

Ze względu na niekrytyczne znaczenie pomiarów napięcia zdecydowano że w tym projekcie całkowicie wystarczający będzie najprostszy dzielnik oparty na dwóch rezystorach. Pozwala on na liniowe podzielenie napięcia w taki sposób, aby nigdy nie przekroczyło ono wartości niebezpiecznej dla mikroprocesora. Do stworzenia dzielnika wykorzystano rezystory z szeregu E24 [10] o wartościach $9.1k\Omega$ oraz $1k\Omega$. Ich połączenia jest pokazane na schemacie 4.10.

Obliczenie dostępnego zakresu pomiarowego przetwornika ADC odbywa się przy pomocy wzoru na dzielnik napięcia [11]. Jest on następujący:

$$V_{out} = \frac{R_2}{R_1 + R_2} \cdot V_{in}$$

Gdzie napięcie wejściowe V_{out} jest maksymalnym napięciem dostępnym na wejściu przetwornika. Zgodnie z informacją zawartą w dokumentacji procesora [3] wybranie tłumienia na poziomie 2.5dB skutkuje otrzymaniem 1250mV jako granicy efektywnego zakresu pomiarowego. Przy czym producent zaznacza że ta wartość zależy od konkretnego egzemplarza i może się ważyć. R_1 oraz R_2 to użyte rezystory w dzielniku. W tym przypadku są to kolejno $9.1k\Omega$ oraz $1k\Omega$.

Dla maksymalnego obsługiwanej napięcia z przedziału wzór to:

$$1.250V = \frac{1k\Omega}{9.1k\Omega + 1k\Omega} \cdot V_{in}$$

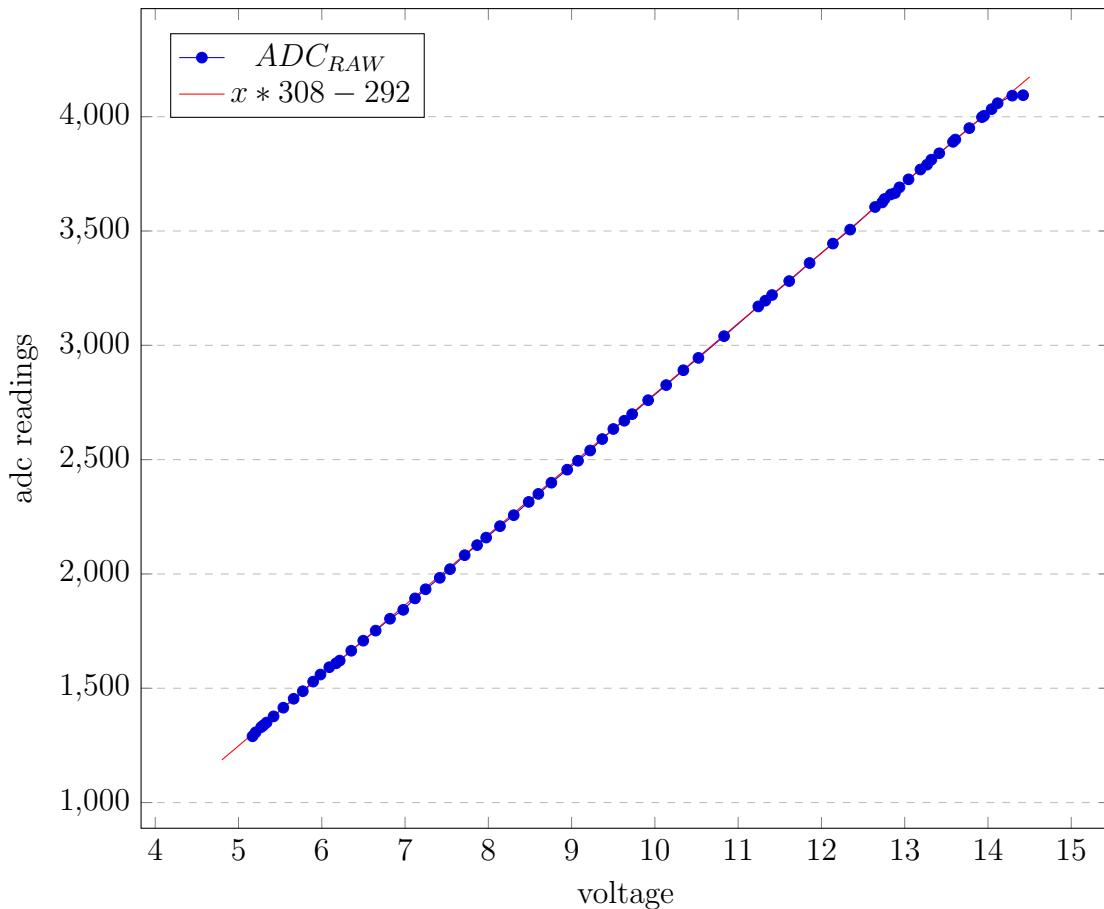
Po przekształceniu go tak aby wyliczyć napięcie przed dzielnikiem otrzymujemy:

$$V_{in} = \frac{1.250V \cdot (9.1k\Omega + 1k\Omega)}{1k\Omega} = 12.625V$$

Dla najniższego napięcia w miesiączącego się w zakresie efektywnego pomiaru (100mV) otrzymamy:

$$V_{in} = \frac{0.100V \cdot (9.1k\Omega + 1k\Omega)}{1k\Omega} = 1.01V$$

Oznacza to że w zakresie zasilania od $1.01V$ do $12.625V$ mamy dużą szansę uzyskać wynik pomiaru bliski rzeczywistemu.



Rysunek 4.17: Pomiary danych ADC do napięcia zasilania

4.11.3 Wyznaczanie współczynnika konwersji

W celu przeliczenia wartości odczytywanych z przetwornika analogowy-cyfrowego na napięcie wyrażone w woltach, niezbędne jest wyznaczenie współczynnika konwersji.

Posiadając wiedzę na temat użytych rezystorów w dzielnicu napięcia, jest możliwe wyliczenie tego współczynnika. Innym sposobem kalibracji przetwornika jest wykonanie serii pomiarów, porównując otrzymane wyniki z zaufanym urządzeniem pomiarowym. Zostały one przedstawione na wykresie 4.17.

Uzyskane pomiary potwierdzają, że prawie cały zakres przetwornika, nie licząc końców, jest liniowy. Pomiary zakończyłem przy około 5.1V ponieważ poniżej tej wartości stabilizator liniowy, użyty do obniżenia napięcia na procesorze, przestawał poprawnie działać, co kończyło się resetami jednostki centralnej.

Dzięki regresji liniowej byłem w stanie wyprowadzić równanie, które w łatwy sposób umożliwia przeliczanie odczytów z ADC na wartość napięcia.

$$y = \frac{ADC_{RAW} + 292}{398}$$

Za ADC_{RAW} wstawiamy wartość pomiaru i otrzymujemy napięcie wyrażone w woltach.

4.11.4 Rozdzielcość pomiaru

Posiadając już możliwość przeliczania surowych odczytów na napięcie, należałoby wyznaczyć rozdzielcość otrzymanych danych. Do obliczenia rozdzielcości napięciowej, czyli najmniejszego możliwego skoku zdolnego do zapisania przed przetwornikiem, musimy wykonać dwa pomiary. Następnie należy porównać różnice obliczonych napięć do różnic surowych danych. Przy pomocy podanego wzoru:

$$\frac{V_1 - V_2}{ADC_1 - ADC_2} = \frac{10.136V - 6.171V}{2826 - 1609} = \frac{3.965V}{1217} = 0.0032V$$

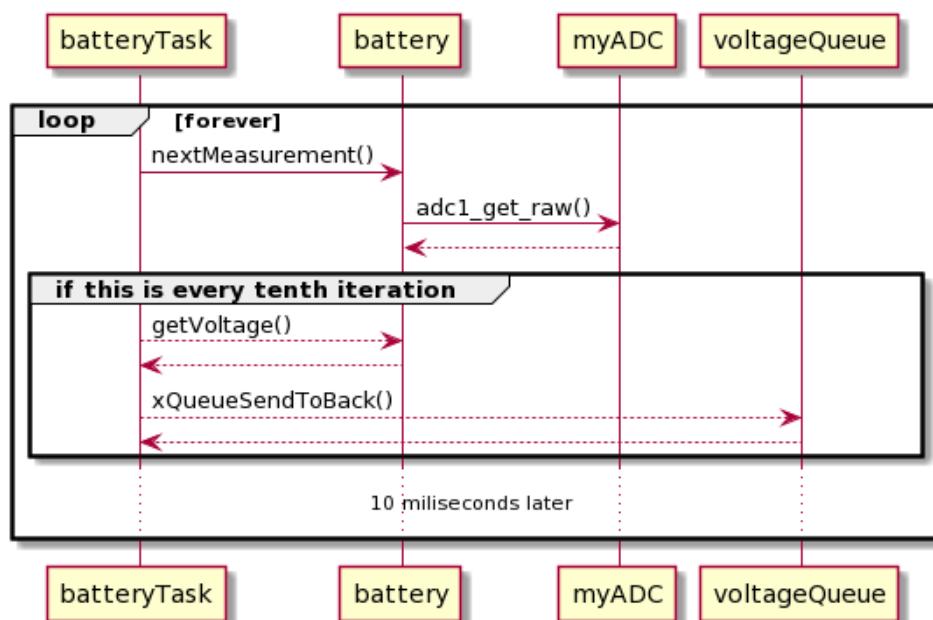
4.11.5 Wykonywanie pomiarów

Wykonywanie pomiarów jest wykonywane na osobnym procesie. Dzięki temu można w łatwy sposób kontrolować częstotliwość mierzenia napięcia i wysyłania danych z ADC na MQTT. Pętla wyzwalająca odczyt wykonuje się raz na 20 milisekund. W każdej iteracji następuje pomiar, a odczytana wartość umieszczana jest w buforze. Co dziesiąta iteracja wylicza średnią z pomiarów, konwertuje wartość na wolty i wysyła dany na MQTT. Diagram aktywności umieszczony został w listingu 4.18.

```

19 /**
20  * Adds another measurement to the buffer
21 */
22 void myADC::nextMeasurement()
23 {
24     static int idx;
25     this->data[idx++] = adc1_get_raw(ADC1_CHANNEL_0);
26     if(idx >= ADC_DATA_SIZE) idx = 0;
27 }
28
29 /**
30  * Calculates the average of the measurements in the buffer
31  * and convert the result into SI units.
32  * @return Voltage in V
33 */
34 float myADC::getVoltage()
35 {
36     uint32_t adc_reading = 0;
37
38     for (size_t i = 0; i < ADC_DATA_SIZE; i++)
39     {
40         adc_reading += this->data[i];
41     }
42
43     return ((float)adc_reading / ADC_DATA_SIZE) / CONVERSION_FACTOR;
44 }
```

Listing 4.11.2: Wyzwalanie pomiaru i przeliczanie wartości



Rysunek 4.18: Diagram aktywności dokonywania pomiarów

4.12 Łączenie z brokerem MQTT

Jedną z zalet wykorzystania oficjalnego frameworka wydawanego przez producenta jest gotowa implementacja warstw abstrakcji ułatwiających obsługę protokołu MQTT. Umożliwia ona łatwą konfigurację i wykorzystanie tego środka komunikacji za pomocą kilku wysokopoziomowych funkcji.

4.12.1 Konfiguracja

Pierwsze co należy zrobić to wypełnić strukturę konfiguracyjną, w której znajduje się aż 165 elementów. Jednak dla prawidłowej pracy wystarczy wypełnić kilka podstawowych pól takich jak dane logowania użytkownika czy adres brokera. Część z nich służy jako wskaźniki do danych przychodzących, bufory, czy flagi niezbędne do funkcjonowania.

Kolejnym krokiem jest stworzenie struktury inicjalizacyjnej przy pomocy poprzedniej struktury. Posłuży nam do tego specjalna funkcja która sprawdza poprawność wprowadzonych przez nas danych, a dane nie definiowane zostają zastąpione wartościami domyślnymi.

Następne musimy wybrać jakie zdarzenia chcemy rejestrować i gdzie mają one trafić. W tym przypadku wybieramy pełną pulę dostępnych sygnałów. Zawartość funkcji obsługującej zdarzenia znajduje się w listingu 4.12.4.

Pozostaje już tylko uruchomić moduł MQTT przekazując powstałą konfigurację do funkcji startowej. Dane zostaną przesłane do nowego procesu, który zajmie się za nas obsługą komunikacji. Wszystkie niezbędne kroki oraz konfiguracja została przedstawiona na listingu 4.12.1.

```

130 void mqtt_client_start()
131 {
132     ESP_LOGI(MQTT_TAG, "Starting MQTT client");
133     esp_mqtt_client_config_t mqtt_cfg = {};
134     mqtt_cfg.host = mqtt_host;
135     mqtt_cfg.port = mqtt_port;
136     mqtt_cfg.username = mqtt_username;
137     mqtt_cfg.password = mqtt_password;
138
139     esp_mqtt_client_handle_t client = esp_mqtt_client_init(&mqtt_cfg);
140     esp_mqtt_client_register_event(client, MQTT_EVENT_ANY, mqtt_event_handler, NULL);
141     esp_mqtt_client_start(client);
142 }
```

Listing 4.12.1: Konfiguracja połączenia MQTT

4.12.2 Obsługa zdarzeń

Wykorzystana warstwa abstrakcji posiada system udostępniający nam możliwość odbierania sygnałów generowanych przez zdarzenia wynikające z pracy klienta MQTT. Skorzystamy z tej funkcjonalności, aby informować użytkownika o stanie komunikacji, a także aby odpowiednio reagować na zdarzenia. Została ona załączona w listingu 4.12.4.

Po otrzymaniu sygnału poprawnego nawiązania połączenia, zaczynamy subskrybować wszystkie tematy które przechowują potrzebne nam dane. Jest to przedstawione na listingu 4.12.2.

```

14 void subscribeAllTopics(esp_mqtt_client_handle_t &client) {
15     esp_mqtt_client_subscribe(client, "edrive/setpoint", 0);
16     esp_mqtt_client_subscribe(client, "edrive/kp", 0);
17     esp_mqtt_client_subscribe(client, "edrive/ki", 0);
18     esp_mqtt_client_subscribe(client, "edrive/kd", 0);
19 }
```

Listing 4.12.2: Subskrybowanie niezbędnych tematów

Kolejnym krokiem jest utworzenie nowego procesu przypisanego do rdzenia drugiego (numeracja jest od zera). Jego zadaniem jest transmisja danych z urządzenia do brokera. Utworzenie kolejnego zadania oraz sam fakt pomyślnego uzyskania połączenia obarczony jest komunikatem diagnostycznym.

Otrzymanie informacji o zakończeniu połączenia wiąże się z odwróceniem zmian dokonanych przez poprzedni sygnał. Po wysłaniu komunikatu na strumień wyjścia, dokonujemy zabicia procesu wysyłającego dane, po uprzednim sprawdzeniu czy taki jeszcze istnieje. Jest to zabezpieczenie przez sytuację, gdy istnieje więcej jak jeden taki proces. Następnie po oczekaniu jednej sekundy, wyzwalana jest próba ponownego łączenia.

Ostatnim ważnym sygnałem jest informacja o zmianie wartości subskrybowanego kanału. Jest ona obrazu przekazywana do stworzonej prze zemnie funkcji. Znajduje się ona w listingu 4.12.3. Jej jedynym zadaniem jest dodawanie odebranych wartości do odpowiednich kolejek priorytetowych w zależności od tematu wiadomości.

```

21 void parseData(std::string &topic, std::string &data) {
22     if(topic == "edrive/setpoint") {
23         int setpoint = std::atoi(data.c_str());
24         xQueueSendToBack(setpointQueue, &setpoint, 0);
25     }
26
27     else if(topic == "edrive/kp") {
28         int kp = std::atoi(data.c_str());
29         printf("New KP: %d\n", kp);
30         xQueueSendToBack(kpQueue, &kp, 0);
31     }
32
33     else if(topic == "edrive/ki") {
34         int ki = std::atoi(data.c_str());
35         printf("New KI: %d\n", ki);
36         xQueueSendToBack(kiQueue, &ki, 0);
37     }
38
39     else if(topic == "edrive/kd") {
40         int kd = std::atoi(data.c_str());
41         printf("New KD: %d\n", kd);
42         xQueueSendToBack(kdQueue, &kd, 0);
43     }
44 }
```

Listing 4.12.3: Odbieranie danych

Pozostałe sygnały służą czysto w celach diagnostycznych. Z tego też powodu nie wykonują one żadnego kodu poza raportowaniem zdarzenia poprzez komunikat na standarydowym strumieniu wyjścia.

```

73 static void mqtt_event_handler(void *handler_args, esp_event_base_t base, int32_t
74   → event_id, void *event_data)
75 {
76     esp_mqtt_event_handle_t event = (esp_mqtt_event_handle_t)event_data;
77     esp_mqtt_client_handle_t client = event->client;
78
79     std::string topic(event->topic, event->topic_len);
80     std::string data(event->data, event->data_len);
81
82     switch ((esp_mqtt_event_id_t)event_id) {
83         case MQTT_EVENT_CONNECTED:
84             ESP_LOGI(MQTT_TAG, "Connected to server");
85             subscribeAllTopics(client);
86             xTaskCreatePinnedToCore(mqttSendingTask, "MQTT_SEND_TASK", 4096,
87             → (void*)client, 5, mqtt_sender, 1);
88             ESP_LOGI(MQTT_TAG, "New MQTT sender task!");
89             break;
90
91         case MQTT_EVENT_DISCONNECTED:
92             ESP_LOGI(MQTT_TAG, "Disconnected from server");
93
94             if(mqtt_sender != nullptr) {
95                 ESP_LOGI(MQTT_TAG, "Killing MQTT sender task!");
96                 vTaskDelete(mqtt_sender);
97             }
98
99             ESP_LOGI(MQTT_TAG, "Waiting 1 second...");
100            vTaskDelay(1000 / portTICK_PERIOD_MS);
101            ESP_LOGI(MQTT_TAG, "Reconnecting...");
102            esp_mqtt_client_reconnect(client);
103            break;
104
105         case MQTT_EVENT_SUBSCRIBED:
106             ESP_LOGI(MQTT_TAG, "Subscribed new topic");
107             break;
108
109         case MQTT_EVENT_UNSUBSCRIBED:
110             ESP_LOGI(MQTT_TAG, "Unsubscribed topic");
111             break;
112
113         case MQTT_EVENT_DATA:
114             parseData(topic, data);
115             break;
116
117         case MQTT_EVENT_ERROR:
118             ESP_LOGI(MQTT_TAG, "Mqtt client error!");
119             break;
120
121         case MQTT_EVENT_BEFORE_CONNECT:
122             ESP_LOGI(MQTT_TAG, "Connecting...");
123
124         default:
125             break;
126     }
127 }
```

Listing 4.12.4: Obsługa sygnałów

4.12.3 Wysyłanie danych

Transmisja informacji odbywa się na osobnym procesie aniżeli reszta modułu MQTT. Logika przekazywania danych jest względnie prosta. Wystarczy odebrać wiadomość z kolejki priorytetowej, a następnie przekazać ją do wysłania z pomocą odpowiedniej funkcji. Na listingu 4.12.5 przedstawione jest wysyłanie wiadomości na wszystkich trzech tematach zarządzanych przez mikrokontroler.

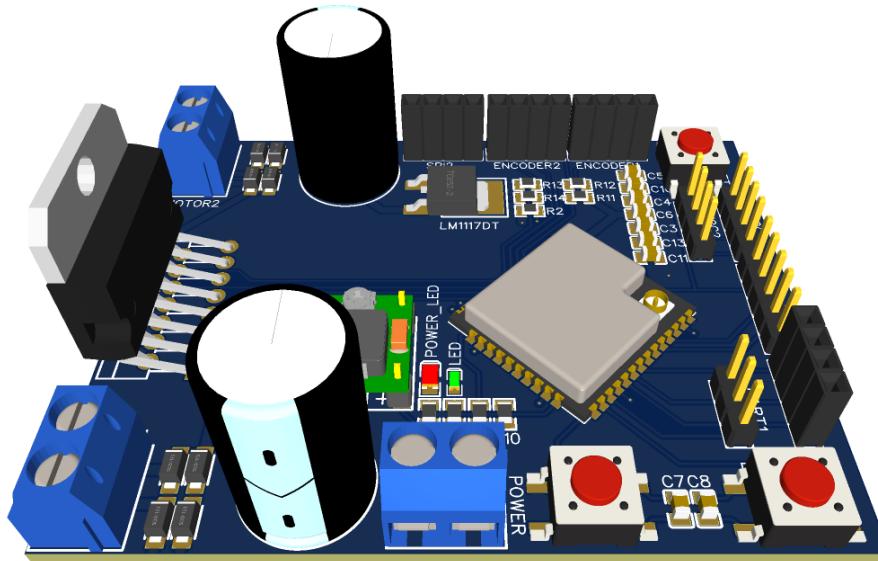
```
46 void mqttSendingTask(void* ptr) {
47     esp_mqtt_client_handle_t client = (esp_mqtt_client_handle_t)ptr;
48
49     int16_t pulses;
50     float voltage;
51     int16_t power;
52
53     while (true)
54     {
55         if(xQueueReceive(pulsesQueue, &pulses, 0)) {
56             std::string data = std::to_string(pulses);
57             esp_mqtt_client_publish(client, "edrive/value", data.c_str(),
58             → data.length(), 0, 0);
59         }
60
61         if(xQueueReceive(voltageQueue, &voltage, 0)) {
62             std::string data = std::to_string(voltage);
63             esp_mqtt_client_publish(client, "edrive/voltage", data.c_str(),
64             → data.length(), 0, 0);
65         }
66         if(xQueueReceive(powerQueue, &power, 0)) {
67             std::string data = std::to_string(power);
68             esp_mqtt_client_publish(client, "edrive/pwm_duty", data.c_str(),
69             → data.length(), 0, 0);
70     }
71     vTaskDelete(NULL);
```

Listing 4.12.5: Wysyłanie danych

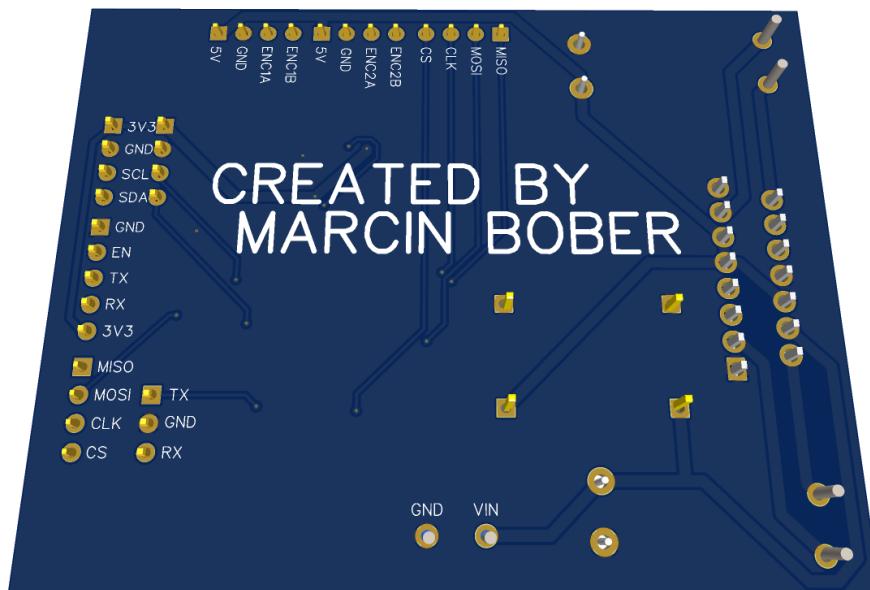
4.13 Gotowe urządzenie

4.13.1 Symulacja

Symulacje gotowego urządzenia zostały przedstawione na rysunkach 4.19 oraz 4.20.



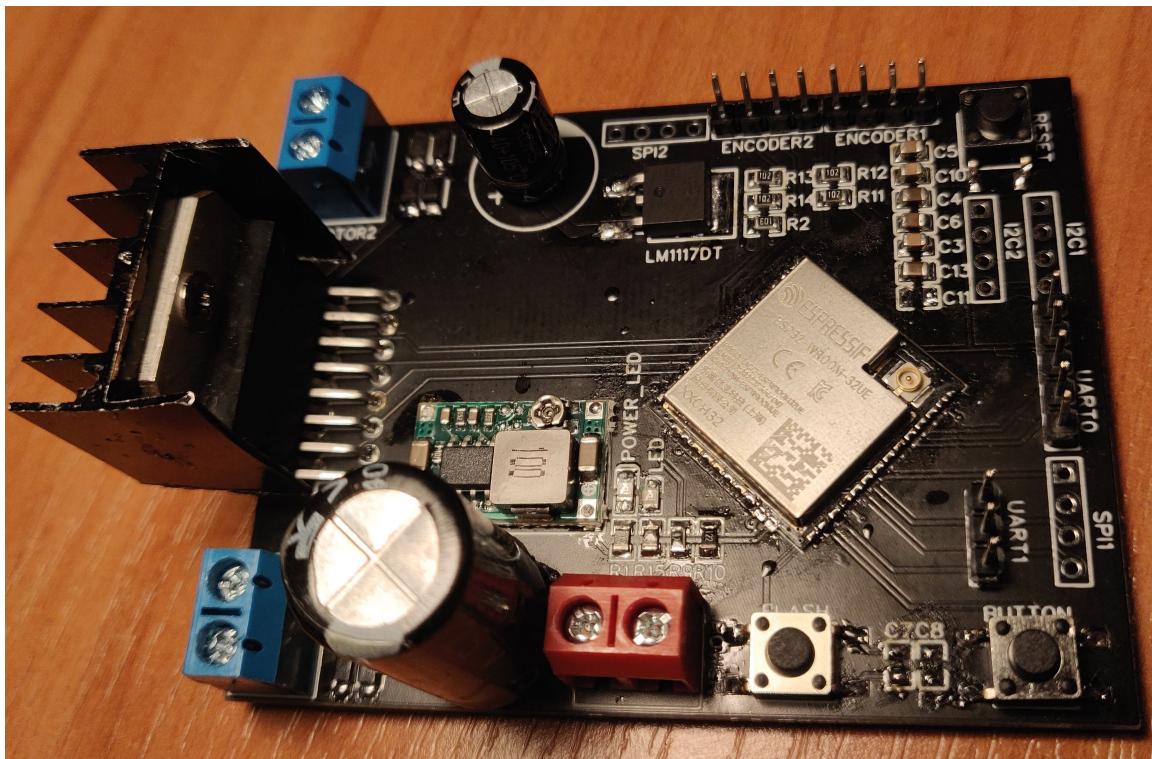
Rysunek 4.19: Symulacja gotowego urządzenia (wierzch)



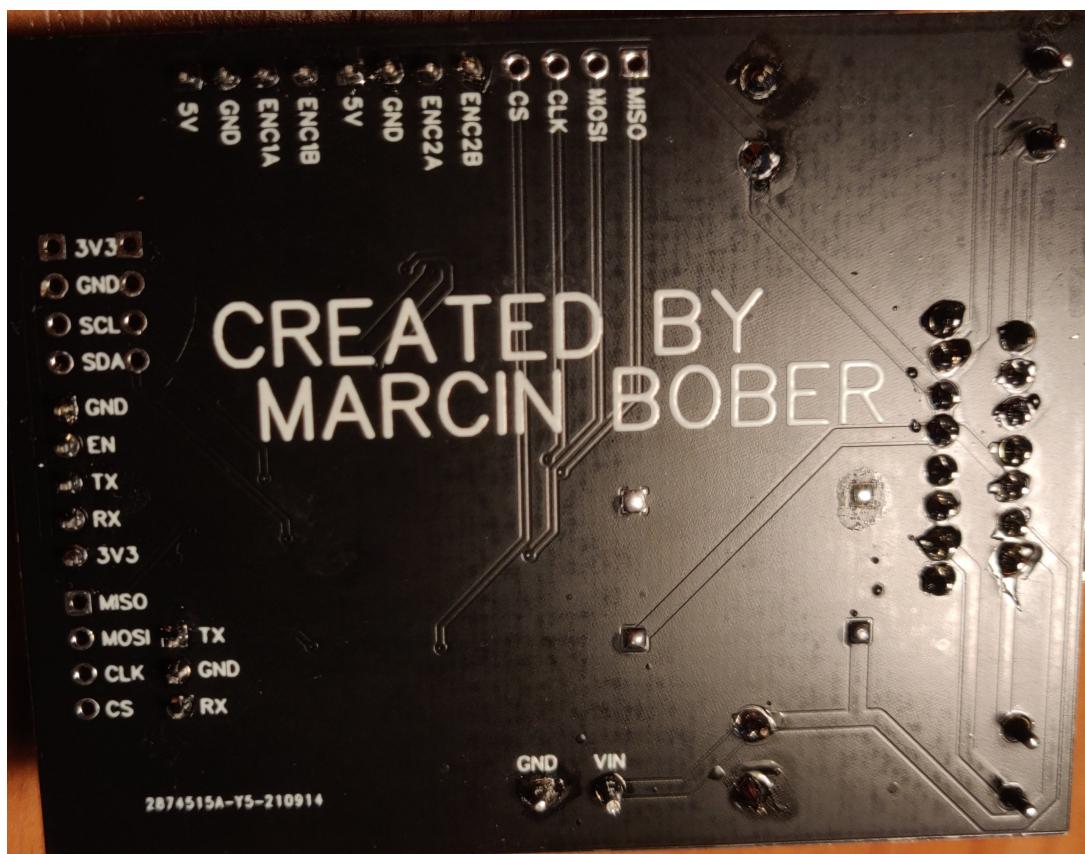
Rysunek 4.20: Symulacja gotowego urządzenia (spód)

4.13.2 Prototyp

Zdjęcia gotowego urządzenia zostały przedstawione na rysunkach 4.21 oraz 4.22.



Rysunek 4.21: Zdjęcie gotowego urządzenia (wierzch)



Rysunek 4.22: Zdjęcie gotowego urządzenia (spód)

Rozdział 5

Szczegółowy opis aplikacji dostępowej

5.1 Grupa docelowa

Aplikacja okienkowa została stworzona, aby zwiększyć dostępność systemu dla osób nie wprawionych w tematy związane z protokołem MQTT. Istnieje pełen przekrój uniwersalnych aplikacji zdolnych do komunikacji za pośrednictwem owego protokołu. Część z nich świetnie naddawałaby się do kooperowania z resztą przygotowanego systemu. Z drugiej strony wszystkie tego typu aplikacje są zazwyczaj nad wyraz rozbudowane oraz wymagają od użytkownika pewnej specyficznej wiedzy i doświadczenia. Z tego też powodu w ramach projektu powstała aplikacja dedykowana opisywanemu systemowi. Gwarantuje ona kompatybilność z resztą elementów zawartych w projekcie jednocześnie posiadając opcje w pełni wykorzystujące wszystkie funkcjonalności systemu. Należy także dodać że projektowana była z myślą o prostocie, intuicyjności w obsłudze i minimalizmie.

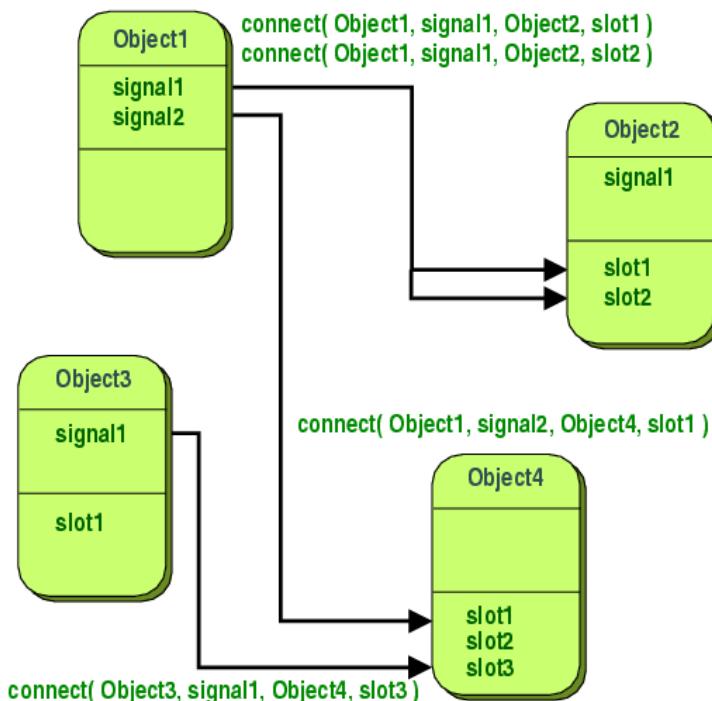
5.2 Wykorzystana technologia

Sprostanie założeniom projektu jest trudnym zadaniem bazując jedynie na standardowych bibliotekach języka C++, ponieważ jednym z kluczowych aspektów projektu jest wykonanie zaawansowanej aplikacji okienkowej. Z tego powodu podjęta została decyzja o wykorzystaniu wysokopoziomowego framework'a, który znaczaco uprościłby rozwój oprogramowania. Przykładem takiego rozwiązania jest bardzo popularny i nowoczesny framework QT. Jego niekomercyjna wersja jest w pełni darmowa i dystrybuowana na licencji otwarto źródłowej. Jego głównym przeznaczeniem jest tworzenie niezwykle rozbudowanych aplikacji okienkowych niewielkim nakładem pracy. Wspiera ono pełen przekrój systemów operacyjnych oraz architektur sprzętowych dzięki czemu program opierający się o tą technologię może być z powodzeniem przenoszony na różne urządzenia. Zdecydowanie najciekawszym aspektem tego oprogramowania jest niecodzienny system sygnałów i slotów. Według wielu początkujących programistów jest on nieintuicyjny. Jednakże wraz z korzystaniem z tego rozwiązania i towarzyszącym temu rosnącym doświadczeniem, pogląd ten jest niejednokrotnie rewidowany.

Poniżej przedstawione jest kilka fragmentów kodu użytego przy budowie aplikacji dostępowej wraz z opisami wyjaśniającymi wszystkie znajdujące się w nim zawiłości.

5.3 System sygnałów i slotów

Wprowadzenie tego rodzaju funkcjonalności rozegrało niebagatelne znaczenie w popularności framework'a QT. W przygotowanej na potrzeby opisywanego projektu aplikacji, system sygnałów i slotów pełni kluczową rolę w jej funkcjonowaniu. Na listingu 5.3.1 został umieszczony fragment programu odpowiadającego za przypisanie sygnałów produkowanych przez liczne elementy interfejsu użytkownika do specjalnie przygotowanych na tę potrzeby funkcji. Nie trudno sobie wyobrazić jak wiele pracy jest w stanie oszczędzić proste łączenie sygnałów ze slotami w porównaniu ze żmudną implementacją karkołomnych rozwiązań na własną rękę. Pozwala to także na omijanie sytuacji, w której nie jeden programista sięgnąłby po rozwiązanie jakim jest wielowątkowość. Sposób działania tego rozwiązania przedstawiony jest na rysunku 5.1.



Rysunek 5.1: System sygnałów i slotów [28]

```

18     connect(ui->actionConnect, &QAction::triggered, this, &MainWindow::actionConnect);
19     connect(ui->actionDisconnect, &QAction::triggered, this,
20             &MainWindow::actionDisconnect);
21     connect(&chartTimer, & QTimer::timeout, this, &MainWindow::drawData);
22     connect(mqtt, &QMqttClient::stateChanged, this,
23             &MainWindow::changeConnectionStatus);
24     connect(mqtt, &QMqttClient::errorChanged, this, &MainWindow::connectionError);
25     connect(ui->actionQuit, &QAction::triggered, this, &MainWindow::close);
26     connect(ui->stopButton, &QPushButton::clicked, this, &MainWindow::stop);
27
28     connect(ui->SetpointCheckBox, &QCheckBox::stateChanged, [this](int state) {
29         if(state == Qt::CheckState::Unchecked) this->chart->setSeriesVisible(0,
30             false);
31         else if(state == Qt::CheckState::Checked) this->chart->setSeriesVisible(0,
32             true);
33     });

```

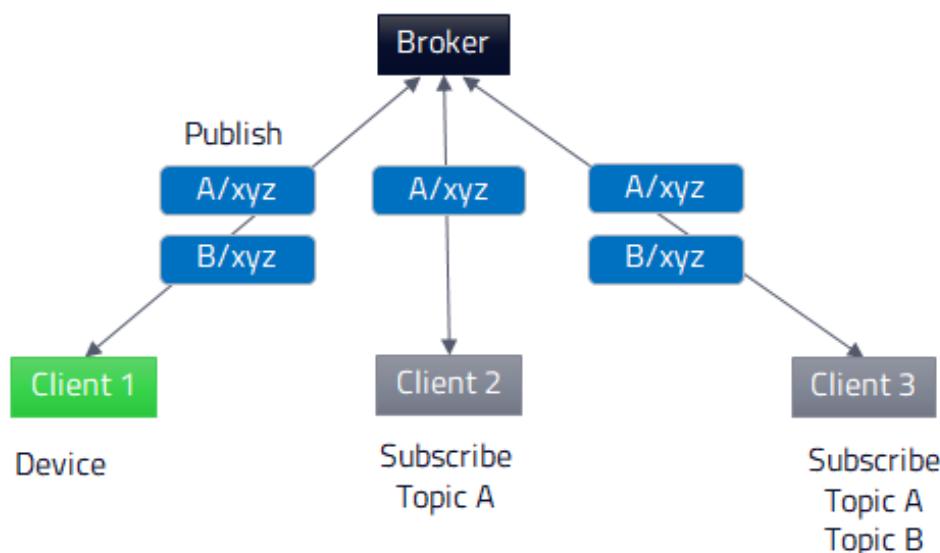
```
31     connect(ui->ValueCheckBox, &QCheckBox::stateChanged, [this](int state) {
32         if(state == Qt::CheckState::Unchecked) this->chart->setSeriesVisible(1,
33             false);
34         else if(state == Qt::CheckState::Checked) this->chart->setSeriesVisible(1,
35             true);
36     });
37
38     connect(ui->DutyCheckBox, &QCheckBox::stateChanged, [this](int state) {
39         if(state == Qt::CheckState::Unchecked) this->chart->setSeriesVisible(2,
40             false);
41         else if(state == Qt::CheckState::Checked) this->chart->setSeriesVisible(2,
42             true);
43     });
44 }
```

Listing 5.3.1: Użycie systemu sygnałów i slotów

5.4 Implementacja komunikacji

5.4.1 Łączenie z brokerem

Podobnie jak w framework'u użytym na poczet przygotowania wsadu do sterownika, tutaj również jest zdefiniowana fenomenalnie prosta w obsłudze abstrakcja. Niemniej jednak w celu utrzymania wysokiej czytelności produkowanego kodu została przygotowana nowa autorska klasa. Dziedziczy ona po klasie zawartej w module QtMqtt. Jej celem jest rozszerzenie klasy bazowej o dodatkową funkcjonalność, która sprawi że inicjalizacja i obsługa połączenia będą jeszcze wygodniejsze. Jako argumenty wywołania należy podać adres i port serwera, a następnie nazwę użytkownika oraz hasło. Cała reszta procesu została zgrabnie ukryta pod warstwą abstrakcji. Omawiany kod przedstawiony został w listingu 5.4.1.



Rysunek 5.2: Schemat działania modułu Qt MQTT [22]

```

21  /**
22   * Connect to mqtt server
23   * @param[in] hostname Server address
24   * @param[in] port Server port
25   * @param[in] username Username
26   * @param[in] password Pusername
27  */
28 void Mqtt::connectToHost(const QString &hostname, const uint16_t port, const QString
29   &username, const QString &password) {
30   this->setHostname(hostname);
31   this->setPort(port);
32   this->setUsername(username);
33   this->setPassword(password);
34   this->QMqttClient::connectToHost();
}

```

Listing 5.4.1: Nawiązanie połączenia z brokerem

5.4.2 Obsługa zdarzeń

Kolejną analogią znaną z frameworka ESP-IDF jest obsługa zdarzeń przychodzących z modułu MQTT. Wysyła on sygnał połączony z funkcją widoczną w listingu 5.4.2 zdający raport o stanie połączenia. Jest on wykorzystywany do aktywowania interfejsu użytkownika, który domyślnie przyjmuje status nieaktywnego. Ponadto jest to wyzwalacz do subskrybowania uprzednio zdefiniowanych tematów.

```

102 /**
103 * Change connection status
104 */
105 void MainWindow::changeConnectionStatus(QMqttClient::ClientState state) {
106     switch (state)
107     {
108         case QMqttClient::Connected: {
109             enableUi(true);
110             ui->actionConnect->setVisible(false);
111             ui->actionDisconnect->setVisible(true);
112             ui->actionDisconnect->setEnabled(true);
113             this->subscribe();
114             break;
115         }
116
117         case QMqttClient::Connecting:
118         case QMqttClient::Disconnected:
119             enableUi(false);
120             ui->actionConnect->setVisible(true);
121             ui->actionConnect->setEnabled(true);
122             ui->actionDisconnect->setVisible(false);
123             this->unsubscribe();
124             break;
125     }
126 }
127

```

Listing 5.4.2: Obsługa zdarzeń

5.4.3 Subskrybowanie tematów

Subskrybowanie tematów jest dość długą funkcją. Wbrew pozorom nie jest ona skomplikowana. Jej zadaniem jest zasubskrybować jedynie cztery tematy. Z tego też powodu wszystko jest powielone czterokrotnie. To właśnie wpływa w znaczący sposób na jej objętość. Dla każdego tematu wywoływana jest metoda na obiekcie MQTT, która przyjmuje temat oraz parametr QoS.

QoS z angielskiego "Quality of Service" określa poziom istotności wiadomości. W tym przypadku użyta została wartość 0. Oznacza to że nie ma gwarancji otrzymania wiadomości. Taka kolej rzeczy nie wydaje się być z żaden sposobem problematyczna ze względu na cykliczność z jaką otrzymywane są ramki. Zgubienie jednej wartości nie musi być istotne w momencie, gdy są one transmitowane z ponadprzeciętnie dużą częstotliwością. W takim systemie pierwszoplanowa jest wydajność i przepustowość. Pozostałe wartości które są możliwe do ustawienia to 1 i 2. Pierwsza z nich sprawia że mamy pewność otrzymania wiadomości, natomiast druga gwarantuje jednokrotność tego zdarzenia.

Produktem wywołania tej metody będzie obiekt typu QMqttSubscription, który następnie należy przy pomocy systemu sygnałów i slotów, połączyć z odpowiednią funkcją.

Funkcja ta jedynie skleja otrzymaną wartość z uprzednio przygotowanym ciągiem znaków i bezpośrednio wyświetla ją w interfejsie użytkownika. Przykład takiej funkcji dostępny jest w listingu 5.4.3.

Na samym końcu znajduje się jeszcze wywołanie funkcji ustawiającej wartości domyślne dla wyżej wymienionych tematów, tak aby można było je umieścić w interfejsie. Cała ta procedura powtórzona jest cztery razy dla czterech różnych tematów. Odnosi się do niej listing 5.4.4.

```

245 /**
246 * Set motor revolutions value
247 * @param[in] value Value to set
248 */
249 void MainWindow::setValue(int value) {
250     engine->setValue(value);
251     QString text(QString::number(tickToRPM(value)));
252     text += " RPM";
253     ui->valueLabelValue->setText(text);
254 }
```

Listing 5.4.3: Przetwarzanie odebranych danych

```

186 // read subscriptions
187 auto valueSubscription = mqtt->subscribe(QMqttTopicFilter("edrive/value"), 0);
188 connect(valueSubscription, &QMqttSubscription::messageReceived,
189     [this](QMqttMessage msg) {
190         this->setValue(msg.payload().toInt());
191     });
192
193 auto voltageSubscription = mqtt->subscribe(QMqttTopicFilter("edrive/voltage"), 0);
194 connect(voltageSubscription, &QMqttSubscription::messageReceived,
195     [this](QMqttMessage msg) {
196         this->setVoltage(msg.payload().toFloat());
197     });
198
199 auto pwmDutySubscription = mqtt->subscribe(QMqttTopicFilter("edrive/pwm_duty"),
200     0);
201 connect(pwmDutySubscription, &QMqttSubscription::messageReceived,
202     [this](QMqttMessage msg) {
203         this->setPwmDuty(msg.payload().toInt());
204     });
205
206 auto currentSubscription = mqtt->subscribe(QMqttTopicFilter("edrive/current"), 0);
207 connect(currentSubscription, &QMqttSubscription::messageReceived,
208     [this](QMqttMessage msg) {
209         this->setTorque(msg.payload().toFloat());
210     });
211
```

Listing 5.4.4: Subskrybowanie tematów

5.4.4 Wysyłanie danych

Wysyłanie danych polega na odbieraniu wartości prosto z interfejsu i przekazywaniu ich do metody obiektu MQTT. Połączone są więc konkretne sygnały elementów interfejsu z daną

lambdą. W roli przykładu, sygnał suwaka ustawiającego wartość zadaną łączymy z lambdą w której wyłuskujemy wartość suwaka. Następnie jest ona zapisywana jako tablica bajtów i publikowana. Dodatkowo jest wyświetlana w postaci liczbowej w interfejsie. Przykład implementacji znajduje się w listingu 5.4.5.

```
208 // write subscription
209 connect(ui->SetpointSlider, &QAbstractSlider::valueChanged, [this](int value) {
210     QByteArray array;
211     array.setNum(value);
212     mqtt->publish(QMqttTopicName("edrive/setpoint"), array);
213     this->setSetpoint(value);
214 });
215
216 connect(ui->KpSpinBox, QOverload<int>::of(&QSpinBox::valueChanged), [this](int i)
217     {
218     QByteArray array;
219     array.setNum(i);
220     mqtt->publish(QMqttTopicName("edrive/kp"), array);
221 });
222
223 connect(ui->KiSpinbox, QOverload<int>::of(&QSpinBox::valueChanged), [this](int i)
224     {
225     QByteArray array;
226     array.setNum(i);
227     mqtt->publish(QMqttTopicName("edrive/ki"), array);
228 });
229
230 connect(ui->KdSpinbox, QOverload<int>::of(&QSpinBox::valueChanged), [this](int i)
231     {
232     QByteArray array;
233     array.setNum(i);
234     mqtt->publish(QMqttTopicName("edrive/kd"), array);
235 });
236
237 this->setDefaultValues();
```

Listing 5.4.5: Wysyłanie danych

5.5 Wykresy

5.5.1 Definicja klasy

Jedną z najważniejszych funkcjonalności aplikacji jest estetyczna prezentacja danych pobranych ze sterownika. Podjęta została decyzja o wykorzystaniu w tym celu prostych wykresów. Framework QT zapewnia moduł QtCharts, który jest zestawem prostych w użyciu komponentów graficznych niezbędnych do stworzenia estetycznych wykresów. Zostało to uszczególnione poprzez wykonanie nowej klasy dziedziczącej po klasie QChart dostępnej w opisywanym module. W swoim konstruktorze tworzy ona wcześniej zdefiniowany wykres z kilkoma seriami danych. Oprócz tego stworzona klasa posiada zdefiniowane metody do czyszczenia zawartości wykresu, dodawania nowych punktów oraz ukrywania nieużywanych serii danych. Kod jest dostępnym w listingu 5.5.1.

```
17  /**
18   * @brief Chart class
19  */
20  class Chart : public QChart
21 {
22     Q_OBJECT
23 public:
24     Chart(const uint pointCount, QGraphicsItem *parent = nullptr, const
25           Qt::WindowFlags wFlags = {});
26     void clear();
27
28 public slots:
29     void addPoint(const int* values);
30     void setSeriesVisible(const int series, const bool enable);
31
32 private:
33     QSplineSeries** series;
34     QValueAxis *axisX;
35     QValueAxis *axisY;
36     uint pointCount;
37 };
```

Listing 5.5.1: Definicja klasy wykresów

5.5.2 Dodawanie punktów do wykresu

Implementacja rysowania wykresów opublikowana przez twórców framework'a QT pozostawia wiele do życzenia. W szczególności kwestią wydajności nie została należycie rozpatrzona czego skutkiem jest niewymiernie wysokie zużycie zasobów komputera podczas renderowania tych obiektów. Zastosowana strategia przeciwdziałająca temu zjawisku zakłada skońzoną liczbę punktów obecnych jednocześnie na wykresie. Z tego też powodu został zaimplementowany trywialny algorytm pozbywający się nadmiarowych obiektów.

Kod przedstawiony w listingu 5.5.2 ma za zadanie włączać kolejne dane do wykresu. Uprzednio musi zostać wyznaczona pozycja punktu na osi X. Następnie dla każdej z 3 serii sprawdzana jest ilość znajdujących się na niej pomiarów. W razie przekroczenia ustalonego limitu, nadmiar elementów jest usuwany z początku kolejki, aby następnie dodać na jej końcu dodać nowy punkt. Przed zakończeniem tej metody wykonywana

jest kalkulacja położenia najbardziej oddalonych od siebie obiektów, aby móc dopasować zakresy wyświetlania wykresu.

```

71  /**
72   * Add point to the chart
73   */
74  void Chart::addPoint(const int* values)
75  {
76      static int xPosition;
77
78      // calculate the position for X in the chart
79      xPosition++;
80
81      for (size_t i = 0; i < 3; i++)
82      {
83          if(this->series[i]->points().size() > this->pointCount)
84              → this->series[i]->remove(0); // remove redundant points
85          this->series[i]->append(xPosition, values[i]); // add new points
86      }
87
88      auto dataCount = (this->series[0])->count();
89
90      if(dataCount) {
91          auto min = this->series[0]->at(0).x();
92          auto max = min + this->pointCount;
93          this->axisX->setRange(min, max);
94      }
95  }
```

Listing 5.5.2: Dodawanie danych do wykresu

5.5.3 Czyszczenie wykresu

Usuwanie danych w wykresów nie należy do najbardziej skomplikowanych. Wystarczy jedynie na każdej z serii wywołać metodę pozbywającą się wszystkich punktów z bufora. Kod znajduje się w listingu 5.5.3.

```

97  /**
98   * Clear chart
99   */
100 void Chart::clear() {
101     for (size_t i = 0; i < 3; i++)
102     {
103         this->series[i]->clear();
104     }
105 }
```

Listing 5.5.3: Usuwanie wszystkich danych z wykresu

5.5.4 Ukrywanie serii

Metoda zmiany widoczności danej serii jest jedynie makrem korzystającym z odziedziczo-nych metod. Dostępna jest pod listgiem 5.5.4.

```
108  /**
109   * Set series visibility
110  */
111 void Chart::setSeriesVisible(const int series, const bool enable)
112 {
113     this->series[series]->setVisible(enable);
114 }
```

Listing 5.5.4: Zmiana widoczności serii

5.6 Abstrakcja silnika

5.6.1 Definicja klasy silnika

W celu ujednoliconego zarządzania danymi odbieranymi i wysyłanymi do sterownika, została utworzona specjalna klasa, która przechowuje wartości z nim związane. Zdecydowanie zwiększa to czytelność kodu ponieważ zmienne dotyczące na przykład prędkości obrotowej nie są porozrzucane po całym programie. Co więcej, zostały stworzone odpowiednie metody które niwelują niebezpieczeństwo przypadkowego nadpisania jakiejś zmiennej. Implementacja została przedstawiona w listingu 5.6.1. Przykładowe użycie metody można zobaczyć w listingu 5.4.3.

```

3  class Engine
4  {
5      public:
6          Engine();
7          ~Engine();
8
9          void setValue(const int value);
10         void setVoltage(const int voltage);
11         void setSetpoint(const int setpoint);
12         void setPwmDuty(const int duty);
13         void setTorque(const float torque);
14         int getValue();
15         int getVoltage();
16         int getSetpoint();
17         int getPwmDuty();
18         float getTorque();
19
20     private:
21         int value, voltage, setpoint, pwmDuty;
22         float torque;
23     };

```

Listing 5.6.1: Klasa abstrakcji silnika

5.6.2 Przeliczanie wartości obrotów

Konwersja liczby otrzymanej impulsów na ilość wykonanych obrotów wymaga od nas posiadania odpowiednich informacji na temat enkodera. Częstotliwości pomiaru impulsów przez sterownik to 100Hz. Oznacza to czas pomiędzy pomiarami równy:

$$T = \frac{1}{f} = \frac{1}{100\text{Hz}} = 10\text{ms}$$

Wynika z tego że ilość obrotów należy pomnożyć stukrotnie, aby uzyskać ilość impulsów na sekundę. Następne mnożenie przez 60 doprowadzi do otrzymania ilości impulsów w okresie jednej minuty. Nie można zapomnieć że wartość impulsów jest czterokrotnie większa ze względu na konfigurację liczników (patrz 4.8.1). Z tego też powodu niezbędne jest podzielenie wyniku przez 4. Ostatecznie całość należy podzielić przez ilość impulsów

przypadających na jeden obrót wałka wychodzącego z przedkładani (224.4PPR/RPM). W ten sposób możliwe jest uzyskanie odpowiedniego przelicznika.

```
27 const float RPM_Multiplier = (100 * 60) / 224.4 / 4;  
28  
29  
30 /**
31 * Convert encoder impulses to RPM
32 * @param[in] tick impulses
33 */
34 inline int tickToRPM(int tick) {
35     // * 100 -> tick / s
36     // * 60 -> tick / min
37     /// 224.4 -> revolution / min
38     /// / 4 (encoder)
39     return tick * RPM_Multiplier;
40 }
```

Listing 5.6.2: Przeliczanie impulsów na obroty

5.7 Interfejs użytkownika

Interfejs zawiera wszystkie elementy niezbędne do wygodnej obsługi urządzenia.



Rysunek 5.3: Interfejs użytkownika

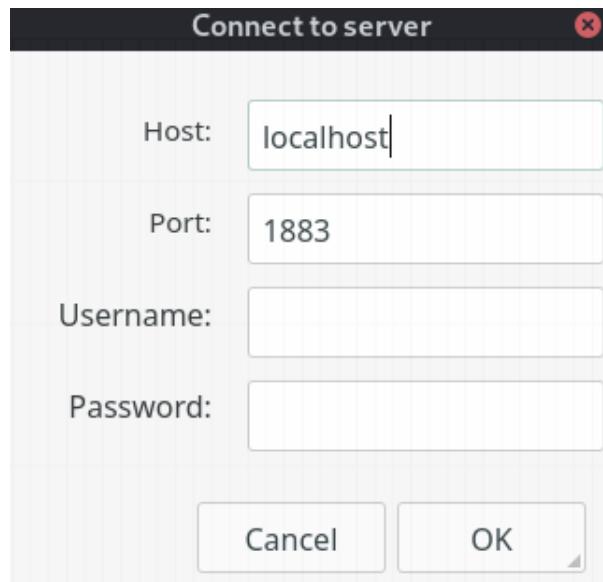
5.7.1 Górná belka

W górnej części aplikacji znajduje się menu kontekstowe. Pozwala ono na wywołanie okna inicjalizującego połączenie z serwerem (patrz 5.4). Do nawiązania pomyślnego połączenia wymagane jest umieszczenie poprawnego adresu oraz portu brokera MQTT. W przypadku dezaktywowanej opcji przyjmowania anonimowych użytkowników wymagane jest także podanie ważnych danych autoryzacyjnych.

5.7.2 Panel parametrów

Dolna część okna podzielona jest na trzy części.

- Panel parametrów regulatora.
- Panel widoczności wykresów.
- Panel z aktualnymi danymi.



Rysunek 5.4: Okno nawiązywania połączenia

Pierwsza z nich udostępnia interfejs do dynamicznej zmiany nastaw regulatora PID. Pozwala to bardzo szybko uzyskać zadowalające nastawy, niezbędne do prawidłowego działania silnika.

Środkowy panel zawiera pola wielokrotnego wyboru służące do ukrywania niepotrzebnych danych na wykresie. Dzięki tej funkcjonalności możliwe jest uniknięcie nakładanie się na siebie wykresów, co znaczenie zwiększa komfort i czytelność prezentowanych informacji.

Ostatni panel, umieszczony z prawej strony zawiera zrób danych zebranych z urządzenia wykonawczego. Można wyróżnić w śród nich:

- napięcie zasilanie urządzenia,
- ustawiony punkt pracy,
- aktualną prędkość obrotową silnika,
- wartość procentową wypełnienia PWM.

5.7.3 Graf

Najatrakcyjniejszym wizualnie elementem jest wykres umieszczony w centrum okna. Wizualizuje on trzy zmienne:

- zadaną wartość obrotów silnika,
- aktualną wartość obrotów silnika,
- wypełnienie PWM.

Obecność wykresu umożliwia wizualną ocenę jakości sterowania.

Rozdział 6

Podsumowane

Powysza praca jest przykładem rozwiązania problemu budowy projektu systemu sensorycznego opartego na protokole MQTT. W pracy znalazły się jedynie najważniejsze fragmenty kodu źródłowego oprogramowania niezbędnego do budowy systemu. Podkreślają one kwestie szczególnie istotne i warte omówienia. Całość kodu wraz z wylewną dokumentacją oraz projekt płytki drukowanej wykonanego urządzenia znajdują się na repozytorium projektu.

Zastosowanie rozwiązań otwartoźródołych znaczaco przyczyniło się do zwiększenia dostępności przedstawionego rozwiązania. Co więcej, wykorzystanie proponowanego systemu i wdrożenie go do środowiska komercyjnego również nie będzie stanowiło dużego obciążenia finansowego ze względu na brak konieczności zakupienia żadnych licencji.

Najważniejsze aspekty niezbędne przy realizacji projektu to:

- zdefiniowanie ogólnych założeń systemu i struktur danych,
- zaprojektowanie płytki drukowanej urządzenia i wykonanie prototypu,
- konfiguracja i konteneryzacja brokera MQTT,
- oprogramowanie prototypu urządzenia,
- stworzenie aplikacji dostępowej we framework'u QT,
- stworzenie dokumentacji kodu.

Dogłębna analiza działania urządzenia i symulowanie jego zachowania pozwoliły na stworzenie prototypu wolnego od wad. Sekcja zasilania bezproblemowo filtryuje wszelkie zakłócenia generowane przez szczotki silnika prądu stałego. Może się ona pochwalić również bardzo szerokim zakresem obsługiwanej napięcia wejściowego. Dzięki temu urządzenie działa stabilnie i przewidywalnie. Nie ma problemu z utratą połączenia z serwerem czy restartami podczas pracy. System błyskawicznie reaguje na zmiany zadanej wartości obrotowej silnika. Opóźnienia transmisji i przetwarzania danych są na tyle małe wydają się być niezauważalne przez człowieka.

Literatura

- [1] Qt Development Frameworks: Dokumentacja biblioteki Qt
<https://doc.qt.io/qt-5/>
Dostęp 14.09.2021
- [2] STMicroelectronics: Dokumentacja L298H
https://www.sparkfun.com/datasheets/Robotics/L298_H_Bridge.pdf
Dostęp 14.09.2021
- [3] Espressif Systems: Dokumentacja mikrokontrolera ESP32 https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf
Dostęp 14.09.2021
- [4] Espressif Systems: Dokumentacja modułu ESP32WROOM32UE https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf
Dostęp 14.09.2021
- [5] OASIS: Dokumentacja standardu MQTT 5.0
<https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf>
Dostęp 14.09.2021
- [6] V.P. Eloranta, J. Koskinen, M. Leppanen, V. Reijonen: Designing Distributed Control Systems: A Pattern Language Approach, Wiley, 2014.
- [7] Kenneth Flamm: Measuring Moore's Law: Evidence from Price, Cost, and Quality Indexes
<https://www.imf.org/-/media/Files/Conferences/2017-stats-forum/session-6-kenneth-flamm.ashx>
Dostęp 15.09.2021
- [8] Espressif Systems: Dokumentacja ESP-IDF
<https://docs.espressif.com/projects/esp-idf/en/latest/esp32/>
Dostęp 15.09.2021
- [9] Amazon Web Services: Dokumentacja FreeRTOS
<https://www.freertos.org/a00106.html>
Dostęp 15.09.2021
- [10] Wikipedia: Szereg wartości
https://pl.wikipedia.org/wiki/Szereg_warto%C5%9Bci
Dostęp 17.09.2021
- [11] Wikipedia: Dzielnik rezystorowy
https://en.wikipedia.org/wiki/Voltage_divider
Dostęp 17.09.2021

- [12] Robert W. Doran: The Gray Code
http://www.jucs.org/jucs_13_11/the_gray_code/jucs_13_11_1573_1597_doran.pdf
Dostęp 27.09.2021
- [13] Siemens: Enkodery inkrementalne
<https://publikacje.siemens-info.com/pdf/56/Motion%20Control%20-%20Uk%C5%82ad%20pomiarowy.pdf>
Dostęp 17.09.2021
- [14] Espressif Systems: Producent SoC
<https://www.espressif.com> Dostęp 19.10.2021
- [15] Robert McDowall: Fundamentals of HVAC Control Systems
- [16] Texas Instruments: Symmetric PWM Outputs Generation with the TMS320C14 DSP
<https://www.ti.com/lit/an/spra278/spra278.pdf> Dostęp 20.10.2021
- [17] FreeRTOS FAQ: Context Switch Times
<https://www.freertos.org/FAQMem.html#ContextSwitchTime> Dostęp 8.11.2021
- [18] Lawrence Williams: Round Robin Scheduling Algorithm
<https://www.guru99.com/round-robin-scheduling-example.html> Dostęp 8.11.2021
- [19] Strona domowa projektu Mosquitto
<https://mosquitto.org/> Dostęp 22.11.2021
- [20] Strona domowa projektu Docker
<https://www.docker.com/> Dostęp 22.11.2021
- [21] Bruno Siciliano, Oussama Khatib: Springer Handbook of Robotics
- [22] Schemat działania modułu Qt MQTT
<https://doc.qt.io/QtMQTT/mqtt-overview.html>
Dostęp 29.09.2021
- [23] Topologia procesora z użyciem FreeRTOS
<https://www.freertos.org/2020/02/simple-multicore-core-to-core-communication-using.html>
Dostęp 29.09.2021
- [24] Schemat podłączenia mostka H
<https://forbot.pl/forum/topic/16-teoria-mostek-h-h-bridge-kompendium-dla-robotyka>
Dostęp 29.09.2021
- [25] Schemat wykorzystanego silnika
<https://www.dfrobot.com/product-1619.html>
Dostęp 29.09.2021
- [26] Schemat wykorzystanego mostka
https://www.sparkfun.com/datasheets/Robotics/L298_H_Bridge.pdf
Dostęp 29.09.2021

- [27] Schemat działania protokołu MQTT
<https://en.wikipedia.org/wiki/MQTT>
Dostęp 29.09.2021
- [28] System sygnałów i slotów
<https://doc.qt.io/qt-5/signalsandslots.html>
Dostęp 29.09.2021
- [29] Wykorzystana jednostka centralna
<https://www.digikey.pl/product-detail/pl/espressif-systems/ESP32-WROOM-32U-16MB/1904-1028-1-ND/9381737>
Dostęp 29.09.2021
- [30] Przebiegi sygnału PWM
<https://developer.android.com/things/sdk/pio/pwm>
Dostęp 20.10.2021
- [31] Możliwe stany procesów w FreeRTOS
<https://microcontrollerslab.com/wp-content/uploads/2017/07/FreeRTOS-tasks-state.png>
Dostęp 8.11.2021

Spis rysunków

2.1	Elementy systemu	4
2.2	Schemat wymiany danych w systemie	6
3.1	Schemat działania protokołu MQTT [27]	8
4.1	Wykorzystana jednostka centralna [29]	12
4.2	Topologia procesora z użyciem FreeRTOS [23]	12
4.3	Możliwe stany procesów w FreeRTOS [31]	13
4.4	Diagram wątków	15
4.5	Ogólny schemat mostka H [24]	19
4.6	Schemat wykorzystanego mostka H [26]	20
4.7	Schemat podłączenia mostka H [24]	20
4.8	Schemat wykorzystanego silnika [25]	21
4.9	Wygląd PCB	22
4.10	Schemat płytki PCB (CPU+PSU)	23
4.11	Schemat płytki PCB (mostek + enkodery)	24
4.12	Schemat płytki PCB (interfejsy)	25
4.13	Pomiar impulsów z oscyloskopu	27
4.14	Przebieg modulacji PWM [30]	29
4.15	Przebieg modulacji PWM [16]	30
4.16	Diagram aktywności procesu PID	36
4.17	Pomiary danych ADC do napięcia zasilania	39
4.18	Diagram aktywności dokonywania pomiarów	41
4.19	Symulacja gotowego urządzenia (wierzch)	46
4.20	Symulacja gotowego urządzenia (spód)	46
4.21	Zdjęcie gotowego urządzenia (wierzch)	47
4.22	Zdjęcie gotowego urządzenia (spód)	47
5.1	System sygnałów i slotów [28]	49
5.2	Schemat działania modułu Qt MQTT [22]	51
5.3	Interfejs użytkownika	60
5.4	Okno nawiązywania połączenia	61

Spis listingów

3.4.1 Utworzenie instancji brokera MQTT w kontenerze	9
3.4.2 Minimalna konfiguracja brokera	10
3.4.3 Dodawanie pierwszego użytkownika	10
4.4.1 Struktura konfiguracyjna wątku	15
4.4.2 Lista wątków	16
4.4.3 Konfiguracja wątków	17
4.4.4 Struktura konfiguracyjna kolejki	17
4.4.5 Konfiguracja kolejek FIFO	18
4.4.6 Inicjalizacja kolejek FIFO	18
4.4.7 Makra do uchwytów kolejek FIFO	18
4.8.1 Konfiguracja licznika impulsów	28
4.9.1 Inicjalizacja PWM	30
4.9.2 Zmiana wypełnienia PWM	31
4.10.1 Pobieranie wartości i czyszczenie rejestrów licznika	32
4.10.2 Pętla regulatora PID	33
4.10.3 Normalizacja wartości regulatora	33
4.10.4 Proces wykonywanie PID	35
4.11.1 Konfiguracja przetwornika ADC	37
4.11.2 Wyzwalanie pomiaru i przeliczanie wartości	40
4.12.1 Konfiguracja połączenia MQTT	42
4.12.2 Subskrybowanie niezbędnych tematów	43
4.12.3 Odbieranie danych	43
4.12.4 Obsługa sygnałów	44
4.12.5 Wysyłanie danych	45
5.3.1 Użycie systemu sygnałów i slotów	50
5.4.1 Nawiązanie połączenia z brokerem	51
5.4.2 Obsługa zdarzeń	52
5.4.3 Przetwarzanie odebranych danych	53
5.4.4 Subskrybowanie tematów	53
5.4.5 Wysyłanie danych	54
5.5.1 Definicja klasy wykresów	55
5.5.2 Dodawanie danych do wykresu	56
5.5.3 Usuwanie wszystkich danych z wykresu	56
5.5.4 Zmiana widoczności serii	57
5.6.1 Klasa abstrakcji silnika	58
5.6.2 Przeliczanie impulsów na obroty	59