

Politechnika Poznańska
Wydział Elektryczny
Instytut Robotyki i Inteligencji Maszynowej

Praca dyplomowa inżynierska

**OPRACOWANIE I WYKONANIE STANOWISKA
LABORATORYJNEGO Z TZW. WAHADŁEM LOTNICZYM
(AEROPENDULUM) (PRACA ZESPOŁOWA)**

Patryk Pilarski
Bartosz Podkański
Jakub Walkowski

Promotor
dr hab. inż. Tomasz Pajchrowski

Poznań, 2021 r.

Tutaj znajdzie się karta pracy dyplomowej;
oryginał, wstawiamy do wersji dla archiwum PP, w pozostałych kopiiach
wstawiamy ksero.

Podziękowania:

Chcielibyśmy podziękować naszemu promotorowi dr hab. inż. Tomaszowi Pajchrowskiemu za pomoc w realizacji pracy dyplomowej, nieustanną dyspozycyjność oraz opiekę promotorską.

Patryk Pilarski

Bartosz Podkański

Jakub Walkowski

Serdeczne podziękowania dla Pana mgr. inż. Adriana Wójcika za poświęcony czas, mertoryczne wspracie, cierpliwość i zaangażowanie w trakcie reazlizacji pracy dyplomowej.

Patryk Pilarski

Bartosz Podkański

Jakub Walkowski

Spis treści

1 Wstęp	1
1.1 Opis zagadnienia poruszonego w pracy	1
1.2 Cel pracy	1
1.3 Zawartość pracy	1
2 Podstawy teoretyczne	2
2.1 Silnik BLDC	2
2.1.1 Budowa silnika	2
2.1.2 Sterowanie silnikiem	3
2.2 Komunikacja	4
3 Budowa stanowiska	5
3.1 Poszukiwanie i projektowanie prototypu	5
3.1.1 Przegląd gotowych rozwiązań	5
3.1.2 Prototypy	7
3.2 Konstrukcja elektryczna	8
3.2.1 Silnik	8
3.2.2 Sterownik	9
3.2.3 Mikrokontroler/raspberry	10
3.2.4 Pierścień ślizgowy	10
3.2.5 Zasilacze	11
3.2.6 Instalacja elektryczna	12
3.2.7 Zabezpieczenia	15
3.3 Konstrukcja mechaniczna	17
3.3.1 Rama	18
3.3.2 Oś obrotu	18
3.3.3 Elementy personalizowane, integracja elementów	19
Elementy drukowane	19
Elementy wycięte laserowo	22
Rozwiązania techniczne i łączenie elementów	22
3.3.4 Osłona	23
3.3.5 Skrzynka elektryczna	24
4 Budowa modelu matematycznego	26
4.1 Identyfikacja parametrów	26
4.2 Model matematyczny na podstawie równań różniczkowych	26
4.3 Testy w pętli otwartej	31
5 Oprogramowanie	35
5.1 Raspberry Pi	36

5.1.1	Komunikacja za pośrednictwem MCP - Motor Control Protocol	40
	Konstrukcja ramki startowej	40
	Konstrukcja ramki odpowiedzi	45
	Przykładowa ramka startowa i ramka odpowiedzi	48
	Wysyłanie i odbieranie ramek	50
	API biblioteki	52
5.1.2	Serwer UDP	54
5.2	STEVAL-SPIN 3201	58
5.3	Enkoder AS5600	61
5.4	System wizyjny	64
6	Sterowanie w pętli zamkniętej, testy	70
6.1	Aparat regulacji	70
6.2	Wykorzystanie modelu w pętli zamkniętej	71
6.3	Implementacja regulatora na stanowisku	72
6.3.1	Wysoka dynamika obiektu	72
6.3.2	Niska dynamika obiektu	72
7	Podsumowanie	74
A	Opis zawartości płyty DVD	75
	Spis tabel	76
	Spis rysunków	77
	Bibliografia	79

Streszczenie

Praca polegała na zaprojektowaniu stanowiska laboratoryjnego, zawierającego tzw. wahadło lotnicze. Stanowisko to powstało z myślą o zajęciach laboratoryjnych, odbywających się w ramach działań dydaktycznych Politechniki Poznańskiej. Stworzenie stanowiska polegało na wykonaniu konstrukcji mechanicznej, układów elektrycznych a także sporządzeniu odpowiedniego oprogramowania. Konstrukcja mechaniczna, jak i elektryczna spełnia warunek bezpieczeństwa, wymagany w przypadku prowadzenia na stanowisku zajęć dydaktycznych. Oprogramowanie przeznaczone na wykorzystane w projekcie Raspberry Pi 4B pozwala na komunikację ze wszystkimi elementami stanowiska. W skład tych elementów wchodzą: enkoder magnetyczny AS5600, będący czujnikiem wychylenia wahadła, silnik BLDC oraz sterownik STEVAL-SPIN 3201 odpowiedzialny za pomiar prędkości silnika, a także regulację jego prędkości oraz kamera pełniąca rolę sprzężenia wizyjnego przy pomiarze kąta. Oprogramowanie zostało również przystosowane do komunikacji z oprogramowaniem Matlab, przy użyciu protokołu sieciowego UDP. W pracy został poruszony także fizyczny aspekt wahadła lotniczego, powstawał jego model matematyczny i został zaimplementowany w programie Simulink. Wykonane testy potwierdziły poprawność przeprowadzonych działań konstrukcyjnych oraz zrealizowanie założeń projektowych.

Abstract

The work consisted in designing a laboratory stand containing the aviation pendulum. This position was created for the laboratory classes taking place as part of the didactic activities of the Poznań University of Technology. The creation of the station consisted in the implementation of a mechanical structure, electrical systems and the preparation of appropriate software. The mechanical and electrical construction meets the safety requirement, which is required in the case of conducting didactic classes at the workplace. The software used in the Raspberry Pi 4B project allows communication with all elements of the station. These elements include: AS5600 magnetic encoder, which is a pendulum position sensor, a BLDC motor, and a STEVAL-SPIN 3201 controller responsible for measuring the speed of the motor, as well as regulating its speed, and a camera that acts as a visual coupling when measuring the angle. The software has also been adapted to communicate with Matlab software, using the UDP network protocol. The work also touched upon the physical aspect of the air pendulum, its mathematical model was developed and implemented in the Simulink program. The performed tests confirmed the correctness of the construction activities carried out and the implementation of the design assumptions.

Rozdział 1

Wstęp

1.1 Opis zagadnienia poruszonego w pracy

W pracy poruszono temat realizacji projektu programistyczno-konstrukcyjnego, który wraz z dołączonym do wahadła napędem lotniczym miał współtworzyć wahadło lotnicze (aeropendulum). Na realizację pracy, w głównej mierze miało składać się dobranie odpowiednich komponentów elektrycznych i zlecenie wykonania konstrukcji mechanicznej firmie zewnętrznej, jednak sposób doboru części sprawił, że nie było to konieczne, a samodzielne złożenie stanowisk pozwoliło przeznaczyć zaoszczędzone środki na zwiększenie liczby powstałych stanowisk.

1.2 Cel pracy

Celem pracy było zaprojektowanie i zbudowanie stanowiska laboratoryjnego z tzw. wahadłem lotniczym. Należało dobrać niezbędne podzespoły jak silnik, sterownik, układ pomiarowy, układ mikroprocesorowy, a całe stanowisko miało posiadać możliwość współpracy z programem Matlab. Należało opracować koncepcję konstrukcji mechanicznej oraz uruchomić układ. Dodatkowym awantażem było zaimplementowanie sprzężenia wizyjnego w postaci kamery.

1.3 Zawartość pracy

W kolejnych rozdziałach zostały opisane składowe elementy projektu, które były niezbędne do zbudowania działającego stanowiska laboratoryjnego. Ogólny zarys stanu wiedzy został przedstawiony przez wszystkich członków grupy, zgodnie z własnym zakresem pracy, w rozdziale drugim. W rozdziale trzecim został opisany proces analizy rynku pod kątem zakupu gotowego stanowiska, projektowania własnego rozwiązania. Jego elementy składowe jak konstrukcja elektryczna oraz mechaniczna zostały opisane przez Patryka Pilarskiego, za podrozdział mówiący o elementach personalizowanych odpowiadał Bartosz Podkański. Rozdział czwarty prezentuje fizyczny punkt widzenia i implementację modelu matematycznego w programie Matlab Simulink wykonany przez Bartosza Podkańskiego. Opisane oprogramowanie, wraz z elementami służącymi do obsługi urządzeń peryferyjnych, takich jak enkoder czy silnik, zostały opisane w rozdziale piątym przez Jakuba Walkowskiego. Rozdział szósty prezentuje pomyślne próby regulacji wahadła w żądanym położeniu oraz charakterystyki obiektu, którego powstanie opisuje niniejsza praca, przeprowadzone przez Bartosza Podkańskiego.

Rozdział 2

Podstawy teoretyczne

2.1 Silnik BLDC

Autor: Patryk Pilarski

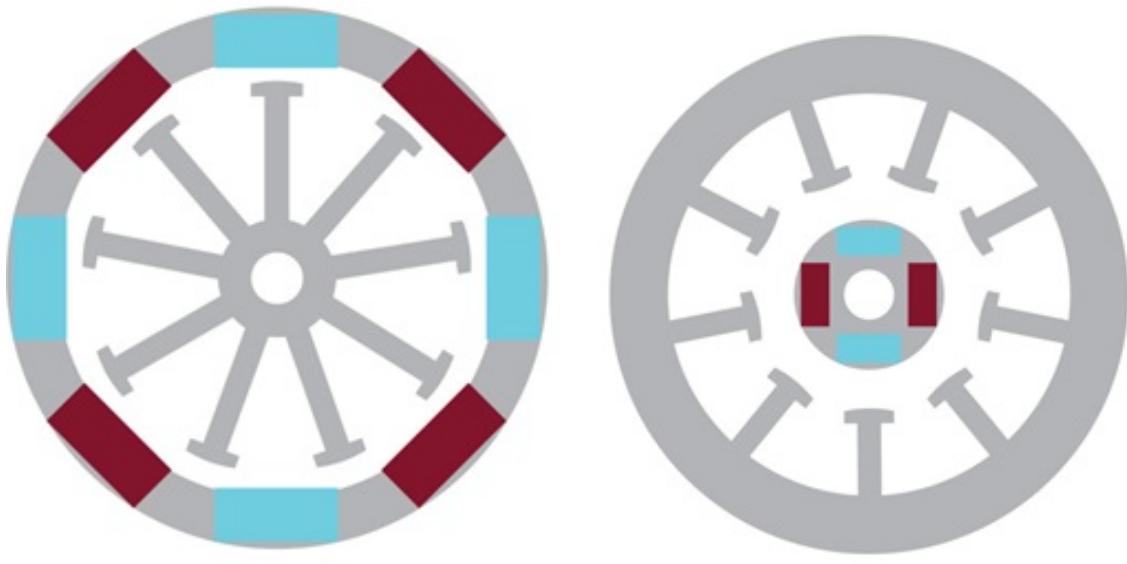
Obecnie bezszczotkowe silniki prądu stałego (z ang. BLDC-Brushless Direct-Current motor) stają się coraz popularniejszymi napędami elektrycznymi. Od dawna stosowane są w napędach CD-ROM czy dyskach HDD, coraz częściej stosowane są w pojazdach, elektrycznych, przemyśle, a także w sprzętach użytku domowego, takich jak suszarka do włosów czy odkurzacz. Zjawisko to występuje dzięki zwiększającej się dostępności komponentów elektronicznych, a także innym zaletom tych silników. Charakteryzują się one bardzo wysoką wydajnością, sprawnością, a także stosunkiem mocy do masy. Dzięki niezawodności, podzespoł ten cieszy się coraz większym uznaniem wśród pracowników działu utrzymania ruchu.

2.1.1 Budowa silnika

Główną zaletą silnika BLDC odróżniającą go od silników szczotkowych, jest brak komutatora odpowiedzialnego za mechaniczne przełączanie zasilania poszczególnych faz. W silnikach bezszczotkowych jego rolę spełnia przekształtnik-sterownik silnika.

Silniki BLDC można podzielić na kilka rodzajów ze względu na konstrukcję rotora:

- „Outrunner- stojan otoczony jest wirnikiem”, do którego przymocowane są magnesy. Stosowany jako napęd pojazdów wielowirnikowych[31].
- „Inrunner- wirnik otoczony jest stojanem”. Stosowany w przemyśle jako następca silników szczotkowych[31].



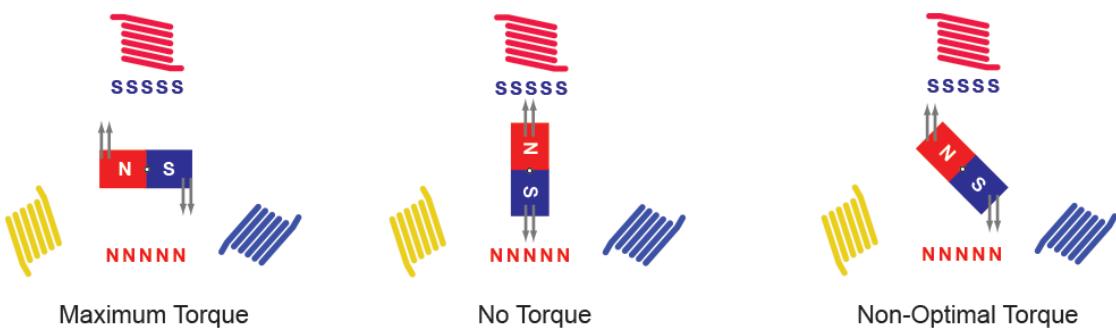
(a) Silnik BLDC typu outrunner

(b) Silnik BLDC typu inrunner

Rysunek 2.1: Rodzaje silników BLDC ; źródło: <https://www.analog.com>

2.1.2 Sterowanie silnikiem

Jak zostało wspomniane we wcześniejszym rozdziale, funkcję komutatora spełnia przekształtnik sterujący zasilaniem silnika. Nazwa napędu wskazuje iż sterowanie silnikiem powinno być realizowane poprzez przesyłanie sygnału trapezoidalnego. Okazuje się jednak, że metoda ta jest nieoptymalna. Większość silników posiada uzwojenia trójfazowe, a więc poszczególne fazy rozłożone są co 120° . Rysunek 2.2 przedstawia rozłożenie cewek oraz magnesów, można z niego wywnioskować, że maksymalny moment występuje gdy magnes wirnika jest oddalony o 90° od pola stojana. Podczas gdy kąt ten jest inny niż 90° następuje marnotrawienie energii, w wyniku czego powstają oscylacje momentu obrotowego silnika, głośna praca, a także szумy w sygnale sterującym.



Rysunek 2.2: Wizualizacja ustawiń pola magnetycznego względem cewek silnika; źródło: <https://www.roboteq.com>

Rozwiązaniem tego zagadnienia jest sterowanie wektorowe FOC (z ang. Field Oriented Control). Polega ono na zastosowaniu sygnałów sinusoidalnych przesuniętych względem siebie o 120° . Dzięki temu możliwa jest kontrola siły pola magnetycznego, w wyniku czego nie występują skoki i oscylacje momentu obrotowego silnika.

2.2 Komunikacja

Autor: Jakub Walkowski

W celu przeprowadzenia komunikacji stanowiska z programem Matlab wykorzystany został serwer uruchamiany na mikrokomputerze Raspberry Pi 4B wykorzystujący protokół warstwy transportowej modelu OSI o nazwie UDP (User Datagram Protocol). W czasie realizacji projektu pod rozwagę brana była również możliwość zaimplementowania protokołu TCP (Transmission Control Protocol). Protokoły te różnią się w sposobie kontroli przepływu danych i właśnie te różnice zaważyły nad wybrianiem UDP. Protokół UDP jako, iż jest protokołem bezpołączniowym, nie gwarantuje dostarczenia, ani sprawdzenia sekwencji wysyłanych danych. Przez brak kontroli przepływu danych dostarczenie ich klientowi nie jest sprawdzane a co za tym idzie szybkość transmisji jest większa. W przypadku naszego stanowiska i regulacji mniejsze opóźnienie w odbiorze danych ma o wiele większe znaczenie, niż utrata jakiegoś z wysłanych pakietów, które do środowiska Matlab wysyłane są nieustannie.

Dodatkowo ze względu na mniejszy narzut danych sterujących, datagram jakim operuje protokół UDP jest mniejszy, co również przyczynia się do zmniejszenia opóźnienia transmisji. Nagłówek datagramu UDP zawiera jedynie 8 bajtów danych, w których skład wchodzi port źródłowy, port docelowy, całkowita długość pakietu (nagłówek i dane) w bajtach oraz suma kontrolna (Tabela 2.1). Wielkość tego nagłówka jest zdecydowanie mniejsza od długości nagłówka protokołu TCP, która wynosi aż 24 bajty (Tabela 2.2).

Bajty			
1	2	3	4
Port źródłowy		Port docelowy	
Długość		Suma kontrolna	
Dane			

Tabela 2.1: Nagłówek datagramu UDP, opracowano na podstawie [16]

Bajty									
1	2	3 4							
Port źródłowy		Port docelowy							
Numer sekwencyjny									
Numer potwierdzenia									
Długość nagłówka	Rezerwa	Znaczniki	Okno						
		U A P P S F							
		R C S S Y I							
		G K H T N N							
Suma kontrolna				Wskaźnik pilności					
Opcje				Wypełnienie					
Numer potwierdzenia									

Tabela 2.2: Nagłówek datagramu TCP, opracowano na podstawie [16]

Z uwagi na to iż zmniejszenie opóźnienia w komunikacji z programem Matlab, który pełni rolę regulatora, podjęta została decyzja o skorzystaniu z protokołu UDP, który okazał się być idealny dla tego zastosowania. [15][16].

Rozdział 3

Budowa stanowiska

Autor: Patryk PilarSKI

3.1 Poszukiwanie i projektowanie prototypu

3.1.1 Przegląd gotowych rozwiązań

Pierwszy etap prac nad stanowiskiem laboratoryjnym rozpoczął przegląd tzw. benchmarków, czyli gotowych rozwiązań na rynku. Początkowo, zaproponowanym rozwiązaniem było wahadło napędzane kołem reakcyjnym. Zaletami tego urządzenia są dopracowana konstrukcja przez firmę zewnętrzną oraz gotowa karta sterująca silnikiem posiadająca moduł wejść i wyjść. Ta cecha z kolei pozwala na komunikację z oprogramowaniem Matlab na komputerach PC.

Skorzystanie z wahadła zapewniłoby oszczędność czasu, poświęcanego na pisanie własnych bibliotek do komunikacji. Co również istotne, producentem jest polska firma Inteco w związku z czym, czas dostawy maszyny byłby stosunkowo krótki. Rozwiązanie to jednak zostało odrzucone ze względu na wysoką cenę tj. 28 tysięcy złotych oraz niezgodność z założeniem wahadła lotniczego.



Rysunek 3.1: Wahadło napędzane kołem reakcyjnym Inteco; źródło: <http://www.inteco.com.pl/>

Kolejnym rozważanym rozwiązańem było wahadło lotnicze zbudowane przez University of Arizona. Spełnia ono podstawowe założenia projektu. Podobnie jak w przypadku wahadła z kołem reakcyjnym, podstawą działania jest oś obrotu umieszczona na jednej nodze.



Rysunek 3.2: Wahadło lotnicze - University of Arizona; źródło: <https://aeropendulum.arizona.edu/>

Zaletą tego urządzenia jest stosunkowo niski koszt jednego urządzenia tj. około 120 USD. Podobnie jak poprzednia propozycja, zestaw zawiera gotowy sterownik

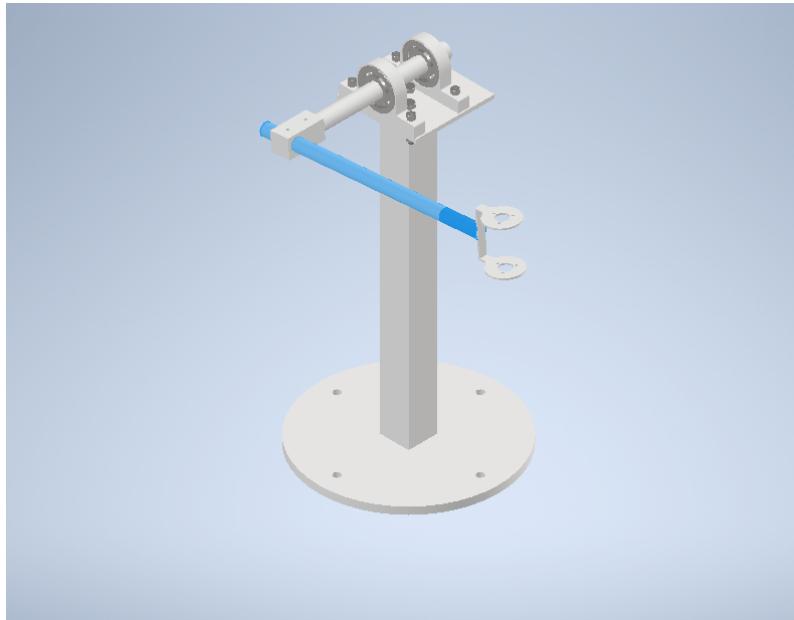
silnika umożliwiający komunikację z komputerem PC oraz oprogramowanie z zaimplementowanym algorytmem sterowania w środowisku Matlab Simulink. Stanowisko zbudowane jest w sposób prosty. Odczyt kąta wychylenia realizowany jest poprzez odczyt napięcia podawanego z potencjometru na przetwornik ADC (ang. Analog to Digital Converter). Dodatkowo nie posiada pierścienia obrotowego - z tych dwóch powodów, nie ma możliwości wykonania pełnego obrotu bez wprowadzenia gruntownych zmian. Kolejną czynnikiem wpływającym na rezygnację z wykorzystania systemu był chwilowy brak wahadeł na stanie i długi czas dostawy (wysyłka z Stanów Zjednoczonych)

W związku z brakiem dostępnych na rynku rozwiązań, spełniających założenia projektu, podjęta została decyzja o zaprojektowaniu oraz budowie własnej konstrukcji. Głównymi założeniami były:

- długość wahadła około *30 cm*,
- możliwość wykonania wielu obrotów,
- siła ciągu, pozwalająca na pokonanie oporów ruchu oraz wykonanie pełnego obrotu wahadła,
- konstrukcja bezpieczna dla użytkownika,
- silnik BLDC zawierający czujniki halla,
- sterownik z rodziny STEVAL-SPIN (dobrany odpowiednio do właściwości silnika),
- możliwość implementacji systemu wizyjnego,
- mikrokomputer Raspberry Pi służący za centralny element zarządzający komunikacją pomiędzy wszystkimi elementami stanowiska serwer,
- enkoder absolutny do odczytu kąta wychylenia.

3.1.2 Prototypy

Pierwszą wersją było wahadło oparte na jednej nodze. Jest to kompaktowa konstrukcja niewielkich rozmiarów, spełniająca swoje zadania oraz pasująca do przebiegu zajęć laboratoryjnych



Rysunek 3.3: Wahadło oparte na jednej nodze [opracowanie własne]

Śmigło kręcące się zgodnie ze swoim prawidłowym kierunkiem obrotów powoduje powstanie ciągu proporcjonalnego do prędkości obrotowej śmigła. Jeżeli śmigło zacznie się obracać w kierunku przeciwnym do jego przeznaczenia, ciąg który powstaje osiąga znacznie mniejszą wartość. Rozwiązaniem tego problemu jest zastosowanie dwóch silników umieszczonych w przeciwnych kierunkach. Aby zaimplementować taki układ należałoby użyć dwóch sterowników, po jednym na każdy silnik. To spowodowałoby znaczny wzrost kosztów, poziomu skomplikowania konstrukcji oraz oprogramowania do komunikacji z komputerem PC. Brak możliwości podłączenia enkodera oraz pierścienia obrotowego na jednym z końców osi obrotu wahadła, było powodem rezygnacji z wykorzystania konstrukcji.

3.2 Konstrukcja elektryczna

Zaprojektowane na potrzeby pracy dyplomowej wahadło lotnicze, służyć będzie jako stanowisko dydaktyczne podczas laboratoriów prowadzonych na Politechnice Poznańskiej, w związku z czym projekt został dostosowany szczególnych wymogów bezpieczeństwa.

3.2.1 Silnik

Kluczowy dla konstrukcji element - silnik elektryczny - został dobrany jako jeden z pierwszych. Aby umożliwić implementację zamkniętej pętli regulacji prędkości obrotowej silnika, niezbędny jest pomiar jego prędkości, co można przeprowadzić na wiele sposobów. Rozważone zostały dwie metody tj. metoda bezczujnikowa, oparta o pomiar siły elektromotorycznej (z ang. Back EMF) oraz zastosowanie silnika z czujnikami Halla.

Metoda bezczujnikowa cechuje się dokładnością, umożliwiającą dokonanie pomiarów z dokładnością do 1%. Implementacja takiego układu w instalacji wahadła okazała się nie spełniać wymogów projektu z przyczyn technicznych. Zastosowanie

silnika z czujnikami Halla cechuje się zaś dokładnością pomiarów wystarczającą do implementacji algorytmu sterowania wahadła.

Biorąc pod uwagę stosunek mocy do masy, oraz prędkość obrotową silnika, uzasadnionym wydaje się zastosowanie silników znajdujących się w dronach. W urządzeniach tych jednak, pomiar prędkości za pomocą czujników halla nie jest praktykowany, co za tym idzie nie ma możliwości kupna gotowego rozwiązania.

Na rynku dostępnych jest wiele silników spełniających założenia projektu pod względem technicznym, jednak wykraczających poza jego budżet. Ostatecznie zastosowanym w konstrukcji elementem jest trójfazowy silnik prądu stałego DF45L024048-A2 wyprodukowany przez firmę Nanotec®[18]. Wyposażony jest w trzy czujniki Hallotronowe o rozstawie 120° , pozwalające na pomiar prędkości z rozdzielczością do tysiąca impulsów na obrót.



Rysunek 3.4: Silnik prądu stałego DF45L024048-A2; źródło: <https://pl.farnell.com>

3.2.2 Sterownik

Do dobranego wcześniej silnika, należało dopasować sterownik. Wybór determinowany był przez obecność kontrolera z rodziny STEVAL-SPIN [19]. Aby układ nie był podatny na uszkodzenia oraz działał niezawodnie podczas zajęć laboratoryjnych, przekształtnik musiał cechować się odpowiednimi danymi znamionowymi m.in zakresem napięć wyjściowych oraz amperażem. Użyty w konstrukcji silnik w piku pobiera do 9.5 A , a jego napięcie znamionowe wynosi 24 V . Sterownik który, spełnia wymagania projektu to STEVAL-SPIN3201. Uniwersalność tego urządzenia pozwala na zasilanie silnika napięciem od 8 V do 45 V oraz do 15 A prądu ciągłego. W związku z parametrami odpowiadającymi danym znamionowym silnika, zastosowany został właśnie ten kontroler.



Rysunek 3.5: Sterownik STEVAL-SPIN3201; źródło: <https://www.st.com/>

3.2.3 Mikrokontroler/raspberry

Podstawową platformą odpowiedzialną za komunikację między kontrolerem silnika, a komputerem PC jest mikrokomputer Raspberry Pi 4B 2GB, z kartą pamięci o pojemności 64GB [3]. Początkowo, w ramach obsługi wielu interfejsów szeregowych, przetworników analogowych (w które Raspberry Pi nie jest wyposażone), rozważane było wykorzystanie mikrokontrolera NUCLEO STM32F746ZG.

W związku, z wykorzystaniem jednego interfejsu szeregowego oraz jednego interfejsu I2C - w ostatecznej wersji projektu, zadecydowano o wykorzystaniu mikrokomputera Raspberry Pi do obsługi komunikacji. Dodatkowo ten mikrokomputer służy jako serwer do komunikacji z programem Matlab, a także przetwarzania obrazu z kamery na kąt wychylenia wahadła.

3.2.4 Pierścień ślizgowy

Istotną cechą stanowiska jest możliwość wykonania wielu obrotów wahadła, bez ryzyka zapłatania przewodów zasilających silnik oraz uszkodzenia jego elementów. W tym celu postanowiono zastosować pierścień ślizgowy.

Czynnikami, które były kluczowe podczas podejmowania decyzji były; ilość linii sygnałowych - minimum osiem żył przesyłających dane; 3 z nich musiały posiadać maksymalne obciążenie prądowe - do 9.5 A prądu chwilowego oraz cena.

Optymalnym rozwiązaniem dla powstania konstrukcji był przelotowy pierścień ślizgowy. Zbudowany jest, w sposób umożliwiający osi obrotu wahadła, przejście przez środek pierścienia. Jego zaletami są prostota montażu i dostęp do końca osi obrotu, dzięki czemu możliwe jest zastosowanie kilku enkoderów. Jednakże zastosowanie w.w. rozwiązanie wiązało się z wysokimi kosztami oraz długim czasem oczekiwania na przesyłkę, co uniemożliwiło wykorzystanie elementu przedstawionego na rysunku 3.6.

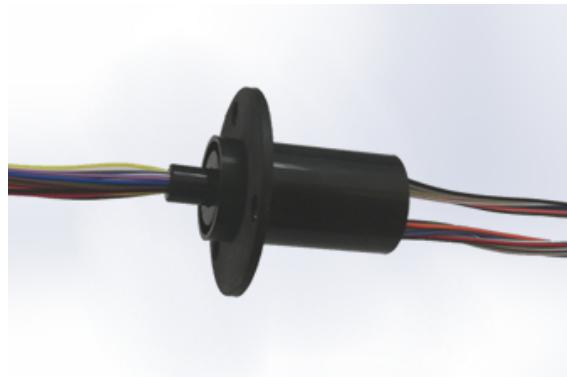


Rysunek 3.6: Przelotowy pierścień ślizgowy; źródło: <https://pl.aliexpress.com>

Równie optymalnym oraz dostępnym cenowo rozwiązaniem jest pierścień ślizgowy, stosowany do budowy domowych elektrowni wiatrowych. Jego dostępność w polskich sklepach internetowych, pozwoliła na szybki zakup elementu. Rozwiązanie to różni się od przelotowego pierścienia obrotowego (rysunek 3.6), konstrukcją tj. przewody w części ruchomej, muszą znajdować się w środku osi obrotu wahadła.

Na rynku dostępne jest rozwiązanie, możliwe do wdrożenia przy budowie konstrukcji. Jest to pierścień wyposażony w trzy linie zasilające silnik o obciążalności prądowej do 10 A oraz pięć linii sygnałowych.

Wariant ten wykraczał poza budżet projektu, w związku z czym zdecydowano o zakupie dwudziestocztero liniowego pierścienia, o maksymalnym prądle 2 A [22]. Aby uzyskać odpowiednią obciążalność prądową zastosowano równolegle połączenie pięciu żył, otrzymując tym samym trzy linie zasilające silnik, pięć linii sygnałowych oraz cztery linie rezerwowe.

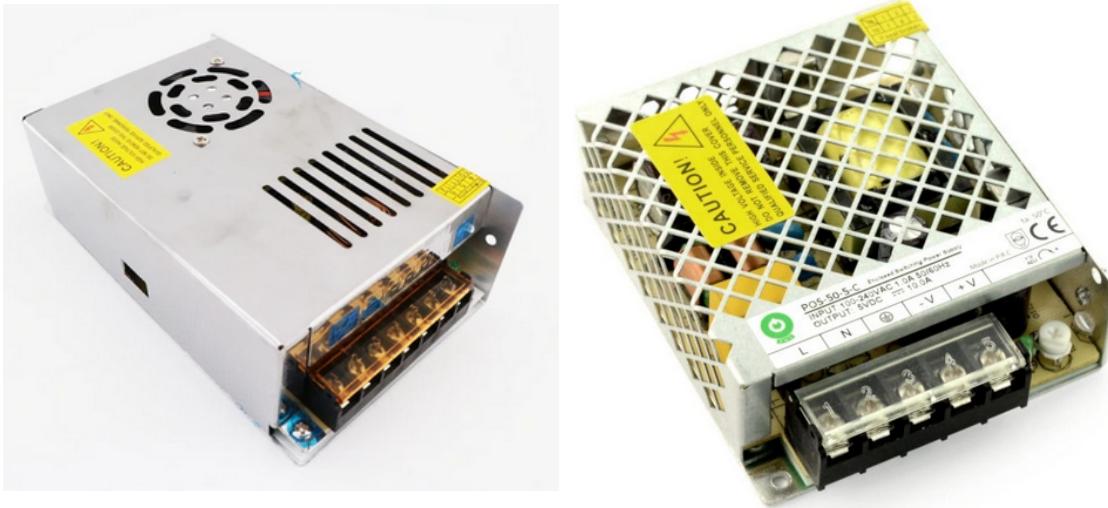


Rysunek 3.7: ZER-X5 24-Channel, 2 A, Capsule Slip Ring; źródło: <https://slip-ring.com>

3.2.5 Zasilacze

Po dobraniu elementów wykonawczych, należało dobrać do nich odpowiednie zasilanie. Pierwszym, rozważanym rozwiązaniem był zasilacz dwukanałowy (po jednym kanale dla linii zasilającej 5 V oraz 24 V). Z wyników przeprowadzonego badania rynku wywnioskowano, że nie występuje wiele zasilaczy o wystarczającej mocy,

miesiączących się w budżecie projektu. W związku z tym, wybrano mniej skomplikowane rozwiązanie, o niższej cenie - mianowicie zakup dwóch zasilaczy. Dobór układów zasilających spełniających potrzeby podzespołów, nie stanowił problemu. Zastosowane zasilacze to S-240-24, o napięciu wyjściowym 24 V i prądzie maksymalnym 10 A, widoczne na fotografiach 3.8a i 3.8b [24, 25].



(a) Zasilacz S-240-24 o napięciu wyjściowym 24 V i prądzie maksymalnym 10 A; źródło: <https://allegro.pl>
(b) Zasilacz montażowy POS-50-5-C 5V/10A/50W; źródło: <https://botland.com.pl>

Rysunek 3.8: Źródła napięcia

3.2.6 Instalacja elektryczna

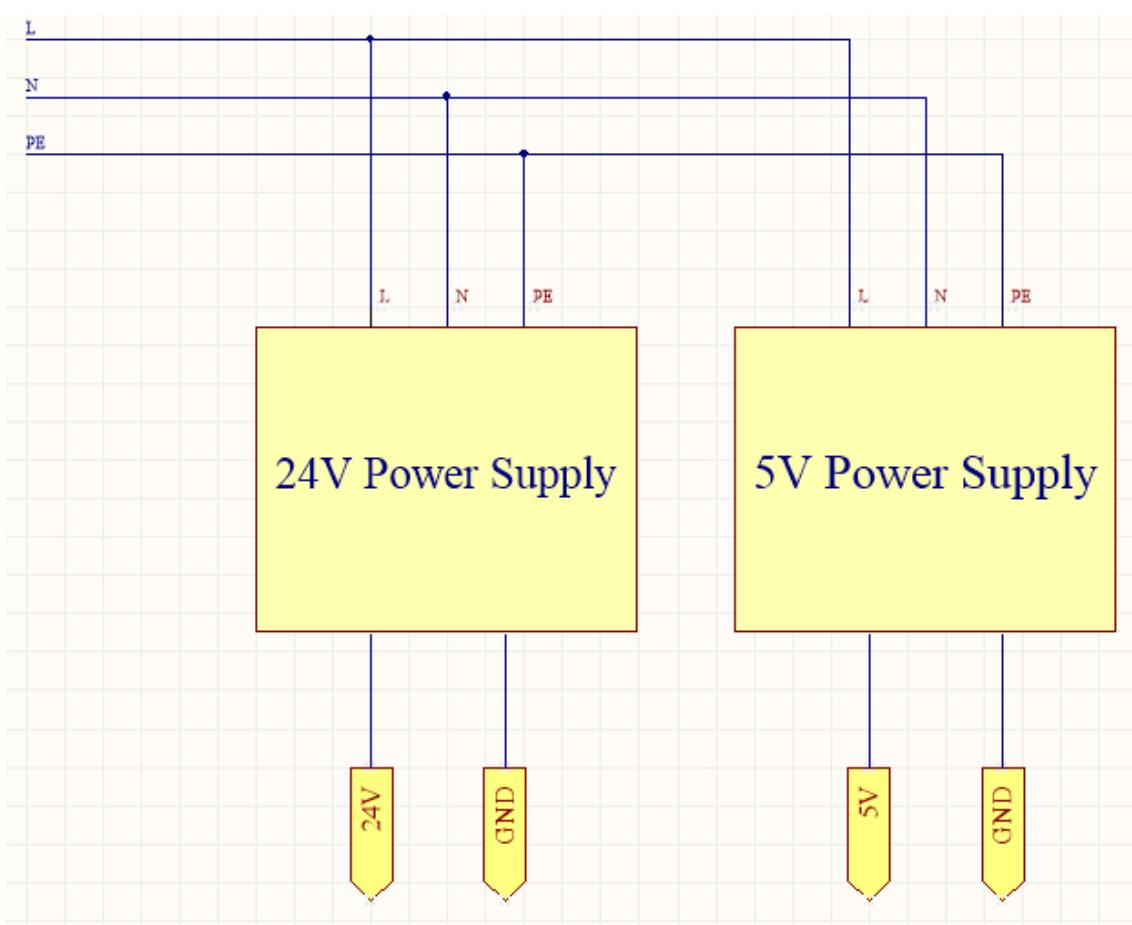
Po doborze potrzebnych komponentów elektrycznych, należało stworzyć odpowiednią instalację elektryczną. Maksymalna ilość prądu w piku, pobierany przez układ to około 12 A_{pk} . Należało więc dobrać przewody zasilające o odpowiedniej obciążalności prądowej.

Stanowisko podłączone zostało do sieci za pomocą kabla zasilającego o maksymalnym prądzie pracy 16 A. Następnie zastosowano zostało gniazdo C14 z wbudowanym wyłącznikiem i bezpiecznikiem. Zasilacze zostały podłączone do wyżej wymienionego gniazda za pomocą kabli LgY $0,75 \text{ mm}^2$, o maksymalnej długotrwałej obciążalności prądowej wynoszącej 12 A.

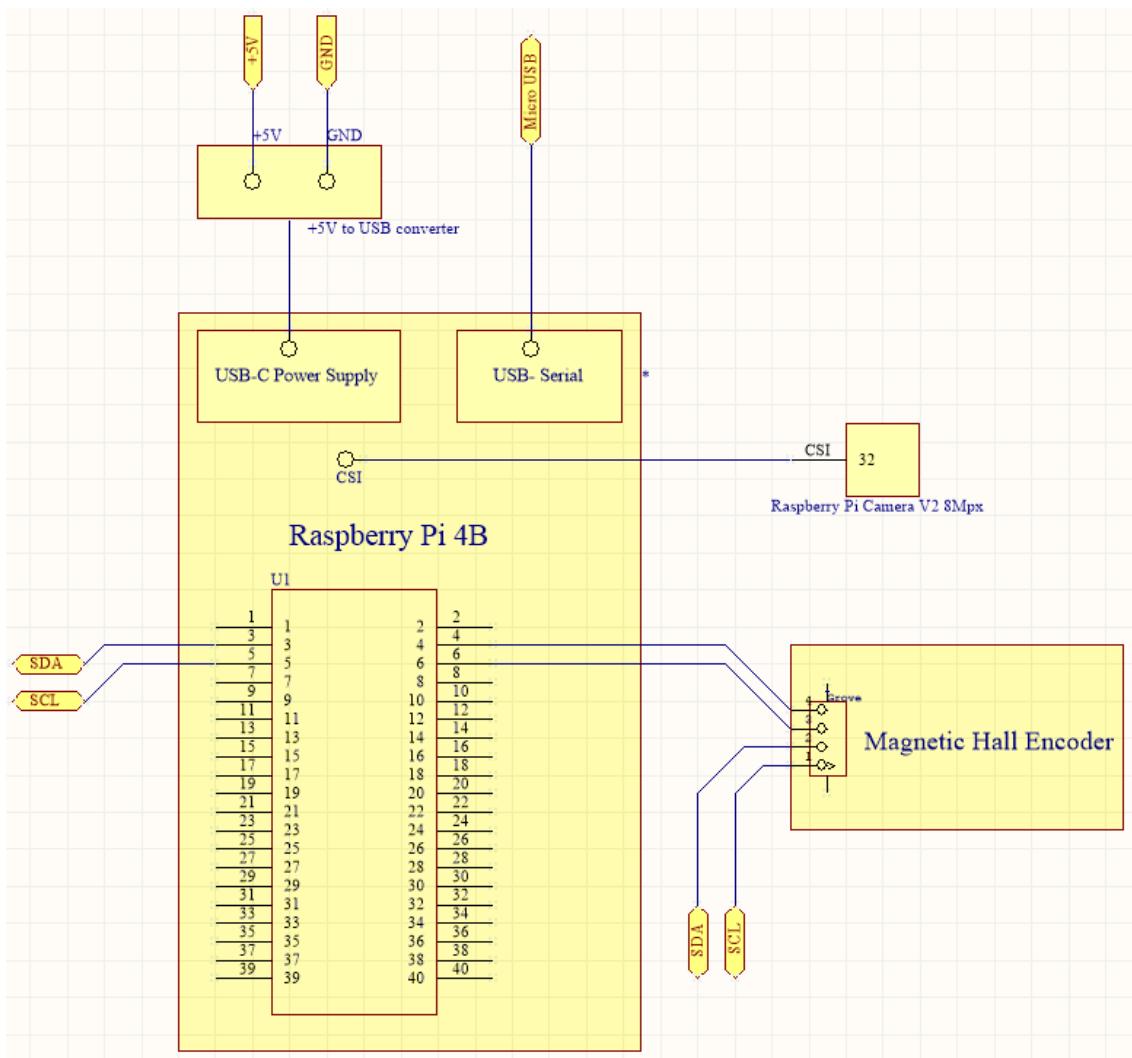
Do zasilania Raspberry Pi zastosowano przejściówkę ze złącza śrubowego na kabel USB (rysunek 3.9a). Umożliwia to podłączenie mikrokomputera przez wbudowane wejście USB-C do przekształtnika napięcia 5 V. Platforma STEVAL-SPIN 3201 wyposażona jest w złącza śrubowe. Aby uzyskać pełną zgodność z wymaganiami projektu, należało zakuć przewody. Wizualizacje schematów elektrycznych znajdują się na rysunkach 3.10, 3.11 oraz 3.12.



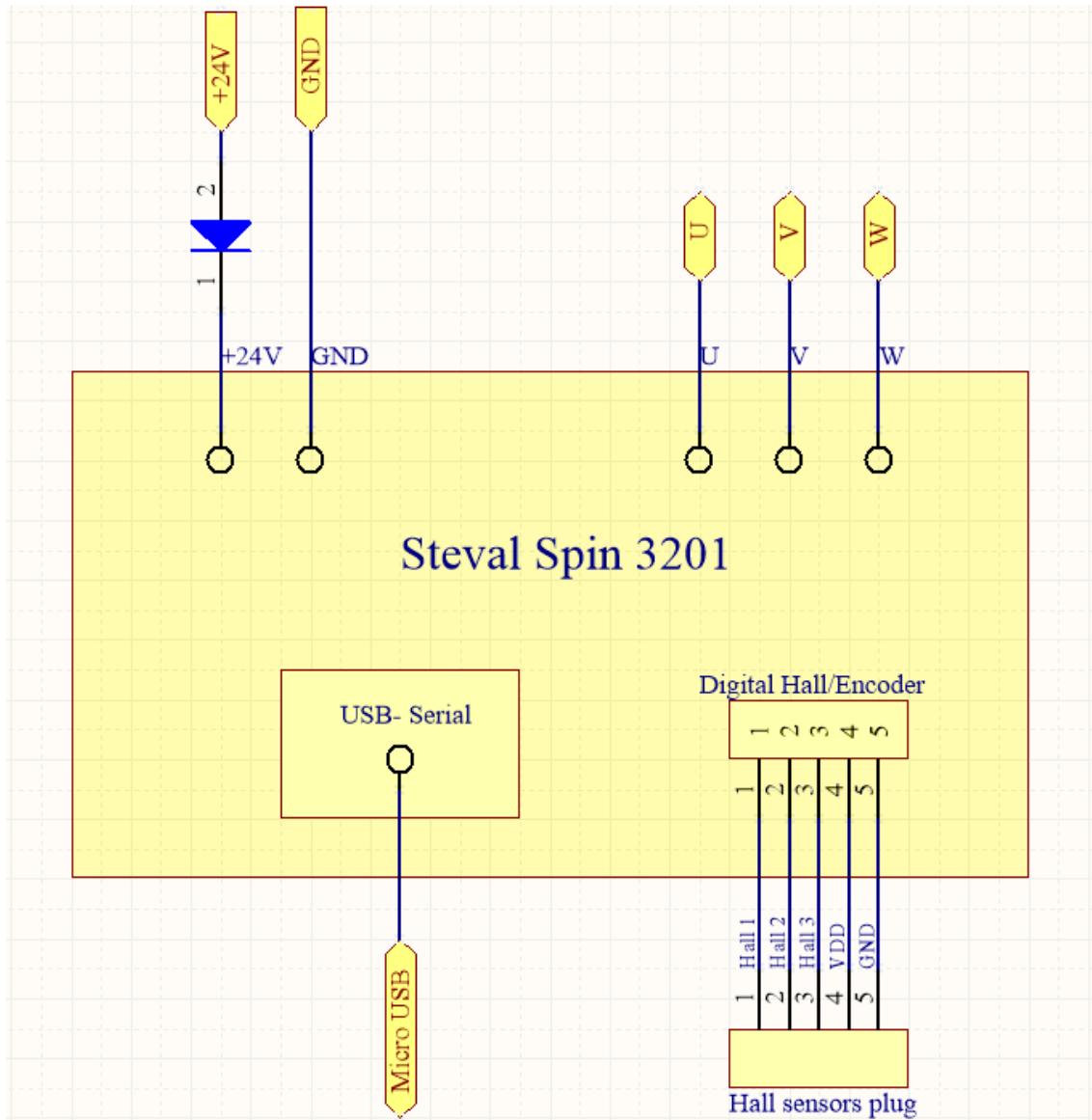
Rysunek 3.9: Elementy instalacji elektrycznej



Rysunek 3.10: Schemat elektryczny połączeń zasilaczy [opracowanie własne]



Rysunek 3.11: Schemat elektryczny Raspberry Pi [opracowanie własne]



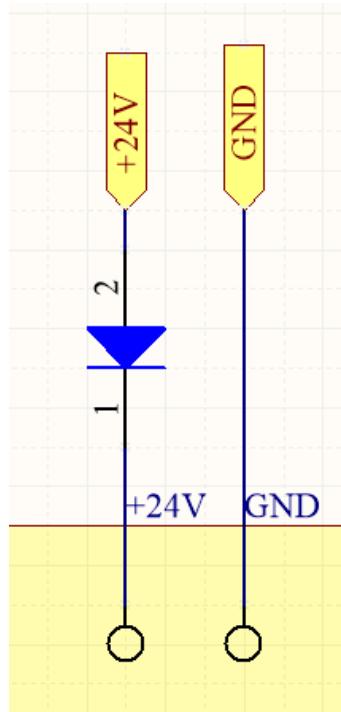
Rysunek 3.12: Schemat elektryczny STEVAL-SPIN3201 [opracowanie własne]

3.2.7 Zabezpieczenia

Aby stanowiska laboratoryjne były bezpieczne i niezawodne, musiały zostać odpowiednio zabezpieczone. Podstawowym zabezpieczeniem przeciwwzwarciowym, po stronie pierwotnej przekształtników napięcia jest ceramiczny bezpiecznik topikowy, o prądzie znamionowym 20 A, umiejscowiony w gnieździe C14, taka wartość bezpiecznika została dostosowana do poboru prądu układu z zapasem 5A. Dodatkowo, oba zasilacze posiadają wbudowane zabezpieczenia przeciążeniowe, zwarciowe i nadnapięciowe po stronie wtórnej. W razie wystąpienia nieprawidłowości w układzie zasilania, przekształtnik wyłącza się i automatycznie powraca do normalnej pracy, po ustąpieniu przyczyny.

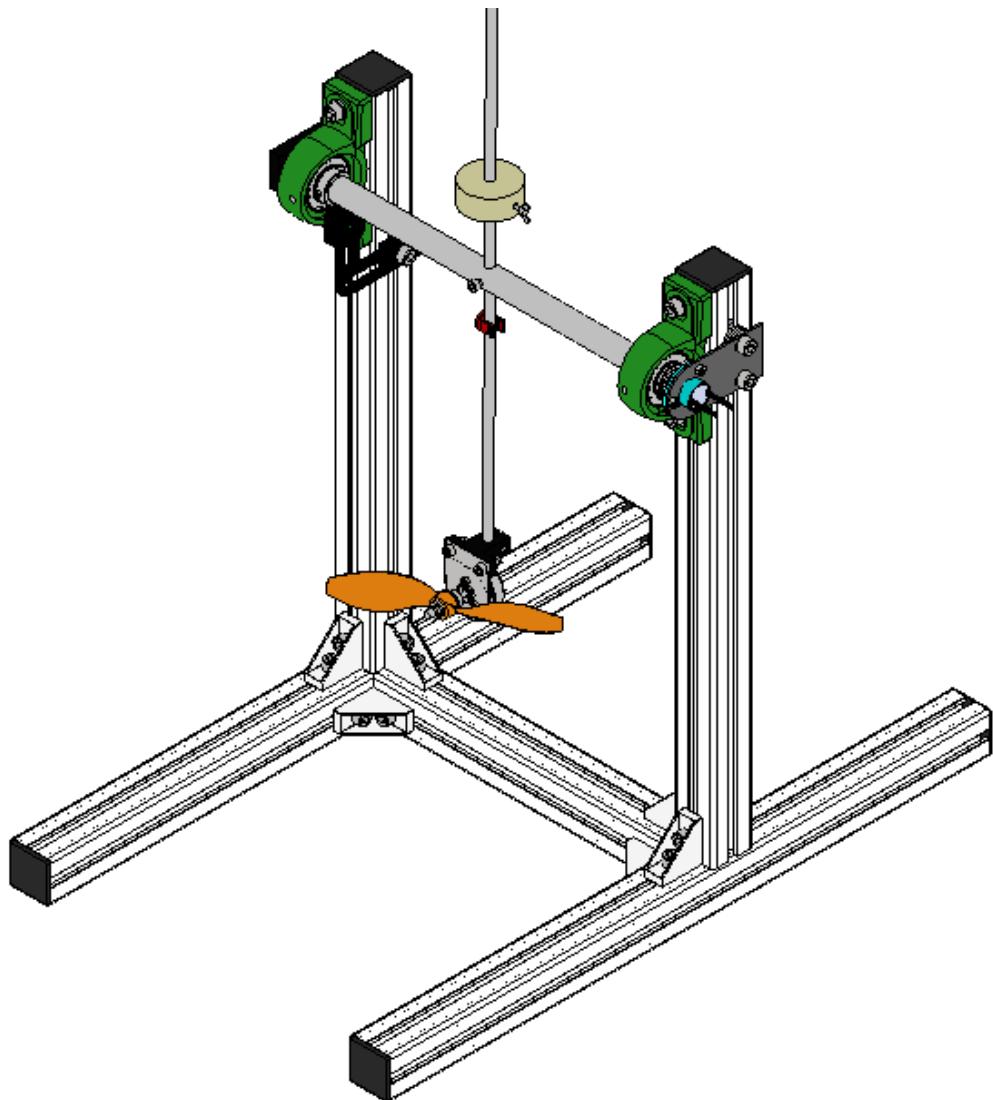
Po uruchomieniu układu kontroler STEVAL SPIN3201, podczas zmniejszania prędkości obrotowej silnika oddawał napięcie do zasilacza. Skutkowało to wzrostem napięcia na linii zasilającej do około 35 V. Zabezpieczenie nadnapięciowe zadzia-

łało prawidłowo - rozłączyło zasilanie. Był to niepożądany efekt, który należało wyeliminować sprzętowo. Zastosowana została więc dioda prostownicza, w sposób przedstawiony na schemacie (Rysunek 3.13), która zapobiega wzrostowi napięcia po stronie wtórnej zasilacza.



Rysunek 3.13: Dioda zabezpieczająca [opracowanie własne]

3.3 Konstrukcja mechaniczna



Rysunek 3.14: Makietka zbudowana w programie Autodesk Inventor [opracowanie własne]

Przed rozpoczęciem budowy stanowiska należało zadecydować o metodzie wytwarzania elementów personalizowanych. Istniała możliwość zaprojektowania konstrukcji, a następnie zlecenia jej wykonania firmom zewnętrznym lub wykonania ich samodzielnie.

Wybrano samodzielne wykonanie elementów, ze względu na możliwość dokonywania równoczesnej weryfikacji czy dany element spełni swoją funkcję oraz czy nie wymaga modyfikacji, dzięki czemu zyskano możliwość dokładnej analizy kolejnych elementów stanowiska. Głównym założeniem podczas tego etapu był zakup gotowych elementów bez konieczności ich modyfikacji i obróbki. Części, które nie były dostępne na rynku wykonano za pomocą druku 3D oraz cięcia laserowego CNC. Równolegle z dobieraniem części powstawała makietka w programie Autodesk Inventor, która pozwoliła na precyzyjne dopasowanie części, a także wyeliminowała

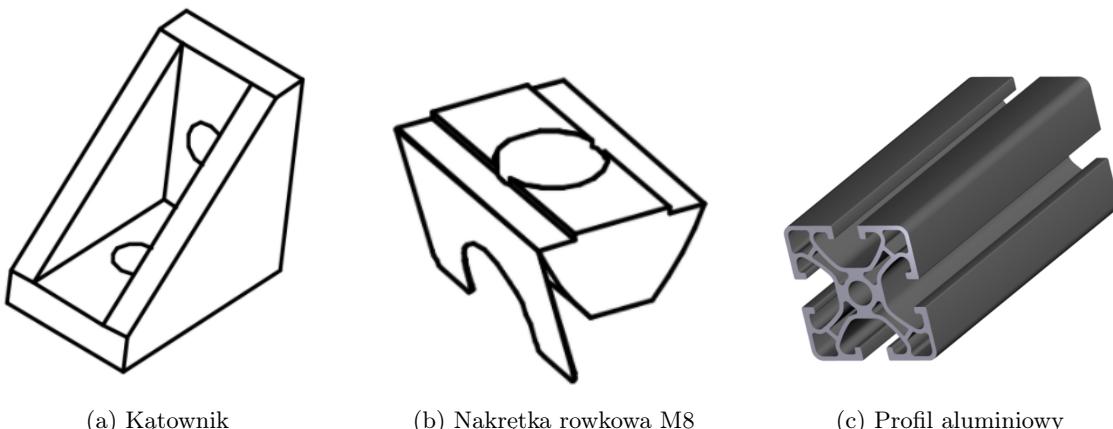
konieczność wprowadzania poprawek w trakcie składania stanowisk.

Projekt konstrukcji mechanicznej rozpoczęto od przeglądu dostępnych rozwiązań na rynku. Cechy, którymi miał się charakteryzować się zespół elementów to m.in.:

- długość wahadła 30 cm ,
- oś obrotu z podparciem na dwóch nogach,
- możliwość wykonania wielu obrotów; przeciwdziałanie płataniu się przewodów,
- bezpieczeństwo dla zdrowia użytkownika.

3.3.1 Rama

Główną część szkieletu stanowi rama. Rozważano dwa rodzaje materiałów, z którego miała zostać wykonana. Pod uwagę wzięto stalowe profile - zaletą była niska cena. Jednak budowa stanowiska z tego materiału wymagała dużego nakładu pracy przy obróbce oraz znacznie utrudnione przyszłe modernizacje stanowiska. Ostatecznie wykorzystano profile aluminiowe, spełniające warunki projektu. Zostały zamówione docięte na wymiar oraz połączone ze sobą za pomocą nakrętek rowkowych M8 i gotowych kątowników. Zaletą tego rozwiązania są nieograniczone możliwości modyfikacji konstrukcji, poprzez umiejscowienie nakrętek w dowolnym miejscu oraz przykręcenie kolejnego elementu.



Rysunek 3.15: Elementy składające się na budowę ramy; źródło: <https://www.ebmia.pl>

3.3.2 Oś obrotu

Jako oś obrotu zastosowana została precyzyjna rura o średnicy 20 mm i grubości ścianki 1.5 mm . Wahadło zostało zamocowane przelotowo przez oś obrotu oraz skontrowane śrubą M5 (Rysunek x). Aby umożliwić swobodny obrót ruchomych elementów, zastosowane zostały łożyska wahliwe w obudowie (Rysunek 3.16.). Dzięki zastosowaniu tego typu łożysk, wyeliminowana została kwestia ustawienia osiowania łożysk względem ich mocowań.



Rysunek 3.16: Łożysko; źródło: <https://www.ebmia.pl>

3.3.3 Elementy personalizowane, integracja elementów

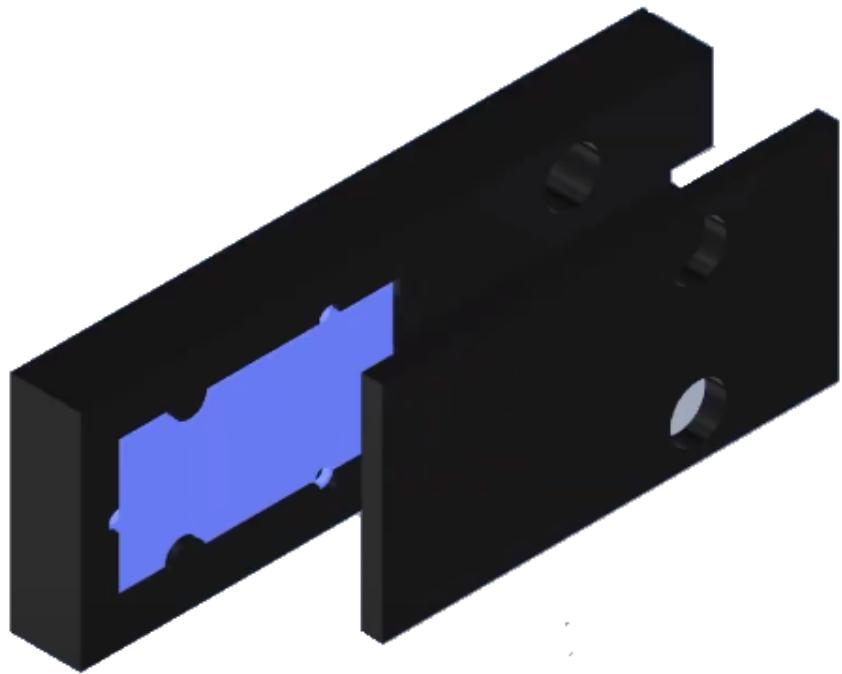
Autor: Bartosz Podkański

Ze względu na brak dostępności dedykowanych rozwiązań na rynku, konieczne było zaprojektowanie niektórych elementów, a także zaplanowanie rozwiązań technicznych połączenia poszczególnych elementów. Niezbędne elementy zostały najpierw zaprojektowane w programie Autodesk Inventor, a następnie w zależności od potrzeb i zastosowania elementu, wykonana techniką drukowania 3D, bądź wycinania laserowego przy pomocy maszyny CNC zrealizowane w zewnętrznej firmie.

Elementy drukowane

- Obudowa enkodera

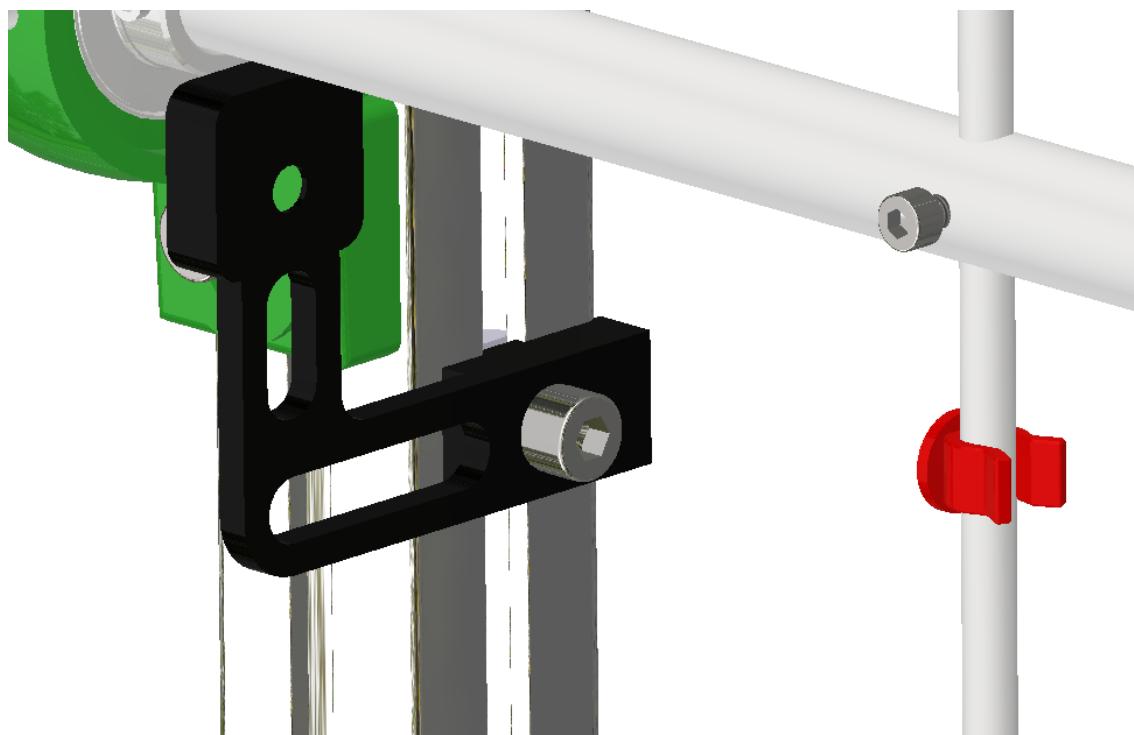
W związku z tym, że producent nie udostępnia dedykowanych obudów do enkodera, podstawa pod enkoder została zaprojektowana i wydrukowana. Składa się z dwóch elementów. Obudowy w której umieszczany jest enkoder oraz pokrywki, dzięki której enkoder stabilnie jest umieszczony w obudowie. Obudowa posiada kanał na przewody i jest przykręcona do nakrętek umieszczonych w profilu aluminiowym za pomocą śrub.



Rysunek 3.17: Zaprojektowana obudowa enkodera [opracowanie własne]

- **Uchwyt kamery**

Podobne rozwiązania jak przy zamontowaniu enkodera zastosowano przy montażu kamery. Udostępniona w internecie wersja uchwytu została zmodyfikowana na potrzeby budowanego stanowiska.

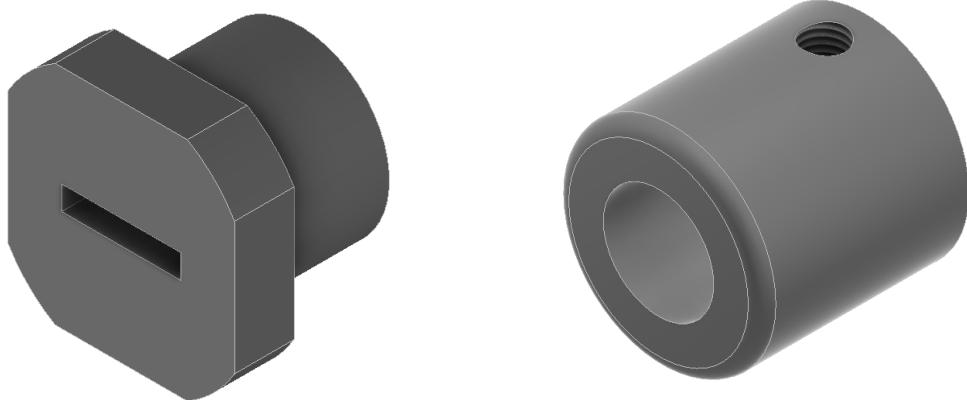


Rysunek 3.18: Uchwyt mocujący kamerę [opracowanie własne]

W zestawie do uchwytu kamery został także wydrukowany czerwony znacznik, który jest zatrzaskiwany na wahadle.

- Elementy montowane na osi - adapter, oprawa magnesu

W celu ulżenia przewodom wychodzącym z pierścienia ślizgowego został zaprojektowany adapter, który na wcisk montowany jest w rurze oraz za pomocą śruby przykręcanym do pierścienia. Drugi koniec osi został uzbrojony w korek montowany na wcisk, do którego również na wcisk, umocowany jest magnes służący enkoderowi jako źródło pola magnetycznego, kształt oprawy został dobrany tak, aby po zablokowaniu wahadła, za pomocą klucza 21 móc skalibrować kąt padającego pola magnetycznego.



(a) Oprawa magnesu

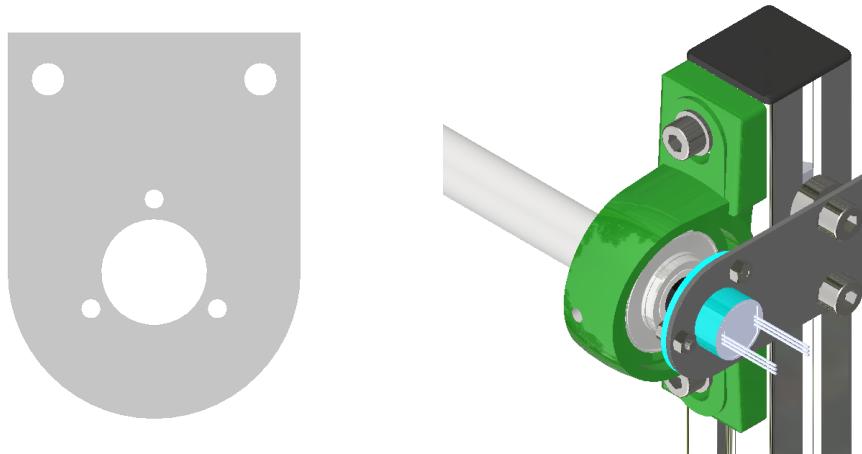
(b) Adapter oś - pierścień

Rysunek 3.19: Elementy montowane po przeciwnieństwach stronach osi [opracowanie własne]

Elementy wycięte laserowo

- Łoże silnika i uchwyt pod pierścień

Decyzja o wycięciu łożyska silnika oraz uchwytu pod pierścień ze stali nierdzewnej była podyktowana potrzebą precyzyjnego wycięcia otworów montażowych, trwałości całego elementu oraz inaczej niż w przypadku elementów drukowanych małą ilością potrzebnego materiału.



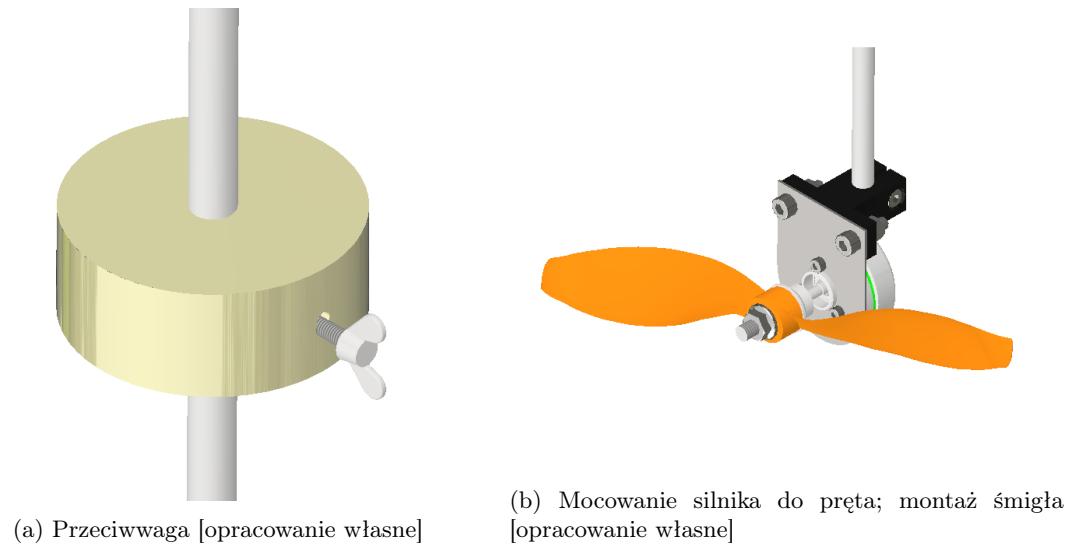
(a) Łoże silnika [opracowanie własne]

(b) Uchwyt mocujący pierścień [opracowanie własne]

Rysunek 3.20: Elementy wycięte za pomocą lasera CNC

Rozwiązania techniczne i łączenie elementów

Silnik wraz z łożem zostało przyjmocowane do wahadła które stanowi aluminiowy pręt o średnicy 8 milimetrów za pomocą wspornika wałka. Śmigło do wału silnika zamontowano za pomocą piasty, wahadło można dodatkowo obciążać przeciwagą mocowaną do pręta przy pomocy śruby motylkowej.



Rysunek 3.21: Sposób mocowania elementów

3.3.4 Osłona

Autor: Bartosz Podkański

Śmigło wirujące z prędkością 5000 obrotów na minutę może stanowić zagrożenie dla zdrowia użytkowników stanowiska, a jednym z podstawowych założeń projektu było zapewnienie bezpieczeństwa osobom pracującym podczas zajęć dydaktycznych ze stanowiskiem. Została podjęta decyzja zastosowania osłony ze szkła akrylowego otaczającej całe stanowisko. Fizyczne zabezpieczenie zostało zamówione w zewnętrznej firmie. Możliwym jest zdemontowanie osłony ze stanowiska, do którego jest przy mocowana za pomocą trzech nakrętek motylkowych (do stanowiska na stałe zostały przytwierdzone gwintowane szpilki).



Rysunek 3.22: Makieta za osłoną; [opracowanie własne]

3.3.5 Skrzynka elektryczna

Zaprojektowany wcześniej układ elektroniki należało estetycznie oraz bezpiecznie zamontować do ramy. W tym celu zastosowano obudowę S-BOX 716-PM o rozmiarach 380x300x120 mm. Takie gabaryty pozwoliły na bezpieczne, wygodne oraz estetyczne połączenie układów elektronicznych. Dodatkowo wyposażona jest w perforowaną płytę montażową, umożliwiającą montaż kolejnych komponentów bez konieczności wykonywania nowych otworów (Rysunek 3.23). Zamontowana została do stanowiska po przeciwej stronie względem użytkownika, aby nie zmniejszać jego pola widzenia podczas pracy przy stanowisku.



Rysunek 3.23: Obudowa S-BOX 716-PM; źródło: <https://elektryzuj.com.pl>

Rozdział 4

Budowa modelu matematycznego

Autor: Bartosz Podkański

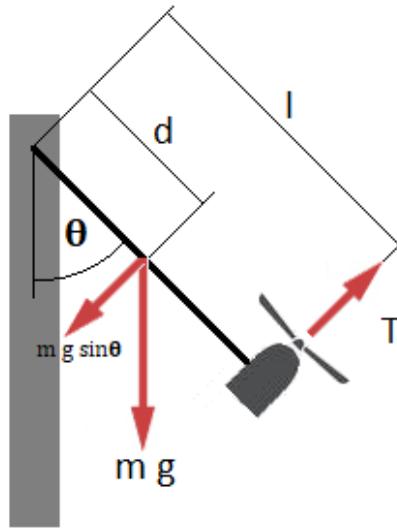
4.1 Identyfikacja parametrów

Stanowisko charakteryzuje parametry, które trzeba przenieść do modelu, niektóre z nich można zmierzyć, obliczyć, inne oszacować albo odczytać z tablic dostarczonych przez producenta.

- $m = 0.18 \text{ kg}$ - masa - silnik 150 gramów, uchwyt 30 gramów,
- $g = 9.81 \text{ m/s}^2$ - przyspieszenie grawitacyjne,
- $l = 0.25 \text{ m}$ - długość wahadła,
- $d = 0.25 \text{ m}$ – odległość osi od środka masy – bez odważnika cała masa jest skupiona w miejscu zamontowania silnika, zamontowanie i manipulacja odważnikiem pozwala na szybką zmianę środka masy wahadła na rzeczywistym stanowisku,
- $c = 0.0060 \text{ Nms/rad}$ - tarcie wiskotyczne występujące na łożyskach – oszacowane na podstawie tabeli, tarcie na pierścieniu ślizgowym z dokumentacji [28],
- $J = ml^2$ – moment bezwładności [kgm^2].

4.2 Model matematyczny na podstawie równań różniczkowych

Schemat wraz z rozkładem sił działających na obiekt został zaprezentowany na rysunku. Silnik wraz ze śmiigłem, który zamontowany jest na końcu aluminiowego pręta generuje ciąg (T) i jest źródłem siły wpływającej na układ.



Rysunek 4.1: Rozkład sił działających na wahadło [opracowanie własne]

Na podstawie rysunku, korzystając zasad dynamiki Newtona można wyprowadzić równanie różniczko-w opisujące układ, które zostało przedstawione w równaniu 4.1[26].,

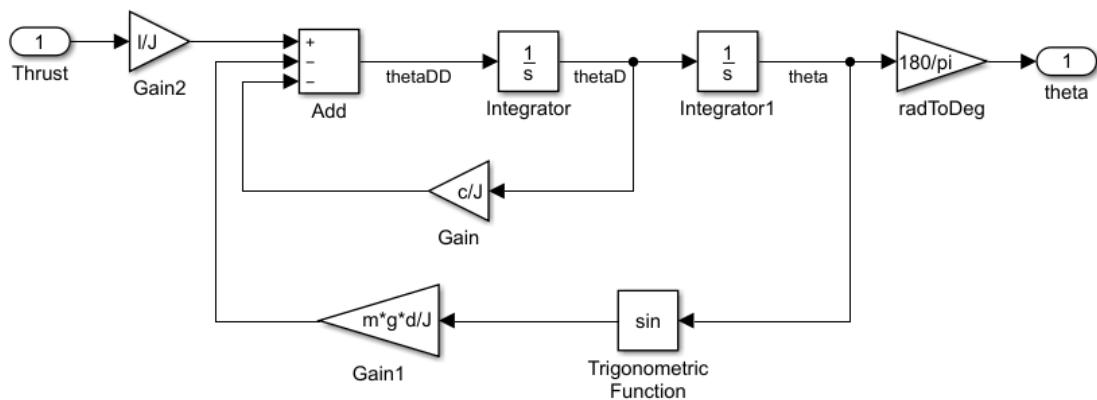
$$J\ddot{\theta} + c\dot{\theta} + mgdsin\theta = lT \quad (4.1)$$

Gdzie:

- m - masa wahadła,
- g - przyspieszenie grawitacyjne,
- d - odległość osi obrotu od środka masy,
- l - długość wahadła,
- c - tarcie wiskotyczne,
- J - moment bezwładności,
- T - ciąg generowany przez śmigło,
- θ - położenie wahadła,
- $\dot{\theta}$ - prędkość kątowa wahadła,
- $\ddot{\theta}$ - przyspieszenie kątowe wahadła.

Równanie zostało przekształcone do postaci, która posłuży do zbudowania modelu w simulinku (4.2) [29].

$$\ddot{\theta} = -\frac{c}{J}\dot{\theta} - \frac{mgd}{J}sin\theta + \frac{l}{J}T \quad (4.2)$$



Rysunek 4.2: Zbudowany model w Simulinku na podstawie równania 4.2 [opracowanie własne]

Kręcące się śmigło generuje siłę o wartości, która jest w stanie przeciwstawić się sile grawitacji, w wyniku czego, kąt wychylenia wahadła może przyjąć dowolną wartość i utrzymywać ją. Na siłę ciągu ma wpływ kilka parametrów są to m.in.: prędkość obrotowa wału silnika, skok oraz średnica śmigła. Aby uzależnić model od obrotów silnika trzeba zmodyfikować wzór i wprowadzić współczynnik α , który konwertuje obroty silnika na siłę ciągu [26].

$$T = \alpha * \omega_{RPM} \quad (4.3)$$

$$[N] = \left[\frac{k \cdot g \cdot m}{s} \right] * \left[\frac{rad}{s} \right] \quad (4.4)$$

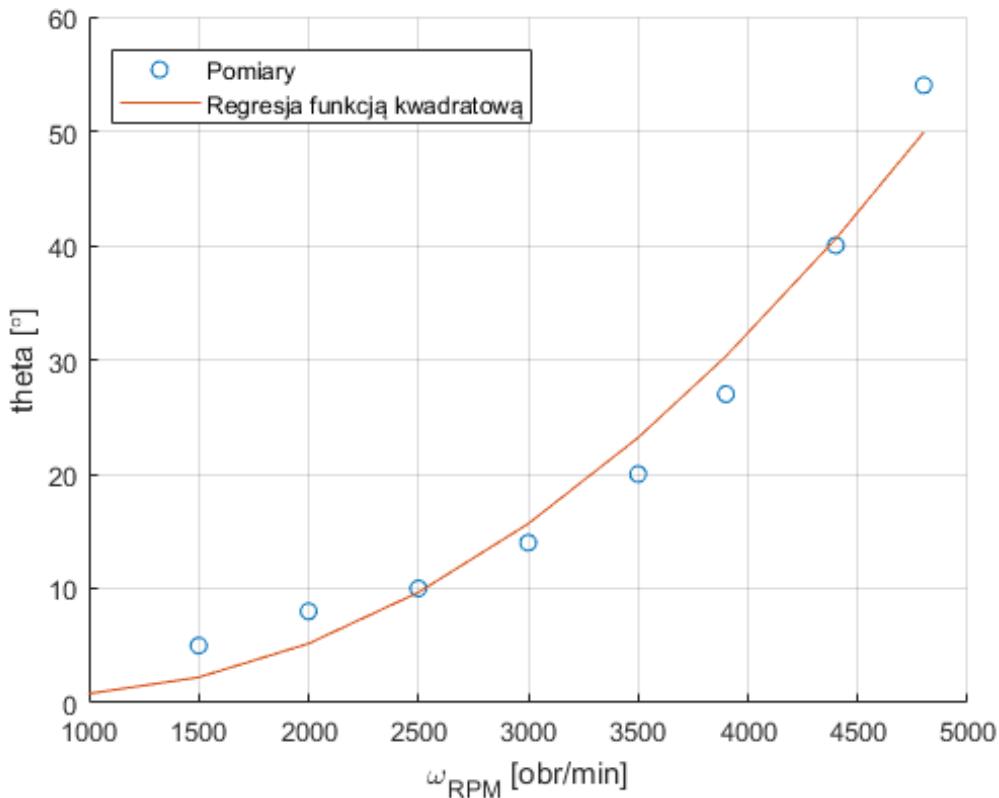
Gdzie:

T - ciąg wytworzony przez układ napędowy,

ω_{RPM} - obroty silnika ,

α - współczynnik konwertujący dla dodatnich obrotów silnika.

Możliwości napędu (maksymalny generowany ciąg; silnik jest w stanie wygenerować obroty rzędu około 5000 obrotów na minutę, w związku z tym dysponuje się skończoną siłą ciągu), można zbadać na specjalnych przeznaczonych do tego hamowaniach tensometrycznych, jednak do wyznaczenia współczynnika można wykorzystać zbudowane wahadło. Aby wyznaczyć współczynnik α należy zdjąć charakterystykę położenia wahadła od prędkości, poprzez zwiększenie prędkości i odczytywanie położenia po ustaleniu się pozycji wahadła.



Rysunek 4.3: Wykres położenia wahadła od obrotów silnika [opracowanie własne]

Zmierzone wartości zostały poddane dopasowaniu do funkcji kwadratowej. Na podstawie tej funkcji, z każdym uruchomieniem symulacji modelu jest liczony współczynnik α , który wprowadzono we wzorze 4.3.

Zbudowane wahadło posiada możliwość osiągania zarówno kątów dodatnich jak i ujemnych, zastosowane śmiegle jest śmigłem działającym prawidłowo dla silnika obracającego się w jedną stronę, jeżeli silnik kręci się w stronę przeciwną niż taka, do której śmiegle jest przystosowane, moc generowanego ciągu spada, dlatego należy wyznaczyć dwie charakterystyki i na ich podstawie obliczać współczynniki α oraz α_{rev} , a w modelu stosować je w zależności od zadanych obrotów silnika. Równanie 4.5 zawiera współczynnik konwertujący dla ujemnych obrotów silnika.

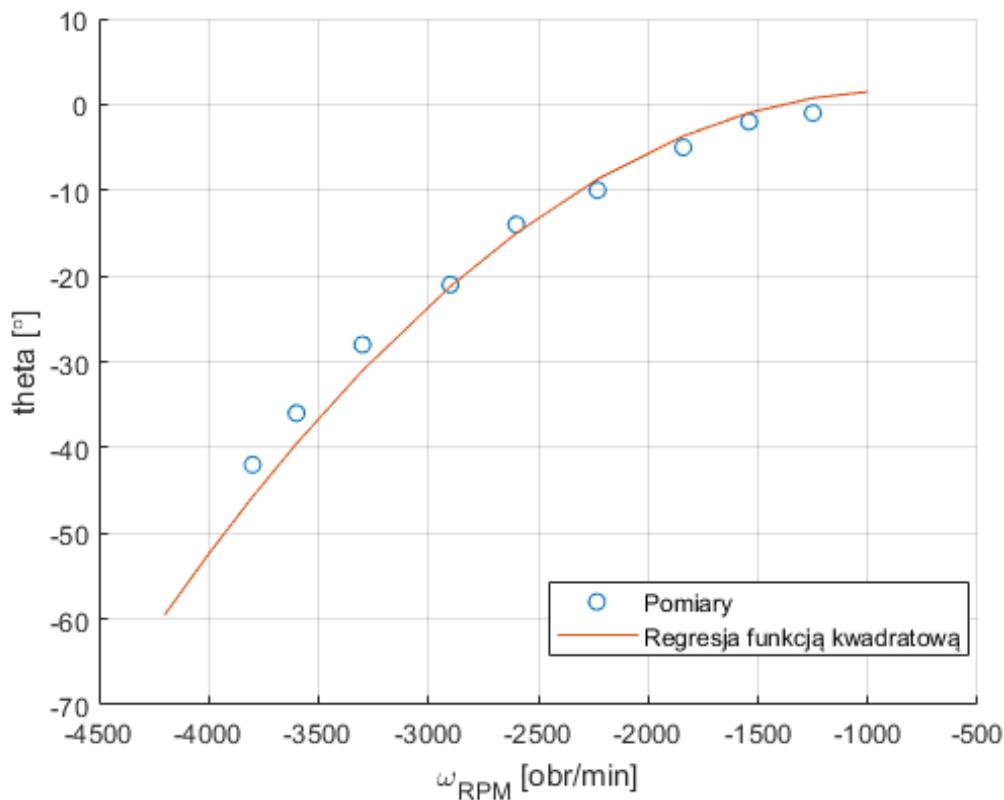
$$T = \alpha_{rev} * \omega_{RPM} \quad (4.5)$$

Gdzie:

T - ciąg wytworzony przez układ napędowy,

ω_{RPM} - obroty silnika ,

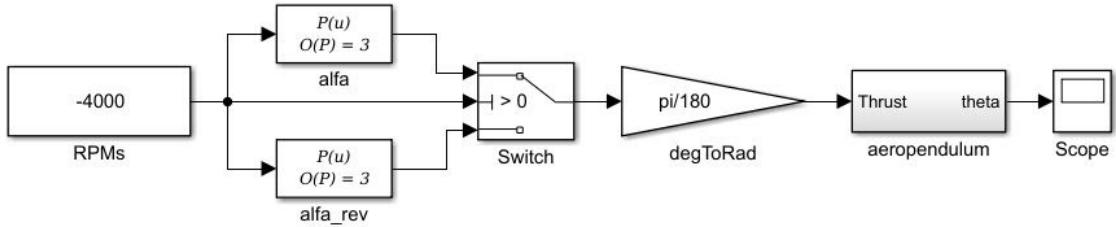
α_{rev} - współczynnik konwertujący dla ujemnych obrotów silnika.



Rysunek 4.4: Wykres położenia wahadła od ujemnych obrotów silnika [opracowanie własne]

Jak wynika z przeprowadzonych doświadczeń charakterystyka dla zadawanych obrotów ujemnych osiąga analogiczne wartości kątów dla mniejszych obrotów, to też śmigło działa zgodnie ze swoim przeznaczeniem dla ujemnych obrotów silnika. Ponadto niewiele poniżej -4000 obrotów wahadło traci równowagę i zaczyna wirować. Należy pamiętać, aby przy zadawaniu prędkości nie przekraczać tej wartości dla wahadła bez założonej przeciwagi. Dla wahadła obciążonego, ta wartość obrotów krytycznych zmniejsza się jeszcze bardziej (zmienia się wtedy moment bezwładności wahadła), ponieważ środek ciężkości wędruje w stronę osi obrotu, w związku z czym ciąg potrzebny do osiągnięcia analogicznych wartości kątów również maleje, zatem współczynnik konwertujacy także się zmienia. Aby rzeczywista wartość współczynnika α pokrywała się z modelem, należy wyznaczyć charakterystykę, za każdym razem, kiedy zostanie dodana przeciwaga, bądź jej pozycja została zmieniona.

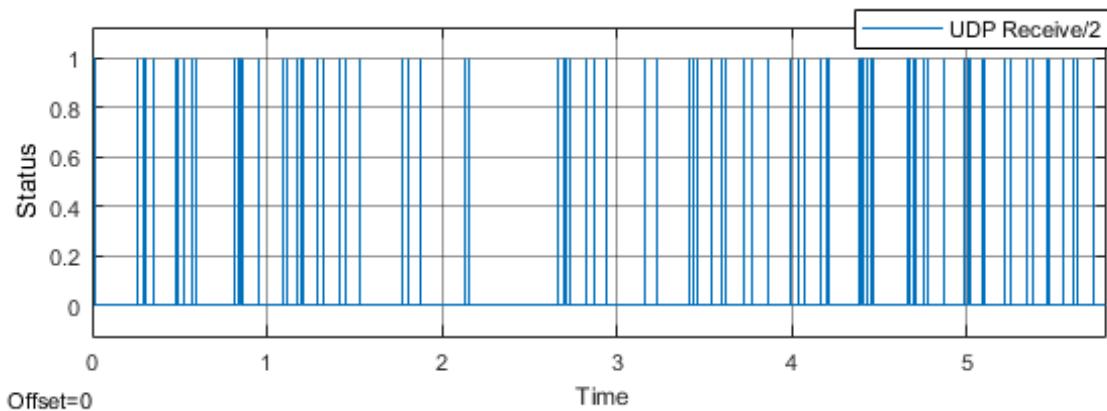
4.3 Testy w pętli otwartej



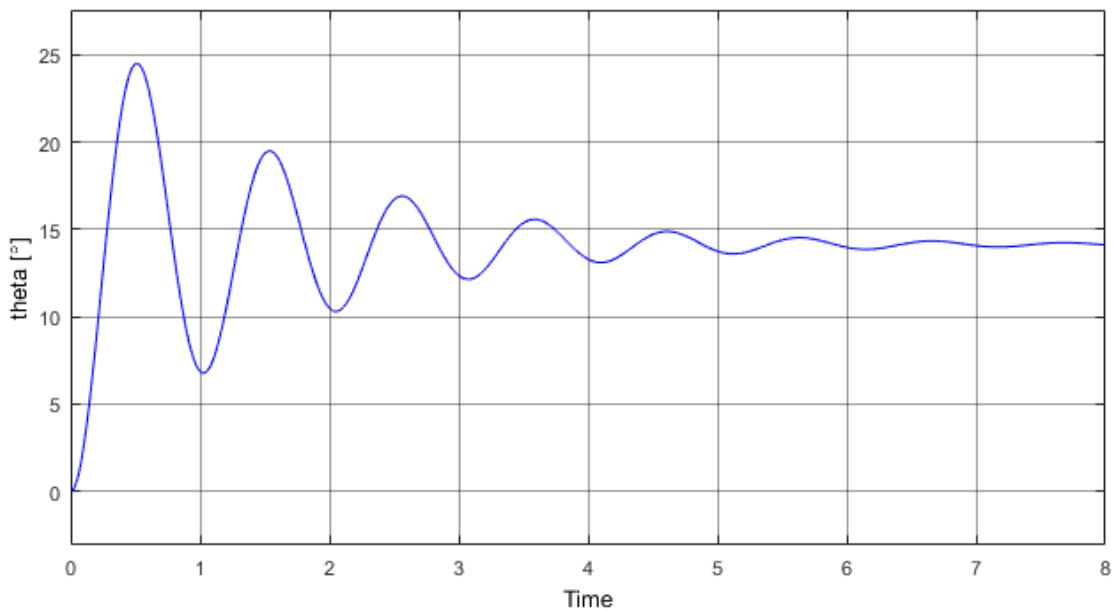
Rysunek 4.5: Zmodyfikowany model [opracowanie własne]

Tak zmodyfikowany model jest w stanie dokładniej odwzorować zbudowane stanowisko i odpowiadać właściwymi charakterystykami na zadane obroty silnika. Stosownym jest wyjaśnienie sensu istnienia bloku przekształcającego sygnał na radiany. Model oblicza położenie przy zastosowaniu miary łukowej kąta, współczynniki α zostały obliczone na podstawie położenia liczonego w stopniach. Sygnał jest mnożony przez odpowiedni wielomian w zależności od obrotów silnika (warunek bloku *Switch*).

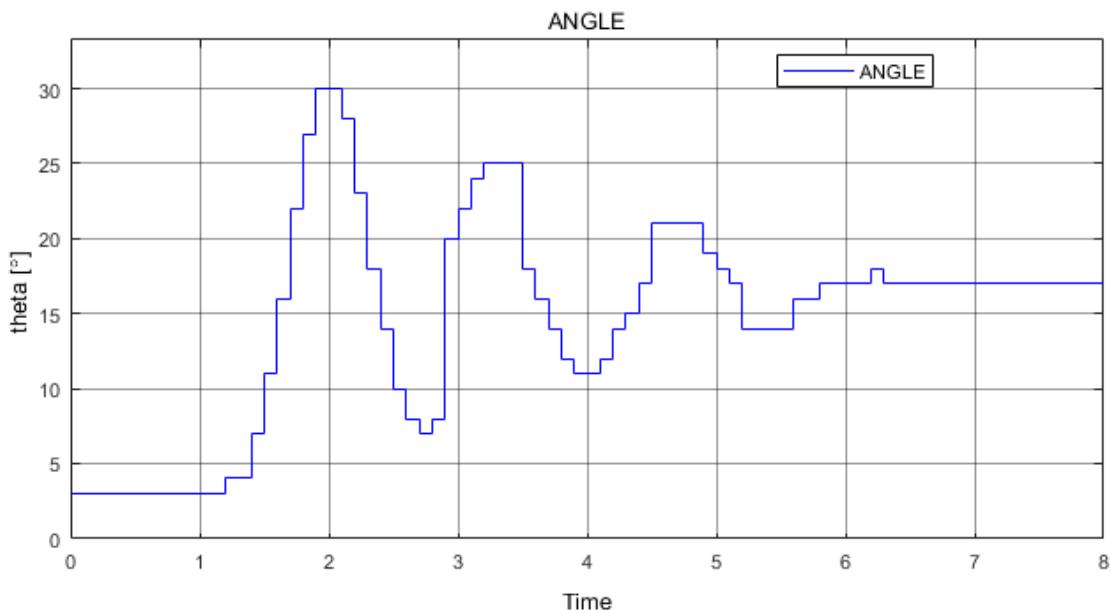
Poniżej znajduje się porównanie odpowiedzi uzyskanych za pomocą symulacji oraz zarejestrowanych za pomocą enkodera na rzeczywistym stanowisku. W symulacji nie uwzględniono opóźnienia pomiarowego. Czas próbkowania może zostać obliczony na podstawie czasu symulacji oraz liczby odebranych próbek. Obliczanie próbkowania w ten sposób jest jednak zależne od liczby poprawnie odebranych pakietów danych, oraz od aktywnego trybu blokowania bloku UDP Receive. Wyłączenie trybu blokowania sprawia, że program ustawia wartość STATUS na 1, jeżeli w buforze znajduje się odpowiednia ilość danych oraz na 0, jeżeli liczba danych jest niewystarczająca, wtedy program czeka na zapełnienie się buforu, pobiera z buforu odpowiednią ilość danych zgodnie z kolejnością ich przybycia i zmienia wartość STATUSTU na 1 (Rysunek 4.6). Jeżeli w trybie blokowania w buforze brakuje danych symulacja jest blokowana do czasu, aż w buforze znajdzie się odpowiednia liczba danych. W ogólności czas odbierania kolejnych próbek nie jest stały, odnotowano zakres od 15 do 90 ms. Występujące rozbieżności mogą wynikać z niedoszacowania parametrów obiektu oraz warunków środowiska, w których znajdowało się stanowisko. Należy mieć na uwadze, że na wahadło wpływają generowane przez śmigło podmuchy powietrza (szczególnie w małych pomieszczeniach; występowanie drgań gdy stanowisko znajduje się za osłoną).



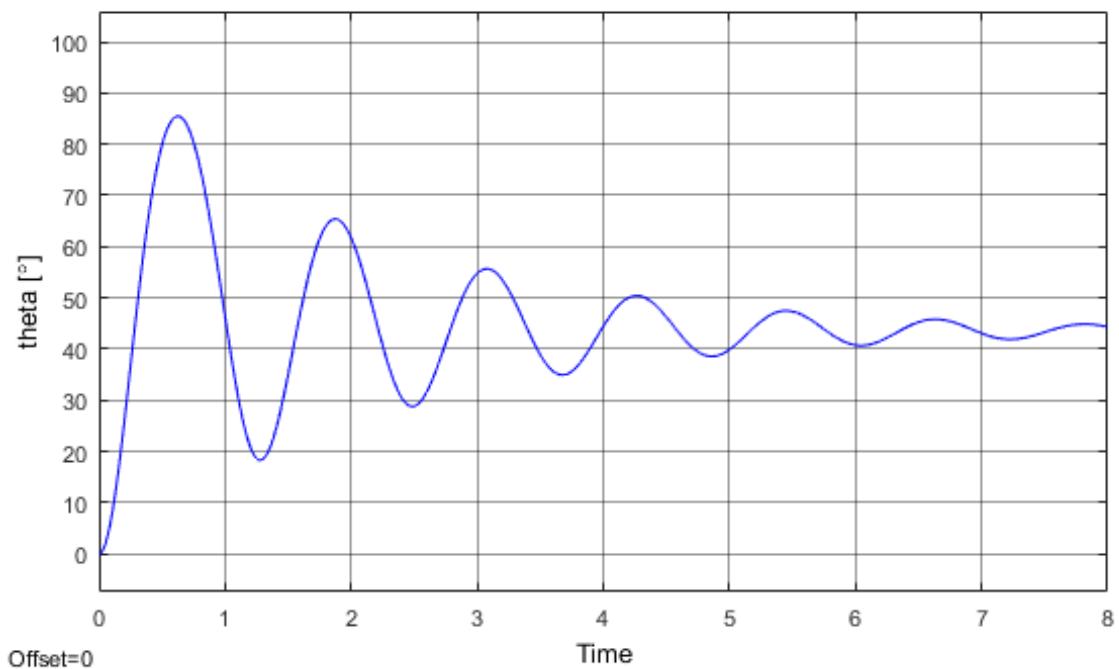
Rysunek 4.6: UDP Receive wyłączony tryb blokujący



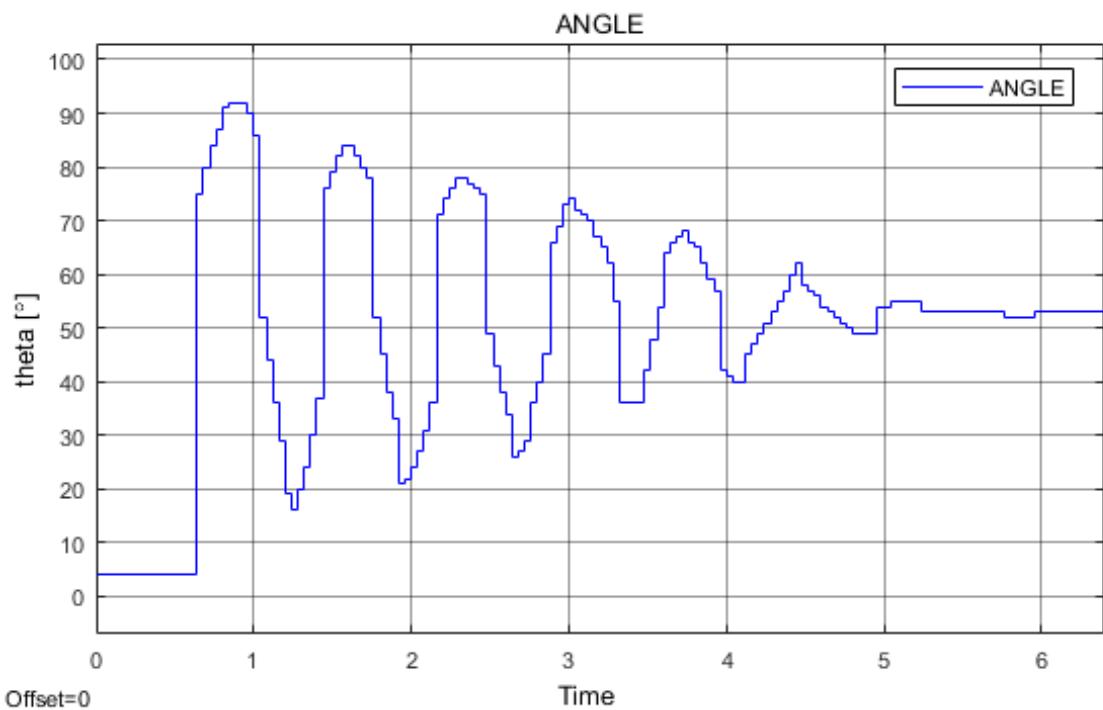
Rysunek 4.7: Odpowiedź modelu dla 3000 obrotów na minutę [opracowanie własne]



Rysunek 4.8: Odpowiedź stanowiska dla 3000 obrotów na minutę [opracowanie własne]



Rysunek 4.9: Odpowiedź modelu dla 4800 obrotów na minutę [opracowanie własne]



Rysunek 4.10: Odpowiedź stanowiska dla 4800 obrotów na minutę [opracowanie własne]

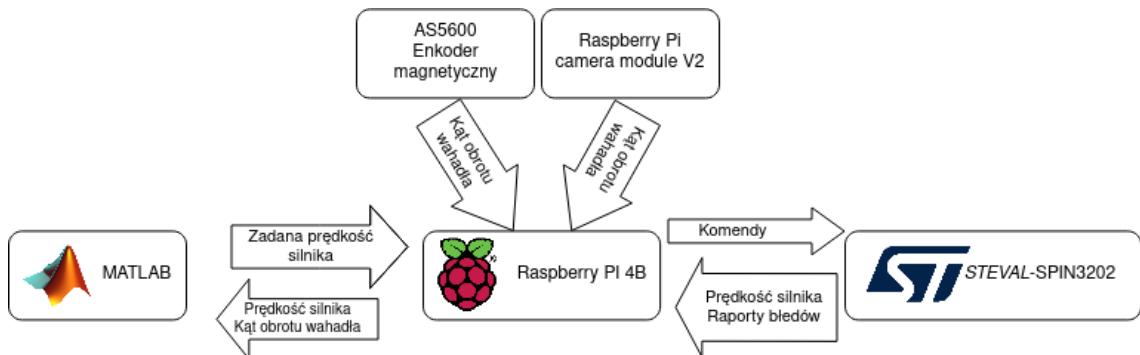
Rozdział 5

Oprogramowanie

Autor: Jakub Walkowski

Zbudowane stanowisko zawiera pięć głównych modułów wymieniających ze sobą dane:

- raspberry Pi 4B,
- sterownik STEVAL-SPIN3201,
- enkoder magnetyczny AS5600,
- program Matlab,
- kamera Raspberry Pi camera module V2.



Rysunek 5.1: Schemat wymiany danych pomiędzy elementami stanowiska [opracowanie własne]

Głównym zadaniem programistycznym pracy, było opracowanie rozwiązań pozwalających na płynną komunikację pomiędzy elementami wchodzącyymi w skład stanowiska. W tym celu należało opracować cztery główne elementy:

- Program uruchamiany na sterowniku STEVAL-SPIN 3201 realizujący regulację prędkości silnika oraz pozwalający na komunikację z nim.
- Bibliotekę dostarczającą narzędzia do obsługi enkodera magnetycznego odczytującego kąt wychylenia wahadła.
- Oprogramowanie dla RPI będącego centralnym elementem wymiany danych pomiędzy wszystkimi elementami stanowiska.

- System wizyjny, przetwarzający obraz na kąt wychylenia wahadła.

Całość prac programistycznych została przeprowadzona wraz z użyciem systemu kontroli wersji. Użyto w tym celu oprogramowanie Git, natomiast repozytorium zdalne umieszczone zostało w serwisie Github [1]. Dostęp do repozytorium zdalnego otrzymali wszyscy członkowie zespołu, co pozwoliło na zorganizowanie wspólkiej i zdalnej pracy, a także ułatwiło dostęp do kodu źródłowego tworzonego oprogramowania.

Całość napisanego kodu źródłowego została opatrzona odpowiednimi komentarzami, pisany w standardzie przeznaczonym dla programu doxygen [2]. Pozwoliło to na wygenerowanie dokumentacji kodu źródłowego tworzonego oprogramowania. Wygenerowana dokumentacja w formacie pdf została dołączona do niniejszej pracy w formie załącznika.

5.1 Raspberry Pi

W celu realizacji komunikacji pomiędzy elementami stanowiska zastosowany został mikrokomputer Raspberry Pi 4B. Wykorzystany model posiada czterordzeniowy procesor Broadcom BCM2711 o architekturze 64-bitowej i taktowaniu 1.5 Ghz. Wyposażony został również w 2GB pamięci RAM oraz slot na kartę Micro-SD wykorzystywanej jako pamięć masowa. W tym przypadku użyta została karta o pojemności 64GB [3]. Głównymi elementami RPI pozwalającymi na opracowanie odpowiednich rozwiązań programistycznych są:

- Moduł bezprzewodowej komunikacji pozwalający na dostęp do sieci WLAN.
- Moduł sieciowy Ethernet pozwalający na dostęp do sieci LAN.
- 40 pinów GPIO pozwalających na skorzystanie między innymi z interfejsu I2C.
- 4 porty USB pozwalające wykorzystać komunikację szeregową.
- Port CSI pozwalający na podłączenie dedykowanej RPI kamery.

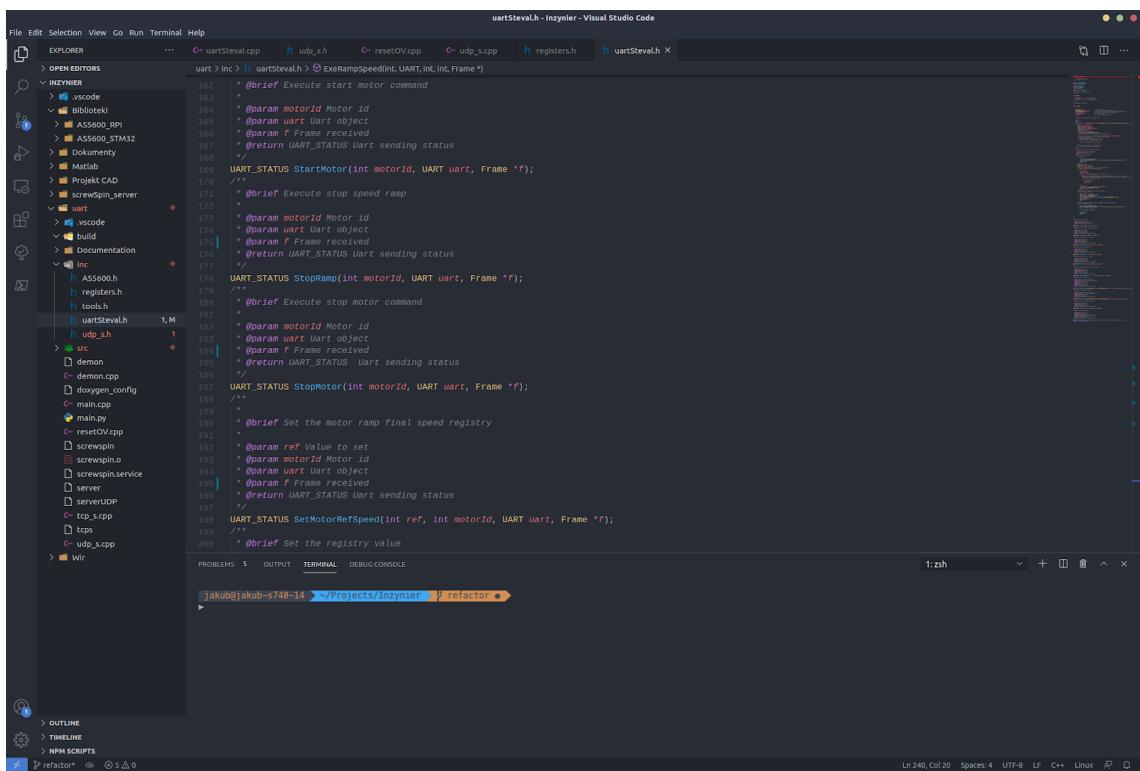
Na RPI zainstalowany został system operacyjny Raspberry Pi OS (dawniej Raspbian) w wersji 5.4. Jest to system operacyjny będący darmową dystrybucją systemu Linux opartą na dystrybucji Debian, która została specjalnie przystosowana do współpracy ze wszystkimi podzespołami RPI [4]. Oprogramowanie zostało zainstalowane w wersji Desktop, która zawiera wbudowany interfejs graficzny.

Istotnym elementem pozwalającym na stworzenie wymaganego oprogramowania był moduł sieci bezprzewodowej. Dzięki niemu możliwa była zdalna realizacja pracy przy użyciu komputerów osobistych znajdujących się w sieci lokalnej razem z RPI, bez potrzeby bezpośredniego korzystania z mikrokomputera. Rozwiązanie to było możliwe dzięki zastosowaniu protokołu secure shell (w skrócie ssh) służącemu do terminalowego, zdalnego łączenia się z komputerami [5].

Z uwagi na sytuację pandemiczną, przypadającą na czas realizacji pracy i fakt posiadania jednego stanowiska w czasie opracowywania oprogramowania, istotne było łączenie się z RPI bez konieczności znajdowania się w tej samej sieci lokalnej.

Rozwiążaniem tego problemu było skonsultowanie się z lokalnym dostawcą internetu z prośbą o ustawienie publicznego IP. Następnie, aby umożliwić łączenie się za pomocą owego publicznego IP przez protokół ssh, wymagane było przekierowanie portu 22, typowego dla tego protokołu, na lokalny adres RPI. Przekierowania tego można dokonać w ustawieniach routera, z którym łączy się mikrokomputer.

Do napisania oprogramowania wykorzystano darmowy edytor kodu źródłowego Visual Studio Code. Edytor ten posiada wiele funkcji ze względu na swoją modułowość, która pozwala instalować różnego rodzaju dodatki wspomagające prace programistyczne. Jednym z nich był C/C++ ułatwiający pisanie programów w języku C i C++. Kolejnym istotnym dla stworzenia pracy rozszerzeniem był dodatek Remote-SSH. Wykorzystuje on protokół ssh, pozwalając dzięki niemu na swobodną zdalną pracę z plikami znajdującymi się w pamięci mikrokomputera.



Rysunek 5.2: Interfejs edytora kodu źródłowego Visual Studio Code [opracowanie własne]

Utworzone oprogramowanie spełnia następujące zadania:

- Komunikacja sieciowa z programem Matlab uruchomiona na komputerze osobistym pozwalająca na wysyłanie do programu wartości mierzonych oraz odbieranie obliczanych w nim wartości zadanego.
- Komunikacja ze sterownikiem STEVAL-SPIN poprzez port szeregowy z wykorzystaniem protokołu MCP (Motor Control Protocol).
- Odbieranie danych o wychyleniu wahadła z enkodera magnetycznego.
- Pomiar kąta wychylenia wahadła za pomocą dedykowanej kamery Raspberry Pi.

Całość oprogramowania zawiera się w jednym pliku wykonawczym, który otrzymywany jest poprzez komplikację z użyciem programu g++ [6]. Wywołanie komplikacji oprogramowania zostało przedstawione w listingu 5.1.

Listing 5.1: Wywołanie komplikacji oprogramowania dla RPI.

```
01. g++ src/tools.cpp src/registers.cpp src/AS5600.cpp src/uartStevel.cpp udp_s.  
     cpp -lwiringPi -lpthread -o serverUDP
```

Uzyskany program uruchamiany jest w systemie Linux jako demon. Jest to proces drugoplanowy, wykonywany wewnątrz środowiska wielozadaniowego systemu operacyjnego bez konieczności interakcji z użytkownikiem. W celu uruchamiania programu wraz ze startem RPI użyty został menadżer systemu i usług dla Linuxa, o nazwie systemd. Pozwala on na tworzenie usług oraz zarządzanie nimi. Udostępnia także szereg możliwości takich jak wystartowanie, zatrzymanie i resetowanie usługi [7].

W celu utworzenia usługi należało sporządzić plik o dowolnej nazwie kończącej się rozszerzeniem .service, a następnie umieścić go w folderze */etc/systemd/system* [8]. Utworzona usługa o nazwie “screwspin” przedstawiona została w listingu 5.2.

Listing 5.2: Plik konfiguracyjny usługi screwspin.service

```
01. [Unit]  
02. Description = Screwspin server service  
03. After = network.target  
04.  
05. [Service]  
06. ExecStart = /home/pi/Inzynier/uart/serverUDP  
07. WorkingDirectory = /home/pi/Inzynier/uart  
08. StandardOutput = inherit  
09. StandardError = inherit  
10. Restart = on-failure  
11. RestartSec = 10s  
12. User = pi  
13.  
14. [Install]  
15. WantedBy = multi-user.target
```

Opis poszczególnych parametrów został umieszczony w tabeli 5.1 [7].

Tabela 5.1: Tabela wybranych parametrów konfiguracyjnych usługi

Nazwa	Opis
Description	opis danej usługi
After	lista programów, po których ma nastąpić uruchomienie usługi
ExecStart	ścieżka do pliku wykonawczego usługi
WorkingDirectory	ścieżka robocza w której operować ma program usługi
StandardOutput	definiowanie połączenia deskryptora pliku 1 (stdout) uruchamianego programu
StandardError	definiowanie połączenia deskryptora pliku 2 (stderr) uruchamianego programu
Restart	wydarzenie, przy którym usługa ma zostać zrestartowana
RestartSec	wartość opóźnienia przed restartem
User	nazwa użytkownika uruchamiającego program usługi
WantedBy	lista zawierająca inne usługi, od których zależy ta usługa

Wystartowanie usługi wywołane z konsoli przedstawiono w listingu 5.3.

Listing 5.3: Start serwisu

```
01. sudo systemctl start screwspin.service
```

Zatrzymanie usługi zostało zaprezentowane w listingu 5.4.

Listing 5.4: Stop serwisu

```
01. | sudo systemctl stop screwspin.service
```

W celu uruchomienia automatycznego startu usługi, wraz ze startem systemu, należy posłużyć się poleceniem zaprezentowanym w listingu 5.5.

Listing 5.5: Uruchomienie automatyczne wraz ze startem systemu

```
01. | sudo systemctl enable screwspin.service
```

Stworzone oprogramowanie zostało dodatkowo wyposażone w system logowania błędów do pliku. System ten opiera się na dwóch funkcjach *start_log()* oraz *stop_log()*. Deklaracja tych funkcji została przedstawiona 5.6.

Listing 5.6: Deklaracje funkcji służących do logowania błędów

```
01. | #define LOG_PATH "/var/log/screwspin/log"
02.
03. /**
04. * @brief Print actual server time
05. *
06. */
07. void print_log_time();
08. /**
09. * @brief Start printing to log file
10. *
11. * @param path Path to log file
12. * @return long two int fd
13. */
14. long start_log(char const *path);
15. /**
16. * @brief Stop printing to log file
17. *
18. * @param fd12 long two int fd
19. */
20. void stop_log(long fd12);
```

W przypadku gdy zaistnieje potrzeba umieszczenia jakiejś informacji w logach należy wywołać funkcję *start_log()*. Po jej wywołaniu następuje otwarcie wskazanego pliku i przekierowanie do niego wyjścia standardowego. Dodatkowo w nowej linii automatycznie dodana zostanie informacja z datą i czasem wystąpienia zdarzenia. Po wywołaniu tej funkcji wszystkie przekazane do wyjścia standardowego komunikaty zostają wpisane do wskazanego pliku. Aby zakończyć procedurę logowania błędów należy wywołać funkcję *stop_log()* przekazując jako argument wartość zwróconą przez *start_log()*. Przykładowe użycie tego mechanizmu zaprezentowano na listingu 5.7.

Listing 5.7: Przykładowe użycie mechanizmu logowania błędu do pliku

```
01. | long log = start_log(LOG_PATH);
02. | printf("[ERROR_UART_DATA]Wrong frame lenght (%d).\n", j);
03. | stop_log(log);
```

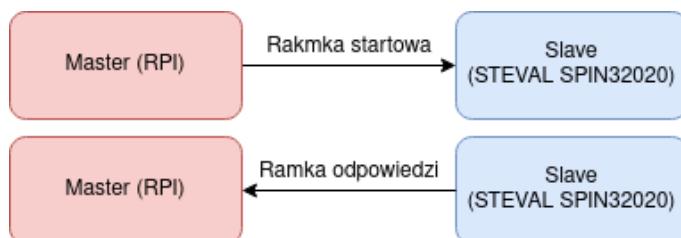
Mechanizm ten był kluczowy podczas prac nad stanowiskiem, dając możliwość łatwego wychwytywania błędów w celu udoskonalenia opracowanego oprogramowania.

5.1.1 Komunikacja za pośrednictwem MCP - Motor Control Protocol

Komunikacja pomiędzy Raspberry Pi a sterownikiem silnika została zrealizowana przy użyciu transmisji szeregowej. Sposób ten został narzucony przez ograniczone możliwości kontrolera, który posiada jedynie port szeregowy. Program obsługujący komunikację po stronie sterownika został w pełni wygenerowany przez program STM32 Motor Control Workbench i opiera się on na protokole MCP. Protokół ten stworzony został specjalnie z myślą o szeregowej transmisji danych pomiędzy układami sterowania silnikiem a innym urządzeniami. Pozwala on na odczyt oraz zapis danych w rejestrach kontrolera czy też wysyłanie komend takich jak stop/start silnika.

Protokół MCP opiera się na porcie szeregowym i wysyłaniu przez niego ramek transmisji UART. Wysyłane po sobie ramki tworzą ze sobą większą strukturę, nazwaną w zależności od roli urządzenia w wymianie danych, ramką startową bądź ramką odpowiedzi [9].

Całość komunikacji opiera się na relacji master - slave, gdzie master w naszym przypadku to RPI a slave sterownik. Komunikacja rozpoczyna się gdy master/RPI wyśle ramkę startową. W momencie w którym ramka zostaje otrzymana sterownik odsyła ramkę odpowiedzi zawierającą np. odczytaną wartość rejestru, informacje o błędzie bądź informację potwierdzającą poprawne wykonanie komendy. Struktura master - slave przejawia się w tym, iż sterownik odpowiada dopiero gdy zostanie wywołany przez urządzenie nadziedzne, w tym przypadku RPI.



Rysunek 5.3: Schemat komunikacji master - slave [opracowanie własne]

Konstrukcja ramki startowej

Biblioteka pozwalająca RPI na komunikację za pomocą protokołu MCP została napisana w języku C++ przy wykorzystaniu biblioteki wiringPi, która pozwala na obsługę pinów GPIO znajdujących się w RPI.[30] Całość komunikacji opiera się na wysyłaniu i odbieraniu przez port szeregowy tzw. ramek danych. Biblioteka przechowuje dane w specjalnej strukturze *Frame* (listing 5.8), która reprezentuje zarówno ramkę startową jak i ramkę odpowiedzi.

Listing 5.8: Definicja struktury *Frame*

```

01. struct Frame
02. {
03.
04.     int motor_id;           /**< Motor id */
05.     FRAME_CODES frame_id;  /**< Frame code from FRAME_CODES enum */
06.     STEVAL_REGISTERS payload_id;  /**< Motor control register id from
07.                                     STEVAL_REGISTERS enum */
08.     int payload_len;        /**< Motor control register payload length */
09.     long long payload;    /**< Data in frame */
10.     int CRC;               /**< Cyclic redundancy check */
  
```

Konstrukcja ramki startowej ma określoną stałą strukturę. W jej skład wchodzi:

- *Frame_start*,
- *Payload_length*,
- *Payload_id*,
- *Payload*,
- *CRC*.

Tabela 5.2: Podstawowa konstrukcja pojedynczej ramki

<i>Frame_start</i>	<i>Payload_length</i>	<i>Payload_id</i>	<i>Payload[0]</i>	...	<i>Payload[n]</i>	<i>CRC</i>
--------------------	-----------------------	-------------------	-------------------	-----	-------------------	------------

Frame_start

Jest to bajt, którego trzy najbardziej znaczące bity zawierają informacje o wybranym silniku, a pozostałe o rodzaju wysyłanej ramki. Obliczanie tego bajtu na podstawie pól w strukturze zostało przedstawione na listingu 5.9.

Listing 5.9: Obliczanie bajtu *frame_start* ramki MCP

```
01. | (motor_id << 5) + (int)frame_id)
```

Wszystkie dostępne *Frame_Id* reprezentowane są przy użyciu klasy wyliczeniowej o nazwie *FRAME_CODE*(listing 5.10).

Listing 5.10: Klasa wyliczeniowa reprezentująca *frame_id* dla ramki MCP

```
01. enum class FRAME_CODES
02. {
03.     SET          = 0x1, /*<< Set register frame. It is used to write a value
04.                  into a relevant motor control variable. */
05.     GET          = 0x2, /*<< Get register frame. It is used to read a value from
06.                  a relevant motor control variable. */
07.     EXE          = 0x3, /*<< Execute command frame. It is used to send a command
08.                  to the motor control object. */
09.     INFO         = 0x6, /*<< Get board info. It is used to retrieve information
10.                  about the firmware currently running on the microcontroller. */
11.     EXE_RAMP     = 0x7, /*<< Exec ramp. It is used to execute a speed ramp. */
12.     GET_REVUP   = 0x8, /*<< Get revup data. It is used to retrieve the revup
13.                  parameters. */
14.     SET_REVUP   = 0x9, /*<< Set revup data. It is used to set the revup
15.                  parameters. */
16.     SET_REF     = 0xA, /*<< Set current references. It is used to set the
17.                  current reference. */
18.     RECEIVE      = 0xF0, /*<< Frame code for received frames. */
19.     ERROR        = 0xFF /*<< Frame with error, error code store in data. */
20. };
```

Dodatkowo klasa *FRAME_CODE* zawiera dwa pola niebędące *Frame_Id*. Są one przeznaczone dla ramek odpowiedzi, gdzie *RECEIVE* informuje o tym, że dana ramka jest ramką odpowiedzi, a *ERROR* informuje iż zawiera ona kod błędu.

Payload_length

Payload_length informuje o tym ile bajtów przesyłanych danych zawiera ramka. Informacja ta wykorzystywana jest zarówno przy ramce startowej jak i ramce odpowiedzi. *Payload_length* dla poszczególnych rejestrów kontrolera zawarty został w klasie wyliczeniowej o nazwie *STEVAL_REGISTERS_LEN*(listing 5.11).

Listing 5.11: Klasa wyliczeniowa reprezentująca *payload_length* dla ramki MCP

```

01. /**
02. * @brief List of relevant motor control register payload length
03. */
04. enum class STEVAL_REGISTERS_LEN
05. {
06.     EXE_CMD      = 1, /*< Payload length on execute command */
07.     GET          = 1, /*< Payload length on get register value */
08.     TARGET        = 3, /*< Target motor, RW */
09.     FLAGS         = 3, /*< Flags, R */
10.     STATUS        = 3, /*< Status, R */
11.     MODE          = 3, /*< Control mode, RW */
12.     SPEED_REF    = 3, /*< Speed reference, R */
13.     SPEED_KP     = 3, /*< Speed KP, RW */
14.     SPEED_KI     = 3, /*< Speed KI, RW */
15.     SPEED_KD     = 3, /*< Speed KD, RW */
16.     TOR_REF      = 3, /*< Torque reference (Iq), RW */
17.     TOR_KP       = 3, /*< Torque KP, RW */
18.     TOR_KI       = 3, /*< Torque KI, RW */
19.     TOR_KD       = 3, /*< Torque KD, RW */
20.     FLUX_REF     = 3, /*< Flux reference (Id), RW */
21.     FLUX_KP      = 3, /*< Flux KP, RW */
22.     FLUX_KI      = 3, /*< Flux KI, RW */
23.     FLUX_KD      = 3, /*< Flux KD, RW */
24.     OBS_C1       = 3, /*< Observer C1, RW */
25.     OBS_C2       = 3, /*< Observer C2, RW */
26.     C_OBS_C1    = 3, /*< Cordic Observer C1, RW */
27.     C_OBS_C2    = 3, /*< Cordic Observer C2, RW */
28.     PLL_KI       = 3, /*< PLL KI , RW */
29.     PLL_KP       = 3, /*< PLL KP, RW */
30.     FLUX_WEAK_KP = 3, /*< Flux weakening KP, RW */
31.     FLUX_WEAK_KI = 3, /*< Flux weakening KI, RW */
32.     FLUX_WEAK_BUS= 3, /*< Flux weakening BUS Voltage allowed percentage
                           reference, RW */
33.     BUS_VOL      = 3, /*< Bus Voltage, R */
34.     HEATSINK_TEMP= 3, /*< Heatsink temperature, R */
35.     MOTOR_POWER   = 3, /*< Motor power, R */
36.     DAC_OUT1     = 2, /*< DAC Out 1, R */
37.     DAC_OUT2     = 2, /*< DAC Out 2, R */
38.     SPEED_MEAS   = 5, /*< Speed measured, R */
39.     TOR_MEAS     = 3, /*< Torque measured (Iq), R */
40.     FLUX_MEAS    = 3, /*< Flux measured (Id), R */
41.     FLUX_WBV     = 3, /*< Flux weakening BUS Voltage allowed percentage
                           measured, R */
42.     RS_NUM       = 2, /*< Revup stage numbers, R */
43.     MAX_APP_SPEED= 5, /*< Maximum application speed, R */
44.     MIN_APP_SPEED= 5, /*< Minimum application speed, R */
45.     IQ_REF       = 3, /*< Iq reference in speed mode, W */
46.     EXP_BEMF_PLL = 3, /*< Expected BEMF level (PLL), R */
47.     OBS_BEMF_PLL = 3, /*< Observed BEMF level (PLL), R */
48.     EXP_BEMF_CORDIC= 3, /*< Expected BEMF level (CORDIC), R */
49.     OBS_BEMF_CORDIC= 3, /*< Observed BEMF level (CORDIC), R */
50.     FF_1Q        = 5, /*< Feedforward (1Q), RW */
51.     FF_1D        = 5, /*< Feedforward (1D), RW */
52.     FF_2          = 5, /*< Feedforward (2), RW */
53.     FF_VQ        = 3, /*< Feedforward (VQ), R */
54.     FF_VD        = 3, /*< Feedforward (VD), R */
55.     FF_VQ_PI_OUT= 3, /*< Feedforward (VQ PI out), R */
56.     FF_VD_PI_OUT= 3, /*< Feedforward (VD PI out), R */
57.     RAMP_FIN_SPEED= 5, /*< Ramp final speed, RW */
58.     RAMP_DUR     = 3, /*< Ramp duration, RW */
59.     RAMP_EXE     = 6 /*< PL for exe ramp frame */
60. };

```

Przedstawiona na listingu 5.11 klasa zawiera dwa dodatkowe pola: *EXE_CMD* oraz *GET*. *EXE_CMD* informuje o *Payload_length* przeznaczonym dla ramki o *Frame_Id* odpowiadającemu wykonaniu komendy, natomiast *GET* wykorzystywane jest przy ramkach wysyłanych w celu odczytu wartości rejestru kontrolera.

Payload_Id

Bajt ten zawiera informacje o tym co reprezentują dane zawarte w *payload*. W większości przypadków jest to identyfikator rejestru kontrolera, reprezentujące parametry sterownika np. wartość prędkości zadanej. W przypadku ramki wykonującej komendę, *payload_id* informuje jaki program powinien zostać zrealizowany. Dostępne komendy przedstawione zostały w tabeli 5.3:

Tabela 5.3: Lista komend wykonywanych przez sterownik

Komenda	ID komendy	Opis
Start Motor	0x01	Polecenie wystartowania silnika niezależnie od jego stanu
Stop Motor	0x02	Polecenie zatrzymania silnika niezależnie od jego stanu
Stop Ramp	0x03	Polecenie zatrzymania aktualnie wykonywanej rampy prędkości
Start/Stop	0x06	Polecenie startu/stopu silnika w zależności od jego stanu
Fault Ack	0x07	Informuje kontroler o wiedzy użytkownika o zaistniałych błędach
Encoder Align	0x08	Polecenie wykonania procedury wyrównania enkodera

Wszystkie *payload_id*, które są możliwe do rozpoznania przez kontroler zostały zawarte w klasie wyliczeniowej o nazwie *STEVAL_REGISTERS*(listing 5.12).

Listing 5.12: Klasa wyliczeniowa reprezentująca *payload_id* dla ramki MCP

```

01. enum class STEVAL_REGISTERS
02. {
03.     NO_REG          = -1,    /**< No registry */
04.     TARGET          = 0x0,   /**< Target motor, RW */
05.     FLAGS           = 0x1,   /**< Flags, R */
06.     STATUS          = 0x2,   /**< Status, R */
07.     MODE            = 0x3,   /**< Control mode, RW */
08.     SPEED_REF       = 0x4,   /**< Speed reference, R */
09.     SPEED_KP        = 0x5,   /**< Speed KP, RW */
10.    SPEED_KI        = 0x6,   /**< Speed KI, RW */
11.    SPEED_KD        = 0x7,   /**< Speed KD, RW */
12.    TOR_REF         = 0x8,   /**< Torque reference (Iq), RW */
13.    TOR_KP          = 0x9,   /**< Torque KP, RW */
14.    TOR_KI          = 0xA,   /**< Torque KI, RW */
15.    TOR_KD          = 0xB,   /**< Torque KD, RW */
16.    FLUX_REF        = 0xC,   /**< Flux reference (Id), RW */
17.    FLUX_KP          = 0xD,   /**< Flux KP, RW */
18.    FLUX_KI          = 0xE,   /**< Flux KI, RW */
19.    FLUX_KD          = 0xF,   /**< Flux KD, RW */
20.    OBS_C1          = 0x10,  /**< Observer C1, RW */
21.    OBS_C2          = 0x11,  /**< Observer C2, RW */
22.    C_OBS_C1        = 0x12,  /**< Cordic Observer C1, RW */
23.    C_OBS_C2        = 0x13,  /**< Cordic Observer C2, RW */
24.    PLL_KI          = 0x14,  /**< PLL KI , RW */
25.    PLL_KP          = 0x15,  /**< PLL KP, RW */
26.    FLUX_WEAK_KP    = 0x16,  /**< Flux weakening KP, RW */
27.    FLUX_WEAK_KI    = 0x17,  /**< Flux weakening KI, RW */
28.    FLUX_WEAK_BUS   = 0x18,  /**< Flux weakening BUS Voltage allowed percentage
                                reference, RW */
29.    BUS_VOL          = 0x19,  /**< Bus Voltage, R */
30.    HEATSINK_TEMP   = 0x1A,  /**< Heatsink temperature, R */
31.    MOTOR_POWER     = 0x1B,  /**< Motor power, R */
32.    DAC_OUT1         = 0x1C,  /**< DAC Out 1, R */
33.    DAC_OUT2         = 0x1D,  /**< DAC Out 2, R */
34.    SPEED_MEAS      = 0x1E,  /**< Speed measured, R */
35.    TOR_MEAS         = 0x1F,  /**< Torque measured (Iq), R */

```

```

36.     FLUX_MEAS      = 0x20, /*< Flux measured (Id), R */
37.     FLUX_WBV       = 0x21, /*< Flux weakening BUS Voltage allowed percentage
   measured, R */
38.     RS_NUM         = 0x22, /*< Revup stage numbers, R */
39.     MAX_APP_SPEED = 0x3F, /*< Maximum application speed, R */
40.     MIN_APP_SPEED = 0x40, /*< Minimum application speed, R */
41.     IQ_REF         = 0x41, /*< Iq reference in speed mode, W */
42.     EXP_BEMF_PLL   = 0x42, /*< Expected BEMF level (PLL), R */
43.     OBS_BEMF_PLL   = 0x43, /*< Observed BEMF level (PLL), R */
44.     EXP_BEMF_CORDIC = 0x44, /*< Expected BEMF level (CORDIC), R */
45.     OBS_BEMF_CORDIC = 0x45, /*< Observed BEMF level (CORDIC), R */
46.     FF_1Q          = 0x46, /*< Feedforward (1Q), RW */
47.     FF_1D          = 0x47, /*< Feedforward (1D), RW */
48.     FF_2           = 0x48, /*< Feedforward (2), RW */
49.     FF_VQ          = 0x49, /*< Feedforward (VQ), R */
50.     FF_VD          = 0x4A, /*< Feedforward (VD), R */
51.     FF_VQ_PI_OUT  = 0x4B, /*< Feedforward (VQ PI out), R */
52.     FF_VD_PI_OUT  = 0x4C, /*< Feedforward (VD PI out), R */
53.     RAMP_FIN_SPEED = 0x5B, /*< Ramp final speed, RW */
54.     RAMP_DUR       = 0x5C, /*< Ramp duration, RW */
55. //Commands:
56.     START_MOTOR    = 0x1, /*< Indicates the user request to start the motor
   regardless of the state of the motor. */
57.     STOP_MOTOR     = 0x2, /*< Indicates the user request to stop the motor
   regardless of the state of the motor. */
58.     STOP_RAMP      = 0x3, /*< Indicates the user request to stop the
   execution of the speed ramp that is currently executed */
59.     START_STOP     = 0x6, /*< Indicates the user request to start the motor
   if the motor is still, or to stop the motor if it runs. */
60.     FAULT_ACT      = 0x7, /*< Communicates the user acknowledges of the
   occurred fault conditions. */
61.     ENCODER_ALIGN  = 0x8 /*< Indicates the user request to perform the
   encoder alignment procedure. */
62. };

```

Przedstawiona na listingu 5.12 klasa zawiera dodatkowe pole o nazwie *NO_REG* informujące o tym iż ramka nie zawiera *payload_id* co występuje w przypadku chociażby ramki odpowiedzi.

Payload

Są to bajty odbierane przez RPI bądź wysyłane do kontrolera. Ich ilość definiowana jest przez *payload_length*. Bajty ułożone są w ramce w kolejności od najmniej znaczącego do najbardziej znaczącego. Jest to przykładowo, wartość zadana prędkości silnika która ma zostać wpisana do rejestru kontrolera. Zastosowany typ zmiennej przeznaczonej do przechowywania *payload* to *long long*. Pozwala on na przechowywanie większej ilości bajtów danych niż podstawowy typ *int* co jest kluczowe w przypadku dużych rozmiarów przesyłanych danych.

CRC

CRC czyli cykliczny kod nadmiarowy. Jest to bajt zawierający liczbę wyliczoną na podstawie wszystkich elementów wchodzących w skład ramki startowej. Jest on stosowany w celu kontroli jakości odbieranych danych. CRC wyliczany jest przed przesaniem danych i wysyłany zaraz potem razem z ramką do kontrolera. Następnie na podstawie otrzymanych informacji sterownik sam oblicza CRC i porównuje z tym które otrzymał. Uzyskanie różnych wartości świadczy o tym iż w czasie transmisji doszło do koruptionu przesyłanych danych. Wartość CRC można obliczyć wykorzystu-

jąc równia:

$$\text{suma} = \text{frame_start} + \text{payload_len} + \sum_{i=0}^n \text{payload}[i] \quad (5.1)$$

$$\text{CRC} = \text{HighByte}(\text{suma}) + \text{LowByte}(\text{suma}) \quad (5.2)$$

gdzie $\text{HighByte}(x)$ to najbardziej znaczący bajt przekazanej wartości, $\text{LowByte}(x)$ to najmniej znaczący bajt przekazanej wartości.

Obliczanie CRC odbywa się w konstruktorze struktury *Frame* przeznaczonym dla ramki startowej (listing 5.13).

Listing 5.13: Konstruktor struktury *Frame* przeznaczony dla ramki startowej

```

01. Frame(int motor_id, FRAME_CODES frame_id, STEVAL_REGISTERS payload_id,
02.         STEVAL_REGISTERS_LEN payload_len, long long payload)
03. {
04.     this->motor_id = motor_id;
05.     this->frame_id = frame_id;
06.     this->payload_id = payload_id;
07.     this->payload_len = (int)payload_len;
08.     this->payload = payload;
09.     //OBLICZANIE CRC
10.     //Czy ramka zawiera frame_id
11.     if(payload_id == STEVAL_REGISTERS::NO_REG)
12.         CRC=((motor_id << 5) + (int)frame_id) + this->payload_len;
13.     else
14.         CRC = ((motor_id << 5) + (int)frame_id) + (int)payload_id + this->
15.             payload_len;
16.     //Sprawdzenie czy ramka zawiera w sobie jakieś dane (payload)
17.     if (payload != NO_DATA){
18.         //Obliczenie CRC z danych zawartych w ramce
19.         for (int i = 0; i < this->payload_len; i++)
20.         {
21.             CRC += int(payload >> i * 8) & 0xff;
22.         }
23.         //Najbardziej znaczący bajt i najmniej znaczący
24.         CRC = ((CRC >> 8) & 0xff) + (CRC & 0xff);
    }
```

W przedstawionym na listingu 5.13 konstruktorze można zauważyć warunek sprawdzający czy ramka zawiera jakieś dane (*payload*). Warunek ten opiera się na porównaniu *payload* z wartością *NO_DATA*. *NO_DATA* to największa wartość jaką może osiągnąć zmienna typu *long long* (listing 5.14).

Listing 5.14: Definicja makra *NO_DATA*

```
01. #define NO_DATA LLONG_MAX
```

Konstrukcja ramki odpowiedzi

Ramka odpowiedzi nieznacznie różni się od ramki startowej. Nie posiada ona *payload_id* oraz jej *frame_start* nie składa się z identyfikatora silnika oraz *frame_id* jak to było w przypadku ramki startowej. *Frame_start* zawiera jedynie informacje o statusie operacji zleconej przez mastera za pomocą ramki startowej. *Frame_start* może przyjąć tylko dwie wartości: 0xF0 oraz 0xFF. W przypadku powodzenia przyjmowana jest wartość 0xF0, natomiast 0xFF informuje o tym, iż po stronie kontrolera wystąpił błąd a wysłana przez niego ramka zawiera kod owego błędu. Możliwe kody błędów przechowywane są w typie wyliczeniowym *STEVAL_ERROR* (listing 5.15).

Listing 5.15: Typ wyliczeniowy reprezentujący możliwe kody błędów otrzymywanych w ramce odpowiedzi

```

01. typedef enum
02. {
03.     BAD_FRAME_ID =      0x1, /*< BAD Frame ID. The Frame ID has not been
04.                         recognized by the firmware */
05.     WRITE_ON_RO =       0x2, /*< Write on read-only. The master wants to write
06.                         on a read-only register */
07.     READ_NOT_ALLOWED = 0x3, /*< Read not allowed. The value cannot be read.
08.                           */
09.     BAD_TARGET =        0x4, /*< Bad target drive. The target motor is not
10.                           supported by the firmware */
11.     OUT_OF_RANGE =      0x5, /*< Out of range. The value used in the frame is
12.                           outside the range expected by the firmware. */
13.     BAD_CMD_ID =        0x7, /*< Bad command ID. The command ID has not been
14.                           recognized */
15.     OVERRUN_ERROR =     0x8, /*< Overrun error. The frame has not been
16.                           received correctly because the transmission speed is too fast. */
17.     TIMEOUT =           0x9, /*< Timeout error. The frame has not been
18.                           received correctly and a timeout
19.                           occurs. This kind of error usually occurs
20.                           when the frame is not correct or is
21.                           not correctly recognized by the firmware. */
22.     BAD_CRC =           0xA, /*< Bad CRC. The computed CRC is not equal to the
23.                           received CRC byte. */
24.     BAD_TARGET_2 =       0xB /*< Bad target drive. The target motor is not
25.                           supported by the firmware */
26. } STEVAL_ERROR;

```

Dodatkowo w typie wyliczeniowym przedstawionym na listingu 5.15 zaimplementowana została funkcja zwracająca komunikat błędu na podstawie *STEVAL_ERROR* (listing 5.16).

Listing 5.16: Definicja funkcji zwracającej komunikat na podstawie *STEVAL_ERROR*

```

01. char const* GetErrorMessage(STEVAL_ERROR errorCode)
02. {
03.     switch(errorCode)
04.     {
05.         case BAD_FRAME_ID:
06.             return "BAD_Frame_ID. The Frame ID has not been recognized by the
07.                 firmware.";
08.         case WRITE_ON_RO:
09.             return "Write_on_read-only. The master wants to write on a read-
10.                 only register.";
11.         case READ_NOT_ALLOWED:
12.             return "Read_not_allowed. The value cannot be read.";
13.         case BAD_TARGET:
14.             return "Bad_target_drive. The target motor is not supported by the
15.                 firmware.";
16.         case OUT_OF_RANGE:
17.             return "Out_of_range. The value used in the frame is outside the
18.                 range expected by the firmware.";
19.         case BAD_CMD_ID:
20.             return "Bad_command_ID. The command ID has not been recognized";
21.         case OVERRUN_ERROR:
22.             return "Overrun_error. The frame has not been received correctly
23.                 because the transmission speed is too fast.";
24.         case TIMEOUT:
25.             return "Timeout_error. The frame has not been received correctly
26.                 and a timeout occurs. This kind of error usually occurs when the
27.                     frame is not correct or is not correctly recognized by the
28.                         firmware.";
29.         case BAD_CRC:
30.             return "Bad_CRC. The computed CRC is not equal to the received CRC
31.                 byte.";
32.         case BAD_TARGET_2:
33.             return "Bad_target_drive. The target motor is not supported by the
34.                 firmware.";
35.     default:

```

```

26.         return "Unknown error.";
27.     }
28. }
```

Do utworzenia ramki odpowiedzi wykorzystywany jest specjalny konstruktor struktury *Frame* przedstawiony na listingu 5.17.

Listing 5.17: Konstruktor struktury *Frame* przeznaczona dla ramki odpowiedzi.

```

01. Frame(int *array, int length)
02. {
03.     if (length < 3)
04.     {
05.         long log = start_log(LOG_PATH);
06.         printf("[ERROR_UART_DATA] %s\n", "Error in Frame constructor.\n"
07.                "Frame length too short.");
08.         stop_log(log);
09.         throw;
10.     }
11.     this->payload = 0;
12.     this->payload_len = array[1];
13.     this->CRC = array[length - 1];
14.     //Czy ramka zawiera jakieś dane i czy nie jest ramk b du
15.     if (this->payload_len > 0 && array[0] != 0xff)
16.     {
17.         int sig = 0;
18.         int nonzero = 0;
19.         bool set = false;
20.         for (int i = 2 + this->payload_len - 1; i >= 2; i--)
21.         {
22.             this->payload = ((int)this->payload << (8 * (this->payload_len
23.                     - 1 - i))) + array[i];
24.             //Pobranie pierwszego bitu znac cego pozwalaj cego na
25.             //stwierdzenie o znaku obliczanej liczby
26.             if (array[i] != 0 && set == false)
27.             {
28.                 sig = array[i] >> 7;
29.                 set = true;
30.                 nonzero = i - 1;
31.             }
32.             if (sig)
33.                 this->payload = -((~this->payload) & (0xffff)) - 1;
34.             this->frame_id = FRAME_CODES::RECEIVE;
35.             this->motor_id = -1;
36.         }
37.         //Sprawdzenie czy odebrana bramka zawiera kod b edu
38.         else if (array[0] == 0xff)
39.         {
40.             this->payload = array[2];
41.             this->frame_id = FRAME_CODES::ERROR;
42.             long log = start_log(LOG_PATH);
43.             printf("[ERROR_UART_DATA] %s\n", GetErrorMessage((STEVAL_ERROR)
44.                     array[2]));
45.             stop_log(log);
46.             this->motor_id = -1;
47.             //throw;
48.         }
49.         //Ramka nie zaczyna si ani od kodu sukcesu(0x0f) ani od kudu b edu
50.         (0xff)
51.         else if (array[0] != 0xf0)
52.         {
53.             long log = start_log(LOG_PATH);
54.             printf("[ERROR_UART_DATA] %s FRAME:", "Error in Frame constructor.\n"
55.                   "Wrong frame start");
56.             for (int i = 0; i < length; i++)
57.             {
58.                 printf("%02x ", array[i]);
59.             }
60.             printf("\n");
61.             stop_log(log);
62.             throw;
```

58. }
59. }

Argumenty jakie przyjmujeowy konstruktor to wskaźnik do tablicy danych zawierającej odebrane przez port szeregowy bajty oraz ilość tych bajtów.

Pierwszą operacją w konstruktorze jest sprawdzenie ilości otrzymanych danych. W przypadku wartości mniejszej od trzech, czyli najmniejszej możliwej poprawnej ramki odpowiedzi, zgłoszany jest wyjątek oraz stosowny komunikat zostaje wpisany do pliku z logami(listing 5.17, wiersze 3-9).

Następnie przypisywane są wartości do odpowiednich pól struktury. Początkowo wartość *payload* zostaje zainicjowana zerem, a *payload_len* oraz CRC zostały bezpośrednio pobrane z tablicy otrzymanych bajtów(listing 5.17, wiersze 10-13).

Kolejnym etapem jest sprawdzenie czy odebrana ramka zawiera jakieś dane oraz czy jej *frame_start* posiada wartość określającą pomyślnie wykonane zadanie. W przypadku gdy warunek jest prawdziwy przypisywane są wartości do odpowiednich pól struktury (listing 5.17, wiersze 14-34).

W przypadku gdy *payload_len* ramki jest większy od 0, co równoznaczne jest z tym iż ramka zawiera *payload*, oraz gdy *frame_start* ma wartość określającą brak błędów (0xF0) następuje obliczenie otrzymanych danych w ramce. Aby je obliczyć do zainicjowanego zerem pola *payload* struktury *Frame* zostają dodawane wartości z tablicy otrzymanych danych w kolejności odwrotnej. Dodawanie poszczególnych wartości poprzedza operacja przesunięcia bitowego o wartość wielokrotności liczby 8 zależnej od indeksu aktualnie dodawanej wartości tablicy (listing 5.17, wiersz 21).

Pomijane są wartości reprezentujące *frame_start*, CRC, oraz *payload_length* tzn. wartości o indeksach 0, 1 oraz o indeksie ostatnim. Dodatkowo w pętli została zaimplementowany warunek, który przy pojawienniu się pierwszego niezerowego bajtu, określa znak otrzymanej liczby, poprzez sprawdzenie wartości pierwszego bitu (listing 5.17, wiersze 23-28).

Wartość jeden bitu przechowywanego zmiennej *sig* oznacza iż otrzymane dane reprezentują wartość ujemną. Ich przeliczenie zostało zaimplementowane w wierszu 31 listingu 5.17.

W przypadku gdy odebrana ramka zawiera informacje o błędzie po stronie kontrolera, wartość *frame_id* ustawniona zostaje jako *FRAME_CODES :: ERROR*, a kod błędu pobierany jest z 2 indeksu tablicy danych. Następnie otrzymany komunikat błędu zostaje zapisany do pliku z logami, a w konstruktorze zgłoszany jest wyjątek.

W przypadku gdy otrzymana ramka nie zaczyna się od 0xF0 bądź od 0xFF(co jednoznaczne jest z błędem w transmisji) zgłoszany zostaje wyjątek oraz błędna ramka zostaje zapisana w pliku z logami.

Przykładowa ramka startowa i ramka odpowiedzi

Rozważając scenariusz, w którym należy pobrać rejestr sterownika odpowiadający finalnej prędkości rampy, ramka startowa wyglądałaby w następujący sposób:

- Frame_start: Wybrany zostaje silnik nr 1 co oznacza że pierwsze 3 bity *frame_start* to 001. Następnie wybierany jest odpowiedni *frame_id* gdzie w przypadku

operacji Get ma on wartość 2. *Frame_start* w całości wygląda następująco :

- binarnie: 00100010,
- heksadecymalnie: 22.

- *Payload_length*: w przypadku operacji Get *payload_length* przyjmuje zawsze wartość 1, a więc *payload_length* wygląda następująco:
 - binarnie: 00000001,
 - heksadecymalnie: 01.
- *Payload_id*: jest to wartość reprezentująca rejestr kontrolera, którego wartość ma zostać odczytana, w przypadku finalnej prędkości rampy znajduje się on w rejestrze o identyfikatorze równym 0x5B, a więc *payload_id* to:
 - binarnie: 1011011,
 - heksadecymalnie: 5B.
- *Payload*: w przypadku operacji Get *payload* nie występuje, dlatego nie jest brany pod uwagę w konstrukcji ramki.
- *CRC*: obliczenie wartości *CRC* przy użyciu równań 5.1 i 5.2 wygląda w następujący sposób:

$$\begin{aligned} CRC = & \text{HighByte}(frame_code + payload_length + payload_id) + \\ & \text{LowByte}(frame_code + payload_length + payload_id) \end{aligned} \quad (5.3)$$

$$CRC = \text{HighByte}(34 + 1 + 91) + \text{LowByte}(34 + 1 + 91) \quad (5.4)$$

$$CRC = \text{HighByte}(126) + \text{LowByte}(126) \quad (5.5)$$

$$CRC = 126 \quad (5.6)$$

Zatem otrzymane *CRC* wynosi:

- binarnie: 1111110,
- hexadecymalnie: 7E.

Przy pomocy tych obliczeń uzyskana ramka startowa w zapisie heksadecymalnym została przedstawiona w tabeli 5.4.

Odpowiedź kontrolera na zapytanie przedstawione w tabeli 5.4 w przypadku wartości 2000 i odczytu rejestru zakończonego sukcesem przedstawiono w tabeli 5.5.

W przypadku gdy odczytywany rejestr dysponowałby jedynie możliwością zapisu, zwrócona ramka błędu zawierałaby kod błędu 0x03 i wyglądałaby w sposób przedstawiony w tabeli 5.6.

Tabela 5.4: Przykładowa ramka startowa w zapisie heksadecymalnym

<i>Frame_start</i>	<i>Payload_length</i>	<i>Payload_id</i>	<i>Payload</i>	<i>CRC</i>
22	01	5B	brak	7E

Tabela 5.5: Przykładowa ramka odpowiedzi w zapisie heksadecymalnym

<i>Frame_start</i>	<i>Payload_length</i>	<i>Payload_id</i>	<i>Payload</i>	<i>CRC</i>
F0	05	brak	D0 07 00 00 00	CD

Tabela 5.6: Przykładowa ramka odpowiedzi zawierająca kod błędu

<i>Frame_start</i>	<i>Payload_length</i>	<i>Payload_id</i>	<i>Payload</i>	<i>CRC</i>
FF	01	brak	03	04

Wysyłanie i odbieranie ramek

Do reprezentowania urządzenia UART służącego do obsługi portu szeregowego stworzono strukturę *UART*.

Listing 5.18: Definicja struktury *UART*

```

01. struct UART
02. {
03.     int baud;           /*< Baud rate*/
04.     const char *device; /*< Device name*/
05.     int fd;             /*< File descriptor*/
06. };

```

Przedstawiona na listingu 5.18 struktura zawiera 3 pola:

- liczba całkowita *baud* odpowiadająca prędkości transmisji szeregowej,
- ciąg znaków *device* określający wybrany port szeregowy,
- liczba całkowita *fd* będąca deskryptorem pliku otwartego połączenia.

Wysyłanie i odbieranie ramek danych odbywa się za pomocą specjalnie przygotowanych funkcji, których deklaracje przedstawiono w lisnigu 5.19. Natomiast do obsługi transmisji szeregowej wykorzystywane są funkcje z biblioteki *wiringSerial* wchodzącej w skład biblioteki *wiringPi*[30].

Listing 5.19: Deklaracje funkcji przeznaczonych do wysyłania i odbierania ramek MCP

```

01. UART_STATUS receive(Frame *cmd, int connection, Frame c);
02. UART_STATUS sendData(int con, long long data, int l);
03. UART_STATUS send(Frame cmd, UART uart, Frame *f);

```

Funkcja *send()* przyjmuje następujące argumenty:

- obiekt *Frame* reprezentujący ramkę wysyłaną,
- obiekt *UART*,
- wskaźnik do obiektu *Frame*, do którego zapisywana będzie ramka odpowiedzi.

Listing 5.20: Definicja funkcji *send* przeznaczona do wysyłania ramki startowej

```

01. UART_STATUS send(Frame cmd, UART uart, Frame *f)
02. {
03.     Frame recCommand = Frame();
04.     int connection = uart.fd;

```

```

05.     serialFlush(connection);
06.     serialPutchar(connection, (cmd.motor_id << 5) + (int)cmd.frame_id);
07.     serialPutchar(connection, (int)cmd.payload_len);
08.     if (cmd.payload_id != STEVAL_REGISTERS::NO_REG)
09.     {
10.         serialPutchar(connection, (int)cmd.payload_id);
11.     }
12.
13.     if ((int)cmd.payload != NO_DATA)
14.     {
15.         int l=cmd.payload_len;
16.         if(cmd.payload_id!=STEVAL_REGISTERS::NO_REG)
17.             l-=1;
18.         sendData(connection, cmd.payload, 1);
19.     }
20.     serialPutchar(connection, cmd.CRC);
21.
22.     UART_STATUS status = receive(&recCommand, connection, cmd);
23.     *f = recCommand;
24.     return status;
25. }
```

Procedura wysłania ramki rozpoczyna się od wywołania funkcji pochodzącej z biblioteki wiringSerial o nazwie *serialFlush()* (listing 5.20 wiersz 5).

Funkcja ta pozwala na zignorowanie wysyłanych bądź oczekujących na odebranie danych w celu skupienia się na aktualnie wysyłanej ramce. Następnie za pomocą funkcji *serialPutchar()* pochodzącej z tej samej biblioteki wysyłane są kolejno elementy ramki (listing 5.20).

Dodatkowo odpowiednie bloki warunkowe sprawdzają czy ramka zawiera *payload_id* oraz *payload*. W przypadku spełnienia tego warunku elementy również zostają wysłane. Procedurę wysyłania *payload* obsługuje funkcja *sendData()* (listing 5.21) przyjmująca następujące argumenty:

- liczbę całkowitą reprezentującą deskryptor pliku otwartego połączenie,
- liczbę całkowitą typu long long zawierającą dane do przesłania,
- *payload_len* będący ilością bitów jakie należy wysłać w ramce.

Listing 5.21: Definicja funkcji *sendData()* służąca do przesyłania *payload* ramki startowej

```

01. UART_STATUS sendData(int con, long long data, int l)
02. {
03.     for (int i = 0; i < l; i++)
04.     {
05.         serialPutchar(con, int((data >> i * 8) & 0xFF));
06.     }
07.     return UART_OK;
08. }
```

Po zakończeniu wysyłania wszystkich elementów ramki przez port szeregowy, następuje wywołanie funkcji *receive()* (listing 5.20, wiersz 22).

Funkcja *receive()* przedstawiona na lisningu 5.22 jako argumenty przyjmuje:

- Wskaźnik do obiektu *Frame* do którego zapisana zostanie odebrana ramka,
- deskryptor pliku połączenie UART.

Listing 5.22: Definicja funkcji *receive()* służącej do odbierania ramki odpowiedzi

```

01. UART_STATUS receive(Frame *cmd, int connection)
02. {
03.     int j = 0;
04.     int *array = new int[10];
05.     delay(5);
06.     while (serialDataAvail(connection))
07.     {
08.         array[j] = serialGetchar(connection);
09.         j++;
10.     }
11.     if (j < 3 || j > 10)
12.     {
13.         long log = start_log(LOG_PATH);
14.         printf("[ERROR_UART_DATA] Wrong frame lenght(%d).\n", j);
15.         stop_log(log);
16.         return UART_FRAME_SIZE_ERROR;
17.     }
18.     try
19.     {
20.         *cmd = Frame(array, j);
21.     }
22.     catch (...)
23.     {
24.         return UART_FRAME_ERROR;
25.     }
26.     return UART_OK;
27. }
```

Procedura pobierania ramki odpowiedzi rozpoczyna się pięciomilisekundowym opóźnieniem (listing 5.22, wiersz 5), które ma na celu danie kontrolerowi czasu na przetworzenie żądania i odesłanie wymaganych danych. Wartość 5 milisekund została wyznaczona empirycznie przy wykorzystaniu mechanizmu logowania błędów. Metoda ta polegała na zmianie opóźnienia i obserwowania czy w logu wyświetlają się błędy o nie odpowiedniej strukturze odebranej ramki a następnie wybraniu najmniejszej wartości w której nie występują błędy.

Po opóźnieniu następuje odebranie danych przy użyciu funkcji biblioteki wiringSerial o nazwie *serialGetChar()*, która pobiera jeden bajt danych z ramki transmisyjnej UART (listing 5.22, wiersz 8). Pobieranie poszczególnych bajtów ramki odpowiedzi odbywa się w pętli *while*, której warunkiem zakończenia jest brak danych do odebrania, określanych przez funkcję *serialDataAvail()* również pochodzącej z biblioteki wiringSerial (listing 5.22, wiersz 6). Odebrane bajty danych przechowywane są w tablicy o rozmiarze 10, czyli takim, który jest w stanie pomieścić największą możliwą ramkę odpowiedzi. Ilość odebranych bajtów zapisywana jest do zmiennej *j*, której wartość zwiększana jest z każdym przejściem pętli. Po odebraniu wszystkich dostępnych danych zostają one przekazane do konstruktora przeznaczonego dla ramek odpowiedzi. Konstruktor ten znajduje się w bloku *try* w celu obsługi wyjątków (listing 5.22, wiersz 20).

API biblioteki

Biblioteka do obsługi protokołu MCP została wyposażona w gotowe funkcje pozwalające na łatwą obsługę podstawowych i najczęściej używanych operacji.

Listing 5.23: Deklaracje funkcji biblioteki do obsługi MCP pozwalającej na obsługę podstawowych i najczęściej występujących operacji

```

01. /**
02.  * @brief Execute start motor command
03. *
```

```

04. * @param motorId Motor id
05. * @param uart Uart object
06. * @param f Received acknowledgment frame
07. * @return UART_STATUS Operation status
08. */
09. UART_STATUS StartMotor(int motorId, UART uart, Frame *f);
10. /**
11. * @brief Execute stop speed ramp
12. *
13. * @param motorId Motor id
14. * @param uart Uart object
15. * @param f Frame received
16. * @return UART_STATUS Uart sending status
17. */
18. UART_STATUS StopRamp(int motorId, UART uart, Frame *f);
19. /**
20. * @brief Execute stop motor command
21. *
22. * @param motorId Motor id
23. * @param uart Uart object
24. * @param f Frame received
25. * @return UART_STATUS Uart sending status
26. */
27. UART_STATUS StopMotor(int motorId, UART uart, Frame *f);
28. /**
29. *
30. * @brief Set the motor ramp final speed registry
31. *
32. * @param ref Value to set
33. * @param motorId Motor id
34. * @param uart Uart object
35. * @param f Frame received
36. * @return UART_STATUS Uart sending status
37. */
38. UART_STATUS SetMotorRefSpeed(int ref, int motorId, UART uart, Frame *f);
39. /**
40. * @brief Set the registry value
41. *
42. * @param reg Motor control register id
43. * @param regL Montrol register payload length
44. * @param motorId Motor id
45. * @param uart Uart object
46. * @param data Value to set
47. * @param f Frame received
48. * @return UART_STATUS Uart sending status
49. */
50. UART_STATUS SetRegistry(STEVAL_REGISTERS reg, STEVAL_REGISTERS_LEN regL, int
51.     motorId, UART uart, int data, Frame *f);
52. /**
53. * @brief Get the registry value
54. *
55. * @param reg Motor control register id
56. * @param regL Montrol register payload length
57. * @param motorId Motor id
58. * @param uart Uart object
59. * @return UART_STATUS Uart sending status
60. */
61. UART_STATUS GetRegistry(STEVAL_REGISTERS reg, STEVAL_REGISTERS_LEN regL, int
62.     motorId, UART uart, Frame *f);
63. /**
64. * @brief Reset error on Motor Controller
65. *
66. * @param motorId Motor id
67. * @param uart Uart object
68. * @param f Received Frame
69. * @return UART_STATUS Uart sending status
70. */
71. UART_STATUS FaultAck(int motorId, UART uart, Frame *f);
72. /**
73. * @brief Execute rampe speed
74. *
75. * @param motorId Motor Id
76. * @param uart Uart object

```

```

75. * @param duration Ramp speed duration
76. * @param finalSpeed Ramp speed final speed
77. * @param f Received frame
78. * @return UART_STATUS Uart sending status
79. */
80. UART_STATUS ExeRampSpeed(int motorId, UART uart, int duration, int finalSpeed,
                           Frame *f);

```

5.1.2 Serwer UDP

Głównym elementem scalającym wszystkie wymagane funkcje jakie ma spełniać oprogramowanie, jest program stanowiący serwer oparty o protokół sieciowy UDP. Podstawową funkcja tego programu jest wysyłanie odebranych danych z czujnika, kamery oraz sterownika przy użyciu tegoż protokołu, a także nasłuchiwanie w celu odebrania danych wysyłanych przez program Matlab, również za pomocą protokołu UDP.

Funkcja startowa *main* przed uruchomieniem serwera rozpoczęta komunikację szeregową ze sterownikiem (listing 5.24, wiersz 5), a także inicjalizowany jest moduł komunikacji I2C z czujnikiem położenia wahadła (listing 5.24, wiersz 14).

Listing 5.24: Funkcja *main* głównego programu RPI

```

01. int main()
02. {
03.     uart.baud = UART_BAUD;
04.     uart.device = UART_DEVICE;
05.     uart.fd= serialOpen(uart.device, uart.baud);
06.     if (uart.fd == -1)
07.     {
08.         long fd12 = start_log(LOG_PATH);
09.         printf("[ERROR_UART] %s", "Uart connection error.\n");
10.         stop_log(fd12);
11.         return UART_CONNECTION_ERROR;
12.     }
13.     int as5600;
14.     AS5600_Init(&as5600);
15.     SERWER_IP = getLocalIp();
16.     Init_ScrewSpin(uart);
17.     server(uart);
18.     return 0;
19. }

```

Dodatkowo przed startem serwera pobierany jest lokalny adres IP mikrokomputera przy pomocy funkcji *getLocalIp()* (listing 5.24, wiersz 15). Rozwiązań to pozwala nieprzejmowanie się ręcznym wpisywaniem adresu dla każdego stanowiska, co czyniowy program uniwersalnym.

Listing 5.25: Definicja funkcji *getLocalIp()* pozwalająca na pobranie lokalnego ip urządzenia

```

01. const char *getLocalIp()
02. {
03.     int fd;
04.     struct ifreq ifr;
05.
06.     fd = socket(AF_INET, SOCK_DGRAM, 0);
07.     //Get IPv4 IP address
08.     ifr.ifr_addr.sa_family = AF_INET;
09.     //Get "eth0"
10.     strncpy(ifr.ifr_name, "wlan0", IFNAMSIZ - 1);
11.     ioctl(fd, SIOCGIFADDR, &ifr);
12.     close(fd);
13.     return inet_ntoa(((struct sockaddr_in *)&ifr.ifr_addr)->sin_addr);
14. }

```

W zależności od rodzaju wykorzystywanego połączenia sieciowego należy skorzystać z identyfikatora “wlan0” lub “eth0”.

Ostatnim etapem przed uruchomieniem serwera jest wywołanie prostej funkcji *Init_ScrewSpin()* mającej za zadanie wystartowanie silnika oraz zadanie rampy o nieznacznej prędkości końcowej (listing 5.24, wiersz 16).

Listing 5.26: Funkcja startująca silnik oraz uruchamiająca rampę o niewielkiej prędkości końcowej

```
01. void Init_ScrewSpin(UART u)
02. {
03.     Frame f, f2;
04.     ExeRampSpeed(1,u,5000,1000,&f2);
05.     StartMotor(1, u, &f);
06. }
```

Po przeprowadzeniu wyżej wymienionych procedurach następuje uruchomienie serwera. Całość programu operuje na dwóch pętlach działających współbieżnie, dzięki zastosowaniu mechanizmu wielu wątków [11]. Jeden wątek ma za zadanie obsługę wysyłania danych, natomiast drugi zajmuje się nasłuchiwaniem i odbieraniem wartości zadanych z programu Matlab(listing 5.27).

Listing 5.27: Deklaracja wątków wysyłania i odbierania danych za pomocą protokołu UDP

```
01. std::thread sending_thread(send_data,socket_,len,u);
02. std::thread receiving_thread(recv_data,socket_,len,u);
03.
04. sending_thread.join();
05. receiving_thread.join();
```

Z uwagi na to iż oba programy operują na komunikacji z kontrolerem przy użyciu jednego portu szeregowego konieczne było zastosowanie mechanizmu blokowania dostępu do wspólnego zasobu. W tym celu wykorzystana została klasa mutex wchodząca w skład biblioteki standardowej języka C++. Zasada działania tego rozwiązania wygląda następująco:

- Wywołaniu funkcji *lock()* przez dany wątek czyni go tym samym właścicielem danego zasobu. W przypadku gdy dany zasób będzie znajdował się we władaniu innego wątku wątek będzie oczekiwał do czasu zwolnienia zasobu.
- Przy użyciu funkcji *unlock()* wątek będący właścicielem danego zasoby zwalnia do niego dostęp tym samym informując iż zakończył on wykonywane na nim działania [10].

Blokowanie zasobu zostało użyte we fragmentach kodu, w których następuje komunikacja za pomocą portu szeregowego sterownika. Przykład użycia w programie zaprezentowano w listingu 5.28

Listing 5.28: Przykładowe użycie klasy mutex

```
01. uart_mtx.lock();
02. ExeRampSpeed(1,u,(int)(abs((int)re-old)),(int)re,&f);
03. uart_mtx.unlock();
```

Główna pętla podprogramu zajmująca się wysyłką danych pomiarowych, tuż przed ich pobraniem z odpowiednich elementów stanowiska, sprawdza czy w czasie pracy sterownika nie wystąpiły po jego stronie żadne błędy. Do tego zadania napisana została funkcja o nazwie *checkForErrors()*(listing 5.29).

Listing 5.29: Funkcja *checkForErrors()* sprawdzająca błędy po stronie sterownika i wpisująca komunikaty błędów do pliku z logami

```

01. void checkForErrors(UART u)
02. {
03.     Frame f, f2, f3;
04.     GetRegistry(STEVAL_REGISTERS::FLAGS, STEVAL_REGISTERS_LEN::GET, 1, u, &f);
05.     long fd;
06.     switch ((int)f.payload)
07.     {
08.         case 2:
09.             fd = start_log(LOG_PATH);
10.             printf("[ERROR] Over voltage\n");
11.             stop_log(fd);
12.             return;
13.         case 32:
14.             fd = start_log(LOG_PATH);
15.             printf("[ERROR] SpeedFeedback\n");
16.             stop_log(fd);
17.             FaultAck(1, u, &f2);
18.             StartMotor(1, u, &f3);
19.             return;
20.         default:
21.             return;
22.     }
23. }
```

Funkcja ta ma za zadanie sprawdzenie odpowiedniego rejestru kontrolera który zawiera informacje o błędzie a następnie owe informacje zostają zapisane do pliku z logami. Jednym z częstych pojawiających się problemów jest błąd o nazwie Speed Feedback. Błąd ten wynika ze słabej dokładności czujników Halla przy niskich obrotach silnika. Gdy pomiar prędkości zostaje uznany przez program sterownika za nieadekwatny zgłoszony jest poważny błąd oraz silnik zostaje zatrzymany. Po zgłoszeniu jakiegokolwiek poważnego błędu interakcja z kontrolerem zostaje zawieszona do momentu uruchomienia funkcji *FaultAck()*, która informuje sterownik o tym, iż użytkownik świadomie jest zaistniałego problemu.

Wydawać by się mogło iż problem ten mógłby znacznie wpływać na możliwości regulacji wahadła. Jednakże zastosowane rozwiązanie zaimplementowane w funkcji *checkForErrors()* pozwoliło na częściowe wyeliminowanie skutków występowania tegoż błędu. Sprawdzanie wystąpienia błędu przy każdym iteracji pętli pozwala szybko zareagować na zaistniałą sytuację. Funkcja w przypadku odebrania błędu Speed Feedback automatycznie wywołuje *FaultAck()* przywracając tym samym możliwość komunikacji ze sterownikiem co pozwala następnie na wywołanie funkcji uruchamiającej silnik.

Rozwiązanie to jest jak najbardziej optymalne. Przestoje w działaniach silnika są bardzo krótkie, a częstotliwość występowania błędu jest na tyle niska, iż rozwiązanie to jest w pełni wystarczające.

Główna pętla podprogramu uruchamianego w osobnym wątku zajmująca się wysyłką danych za pomocą protokołu UDP przedstawiona została w listingu 5.30.

Listing 5.30: Główna pętla wątku zajmująca się wysyłką danych za pomocą protokołu UDP

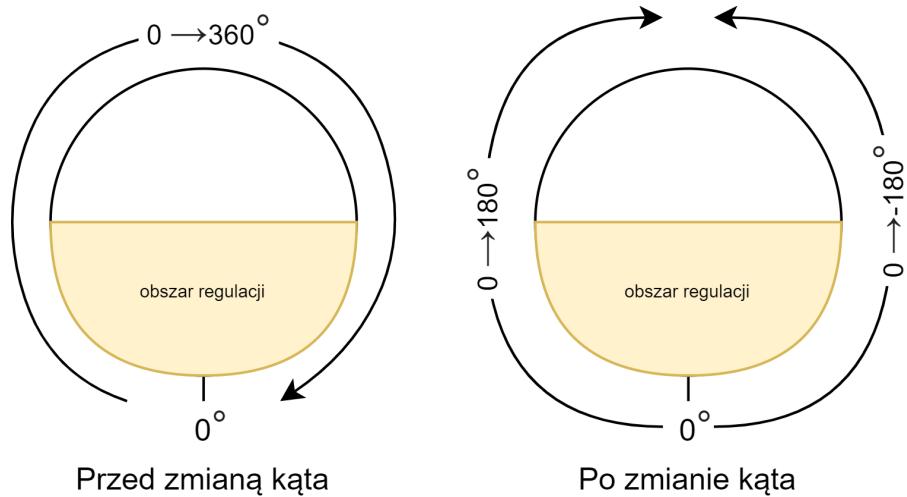
```

01. while (1)
02. {
03.     Frame f;
04.     uart_mtx.lock();
05.     checkForErrors(u);
06.     GetRegistry(STEVAL_REGISTERS::SPEED_MEAS, STEVAL_REGISTERS_LEN::GET, 1,
07.                 u, &f);
08.     uart_mtx.unlock();
09.     float d = convertRawAngleToDegrees(getRawAngle());
```

```

09.
10.     int dd = (int)d;
11.     if(dd>180)
12.         dd= dd-360;
13.
14.     int Amount [] = {(int)f.payload, dd};
15.
16.     std::unique_lock<std::mutex> ul(mtx);
17.     cv_client.wait(ul, [] { return clientIsSet; });
18.     client.sin_port = htons( C_PORT );
19.     if (sendto(socket_, &Amount, sizeof(Amount), 0, (struct sockaddr *)&
19.             client, len) < 0)
20.     {
21.         long l = start_log(LOG_PATH);
22.         printf("[SERVER] [ERROR] sendto() ERROR");
23.         printf("%s\n", strerror(errno));
24.         stop_log(l);
25.     }
26. }
```

Dane przeznaczone do wysyłki pobierane są z czujnika wychylenia wahadła (listing 5.30, wiersz 8) oraz ze sterownika silnika (listing 5.30, wiersz 6). W czasie gdy wątek ten komunikuje się przy użyciu portu szeregowego zasób ten zostaje забlokowany dla innych wątków przy użyciu klasy *mutex* (listing 5.30, wiersz 4). Dane pobrane z czujnika wychylenia przeliczane są na stopnie, a następnie przeprowadzane jest odpowiednie przekształcenie tych danych w celu przesunięcia punktu zerowego pomiaru (listing 5.30, wiersze 11-12). Schemat przesunięcia kąta zaprezentowano na rysunku 5.4.



Rysunek 5.4: Schemat obrazujący przesunięcie kąta po odebraniu z czujnika wychylenia [opracowanie własne]

Zabieg ten ma na celu pozbycie się drastycznej zmiany wartości wychylenia z 0° na 360° w obszarze regulacji poprzez przesunięcie go poza ten obszar.

W celu wysłania danych za pomocą protokołu UDP niezbędny jest adres IP klienta który ma być odbiorcą wysyłanych pakietów danych. Aby uczynić program uniwersalny bez konieczności wpisywania adresu IP komputera łączącego się ze stacją opracowano odpowiednie rozwiązanie tegoż problemu. Polega ono na zapisaniu danych klienta, który wysłał dane do serwera. W podejściu tym działanie wątku obsługującego wysyłanie zostaje zawieszone do momentu, aż wątek odbiera-

jący dane nawiąże połączenie z klientem. Mechanizm ten zrealizowany został przy użyciu zmiennej warunkowej *cv_client* (listing 5.30, wiersz 17) [12].

Dane wysyłane są przez RPI w postaci dwu elementowej tablicy zawierającej wartość prędkości silnika oraz kąt wychylenia wahadła (listing 5.30, wiersz 14). W przypadku wystąpienia błędu, informacja o nim zostaje zapisana w pliku z logami.

Wątek obsługujący odbieranie danych ma za zadanie na podstawie uzyskanej prędkości z programu Matlab uruchomić rampę prędkości. Czas trwania rampy ustalany jest na podstawie różnicy prędkości aktualnie zadawanej z prędkością poprzednia (listing 5.31, wiersz 14). Pętla główna podprogramu zajmującego się odbiorem danych została przedstawiona na listingu 5.31.

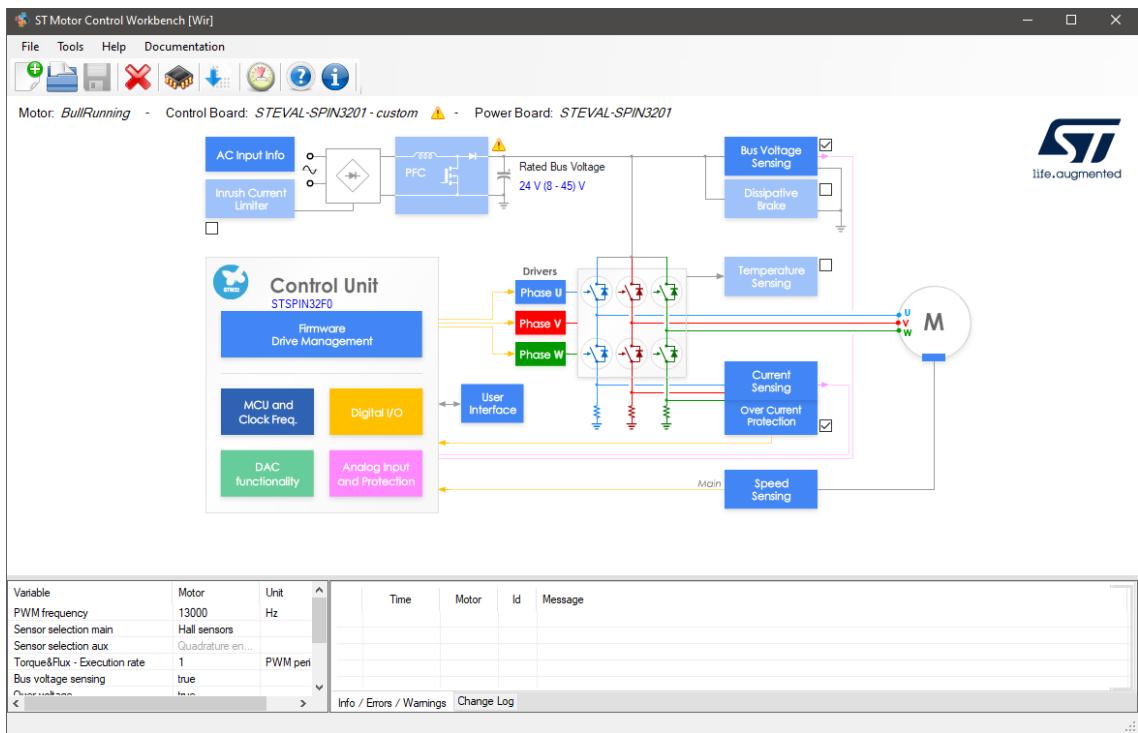
Listing 5.31: Główna pętla wątku zajmująca się odbiorem danych za pomocą protokołu UDP

```

01. while (1)
02. {
03.     double re;
04.     Frame f,f2;
05.     if (recvfrom(socket_, &re, sizeof(double), 0, (struct sockaddr *)&
06.             client, &len) < 0)
07.     {
08.         long l = start_log(LOG_PATH);
09.         printf("[SERVER] ERROR] recvfrom() ERROR");
10.         printf("%s\n", strerror(errno));
11.         stop_log(l);
12.         exit(4);
13.     }
14.     uart_mtx.lock();
15.     ExeRampSpeed(1,u,(int)(abs((int)re-old)),(int)re,&f);
16.     uart_mtx.unlock();
17.     old = (int)re;
18.     clientIsSet = true;
19.     cv_client.notify_one();
}
```

5.2 STEVAL-SPIN 3201

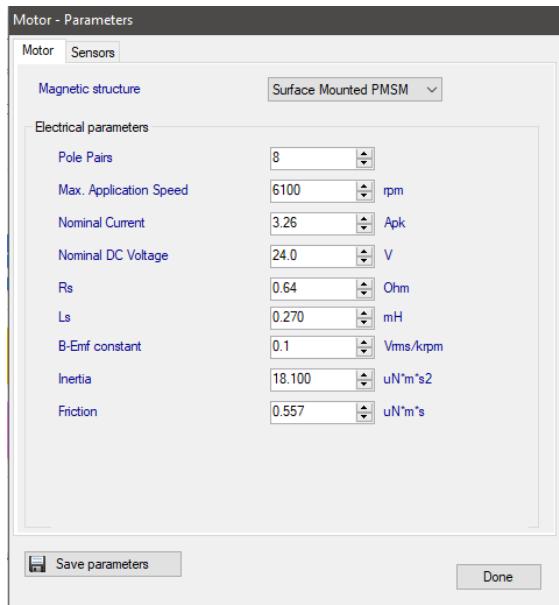
Oprogramowanie do regulacji prędkości silnika oraz komunikacji ze sterownikiem zostało w pełni wygenerowane przy pomocy narzędzie STM32 Motor Control Workbench. Jest to program przeznaczony na komputery osobiste pozwalające zmniejszyć wymagany nakład pracy potrzebny na opracowanie oprogramowania przeznaczonego dla sterowników z rodziny STM. Pozwala on na wygodne konfigurowanie gotowego projektu przy użyciu GUI, zainicjalizowanie odpowiednich dla żądanej aplikacji bibliotek, konfigurowanie parametrów silnika jak i parametrów samej regulacji [13].



Rysunek 5.5: Interfejs programu ST Motor Control Workbench

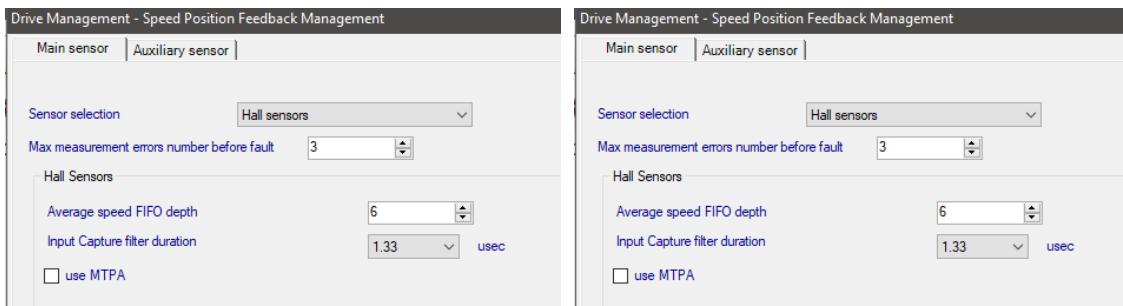
W celu stworzenia odpowiedniego dla naszej pracy projektu musielismy skonfigurować następujące elementy:

- parametry silnika:



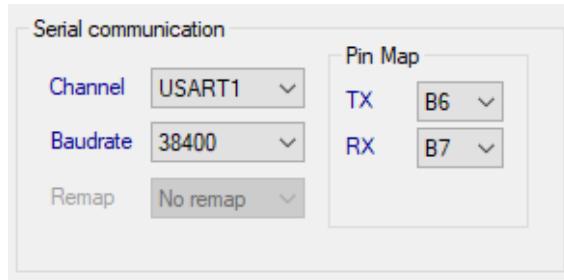
Rysunek 5.6: Interfejs programu ST Motor Control Workbench przeznaczony do konfiguracji parametrów silnika

- sposób pomiaru prędkości dokonywanej przez sterownik gdzie w naszym przypadku wybrana została opcja pomiaru przy użyciu czujników Hall-a:



Rysunek 5.7: Interfejs programu ST Motor Control Workbench przeznaczony do konfiguracji pomiaru prędkości silnika

- Szybkość transmisji służącej do komunikacji szeregowej:



Rysunek 5.8: Interfejs programu ST Motor Control Workbench przeznaczony do konfiguracji parametrów komunikacji ze sterownikiem

Po skonfigurowaniu odpowiednich parametrów można rozpocząć procedurę generowania gotowego projektu. W tym celu należy wybrać rodzaj środowiska programistycznego dla jakiego danego projektu ma zostać wygenerowany. Wybrane przez nas środowisko to STM32CubeIDE, będące specjalnie opracowanym edytorem kodu źródłowego dla produktów z rodziny STM32 opartym na środowisku Eclipse. Z uwagi na brak konieczności zmian w kodzie projektu, środowisko to zostało użyte jedynie w celu przeglądu, debugu i komplikacji programu.

Wygenerowane oprogramowanie dla sterownika STEVAL-SPIN zapewnia:

- regulację prędkości silnika w zamkniętej pętli sterowania,
- pomiar prędkości silnika przy pomocy czujników Hall-a,
- kontrolę błędów,
- pomiary prądów i napięcia w celu ochrony przed przekroczeniem dozwolonych zakresów,
- komunikację przez port szeregowy przy użyciu MCP (Motor Control Protocol).

Po skompilowaniu programu przez środowisko STM32CubeIDE, gotowy plik został wgrany do sterownika, poprzez znajdujący się na płytce programator. W tym celu użyty został specjalny program o nazwie STM32 ST-LINK utility, będący aplikacją do programowania mikrokontrolerów STM32.

Kolejnym udogodnieniem wynikającym z zastosowania gotowego rozwiązania, pochodzącego z programu STM32 Motor Control Workbench, jest zaimplementowany

w nim podprogram o nazwie Monitor. Jest to aplikacja pozwalająca na obsługę kontrolera z wykorzystaniem gotowego przyjaznego użytkownikowi GUI. Program ten był kluczowy w realizacji stanowiska w początkowych etapach prac, gdy biblioteka do obsługi kontrolera z poziomu RPI nie była jeszcze gotowa. Pozwoliła ona na przetestowanie wahadła zanim elementy takie jak RPI, kamera czy enkoder nie zostały jeszcze zaimplementowane. Aplikacja Monitor pozwala na interakcje z kontrolerem na wiele sposobów, między innymi jest to:

- podgląd rejestrów sterownika,
- zadawanie prędkości silnika,
- podgląd błędów,
- podgląd prędkości mierzonej silnika,
- uruchamianie rampy prędkości z możliwością ustawienia czasu jej trwania.



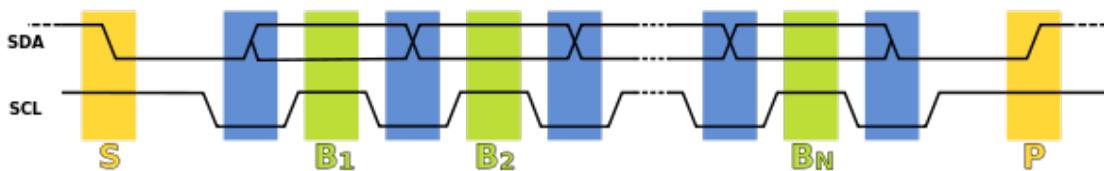
Rysunek 5.9: Interfejs programu Monitor służącego do komunikacji ze sterownikiem

5.3 Enkoder AS5600

W celu pomiaru kąta wychylenia wahadła zastosowany został moduł Grove - 12-bit Magnetic Rotary Position Sensor. Układ ten wyposażony jest w układ scalonu AS5600 będący 12-bitowym bezkontaktowym potencjometrem. Z uwagi na fakt iż dostarczona przez producenta biblioteka do obsługi owego czujnika przeznaczona jest jedynie na platformę mikrokontrolerów Arduino, należało podjąć się przepisania jej na platformę wykorzystaną w projekcie tj. RPI. Jako wzór posłużyła udostępniona przez producenta biblioteka przeznaczona na platformę Arduino [17].

Komunikacja czujnika z RPI odbywa się za pośrednictwem interfejsu I2C. Magistrala ta wyposażona jest dwa złącza, jedno przeznaczone do przesyłu danych

(SDA) natomiast druga jest linią zegarową (SCL). Występowanie linii zegarowej świadczy o tym iż interfejs I2C obsługuje wymianę danych synchronicznie za pośrednictwem zegara, w przeciwieństwie do chociażby zastosowanego w komunikacji RPI - STEVAL-SPIN, portu szeregowego, gdzie mamy do czynienia z transmisją asynchroniczną. Diagram obrazujący synchroniczną wymianę danych za pośrednictwem zegara zaprezentowano na rysunku 5.10



Rysunek 5.10: Diagram obrazujący synchroniczną wymianę danych za pośrednictwem zegara; źródło: <https://i2c.info/>

Wymiana danych przy użyciu I2C opiera się na relacji master - slave, gdzie masterem w naszym przypadku jest RPI, a slave czujnik. W interfejsie tym występuje również możliwość podpięcia kilku urządzeń pełniących rolę slave do jednego мастera. W tym celu wszystkie urządzenia należy wprowadzić w tą samą linię transmisyjną oraz linię zegarową. Przy takim połączeniu urządzeń wymiana danych jest możliwa dzięki wysłanemu przez mastera, przed każdym zapytaniem, bajtu będącego unikalnym adresem wybranego urządzenia. Dane wysłane przez mastera dostarczane są fizycznie do każdego z urządzeń, natomiast odpowiada na nie jedynie urządzenie o danym adresie. Funkcja ta nie została jednak przez nas użyta ze względu na fakt jednego tylko urządzenia wchodzącego w skład stanowiska, które komunikuje się z wykorzystaniem interfejsu I2C. Daje to jednak możliwości na przyszłą rozbudowę stanowiska [14].

Biblioteka pozwalająca RPI na komunikację z czujnikiem została napisana w języku C++ z wykorzystaniem biblioteki wiringPi. Została ona wyposażona w szereg różnych funkcji, takich jak na przykład zmniejszenie obszaru pomiarowego, w celu uzyskania większej rozdzielczości odczytywanych wartości kąta obrotu. W naszym przypadku wykorzystywany jest pomiar pełnego obszaru dlatego zaprezentowane zostaną wybrane możliwości biblioteki.

Do zainicjalizowania systemu I2C służy funkcja *AS5600_Init()* (listing 5.32).

Listing 5.32: Funkcja inicjalizująca interfejs I2C dla czujnika AS5600

```

01. AS5600_STATUS AS5600_Init(int *fd) {
02.     *fd,as5600 = wiringPiI2CSetup(AS5600_ADDRESS);
03.     return AS5600_OK;
04. }
```

Przyjmuje ona za argument wskaźnik do liczby całkowitej będącej deskryptorem pliku zainicjalizowanego połączenia. W celu inicjalizacji I2C użyto funkcji wchodzącej w skład biblioteki wiringPi. Jako jej argument przekazywany jest adres czujnika, którego wartość wynosi 0x36. Zwrotnica przez funkcję *AS5600_Init()* wartość to pole specjalnie przygotowanego typu wyliczeniowego *AS5600_STATUS* reprezentującego pomyślność wykonanych operacji (listing 5.33).

Listing 5.33: Typ wyliczeniowy reprezentujący pomyślność wykonanych operacji

```

01. /**
02. * Initialize AS5600
03. * @param[out] fd
04. * @return AS5600 status
05. */
06. typedef enum {
07.     AS5600_OK = 0, AS5600_ERROR = 1
08. } AS5600_STATUS;

```

W celu uzyskania danych o pomiarze kąta napisana została funkcja o nazwie *getRawAngle()*(listing 5.34).

Listing 5.34: Funkcja pobierająca kąt z enkodera magnetycznego

```

01. /**
02. * start, end, and max angle settings do not apply
03. * @return value of raw angle register
04. */
05. uint16_t getRawAngle()
06. {
07.     uint16_t result;
08.     readTwoBytes(&result, _raw_ang);
09.     return result;
10. }

```

Funkcja zwraca nie przeliczoną wartość kąta, czyli liczbę z zakresu czujnika tj. od 0 do 4095. W celu odczytania wartości użyta została funkcja *readTwoBytes()* do której przekazuje się wskaźnik zmiennej reprezentującej wartość odebraną oraz interesujący nas rejestr czujnika(listing 5.36). W tym przypadku przekazany rejestr jest połączeniem rejestrów *raw_ang_hi* (high byte kąta) oraz *raw_ang_lo* (low byte kąta)(listing 5.35).

Listing 5.35: Makra zawierające rejesty enkodera

```

01. #define _raw_ang_hi      0x0c
02. #define _raw_ang_lo      0x0d
03. #define _raw_ang          0xc0d

```

Funkcja odczytująca dane ze wskazanych rejestrów używa funkcji z biblioteki *wiringPi I2CReadReg16()* pozwalającej na odczyt dwóch bajtów z urządzenia.

Listing 5.36: Funkcja służąca do odczytania dwóch bajtów danych z rejestrów enkodera magnetycznego

```

01. ****
02. * Reads two bytes register from i2c
03. * @param[out] Result
04. * @param[in] in_addr
05. * @return AS5600 status
06. ****
07. AS5600_STATUS readTwoBytes(uint16_t *Result, uint16_t in_addr) {
08.     int response = wiringPiI2CReadReg16(as5600, in_addr);
09.     if (response != -1) {
10.         *Result = response;
11.         return AS5600_OK;
12.     }
13.     return AS5600_ERROR;
14. }

```

Wartości otrzymywane są wartościami “surowymi”, dlatego należy przeliczyć je na stopnie. Dane uzyskane z czujnika są reprezentowane przez wartości z zakresu od 0 do

4095 dlatego wzór pozwalający odpowiednio przeliczyć dane wygląda następująco:

$$\alpha = \alpha_{raw} * \frac{360}{4095} \quad (5.7)$$

Funkcja dokonująca tych obliczeń została przedstawiona na listingu 5.37.

Listing 5.37: Funkcja konwertująca dane z enkodera na stopnie

```

01. /**
02. * @brief Convert raw angle from sensor to degrees
03. *
04. * @param newAngle
05. * @return float
06. */
07. float convertRawAngleToDegrees(uint16_t newAngle)
08. {
09.     /* Raw data reports 0 - 4095 segments, which is 0.087 of a degree */
10.     float retVal = (float)(newAngle) * 0.087;
11.     return retVal;
12. }
```

5.4 System wizyjny

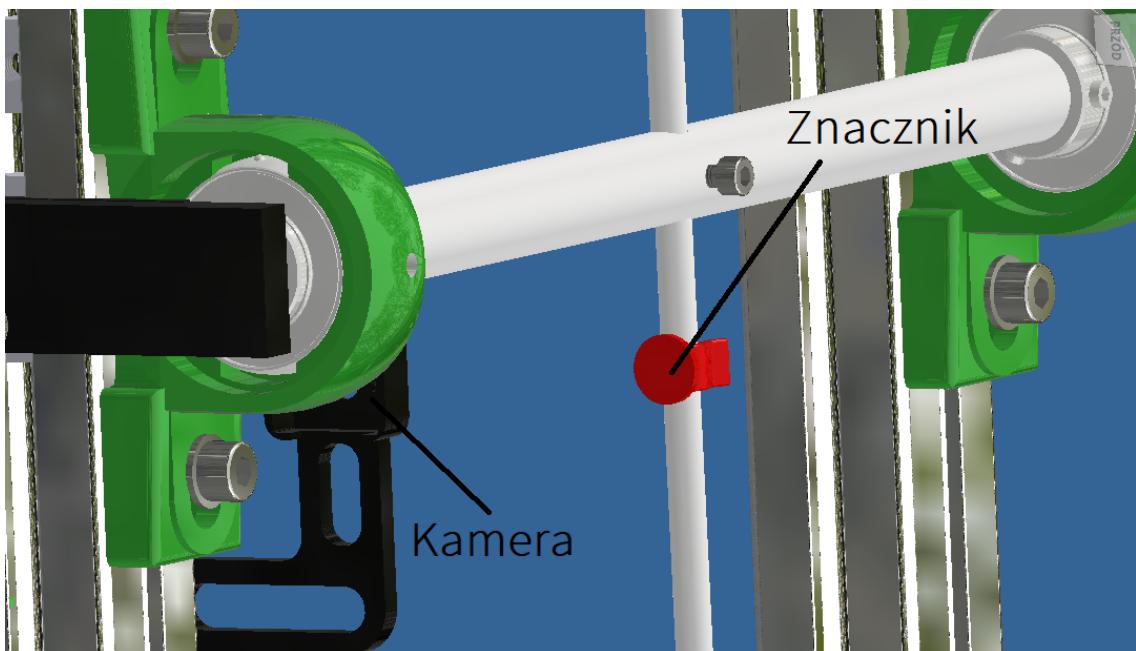
Autor: Patryk Pilarski

Jako dodatkowa forma pomiaru kąta wychylenia wahadła wdrożony został system wizyjny. Aby umożliwić jego implementację, zastosowany został moduł z kamerą Raspberry Pi Camera HD v2 8MPx [20]. Jak wskazuje na to nazwa, kamera posiada matrycję o rozdzielczości 8Mpx a także wspiera kilka trybów nagrywania, takich jak: HD 1080p / 30 fps, 720p / 60 fps, 640 x 480p / 90 fps. Komunikacja z tym modułem realizowana jest za pomocą dedykowanego złącza CSI.

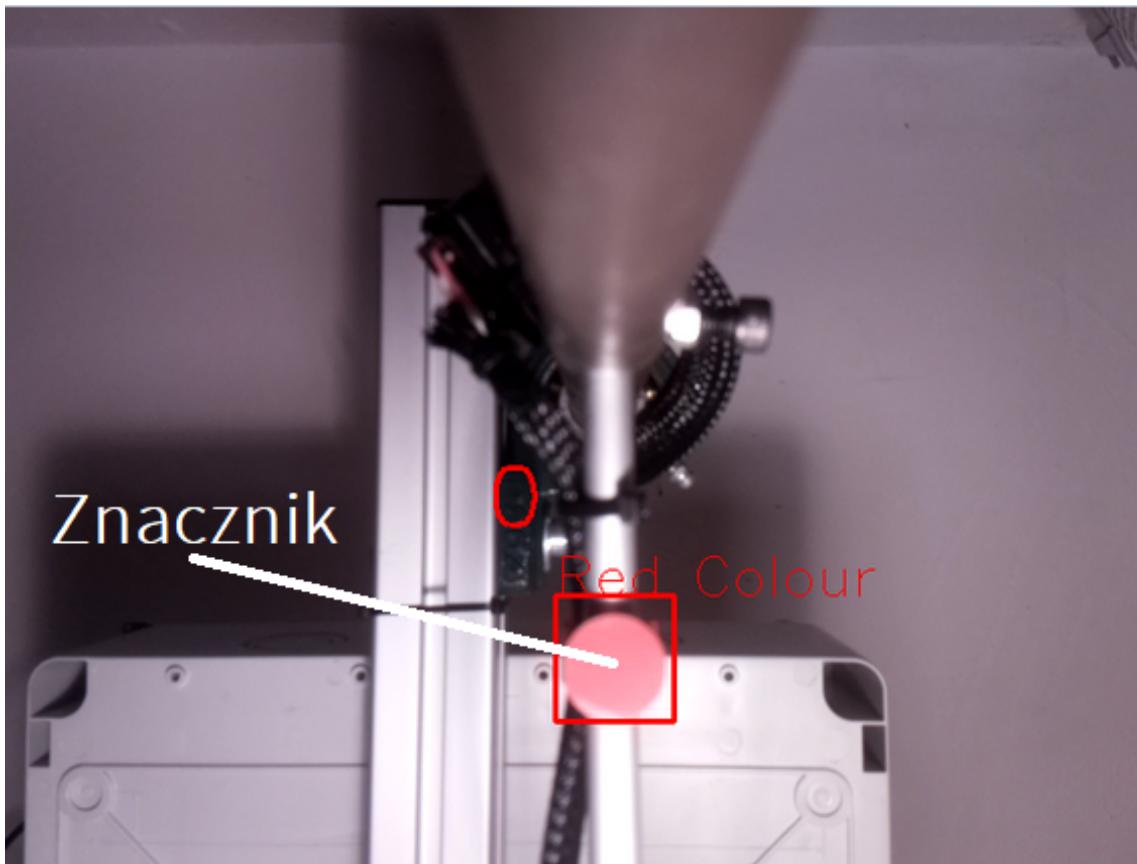


Rysunek 5.11: Camera HD v2 8MPx; źródło: <https://botland.com.pl>

W celu obsługi obrazu z kamery został stworzony skrypt w języku Python. Zdecydowano się na to środowisko ze względu na rozbudowaną dokumentację dla zastosowanego modułu oraz jego prostotę. Główną cechą tego programu jest zastosowanie biblioteki OpenCV, która służy do przetwarzania obrazu w czasie rzeczywistym [23]. W przypadku tego projektu zastosowana została w celu wykrywania czerwonego znacznika umieszczonego na wahadle.



Rysunek 5.12: Kamera i znacznik przedstawiony na makiecie



Rysunek 5.13: Widok z kamery zamontowanej na stanowisku

Do obsługi modułu kamery zastosowana została biblioteka PiCamera napisana w języku Python [21]. W listingu 5.38 przedstawiony jest sposób inicjowania kamery dla rozdzielczości obrazu 640x480 oraz 32FPS (od ang. *frames per second*)

Listing 5.38: Inicjowanie kamery

```

01. # BEGIN camera init #
02. camera = PiCamera()
03. camera.resolution = (640, 480)
04. camera.framerate = 32
05. rawCapture = PiRGBArray(camera, size=(640, 480))
06. time.sleep(0.1)
07. # END camera init #

```

Pętla główna (listing 5.39) odbiera kolejne ramki obrazu z kamery, następnie obraz przetwarzany jest na tablicę pikseli w formacie kolorów BGR i konwertowany do alternatywnej reprezentacji kolorów zwanej HSV (z ang. *Hue Saturation Value*).

Listing 5.39: Pętla główna programu

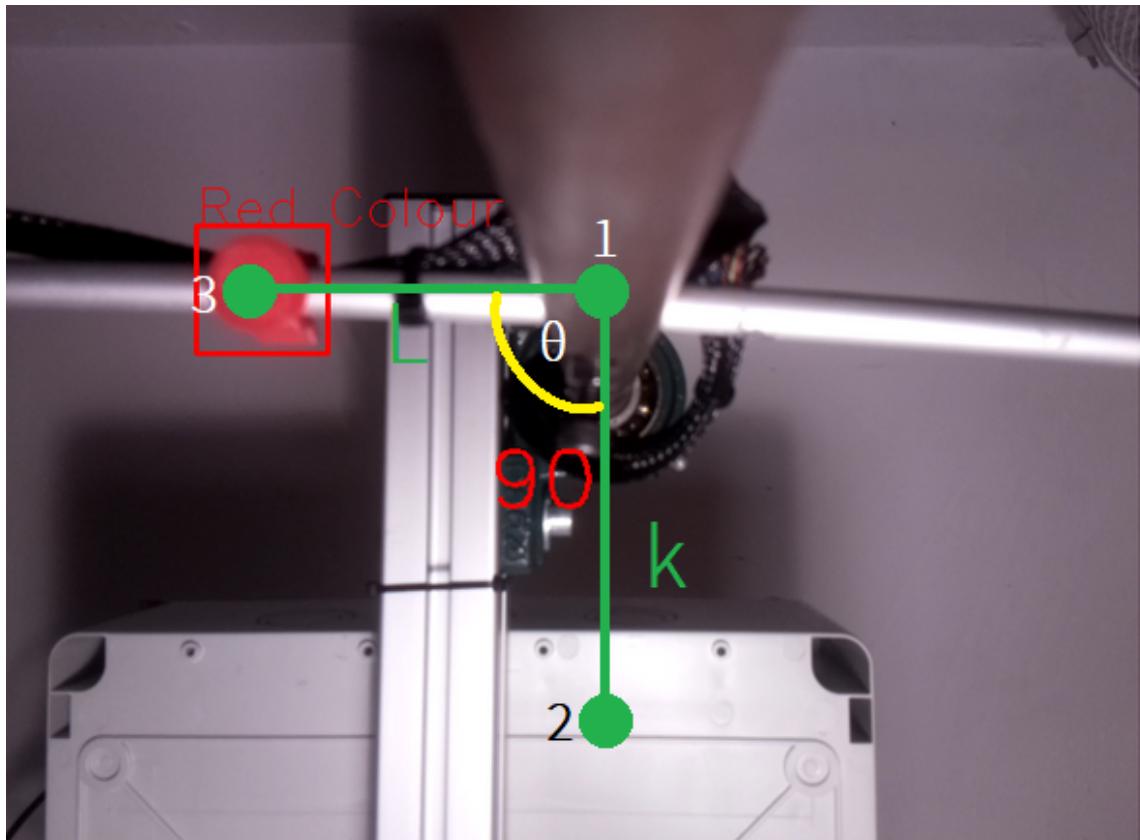
```

01. # # Main loop
02. for frame in camera.capture_continuous(rawCapture, format="bgr",
   use_video_port=True):
03.
04.     img = frame.array
05.     hsv_frame = cv.cvtColor(img, cv.COLOR_BGR2HSV)

```

Aby umożliwić odczyt kąta z obrazu otrzymanego z kamery, wyznaczone zostały trzy punkty. Dwa z nich mają na stałe przypisane wartości pikseli dla konkretnego

obrazu, natomiast trzeci punkt jest wykrywany poprzez rozpoznanie koloru znacznika. Rysunek 5.14 przedstawia układ punktów oraz prostych na obrazie z kamery.



Rysunek 5.14: Układ punktów i prostych

Kąt θ , który należy wyznaczyć, to kąt pomiędzy prostymi k i l przechodzącymi przez te punkty, obliczony za pomocą wzoru 5.8.

$$\tan\theta = \left| \frac{m_1 - m_2}{1 + m_1 m_2} \right| \quad (5.8)$$

gdzie:

$\tan\theta$ - kąt między dwoma prostymi

m_1 - współczynnik nachylenia prostej k

m_2 - współczynnik nachylenia prostej l

Wzór 5.8 służy do wyznaczenia kąta między dwoma prostymi, tylko dla $0 < \theta < 90$, warunek ten został uwzględniony w algorytmie zawartym w listingu 5.40, poprzez ograniczenie maksymalnego kąta wychylenia do 90, oraz minimalnego do -90.

Listing 5.40: Funkcja getAngle()

```

01. def getAngle(pointsLists):
02.     point1, point2, point3 = pointsList[-3:]  # # Get last three elements from
03.     # pointLists
04.     m1 = slope(point1, point2)  # # Calculate slope m1
05.     m2 = slope(point1, point3)  # # Calculate slope m2
06.     radiansAngle = math.atan((m2 - m1) / (1 + (m2 * m1)))  # # Calculate angle
07.     # between lines with slopes m1 and m2
08.     degreesAngle = round(math.degrees(radiansAngle))

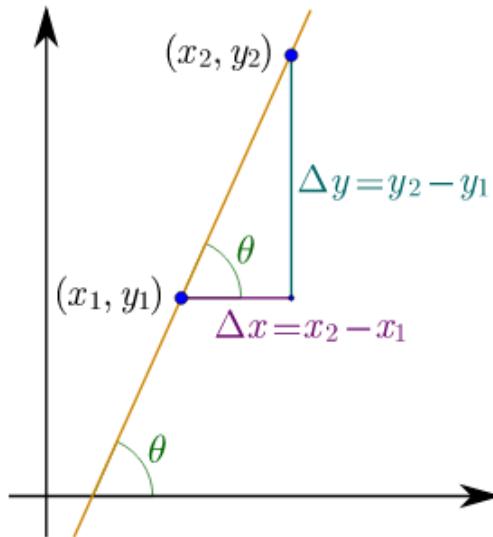
```

```

08.
09.     if (point1[1] >= point3[1]) and (point1[0] < point3[0]): # # set minimum
10.         angle
11.         degreesAngle = -90
12.     elif (point3[1] < point1[1]) and (point1[0] > point3[0]): # # set maximum
13.         angle
14.         degreesAngle = 90

```

Funkcja *slope()* (listing 5.41) służy do obliczenia współczynników nachylenia m_1 i m_2 , według wzoru 5.9.



Rysunek 5.15: Obliczanie współczynników kierunkowych

$$m = \frac{\Delta y}{\Delta x} \quad (5.9)$$

Gdzie $\Delta x \neq 0$, co zostało uwzględnione w algorytmie. Aby uniknąć dzielenia przez 0, do wartości x, dodany został jeden piksel - nie wpływa to znacząco na jakość obliczeń. Implementacja wzoru w programie:

Listing 5.41: Funkcja *slope()*

```

01. #####
02. # Calculates slope of line passing points pt1 and pt2
03. # @param[in] pt1, pt2
04. # @return gradient
05. #####
06. def slope(point1, point2):
07.     if point2[0] == point1[0]:
08.         point1[0] += 1
09.     return (point2[1] - point1[1]) / (point2[0] - point1[0])

```

Wspomniany wcześniej punkt numer 3 rozpoznawany jest na podstawie koloru znacznika. Do implementacji tej części programu została użyta biblioteka OpenCV. W pierwszej kolejności wyznaczany jest zakres kolorów dla czerwonej barwy.

Listing 5.42: Minimalny oraz maksymalny zakres koloru czerwonego

```

01. ## Set range for red color
02. lower_limit_color = np.array([135, 86, 109]) #lower case of color
03.

```

```

04. upper_limit_color = np.array([180, 255, 255])#upper case of color
05.
06. color_mask = cv.inRange(hsv_frame, lower_limit_color, upper_limit_color)
      # set range of red color

```

Następnie za pomocą wbudowanej w bibliotekę OpenCV funkcji *findContours()*, wyszukiwany jest kontur zdefiniowanych wcześniej zakresów koloru czerwonego.

Listing 5.43: Funkcja *findContours()*

```

01. contours, hierarchy = cv.findContours(color_range, cv.RETR_TREE, cv.
      CHAIN_APPROX_SIMPLE)

```

W celu poprawnego wykrywania koloru, sprawdzana jest powierzchnia wyszukanego konturu. Jeżeli wynosi więcej niż 300 pikseli i mniej niż 1000 pikseli, zostaje uznany jako znaleziony znacznik.

Listing 5.44: Wyszukiwanie konturu czerwonego koloru

```

01. # # Contur tracking red color
02. conturs= cv.findContours(color_range, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)
03. for pic, contur in enumerate(conturs):
04.     area = cv.contourArea(contur)
05.     if (area > 300) and (area < 1000): # # check size of contur
06.         x, y, w, h = cv.boundingRect(contur)
07.         img = cv.rectangle(img, (x, y), (x + w, y + h), (0, 0, 255), 2)
08.         point3 = [x, y] # # write point3 coordinates
09.         cv.putText(img, "RedColor", (x, y), cv.FONT_HERSHEY_SIMPLEX, 1.0, (0,
0, 255))
10.
11.     pointsList = [point1, point2, point3]

```

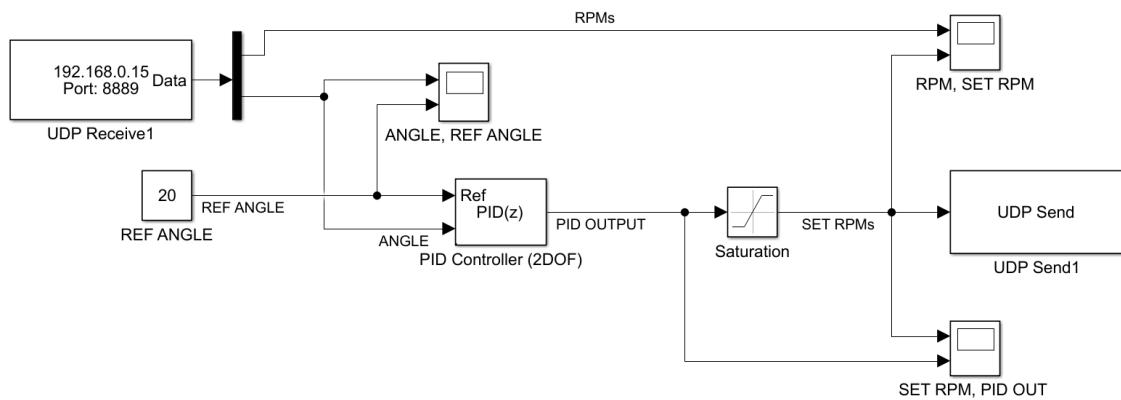
Rozdział 6

Sterowanie w pętli zamkniętej, testy

Autor: Bartosz Podkański

6.1 Aparat regulacji

Główym założeniem początkowym, była zdolność osiągania przez wahadło zadanego kąta wychylenia. Aby to osiągnąć, należało zaopatrzyć obiekt w regulator. Nie został on jednak zaimplementowany na mikrokomputerze, zostały wykorzystane do tego dane (aktualna prędkość oraz kąt odczytany z enkodera) przesyłane po protokole UDP i odbierane przez komputer PC, na którym zainstalowane zostało środowisko Matlab oraz pakiet numeryczny Simulink (jedno z założeń mówiło o współpracy z wyżej wymienionym środowiskiem) [27].



Rysunek 6.1: Aparat regulacji zbudowany w Simulinku

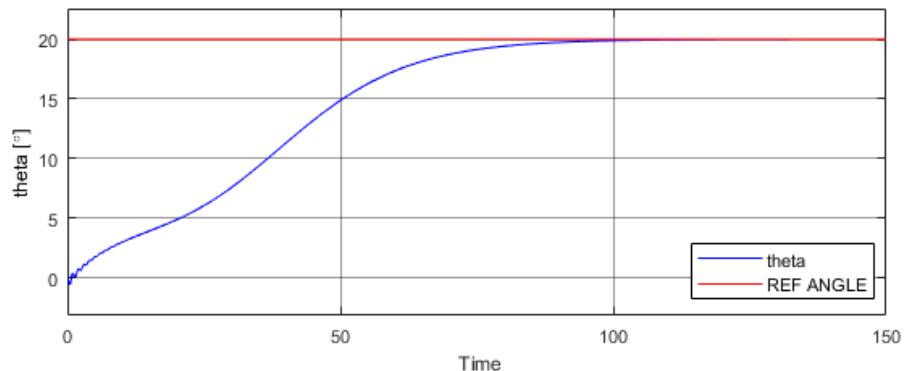
Do zbudowania aparatu regulacji, który miał posłużyć do regulacji kąta, użycie gotowego regulatora PID z biblioteki programu Simulink. Dane odbierane są za pomocą bloku UDP Receive, który wymaga wypełnienia pól: adres serwera, port, rozmiar odebranych danych, typ danych oraz forma zapisu danych (Little/Big Endian). Regulator na wejściu przyjmuje aktualnie odczytany kąt, oraz kąt zadany przez użytkownika. Wyjście regulatora stanowią obroty silnika, które są odsyłane z powrotem do Raspberry Pi, za pomocą bloku UDP Send. Blok ten należy zaopatrzyć w adres serwera oraz odpowiedni port. Za blokiem regulatora znajduje się blok ograniczający obroty. Wartości tego bloku powinny zostać wyznaczone przy pomocy

modelu, dla wahadła bez zamontowanej przeciwwagi jest to zakres (-4000, 5000). Ograniczenie to ma zapewnić nie przekraczanie przez wahadło krytycznych obrotów śmigła, które mogą spowodować wirowanie wahadła.

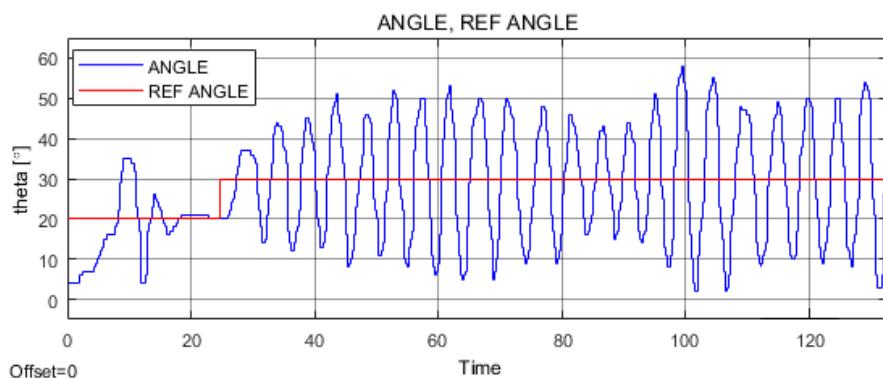
6.2 Wykorzystanie modelu w pętli zamkniętej

Posiadanie modelu obiektu niesie szereg zalet, które zapewniają komfort przewidzenia zachowania prawdziwego obiektu, oraz daje szansę wykluczenia ewentualnych błędów w trakcie użytkowania wahadła (wyznaczanie wartości krytycznych śmigła) i przyspiesza przeprowadzanie eksperymentów.

Jeżeli chodzi o współpracę regulatora z modelem w simulinku to liniowy regulator nie najlepiej radzi sobie z nieliniowym modelem wahadła. Znalezienie wzmacnień regulatora jest zadaniem trudnym i nietrywialnym, dla niewielkich zmian wartości nastaw, regulator odpowiada dużymi zmianami charakterystyk, ponadto odszukane za pomocą symulacji parametry nie zawsze prawidłowo funkcjonują na prawdziwym obiekcie. W związku z różnymi charakterystykami wartości ciągu od obrotów w zależności od kierunku obrotów śmigła, nierzadko trzeba wyznaczać pary nastaw regulatorów w zależności od znaku wartości kąta, który chcemy uzyskać, ponadto znalezienie wzmacnień dla jednego kąta referencyjnego, nie zawsze jest prawidłowe dla całego zakresu i po zmianie kąta zadanego na inny, regulator nie potrafi zadziałać prawidłowo.



Rysunek 6.2: Odpowiedź modelu dla nastaw regulatora: $K_p = 0.01$, $K_i = 5$

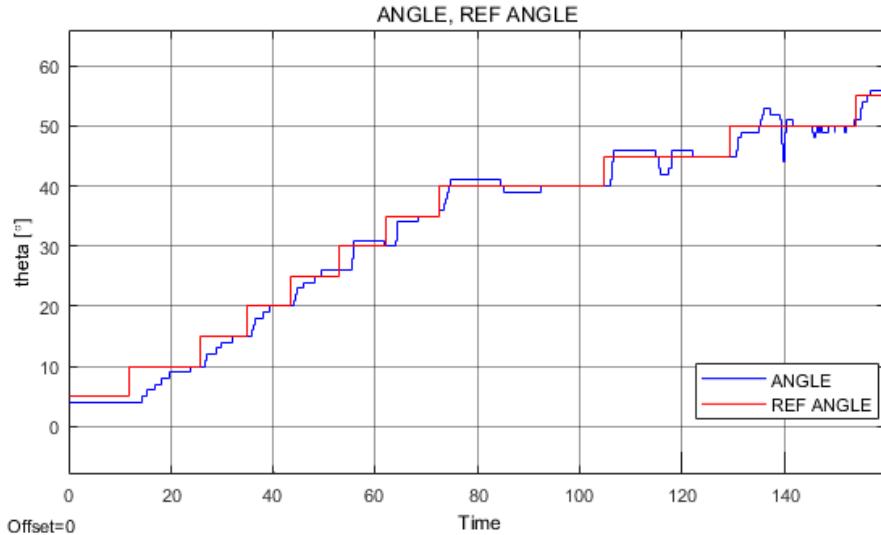


Rysunek 6.3: Odpowiedź stanowiska dla nastaw regulatora: $K_p = 0.01$, $K_i = 5$; osiągnięcie kąta 20°; oscylacje po zmianie na kąt zadany równy 30°.

6.3 Implementacja regulatora na stanowisku

6.3.1 Wysoka dynamika obiektu

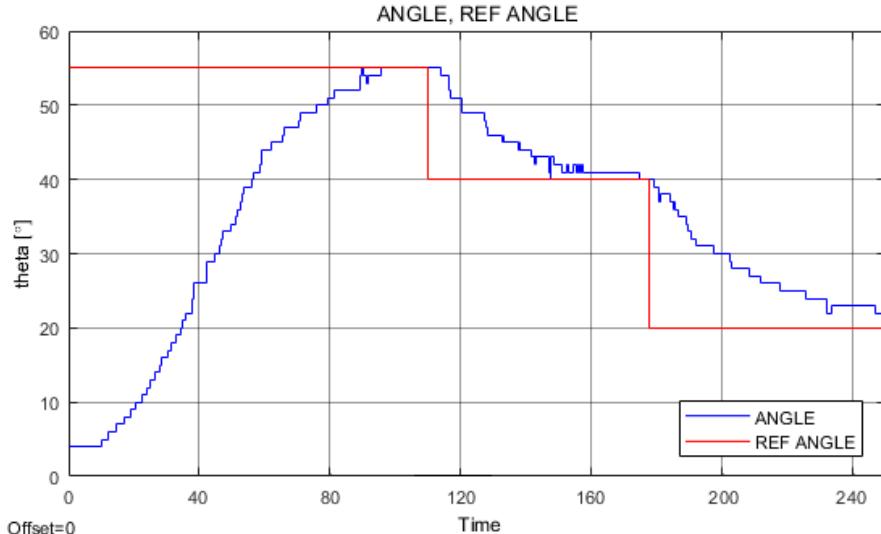
Kluczowym elementem wydaje się być człon całkujący i jego wpływ na dynamikę obiektu, która źle dobrana potrafi doprowadzić do wirowania całego wahadła. Duża wartość K_i jest nastawą regulatora odpowiednią dla małych zmian kąta.



Rysunek 6.4: Dotarcie do kąta 55° poprzez stopniowe zmiany kąta zadanego

6.3.2 Niska dynamika obiektu

Niska wartość parametru K_i sprawia, że zadany kąt osiągany jest w długim czasie zarówno dla niewielkich jak i dużych zmian kąta, zapewnia jednak konsekwentne osiąganie zadanego kąta bez ryzyka wpadnięcia wahadła w wirowanie.



Rysunek 6.5: Niska dynamika zapewnia brak ryzyka wpadnięcia wahadła w wirowanie

Idealnym rozwiążaniem wydaje się zastosowanie sterowania adaptacyjnego, w

którym dynamika obiektu zależy od wielkości zmiany kąta. Nieliniowe stanowisko niejednokrotnie bywa problemem dla liniowego regulatora. Ponadto, obiekt dla takich samych nastaw, jest w stanie zachować się całkiem inaczej w kolejnej próbie wyregulowania stanowiska.

Rozdział 7

Podsumowanie

Przeprowadzone testy i próby implementacji regulatora dla obiektu jakim jest wahadło lotnicze zakończyły się sukcesem i pozytywnym akcentem zwieńczyły projekt realizowany w ciągu ostatnich miesięcy. Jeżeli chodzi o projektowanie konstrukcji mechanicznej czy dobrów elementów elektrycznych i stworzenie pod nie konstrukcji elektrycznej, przydana okazała się analiza rynku i dostępnych rozwiązań. Istotnym była również liczba osób zaangażowanych w projekt, dawało to możliwość przedyskutowania pomysłów i proponowanych rozwiązań. Pomocnym okazała się stworzona makieta, dzięki której większość rzeczy była dopracowana, co nie tylko zaoszczędziło czas ale również dawało ogólny pogląd na tworzące się stanowisko. Zbudowany przedmiot jest w stanie dostarczyć w przyszłości rozległych informacji na temat badania obiektów nieliniowych, a przy zastosowaniu bardziej zaawansowanych metod regulacji, zakres pozycji osiąganych przez wahadło wzrośnie, umożliwiając stworzenie wahadła odwrotnego. Kwestią, o której warto wspomnieć, jest możliwość rozbudowy stanowiska w przyszłości. Pięć przewodów pierścienia ślizgowego, które nie zostały wykorzystane, pozwalają na zamontowanie na samym wahadle dodatkowych rozwiązań technicznych jak akcelerometr, czy czujnik drgań. Ponadto przewód, za pomocą którego podłączony jest enkoder posiada w rezerwie kolejne cztery przewody pozwalające na rozbudowe. Ważną modernizacją, która w przyszłości powinna zostać rozwinięta jest przystosowanie sterownika STEVAL do pracy z silnikiem, którego nadmiar energii wytworzonej podczas hamowania odkłada się na magistrali sterownika, powodując błąd *OverVoltage*, na przykład przez zastosowanie rezystora hamującego (na ten moment została wydłużona rampa silnika). Wygenerowana dokumentacja oprogramowania może w przyszłości istotnie ułatwić sprawę modyfikacji czy ulepszeń, o których wyżej wspomniano.

Dodatek A

Opis zawartości płyty DVD

- wersja edytowalna pracy,
- kod źródłowy oprogramowania dla Rpi,
- dokumentacja wygenerowana przy użyciu *Doxxygen* dla stworzonego oprogramowania,
- modele CAD,
- m-pliki i modele symulacyjne,
- plik wideo prezentujący montaż stanowiska,
- kosztorys w formie tabeli excela,
- instrukcje dydaktyczne wybranych zajęć,
- pliki projektu dla programu STM Motor Control Workbench i wygenerowany projekt dla płytki STEVAL-SPIN3201,

Spis tablic

2.1 Nagłówek datagramu UDP	4
2.2 Nagłówek datagramu TCP	4
5.1 Tabela wybranych parametrów konfiguracyjnych usługi	38
5.2 Podstawowa konstrukcja pojedynczej ramki	41
5.3 Lista komend wykonywanych przez sterownik	43
5.4 Przykładowa ramka startowa w zapisie heksadecymalnym	49
5.5 Przykładowa ramka odpowiedzi w zapisie heksadecymalnym	50
5.6 Przykładowa ramka odpowiedzi zawierająca kod błędu	50

Spis rysunków

2.1	Rodzaje silników BLDC	3
2.2	Wizualizacja ustawień pola magnetycznego	3
3.1	Wahadło napędzane kołem reakcyjnym Inteco	6
3.2	Wahadło lotnicze - University of Arizona	6
3.3	Wahadło oparte na jednej nodze	8
3.4	Silnik prądu stałego DF45L024048-A2	9
3.5	Sterownik STEVAL-SPIN3201	10
3.6	Przelotowy pierścień ślizgowy	11
3.7	Pierścień ślizgowy ZER-X5 24-Channel	11
3.8	Źródła napięcia	12
3.9	Elementy instalacji elektrycznej	13
3.10	Schemat elektryczny połączeń zasilaczy	13
3.11	Schemat elektryczny Raspberry Pi	14
3.12	Schemat elektryczny STEVAL-SPIN3201	15
3.13	Dioda zabezpieczająca	16
3.14	Makieta zbudowana w programie Autodesk Inventor	17
3.15	Elementy składające się na budowę ramy	18
3.16	Łożysko	19
3.17	Obudowa enkodera	20
3.18	Uchwyt mocujący kamerę	21
3.19	Elementy montowane po przeciwnie stronach osi	22
3.20	Elementy wycięte za pomocą lasera CNC	22
3.21	Sposób mocowania elementów	23
3.22	Osłona ze szkła akrylowego	24
3.23	Obudowa elektroniki	25
4.1	Rozkład sił	27
4.2	Model simulink	28
4.3	Wykres położenia wahadła od obrotów silnika	29
4.4	Wykres położenia wahadła od ujemnych obrotów silnika	30
4.5	Zmodyfikowany model	31
4.6	UDP Tryb nieblokujący	32
4.7	Odpowiedź modelu na skok 3000 RPM	32
4.8	Odpowiedź stanowiska na skok 3000 RPM	33
4.9	Odpowiedź modelu na skok 4800 RPM	33
4.10	Odpowiedź stanowiska na skok 4800 RPM	34
5.1	Schemat wymiany danych pomiędzy elementami stanowiska	35
5.2	Interfejs edytora kodu źródłowego Visual Studio Code	37

5.3 Schemat komunikacji master - slave	40
5.4 Schemat obrazujący przesunięcia kąta	57
5.5 Interfejs edytora	59
5.6 Interfejs programu ST Motor Control Workbench	59
5.7 Interfejs programu ST Motor Control Workbench przeznaczony do konfiguracji pomiaru prędkości silnika	60
5.8 Interfejs programu ST Motor Control Workbench przeznaczony do konfiguracji parametrów komunikacji ze sterownikiem	60
5.9 Interfejs programu Monitor służącego do komunikacji ze sterownikiem	61
5.10 Diagram obrazujący synchroniczną wymianę danych za pośrednictwem zegara	62
5.11 Camera HD v2 8MPx	64
5.12 Kamera i znacznik przedstawiony na makiecie	65
5.13 Widok z kamery zamontowanej na stanowisku	66
5.14 Układ punktów i prostych	67
5.15 Obliczanie współczynników kierunkowych	68
6.1 Aparat regulacji zbudowany w Simulinku	70
6.2 Odpowiedź modelu dla nastaw regulatora: $K_p = 0.01$, $K_i = 5$	71
6.3 Regulacja stanowiska - oscylacje	71
6.4 Regulacja stopniowa	72
6.5 Niska dynamika obiektu	72

Bibliografia

- [1] Git - oficjalna dokumentacja,
<https://git-scm.com/doc>
- [2] Doxygen - oficjalna domumentacja,
<https://www.doxygen.nl/manual/index.html>
- [3] Raspberry Pi 4B - oficjalna strona specyfikacji produktu,
<https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/>
- [4] Raspberry Pi Os - oficjalna strona systemu operacyjnego,
<https://www.raspberrypi.org/software/operating-systems/>
- [5] Raspberry Pi oficjalna dokumentacja - ssh,
<https://www.raspberrypi.org/documentation/remote-access/ssh/>
- [6] GCC - oficjalna dokumentacja,
<https://gcc.gnu.org/onlinedocs/>
- [7] Justin Ellingwood, Understanding Systemd Units and Unit Files,
<https://www.digitalocean.com/community/tutorials/understanding-systemd-units-and-unit-files>
- [8] Raspberry Pi oficjalna dokumentacja - systemd,
<https://www.raspberrypi.org/documentation/linux/usage/systemd.md>
- [9] UM1052, User manual, STM32F PMSM single/dual FOC SDK v4.3,
https://www.st.com/resource/en/user_manual/cd00298474-stm32f-pmsm-single-dual-foc-sdk-v43-stmicroelectronics.pdf
- [10] Klasa mutex, dokumentacja biblioteki standardowej języka C++,
<https://en.cppreference.com/w/cpp/thread/mutex>
- [11] Klasa thread, dokumentacja biblioteki standardowej języka C++,
<http://www.cplusplus.com/reference/thread/thread/>
- [12] Klasa condition_variable, dokumentacja biblioteki standardowej języka C++,
https://en.cppreference.com/w/cpp/thread/condition_variable
- [13] UM2374, User manual, Getting started with STM32 motor control SDK v5.0,
<https://www.st.com/en/embedded-software/x-cube-mcsdk.html#documentation>
- [14] I2C Info – I2C Bus, Interface and Protocol,
<https://i2c.info/>
- [15] Fall, Kevin R., W. Richard Stevens, 2013,
TCP/IP od środka: protokoły Gliwice: Grupa Wydawnicza Helion

-
- [16] Janina Rudowicz-Nawrocka, TCP/IP – formaty ramek, datagramów, pakietów...
http://www.au.poznan.pl/jankar/SK-TCP_IP-ramki.pdf
 - [17] Biblioteka dla enkodera magnetycznego AS5600,
https://github.com/Seeed-Studio/Seeed_Arduino_AS5600
 - [18] Dokumentacja techniczna dla silnika DF45L024048-A2,
<http://www.farnell.com/datasheets/1996061.pdf>
 - [19] Przegląd sterownika STEVAL-SPIN 3201,
https://www.st.com/resource/en/data_brief/steval-spin3201.pdf
 - [20] Dokumentacja techniczna dla modułu Raspberry Pi Camera HD v2 8MPx,
<http://www.farnell.com/datasheets/2056179.pdf>
 - [21] Dokumentacja dla biblioteki PiCamera,
<https://picamera.readthedocs.io/en/release-1.13/recipes1.html>
 - [22] Dokumentacja techniczna dla pierścienia ślizgowego,
<https://www.senring.com/capsule-slip-ring/m220.html>
 - [23] OpenCV- oficjalna dokumentacja,
<https://docs.opencv.org/>
 - [24] Specyfikacja techniczna zasilacza POS-50-C,
<http://pospower.pl/wp-content/uploads/POS-50-C-spec-PL-R2.pdf>
 - [25] Specyfikacja techniczna zasilacza sp-240-24,
<https://www.meanwell-web.com/content/files/pdfs/productPdfs/MW/SP-240/SP-240-spec.pdf>
 - [26] Eniko T. Enikov, Associate Professor, University of Arizona
<https://aeropendulum.arizona.edu/>
 - [27] The MathWorks, Inc.
<https://www.mathworks.com/>
 - [28] JTEKT Corporation
<https://koyo.jtekt.co.jp/en/support/bearing-knowledge/8-4000.html>
 - [29] Silvio Simani, Simulink – Modeling Dynamic Systems
http://www.silviosimani.it/simulink_modelling_dynamic_systems.pdf
 - [30] WiringPi - oficjalna dokumentacja
<http://wiringpi.com/reference/>
 - [31] Wikipedia, Brushless DC electric motor
https://en.wikipedia.org/wiki/Brushless_DC_electric_motor