

**Politechnika Warszawska**

W Y D Z I A Ł E L E K T R Y C Z N Y



Instytut Elektrotechniki Teoretycznej i Systemów Informacyjno-Pomiarowych

# Praca dyplomowa inżynierska

na kierunku Automatyka i Robotyka Stosowana

Oprogramowanie robota modułarnego opartego na mikrokontrolerze STM32

**Feliks Brzeziński**

numer albumu 319134

promotor

dr inż. Łukasz Makowski

WARSZAWA 2025



# **Oprogramowanie robota modułarnego opartego na mikrokontrolerze STM32**

## **Streszczenie**

Przedmiotem pracy jest opracowanie oprogramowania dedykowanego dla robota modułarnego oraz dobór komponentów elektronicznych kluczowych dla jego działania. Stworzono oprogramowanie open-source w języku C, które umożliwia realizowanie podstawowych funkcji urządzenia takich jak: działanie aktuatorów, wyznaczanie własnej pozycji czy archiwizacji stanów wewnętrznych oraz wykonywanych operacji.

W drugim rozdziale pracy przybliżono konstrukcję już istniejącego robota modułarnego, a następnie zdefiniowano peryferia potrzebne do jego poprawnej pracy w oparciu o budowę mechaniczną urządzenia. W skład elektroniki wchodzą: dwa servomechanizmy ST3020, dwa akcelerometry ADXL345, karta microSD, dwa nadajniki-odbiorniki MCP2551 oraz wyświetlacz LCD. By dokonać wyboru adekwatnego mikrokontrolera, zestawiono ze sobą jedne z najpopularniejszych rodzin mikrokontrolerów. Pozwoliło to na wyłonienie płytki deweloperskiej spełniające wszystkie wymogi układu – rozwiązanie firmy STMicroelectronics Nucleo-F446RE.

Na drodze analizy warunków pracy robota dokonano również wyboru protokołów, które pozwoliły na zrealizowanie połączenia międzymodułowego oraz ze światem zewnętrznym – odpowiednio protokół CAN oraz UART. Rozważano również silne oraz słabe strony obu mediów transmisji, co umożliwiło zdefiniowanie formatu wymiany danych, zapewniającego płynną pracę urządzenia.

Dokonano pomiarów i porównano dwie zastosowane metody wyznaczania kąta obrotu serwomechanizmu. W ramach przeprowadzania testów komunikacji, zaprojektowano panel sterowniczy pozwalający na podstawową obsługę robota. Przedstawiono możliwe kierunki rozwoju projektu oraz podsumowano rezultaty pracy.

**Słowa kluczowe:** robot modułarny, mikrokontroler STM32, CAN, UART



# **Software development for a modular robot based on the STM32 microcontroller**

## **Abstract**

Focus of this thesis is the development of software dedicated to a modular robot and selection of the electronic components key to its operation. Open-source firmawre was written in C language, and consisted of implementation of basic device functionalities such as: operation of actuators, determining its own position and logging internal states and performed operations.

The second chapter of the work focuses on the construction of an existing modular robot, and then defines the peripherals necessary for its correct operation based on the mechanical structure of the device. Used electronic elements include: two ST3020 servomechanisms, two ADXL345 accelerometers, a microSD card, two MCP2551 transceivers and an LCD display. In order to select an adequate microcontroller, some of the most popular microcontroller families were compared. As a result, development board that meets all the requirements of the system was chosen – Nucleo-F446RE solution produced by STMicroelectronics.

By analyzing the robot's operating conditions, the protocols that allowed for the connection between modules and external devices were selected - CAN and UART, respectively. The strengths and weaknesses of both transmission media were considered, which allowed for defining the data exchange format that ensures smooth operation of the devices.

Using acquired measurements two methods of determining the servo's rotation angle were compared. For the purpose of testing external and internal communication lines in addition to controlling the robot, control panel was made. Possible directions of project development were presented and the results of the work were summarized.

**Keywords:** modular robot, STM32 microcontroller, CAN, UART



# Spis treści

<b>1 Wprowadzenie</b>	<b>9</b>
1.1 Idea robota modułarnego . . . . .	9
1.2 Istniejące roboty modularne i ich zastosowania . . . . .	9
1.2.1 Zestawienie robotów . . . . .	12
1.3 Trudności w projektowaniu robotów modularnych . . . . .	13
<b>2 Robot modularny</b>	<b>15</b>
2.1 Magistrala CAN . . . . .	16
2.1.1 Zasada działania protokołu . . . . .	17
2.1.2 Arbitraż . . . . .	18
2.2 Dobór komponentów elektronicznych . . . . .	19
2.2.1 Dobór peryferiów . . . . .	19
2.2.2 Dobór mikrokontrolera . . . . .	20
<b>3 Oprogramowanie i testy robota</b>	<b>23</b>
3.1 Program główny . . . . .	23
3.1.1 Podstawowe struktury i typy wyliczeniowe . . . . .	25
3.2 Podstawy komunikacji z modułami . . . . .	28
3.2.1 Wiadomości w komunikacji UART . . . . .	28
3.3 Obsługa serwomechanizmu . . . . .	31
3.3.1 Zapis do pamięci serwa . . . . .	32
3.3.2 Odczyt pamięci serwa . . . . .	35
3.3.3 Polecenie ping serwomechanizmu . . . . .	37
3.3.4 Funkcje pomocnicze . . . . .	38
3.4 Obsługa akcelerometru . . . . .	40
3.4.1 Komunikacja z akcelerometrem . . . . .	41
3.4.2 Konfiguracja akcelerometru . . . . .	42
3.4.3 Pomiary i przetwarzanie danych . . . . .	42
3.4.4 Test sprawności akcelerometru . . . . .	44
3.4.5 Przetwarzanie odczytów . . . . .	45

3.5 Obsługa karty microSD . . . . .	46
3.5.1 Zapis danych na kartę . . . . .	46
3.5.2 Alarm RTC . . . . .	48
3.6 Obsługa wyświetlacza LCD . . . . .	49
3.7 Testy oprogramowania . . . . .	50
3.7.1 Porównanie pomiarów kąta obrotu serwomechanizmu . . . . .	51
<b>4 Podsumowanie</b>	<b>53</b>
4.1 Przyszłość projektu . . . . .	55
<b>Bibliografia</b>	<b>57</b>
<b>Spis rysunków</b>	<b>59</b>
<b>Spis tabel</b>	<b>61</b>

# Rozdział 1

## Wprowadzenie

Główym celem pracy jest opracowanie funkcjonalnego oprogramowania dedykowanego dla robota moduarnego. By osiągnąć taki rezultat należało: zdefiniować oraz zaimplementować komunikację wewnętrzną i zewnętrzną, obsługiwać peryferia, wdrożyć mechanizmy pozwalające na diagnostykę i serwisowanie urządzenia. W zakresie pracy mieściło się również dobranie komponentów elektronicznych, w tym serca układu – mikroprocesora.

### 1.1 Idea robota moduarnego

Od początku lat dwutysięcznych obserwuje się rozwój nowej dziedziny – robotyki modularnej. Na chwilę obecną należy zaliczyć ją do niszowych specjalizacji z zakresu robotyki, jednak zaczyna ona zyskiwać na popularności [19]. W momencie pisania pracy nie istnieje ogólnoprzyjęta definicja robota moduarnego, jednak źródła zazwyczaj są zgodne co do charakterystyki urządzeń. Roboty muszą składać się z modułów: posiadających niezależne aktuatora oraz zdolnych do łączenia się i rekonfiguracji [11, 19, 20]. Każdy z członów traktowany jest jako niezależny w pełni funkcjonalny system, co wymusza pośrednio optymalizację konstrukcji mechanicznej oraz układów elektronicznych na poziomie pojedynczego segmentu. Inherentna zdolność tworzenia dowolnych topologii poprzez różne sposoby łączeń modułów sprawia, że roboty modułarne mają lepsze zdolności adaptacyjne oraz wyższą niezawodność w stosunku do klasycznych robotów [19–21]. Charakter urządzeń sprawia, że nadają się one do dowolnych środowisk cechujących się dużą zmiennością.

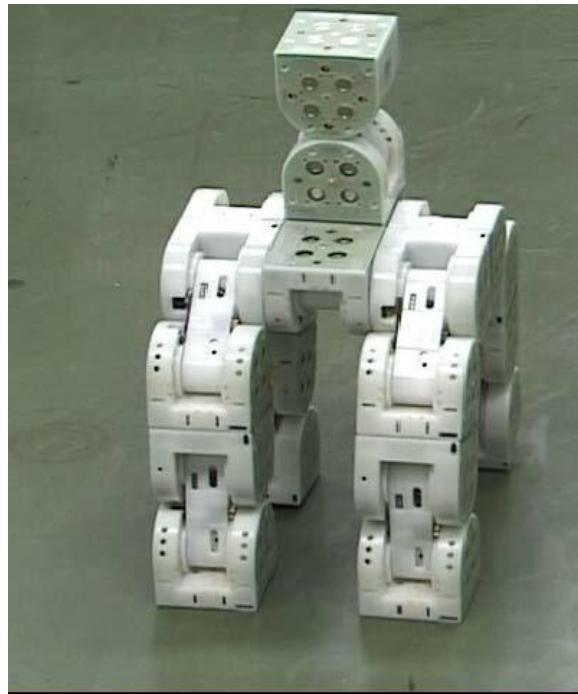
### 1.2 Istniejące roboty modułarne i ich zastosowania

W przeszłości powstało kilka urządzeń, które można zaliczyć do kategorii prototypów (ang. *proof of concept*). Nie posiadają one konkretnych aplikacji i służą bardziej do badania możliwości i praktyczności robotów modułarnych. Jednym z pierwszych przykładów takiego robota jest M-TRAN II [12]. Była to stosunkowo niewielka konstrukcja o kształcie przypominającym złożenie sześcianu oraz cylindra (rysunek 1). Umożliwiała dynamiczne łączenie się członów z wykorzystaniem magnesów.

## Rozdział 1. Wprowadzenie

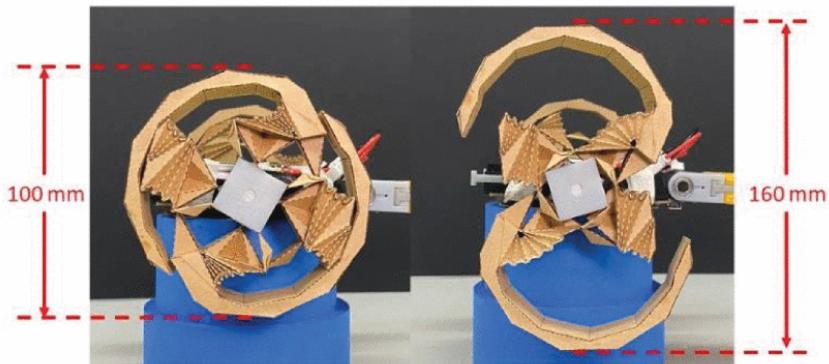
---

Moduły mogły być łączone w dowolne topologie, w tym takie umożliwiające ruch przypominający wicie się. Głównym tematem eksplorowanym przez badaczy była tematyka rekonfiguracji – zarówno pod względem fizycznym jak i procesu planowania kształtów robota.



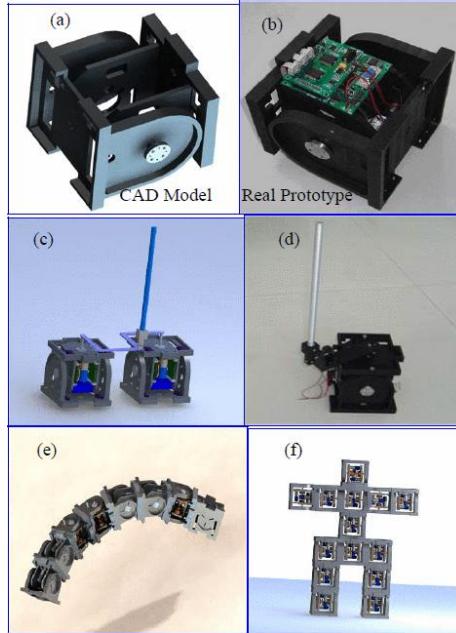
**Rysunek 1.** Robot M-TRAN II [12]

Innym ciekawym projektem był robot Orimo [4], który jest skrzyżowaniem się dziedzin robotyki modularnej oraz robotyki miękkiej (ang. *soft robotics*). Jego koła zostały wykonane z kartonu oraz złożone techniką origami (rysunek 2). Dzięki temu urządzenie jest w stanie zmienić swój kształt umożliwiając dwa sposoby lokomocji – kroczenie oraz jeżdżenie. Robot miał być odpowiedzią na problem znalezienia uniwersalnej metody poruszania się robotów.



Rysunek 2. Robot Orimo [4]

W literaturze można znaleźć również bardziej użytkowe koncepcje wykorzystania robotów modułarnych. Zaczynając od sektora edukacyjnego, gdzie uniwersalność oraz małe koszta sprawiają, że urządzenia tego typu są idealnym narzędziem do nauki. Przykładem takiej konstrukcji jest robot *NEURobot* [9], który w swoim założeniu ma wspierać nauczenie z zakresu teorii sterowania. Autorzy zapewnili możliwość sterowania urządzeniem z poziomu aplikacji *Matlab*, co pozwala na implementację złożonych algorytmów. Dodatkowo udostępniono narzędzie symulujące zachowanie cyfrowego bliźniaka. Konstrukcja robota oraz jego przykładowe topologie widoczne są na rysunku 3.



Rysunek 3. Robot NEURobot [9]

Przemysł kosmiczny jest kolejną branżą, której charakterystyka sprzyja robotom modułarnym. Projektem wyspecjalizowanym i dostosowanym do tej aplikacji jest robot *WORMS* stworzony w ramach programu 2022 NASA *BIG Idea Challenge* [14]. Twórcy zakładają istnienie trzech

generacji urządzeń, każda kolejna o bardziej złożonej konstrukcji oraz wyposażona w większą ilość czujników. W ramach pokazu możliwości stworzono sześciocionżego robota kroczącego (rysunek 4). Przeprowadzono również teoretyczną analizę czterech topologii, eksplorując ich zastosowania oraz właściwości (skalowalność, wytrzymałość etc.).



Rysunek 4. Robot WORMS [14]

### 1.2.1 Zestawienie robotów

Na podstawie wcześniejszej przytoczonej literatury widoczne jest, że roboty modularne są bardzo zróżnicowane pod względem ich budowy mechanicznej, ale również ich zastosowań. W tabeli 1 zestawiono cechy robotów kluczowe dla tej pracy.

	Orimo (2024)	WORMS (2022)	NEURobot (2009)	M-TRAN II (2003)
MCU	ESP32 S3 Wroom	Raspberry PI	TMS320F2812DSP	TMPN3120FE5M, PIC16F873/877
Komunikacja zewnętrzna	Wifi	Wifi	RF	LON
Komunikacja wewnętrzna	Brak/Wifi	Obecna, niezdefiniowana	Brak/RF	Asynchroniczna szeregowa
Sensory położenia	IMU, enkodery	–	Enkodery	Akcelerometr
Dodatkowe perfyferia	czujnik prądu	LIDAR	czujnik IR	–

Tabela 1. Zestawienie wybranych cech robotów modularnych

Chociaż definitive i dokładniejsze określenie cech wspólnych właściwych dla robotów modularnych wymagałoby przeanalizowanie większej ilości artykułów, przytoczone projekty mogą pozwolić na wyciągnięcie kilku wniosków.

Wspólnym mianownikiem okazuje się faworyzowanie komunikacji bezprzewodowej w przypadku komunikacji zewnętrznej (panel sterowniczy-robot). Prawdopodobnie wynika to z wygody takiego rozwiązania – nie trzeba martwić się o prowadzenie przewodów łączeniowych a także urządzenie sterujące nie musi być wyposażone w specjalny interfejs (jak w przypadku Wifi). Kolejną własnością łączącą roboty jest fakt wyznaczania swojej pozycji na podstawie pomiarów otrzymywanych z akcelerometrów i/lub enkoderów.

To co znacznie różni roboty między sobą to dobór mikrokontrolerów Microcontroller Unit (MCU) oraz dodatkowych peryferiów. Autorzy projektów nie podejmują się polemiki na temat ich doboru. W przypadku komponentów elektronicznych można zauważać trend dostosowywania robota do konkretnej aplikacji. W przypadku robota *WORMS* do konstrukcji dodano Light Detection and Ranging (LIDAR), który umożliwia implementację bardziej złożonych algorytmów lokalizacji i nawigacji – są one potrzebne by urządzenie mogło pracować w nieznanym środowisku takim jak Książyc. W projekcie *NEURobot* autorzy uwzględnili możliwość złożenia z modułów robota mobilnego. W takiej aplikacji potrzebne są sensory zbliżeniowe, w tym przypadku czujniki podczerwieni Infrared (IR). Być może decyza o wyborze konkretnych mikrokontrolerów była warunkowana głównie budową robota, jednak równie prawdopodobnymi powodami mogły być: preferencje programistów, dostępność na rynku oraz cena, obecność narzędzi/bibliotek dostosowanych do konkretnej rodziny mikrokontrolerów.

## 1.3 Trudności w projektowaniu robotów modularnych

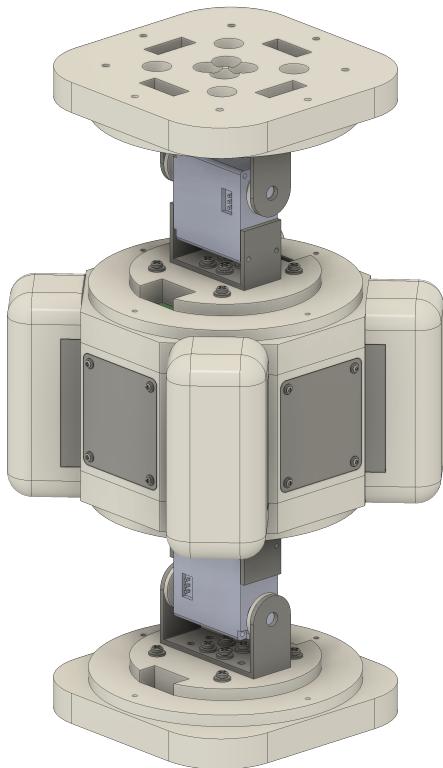
Główymi trudnościami związanymi z projektowaniem robotów modularnych są nie tylko kwestie mechaniczne takie jak wytrzymałość połączeń czy wielkość i waga modułów, ale również zagadnienie sterowania takimi układami o rozproszonej logice. Możliwe, że razem z rosnącym trendem zainteresowania tematyką sztucznej inteligencji zaobserwuje się rozwój i tej dziedziny. Standardowe podejście macierzowego opisu kinematyki robota bardzo szybko staje się niemożliwe do zrealizowania, dlatego dość popularną praktyką jest stosowanie algorytmów ewolucyjnych, nauczania ze wzmacnianiem czy sieci neuronowych [1].



## Rozdział 2

### Robot modularny

Bryłę robota moduarnego, który stanowi podstawę do rozważań tej pracy, opracował Kacper Olszewski w ramach swojej pracy inżynierskiej [15]. *Gizmo*, bo taką nazwę nosi, składa się z dwóch segmentów: członu sterowania i członu ruchowego. Wizualizacje pojedynczego modułu są widoczne na rysunkach 5a oraz 5b.



(a) Robot *Gizmo* z założonymi tubami



(b) Robot *Gizmo* bez tub

Człon sterowania, będący centralnym elementem struktury, jest pusty w środku. Dzięki temu można umieścić w jego wnętrzu płytki PCB oraz oczujnikowanie. Ograniczenia przestrzenne

w kontekście projektu obwodu drukowanego są dość niewielkie – konstrukcja pozwala na montaż płytki o wymiarach maksymalnych 7x7cm. Jeśli taka powierzchnia nie byłaby wystarczająca, możliwe jest równoległe połączenie kilku takich płyt o nakładając je jedna na drugą. Dodatkową przestrzenią użytkową robota stanowią wymienialne panele (szare prostokąty), na których można umieścić np.: wyświetlacz czy wiatraczek chłodzący wewnętrze urządzenia.

Człony ruchowe mają trzy główne funkcje: umożliwienie ruchu, zapewnienie połączenia elektrycznego oraz mechanicznego. Rolę aktuatorów pełnią serwomechanizmy, które jednocześnie są fizycznym łącznikiem między podstawami robota (konektorami) a członem sterującym. Konektory, widoczne na końcach urządzenia, posiadają złącza typu gold pin, dając łącznie osiem połączeń elektrycznych. Ich wyprowadzenia służą do uwspólnienia zasilania oraz magistrali komunikacyjnej robota. Ostatnim elementem konstrukcyjnym jest materiałowa tuba, w którą wszyte są przewody. Jej zadaniem jest przekazywanie sygnałów między konektorami a członem sterowania oraz ochrona układu przed ewentualnym pyłem.

## 2.1 Magistrala CAN

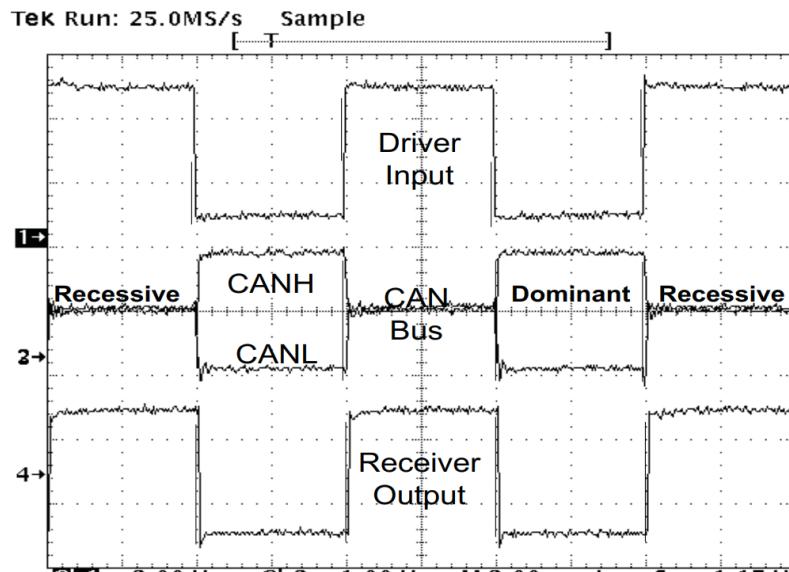
Jedną z priorytetowych kwestii było opracowanie metody komunikacji między modułami. Poszukiwany protokół powinien charakteryzować się:

- Brakiem nadzorzonego urządzenia – zakłada się, że każdy moduł może zażądać informacji od innych współpracujących segmentów. Zatem nie może zaistnieć sytuacja, w której protokół zezwala na jedynie jednego zarządcę magistrali.
- Małą ilość przewodów – ze względu na konstrukcję konektorów niemożliwe jest wprowadzenie większej liczby pinów, sumarycznie protokół może wykorzystywać cztery z nich (wliczając uziemienie).
- Dopuszczalną długością magistrali rzędu kilkudziesięciu metrów – ważne jest by istniała możliwość dołączenia do jednej magistrali wielu modułów, wiąże się to oczywiście ze stosunkowo dużą długością linii.
- Odpornością na zakłócenia elektromagnetyczne.

Do wyżej wspomnianego profilu pasuje CAN (*Controller Area Network*). Protokół zezwala na nadawanie przez dowolne urządzenie na magistrali, tak długo jak jest ona wolna lub wysyłana wiadomość ma wyższy priorytet. Maksymalna odległość między krańcowymi węzłami jest zależna od prędkości transmisji. Dla najwyższej z nich wynoszącej 1 Mb/s, długość magistrali może mieć do 40 m, natomiast dla 50 kb/s dopuszczalna jest odległość aż 1 km [6]. Interfejs CANa składa się z dwóch przewodów zazwyczaj sygnowanych jako CAN\_H oraz CAN\_L. Ponieważ protokół wykorzystuje sygnały różnicowe, jest on bardziej odporny na zakłócenia elektromagnetyczne [7] niż typowe magistrale jednostronne (ang *single-ended*).

### 2.1.1 Zasada działania protokołu

Tak jak wcześniej wspomniano protokół CAN wykorzystuje sygnały różnicowe. Magistrala może być w dwóch stanach: dominującym ( $\Delta V = V_{CAN\_H} - V_{CAN\_L} \simeq 2V$ ) lub recesywnym ( $\Delta V = 0V$ ). Co jest dość nietypowe, logika CANa jest odwrócona w porównaniu ze standardową interpretacją, gdzie dodatnie napięcie stanowi logiczną jedynkę. Przykładowe stany magistrali CAN oraz ich wartości logiczne widoczne są na rysunku 6.



Rysunek 6. Interpretacja stanów magistrali CAN [7]

W ramach protokołu możliwe jest wysyłanie 4 rodzajów ramek: danych (ang. *data frame*), błędów (ang. *error frame*), opóźniająca (ang. *overload frame*), żądania (ang. *remote frame*) [13]. Różnice w strukturze wiadomości są dość niewielkie, ale ze względu na użycie w projekcie wyłącznie ramek danych nie zostaną omówione. Poniżej widoczne są pola bitowe ramki danych 2.1.

Start	Pole arbitracji	RTR	IDE	DLC	Pole danych	CRC	ACK	EOF
-------	-----------------	-----	-----	-----	-------------	-----	-----	-----

Ramka danych 2.1. Struktura ramki danych CAN2.0A

**Bit startu** (logiczne 0), ma za zadanie zasygnalizowanie urządzeniom początku nowej transmisji. Opadające zbocze sygnału pozwala również na synchronizację.

**Pole arbitracji**, może mieć długość 11 lub 27 bitów zależnie od wybranego formatu (odpowiednio CAN 2.0A oraz CAN 2.0B). Z perspektywy logiki układu stanowi identyfikator wiadomości lub urządzenia. Jednocześnie w warstwie fizycznej, jego wartość stanowi również priorytetie wiadomości.

**RTR** (ang. *Remote-Transmission-Request*), jednobitowe pole które sygnalizuje czy wysyłana wiadomość zawiera dane (wówczas przyjmuje wartość logicznego 0)

**IDE** (ang. *Identifier Extension Flag*), jednobitowe pole informujące o formacie identyfikatora (logiczne 0 dla CAN 2.0A),

**DLC** (ang. *Data Length Code*), ma długość 4 bitów a jego wartość wskazuje na ilość wysyłanych bajtów danych. Maksymalna ilość danych, która może zostać przekazana w praktyce wynosi osiem. W przypadku, gdy w polu DLC pojawi się wartość większa, jest ona nadal interpretowana jako osiem [3].

**Pole danych** (ang. *data field*), możliwa jest transmisja od zera do ośmiu bajtów danych.

**Pole sumy kontrolnej CRC** (ang. *CRC – Cyclic Redundancy Check*), składa się z 15 bitów sumy kontrolnej oraz jednego bitu separującego (logiczna 1)

**Pole potwierdzenia ACK** (ang. *acknowledge field*), składa się z dwóch bitów. Pierwszy z nich informuje o poprawności przyjęcia wiadomości przez odbiorniki na magistrali (logiczne 0 przy poprawnej transmisji). Drugi stanowi separację i przyjmuje wartość logicznej jedynki.

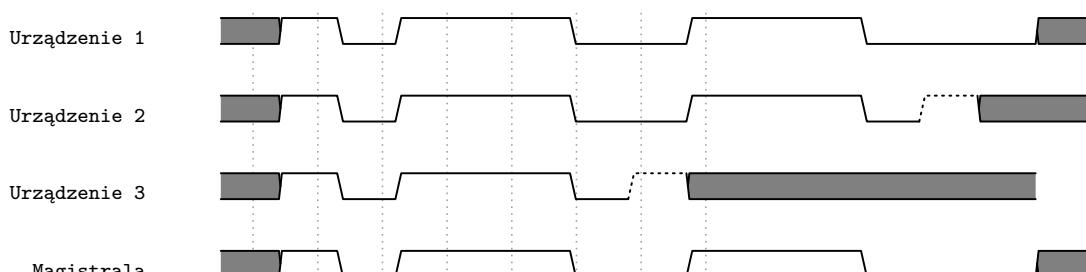
**Pole końcowe EOF** (ang. *end of frame*), składa się z 10 bitów każdy przyjmujący wartość logicznej jedynki.

### 2.1.2 Arbitraż

W przypadku zaistnienia sytuacji, w której dwa lub więcej urządzeń próbuje nadawać jednocześnie o pierwszeństwie decyduje proces arbitrażu. CAN jako protokół korzysta z mechanizmów zbiorczo nazwanych jako Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority (CSMA/CD + AMP). Oznacza to, że:

- Magistrala jest dzielona między wieloma równoważnymi węzłami.
- Kolizje wykrywane są przez porównywanie stanu magistrali z zawartością, którą chce wysłać dany węzeł – gdy są one różne oznacza to, że urządzenie przegrało arbitraż.
- Wynik arbitrażu dyktowany jest przez priorytet wiadomości – w przypadku CANa dominującą wartością jest logiczne zero.

Arbitraż występuje na podstawie zawartości pola arbitracji wiadomości.



**Ramka danych 2.2.** Przykładowy przebieg arbitrażu

Powyżej przedstawiono przykładowe rozstrzygniecie arbitrażu 2.2 między trzema urządzeniami. Widoczne jest, że w pierwszej kolejności urządzenie 3 przegrywa z 1 oraz 2. Następnie zachodzi podobna sytuacja, gdzie wygrywa urządzenie 1. Stan magistrali jest zawsze zgodny z treścią wiadomości, która wygra arbitraż.

Zrozumienie procesu arbitrażu oraz priorytetyzacji wiadomości jest kluczowe dla komunikacji w sieci wewnętrznej robotów. W przypadku znacznej ilości urządzeń na magistrali istotnie jest, które transmisje będą wysyłane zawsze a które mogą być pominięte. Rozkazy globalne informujące np.: o krytycznych błędach, awaryjnym zatrzymaniu czy prośbie o nadanie identyfikatora, powinny rozpoczynać się od logicznych zer. Należy zwrócić również uwagę, na fakt pewnej nierówności wynikającej z nadawania "wag" wiadomościom – przy zwiększym ruchu na magistrali może zajść sytuacja, w której wiadomości o niższym priorytecie są zawsze pomijane.

## 2.2 Dobór komponentów elektronicznych

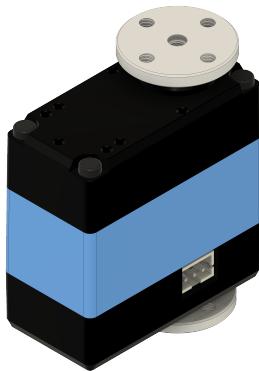
By zapewnić poprawne funkcjonowanie robota potrzebne jest osiągnięcie symbiozy między projektem bryły i elektroniką. Łatwo wyobrazić sobie sytuację, w której robot ma najwyższej jakości konstrukcję, jednak jego oczujnikowanie nie pozwala na dokładne wysterowanie urządzenia. Analogicznie nawet przy dobrze dobranych elementach elektronicznych, słabe spasowanie komponentów czy ich mała wytrzymałość doprowadzi do niesatysfakcjonującej pracy układu. Zatem wybór elektroniki powinien być warunkowany zarówno budową robota jak i charakterystyką jego pracy.

### 2.2.1 Dobór peryferiów

Pierwszymi istotnymi peryferiami są serwomechanizmy ST3020 firmy Waveshare [7]. Ich dobór warunkowany był wymaganiami mechanicznymi oraz faktem posiadania podzespołów pozwalających na wewnętrzny pomiar kąta obrotu. Wyznaczanie fizycznych parametrów pracy robota takich jak maksymalne momenty siły czy obrotowe nie są elementem tej pracy i zostały przyjęte na podstawie wniosków wyciągniętych podczas procesu projektowania mechanicznego robota [15]. W przeszłości projektu próbowało opracować metodę wyznaczania kąta obrotu serwomechanizmu w oparciu o zewnętrzne czujniki. Kwestia ta okazała się nietrywialna w przypadku zrezygnowania ze stosunkowo drogich enkoderów absolutnych dobrej jakości. Alternatywne podejście, takie jak użycie czujników Halla w połączeniu z magnesami diametalnie namagnesowanymi czy pomiar w oparciu o potencjometr, dawały wyniki obarczone dużym błędem bądź stanowiły metodę pomiaru wymagającą kalibrację każdego z układów. Dlatego zdecydowano się na wybór serwomechanizmu z wbudowanym układem pomiaru kąta.

Wybrany serwomechanizm posiada jednak dwie istotne wady: brak dokumentacji technicznej oraz "głuchą" komunikację. Pierwsza z nich sprawia, że algorytm pracy serwa jest niejawnym oraz nieznanym, a dokładne parametry podzespołów. Wynikowo należy podejść sceptycznie zarówno do jakości wysterowania urządzenia jak i pomiaru kąta jego nachylenia. Drugi problem stanowi obsługa

serwomechanizmu oparta na komunikacji za pomocą protokołu UART (ang. *universal asynchronous receiver-transmitter*), który nie posiada w swojej formule bitów potwierdzających odbiór wiadomości przez inne urządzenia na magistrali. Jest to słabość, która mogłaby zostać zmitygowana przez odpowiednią strukturę wymiany informacji z urządzeniem, jednak producent nie przewidział takiej możliwości.



Rysunek 7. Model serwomechanizmu ST3020 udostępniony przez producenta

Oba te powody złożyły się na decyzję dodania dwóch dodatkowych elementów elektronicznych – akcelerometrów ADXL345. Dzięki nim możliwe było dokonanie alternatywnego pomiaru kąta obrotu każdego z serwomechanizmów lub pośrednia weryfikacja ich pracy przez pomiar orientacji modułu. Warto dodać, że ze względu na charakter elementu taki pomiar musi być wykonany gdy układ jest statyczny, inaczej wyniki będą zaburzone przez przyśpieszenie samego robota lub drgania mechaniczne związane z jego pracą. W przyszłości akcelerometry mogą zostać również wykorzystane jako sygnalizator kolizji czy jako zabezpieczenie przed zbyt gwałtownym ruchem urządzenia.

Roboty modularne charakteryzują się zdcentralizowaniem informacji, bowiem każdy z modułów jest autonomicznym urządzeniem, a ich stany wewnętrzne choć mogą być przekazywane dalej w ogólności są niejawne. Oznacza to, że serwisowanie czy diagnostyka problemów w robocie złożonym choćby z kilku modułów może okazać się trudna. Dlatego bardzo istotne jest, by istniała możliwość odtworzenia sekwencji stanów pracy każdego z segmentów. W znacznej mierze jest to problem, który należy rozwiązać oprogramowaniem, jednak nadal wymaga on odpowiedniego wyposażenia robota. Archiwizację umożliwia karta microSD, dzięki której możliwy jest trwały zapis danych. W założeniu, wszystkie operacje robota powinny być na niej przechowywane. Natomiast obserwację bieżących parametrów pracy robota umożliwia wyświetlacz LCD, który docelowo będzie umieszczony na jednym z wymienialnych paneli członu sterującego.

### 2.2.2 Dobór mikrokontrolera

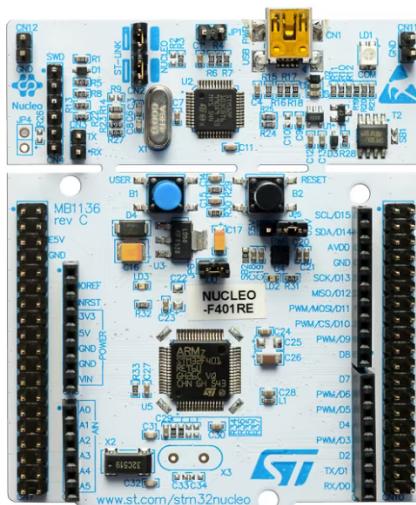
Ostatnim, oraz najważniejszym, elementem wymagającym doboru jest mikroprocesor/mikrokontroler. Wcześniej wspomniane peryferia definiują minimalną ilość oraz rodzaj interfejsów: SDIO (karta microSD), SPI (wyświetlacz), 2x UART (serwomechanizmy), 2x I<sup>2</sup>C (akcelerometry). Dodatkowo

znacznym atutem byłoby posiadanie przez układ wbudowanej obsługi magistrali CAN – co prawda możliwe jest stosowanie przejściówek, np.: UART-CAN, jednak ułatwiłoby to znacznie proces tworzenia oprogramowania. Biorąc pod uwagę charakter pracy robota, musi mieć on stosunkowo dużą pamięć ROM oraz RAM. Można spodziewać się znacznego obciążenia ze strony głównej magistrali CAN, a także z wykonywania operacji matematycznych opartych o macierze oraz funkcje trygonometryczne – dotyczących rozwiązywanie zagadnienia kinematyki, kinematyki odwrotnej czy wyliczanie orientacji modułów. Należy również wyjść z założenia, że robot będzie miał dość obszerne oprogramowanie, wynika to z dużej ilości peryferiów do obsłużenia, a także potrzeby implementacji bardziej zaawansowanej logiki systemu.

By przyśpieszyć proces prototypowania modułów zdecydowano się na wykorzystanie gotowej płytka. Analizie poddano trzy rodziny mikrokontrolerów: Arduino, ESP32 oraz STM32. Są to jedne z najpopularniejszych serii płyt na rynku.

Największą zaletą Arduino jest duże wsparcie ze strony społeczności hobbystów, ale również producentów, którzy zazwyczaj dołączają dedykowane biblioteki obsługujące ich urządzenia. Możliwe jest znalezienie modeli płyt, które spełniałyby oczekiwania związane z pamięciami ROM i RAM a także ilością dostępnych pinów i interfejsów. W porównaniu z innymi produktami w podobnym przedziale cenowym można zauważyc: niższe częstotliwości taktowania zegara (zazwyczaj wynoszące kilkadziesiąt MHz), mniejszą ilość i konfigurowalność przerwań, brak bardziej zaawansowanych funkcji takich jak DMA. Dodatkowym problemem, który ostatecznie zdyskwalifikował użycie Arduino, jest utrudniony proces diagnostyki kodu (ang. *debugging*). Jedynie kilka płyt ma oficjalne wsparcie tej funkcjonalności, w przypadku innych modeli należy użyć zewnętrznych bibliotek.

ESP32 cierpi na podobny problem z debuggingiem, jednak tym razem potrzebny jest zazwyczaj zewnętrzny programator podpinany do pinów JTAG. Powodem odrzucenia tej rodziny mikrokontrolerów jest fakt posiadania zbyt małej ilości pinów.



Rysunek 8. Płytki STM32 Nucleo-F446RE, zdjęcie ze strony producenta

W ofercie STMicroelectronics znajduje się mnogość modeli płyt zróżnicowanych zarówno pod względem interfejsów jak i ilości dostępnych pinów. Zdecydowano się na rozwiązanie z serii Nucleo, dokładniej Nucleo-F446RE 8. Urządzenie spełnia wymagania wynikające z doboru elementów oraz posiada stosunkowo dużą pamięć RAM (128 kB SRAM) oraz ROM (512 kB flash) [16]. Mikrokontroler ma wbudowane dwa interfejsy magistrali CAN, co stanowi dodatkowy atut. Korzystny jest fakt zapewnienia przez producenta dedykowanego środowiska programistycznego *STM32CubeIDE*, które współpracujące ze wszystkimi produktami tej firmy. Oprogramowanie zaspokaja podstawowe potrzeby związane z pisaniem kodu przeznaczonego dla systemów wbudowanych, zawiera opcje: przechodzenia programu 'krok po kroku', stawiania punktów przerwania czy podglądzania zawartości pamięci. Oprócz tego umożliwia prostą konfigurację pinów oraz zegarów urządzenia. Tak jak w większości układów STM32, możliwy jest debugging wykorzystując wyłącznie połączenie USB z komputerem hostem. Zasadniczą wadą wyboru rodziny STM32 jest istnienie ograniczonej ilości przykładowych kodów czy gotowych rozwiązań – wynika to z wyższego progu wejścia w stosunku do platform takich jak Arduino. Dodatkowym kryterium doboru było posiadanie przez mikroprocesor jednostki zmienoprzecinkowej (ang. *FPU, floating point unit*). Jest to cecha charakterystyczna architektur ARM-Cortex M4 oraz ARM-Cortex M7 [2]. Jej obecność na pokładzie mikrokontrolera przyśpieszy obliczenia związane z funkcjami trygonometrycznymi czy macierzami.

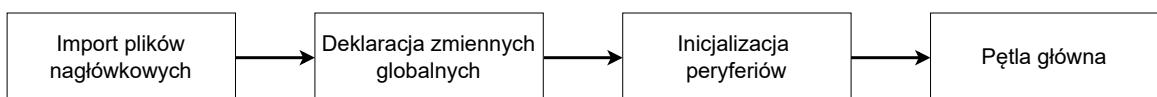
## Rozdział 3

# Oprogramowanie i testy robota

Oprogramowanie robota zostało napisane w języku C z użyciem dedykowanego środowiska *STM32CubeIDE*. Zgodnie z dobrymi praktykami programistycznymi wykorzystano system kontroli wersji *git*, a samo repozytorium znajduje się na stronie *Github*. Kod jest publiczny oraz udostępniony na licencji typu open-source (GPLv3), można go znaleźć pod tym adresem.

### 3.1 Program główny

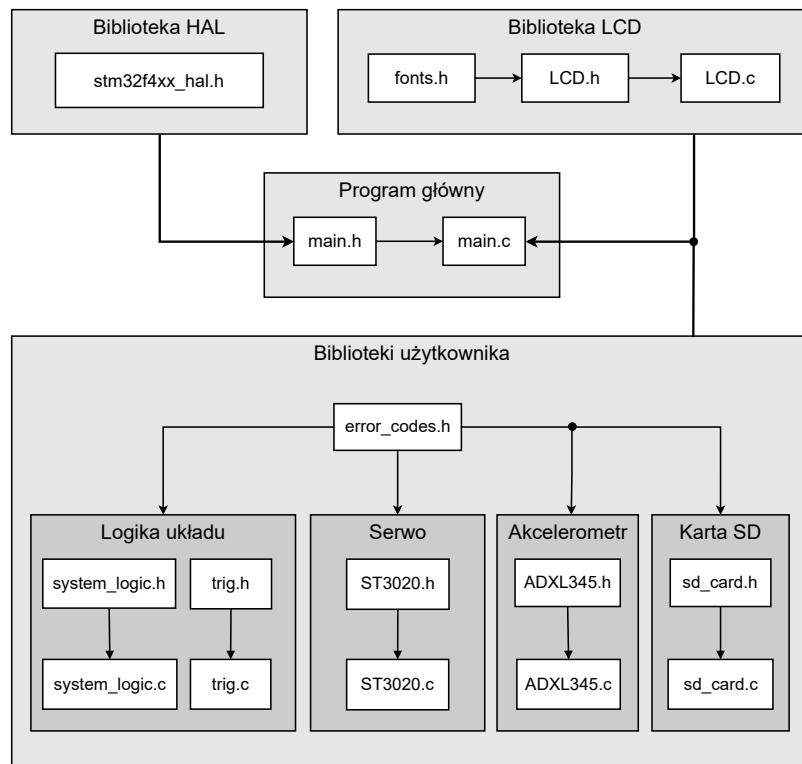
Program główny składa się z kilku podstawowych segmentów, a jego uogólniony przebieg jest zilustrowany na rysunku 9.



Rysunek 9. Kolejne etapy przebiegu programu głównego

**Import plików nagłówkowych** W pierwszym etapie importowane są pliki nagłówkowe użytkownika oraz biblioteki Hardware Abstract Layer (HAL). HAL jest biblioteką opakowującą (ang. *wrapper library*) funkcje operujące na rejestrach. Jej użycie pozwala na bardziej przyjazne użytkownikowi tworzenie oprogramowania dla mikrokontrolerów z rodziny STM32. Pomimo, że używanie pośredniego oprogramowania ogranicza kontrolę nad zachowaniem układu, w niniejszej pracy zdecydowano się na jego użycie – wynika to ze stopnia skomplikowania tworzenia własnych bibliotek najniższego stopnia abstrakcji (ang *bare-metal*) oraz ilości użytych interfejsów. Oprócz HAL użyto również gotowej biblioteki do obsługi wyświetlacza LCD zapewnionej przez producenta.

By zachować porządek oraz przejrzystość kodu, wszystkie większe funkcjonalności rozdzielono na niezależne pliki nagłówkowe. Schemat importowania plików nagłówkowych jest przedstawiony na rysunku 10. Dla zmniejszenia redundancji kodu i zapobiegnięcia wzajemnego importowania się plików dodano spajającą bibliotekę *system\_logic.c* oraz *system\_logic.h*, która zarządcą również logiką układu



Rysunek 10. Schemat importowania plików nagłówkowych

**Deklaracja zmiennych globalnych** W kolejnym kroku deklarowane są zmienne globalne. Użyto ich głównie do deklaracji buforów, struktur związanych z pracą peryferiów oraz zmiennych definiujących pracę robota (np.: identyfikator lub rolę). Zgodnie z zasadami "czystego kodu" (ang. *clean code*) ich ilość jest ograniczona do minimum. Było to możliwe dzięki napisaniu jak najbardziej ogólnych funkcji znajdujących się w plikach nagłówkowych.

**Inicjalizacja peryferiów** Następnie zachodzi inicjalizacja peryferiów, zegarów oraz innych układów potrzebnych do poprawnej pracy urządzenia. Funkcje oraz kolejność ich wykonywania są definiowane przez środowisko programistyczne na podstawie konfiguracji wynikającej z pliku .ioc. W tym segmencie znajdują się również funkcje inicjalizujące urządzenia oraz ich maszyny stanów. Konfigurują one pracę akcelerometrów, serw, karty microSD, wyświetlacza LCD oraz magistrali CAN oraz odpowiadają za uruchomienie podzespołów w odpowiednich trybach.

**Pętla główna** Ostatnim etapem pracy programu jest pętla główna programu, w której zarządzana jest praca peryferiów. Obsługa każdego z nich oparta jest o maszynę stanów, która z kolei jest zależna od komunikacji wewnętrznej (moduł-moduł) oraz zewnętrznej (moduł-panel sterowniczy).

### 3.1.1 Podstawowe struktury i typy wyliczeniowe

W pliku `system_logic.c` zdefiniowano kilka struktur oraz typów wyliczeniowych stanowiących podstawę do pracy robota. Pracę wszystkich układów uogólniono do wspólnej maszyny stanów opartej o typ `PERIPHERAL_STATE`, kodów błędów typu `ReturnCode` oraz kodów komend `COM_COMMAND`. Wszystkie stany `PERIPHERAL_STATE` przedstawione są w tabeli 2.

Wartość [hex]	Nazwa stanu w kodzie	Nazwa użytkowa stanu	Opis
0x00	PER_IDLE	Czuwanie	Urządzenie oczekuje na nowe polecenie
0x01	PER_DONE	Finalizacja	Urządzenie zakończyło wykonywanie polecenia z sukcesem
0x02	PER_WORKING	Praca	Urządzenie jest w trakcie wykonywania polecenia
0x03	PER_WAITING	Oczekiwanie	Urządzenie czeka na dane (używane jedynie w obsłudze magistrali CAN)
0x04	PER_ERROR	Błąd	W trakcie wykonywania polecenia lub przyjmowania polecenia wystąpił błąd
0x05	PER_FAIL	Awaria	Urządzenie przekroczyło limit następujących po sobie nieudanych operacji (kreślony stała <code>MAX_ERROR_COUNT = 5</code> )

Tabela 2. Spis możliwych stanów urządzeń

W celu przechowywania obecnego stanu peryferium wykorzystywana jest struktura `peripheral_state`, której definicja widoczna jest na listingu 1.

```

1 typedef struct peri_states{
2     COM_COMMAND cmd;
3     PERIPHERAL_STATE state;
4     ReturnCode last_code;
5     uint8_t error_count;
6 }peripheral_state;
```

Listing 1. Definicja struktury `peripheral_state` w pliku `system_logic.h`

W strukturze przechowywane są informacje o: obecnie wykonywanej komendzie (`cmd`), obecnym stanie (`state`), ostatnim rezultacie operacji (`last_code`) oraz liczbie błędów które nastąpiły po sobie (`error_count`). Sama obsługa maszyny stanów ogranicza się do funkcji `PeripheralUpdateState`.

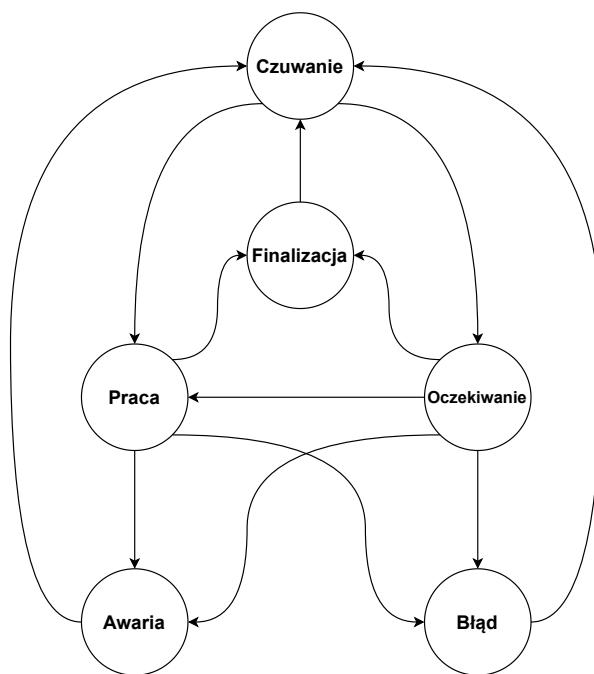
```

1 void PeripheralUpdateState(peripheral_state* per, ReturnCode status)
2 {
3     if( status != G_SUCCESS)
4     {
5         if ( per->error_count >= MAX_ERROR_COUNT)
```

```
6     per->state = PER_FAIL;
7 else
8 {
9     per->error_count += 1;
10    per->state = PER_ERROR;
11 }
12 }
13 else
14 {
15     per->error_count = 0;
16     per->state = PER_DONE;
17 }
18
19 per->last_code = status;
20 }
```

**Listing 2.** Definicja funkcji PeripheralUpdateState

Funkcja widoczna na listingu 2 zapisuje ostatni status operacji do struktury peripheral\_state. W przypadku wystąpienia niepowodzenia inkrementowany jest licznik błędów, a urządzenie jest wprowadzane na jego podstawie w stan błędu lub awarii. W przeciwnym wypadku licznik błędów jest zerowany i urządzenie przechodzi w stan finalizacji. W ogólności wszystkie możliwe przejścia między stanami zilustrowane są na rysunku 11



**Rysunek 11.** Legalne przejścia między stanami

Osiągnięcie stanu awaryjnego docelowo powinno wymagać ponownego uruchomienia całego układu oraz serwisowania go. Jednak w obecnej wersji oprogramowania stany awaryjne nie są utylizowane. Wynika to z faktu, że w trakcie debuggingu urządzenie może być celowo zmuszane do zwracania błędów w trakcie testów i jego resetowanie byłoby czasochłonne.

Jeśli chodzi o kody błędów (typu `ReturnCode`) przyjmują one wartości 8-bitowe co daje przestrzeń na 255 kodów (plus jeden kod zarezerwowany dla poprawnych operacji). Znaczna większość funkcji znajdujących się w plikach nagłówkowych zwraca błędy. Kody są na tyle różnicowane by w przypadku wystąpienia problemu dało się określić dokładny obszar oprogramowania z którego pochodzi. W przyszłości obsługa błędów może być rozbudowana i rozszerzona o przypisywanie im wag. Wybrane błędy wraz z ich opisami widoczne są w tabeli 3.

Nazwa błędu w kodzie	Wartość hex	Opis
G_ERROR	0x00	Ogólny kod błędu
G_SUCCESS	0x01	Kod sukcesu
C_MOUNT_ERROR	0x02	Nie udało się poprawnie zainicjalizować FATFS
G_FILE_WRITE	0x03	Nie udało się dokonać zapisu na kartę microSD
G_FILE_READ	0x04	Nie udało się otworzyć pliku na karcie microSD
G_COM_OVERFLOW	0x05	Przekazano ilość danych przekraczającą długość buferu magistrali CAN/UART
G_COM_TRANSMIT	0x06	Nie udało się wysłać wiadomości po magistrali CAN/UART
G_COM_RECEIVE	0x07	Nie udało się otrzymać poprawnej wiadomości z magistrali CAN/UART
C_RTC_ERROR	0x08	Błąd komunikacji z RTC
C_UART_TRANSMIT	0x09	Błąd komunikacji na magistrali UART serwomechanizmu

Tabela 3. Pierwsze dziesięć kodów błędów

Ostatnim istotnym fragmentem logiki stojącej za obsługą maszyny stanów są kody operacji. Pozwalają one na jednoznaczne ustalenie rozkazu, które ma wykonać peryferium oraz bezpośrednio wskazują na rodzaj urządzenia, które ma go wykonać. Spis wszystkich obsługiwanych komend znajduje się w tabeli 4.

Wartość [hex]	Nazwa operacji w kodzie	Peryferium	Opis
0x00	COM_IDLE	Wszystkie	Brak rozkazu
0x01	COM_SERVO_POS_READ	Serwomechanizm	Odczytaj pozycje enkodera serwomechanizmu
0x02	COM_SERVO_POS_SET	Serwomechanizm	Ustaw docelową pozycję serwomechanizmu
0x03	COM_SERVO_READ_TEMP	Serwomechanizm	Odczyt temperatury wewnętrz obudowy serwomechanizmu
0x04	COM_SERVO_PING	Serwomechanizm	Sprawdź status serwomechanizmu
0x05	COM_ACCANGLES_READ	Akcelerometr	Odczytaj zmierzone kąty nachylenia
0x06	COM_ACC_PING	Akcelerometr	Odczytaj ostatni status akcelerometru

Tabela 4. Spis dostępnych komend

## 3.2 Podstawy komunikacji z modułami

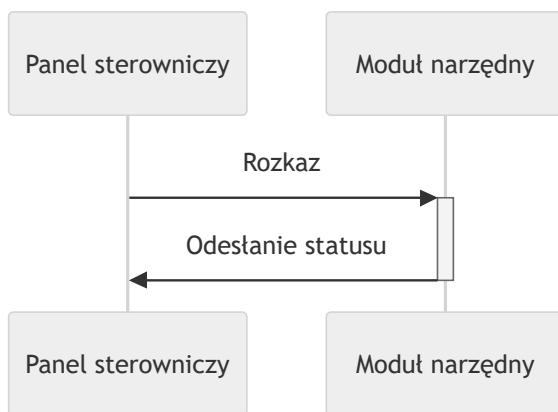
W przyszłości projektu możliwa będzie implementacja prawdziwej autonomii modułów, zarządzających swoim ruchem niezależnie od reszty układu. Wówczas komunikacja między modułowa sprowadzałaby się w znacznej większości do wymiany informacji o stanach wewnętrznych segmentów. Ze względu na zaawansowanie zagadnienia takiego sterowania nie jest ono elementem tej pracy. Zrealizowano jednak system komunikacji, który w przyszłości może zostać rozbudowany do takiej formy docelowej. W obecnej odsłonie w robocie znajduje się jeden moduł pełniący rolę kontrolera – w efekcie odpowiada on za koordynowanie ruchem na magistrali CAN oraz UART. Pozostałe moduły pełnią rolę podrzędną wykonując rozkazy odczytywane po magistrali CAN.

### 3.2.1 Wiadomości w komunikacji UART

Komunikacja moduł-panel sterowniczy została realizowana za pomocą protokołu UART, a w wymianie danych panel sterowniczy pełni rolę nadzorującą. Wybór tego sposobu komunikacji był uwarunkowany dwoma głównymi czynnikami. Pierwszym z nich jest prostota tego rodzaju transmisji, oprócz bitów stopu, startu oraz opcjonalnego bitu parzystości ramka zawiera jedynie dane. Oznacza to, że konfiguracja, uruchomienie oraz obsługa interfejsu jest prostsza w stosunku do SPI czy I<sup>2</sup>C. Drugim powodem jest powszechność interfejsu UART. Jest on obecny w większości mikrokontrolerów, a ponadto można nabyć przejściówki USB-UART, które umożliwią transmisję również z poziomu komputera personalnego przy niewielkim nakładzie finansowym. Daje to pole do przyszłej rozbudowy systemu – obecna komunikacja przewodowa może zostać łatwo zamieniona w bezprzewodową dodając na pokładzie robota odbiornik-nadajnik, który powtarzałby wiadomości przesypane po Wifi, BLE czy jakimkolwiek innym protokołem bezprzewodowym.

Jednocześnie trzeba mieć na uwadze, że ta metoda transmisji nie jest idealna. Pierwszym problemem jest asynchroniczność protokołu, powodująca możliwe błędne odczytywanie danych przy długich transmisjach w wyniku różnic próbkowania sygnału. Drugim znacznym minusem jest brak informacji zwrotnej o otrzymaniu wiadomości przez urządzenie na magistrali.

Mając na uwadze powyższe zdefiniowano dwa podstawowe formaty budowy wiadomości transmisji UART – ramkę rozkazu oraz ramkę odpowiedzi. Ta druga ma na celu zmitygowanie problemu, którym jest brak potwierdzenia przyjęcia transmisji – niezależnie od poprawności danych przychodzących do modułu, panel sterowniczy powinien otrzymać odpowiedź. Przykładowa wymiana między urządzeniami jest widoczna na ilustracji 12.



Rysunek 12. Generalny schemat wymiany wiadomości

Maksymalny czas między ostatnim bajtem rozkazu i pierwszym bajtem odpowiedzi wynosi 5s i jest zdefiniowany w logice panelu sterowniczego.

Ilość bajtów = N	ID modułu	Kod operacji	...	Suma kontrolna
------------------	-----------	--------------	-----	----------------

Ramka danych 3.1. Ogólna ramka rozkazu

Budowa ramki rozkazu jest widoczna na schemacie 3.1. Wiadomość rozpoczyna się od pojedynczego bajtu informującego o liczbie wszystkich danych, które powinny zostać otrzymane w ramach tej wiadomości. Następnie nadawany jest 8-bitowy identyfikator modułu, do którego powinien dotrzeć rozkaz. Później transmitowany jest kod operacji. Kolejne bajty zależne są od rodzaju wydawanego rozkazu. Całość kończona jest sumą kontrolną (ang. *checksum*), która jest negacją sumy wszystkich poprzedzającej ją bajtów. Najdłuższe poprawne transmisje liczą 10 ramek UART oznacza to, że problem dotyczący "gubienia" danych nie powinien zachodzić. Jednak gdyby nastąpił, przed przyjęciem niepoprawnych danych chroni dodane pole sumy kontrolnej. W obecnej wersji istnieją dwa formaty rozkazów widoczne na schematach 3.2 oraz 3.3.

**Ramka rozkazu SERVO\_SET\_POS**

0	1	2	3	4	5	6	7	8	9
0xA	ID modułu	Kod operacji	ID serwa	Przyśpie-szenie	LB pozycji	HB pozycji	LB prędkości	HB prędkości	Suma kontrolna

**Ramka danych 3.2.** Ramka rozkazu SERVO\_SET\_POS

**Ramka reszty rozkazów**

0	1	2	3	4
0x5	ID modułu	Kod operacji	ID peryferium	Suma kontrolna

**Ramka danych 3.3.** Ramka reszty rozkazów

Ramka wiadomości ma analogiczną strukturę i jest widoczna na schemacie 3.4. W jej przypadku pole komendy operacji zostało zamienione na pole statusu. Jest tak dlatego, że kod operacji jest zbędny do zidentyfikowania ramki rozkazu, na którą odpowiadają przychodzące dane. Wynika to z konstrukcji panelu, który zakłada wydawanie instrukcji pojedynczo. By umożliwić wymianę wielu transmisji w jednym momencie należało by na przykład rozszerzyć obie ramki o unikalny identyfikator.

Ilość bajtów = N	ID modułu	Status operacji	...	Suma kontrolna
------------------	-----------	-----------------	-----	----------------

**Ramka danych 3.4.** Ogólna ramka wiadomości

Obecnie występują cztery formaty ramek wiadomości, są one widoczne na schematach: 3.5, 3.6, 3.7 oraz 3.8.

0	1	2	3	4	5
0x6	ID modułu	0x1	Górny bajt pozycji	Dolny bajt pozycji	Suma kontrolna

**Ramka danych 3.5.** Ramka odpowiedzi dla SERVO\_READ\_POS

0	1	2	3
0x4	ID modułu	Status	Suma kontrolna

**Ramka danych 3.6.** Ramka statusowa

### 3.3. Obsługa serwomechanizmu

0	1	2	3	4	5	6	7	8	9
0x0A	ID modułu	0x01	Część całości kąta A	Część ułamkowa kąta A	Część całości kąta B	Część ułamkowa kąta B	Część całości kąta C	Część ułamkowa kąta C	Suma kontrolna

**Ramka danych 3.7.** Ramka odpowiedzi dla ACC\_ANGLES\_READ

0	1	2	3
0x05	ID modułu	Status	ID peryferium

**Ramka danych 3.8.** Ramka odczytu pozostałych danych

### 3.3 Obsługa serwomechanizmu

W strukturze serwa zintegrowane są enkoder absolutny oraz czujnik temperatury. Pozwalają one na zbieranie podstawowych informacji o pracy serwa. Wszelka komunikacja z urządzeniem jest możliwa przez UART w wydaniu jednoprzewodowym – oznacza to, że transmisja i odbiór ramek odbywa się w trybie pół-dupleksu (ang. *half-duplex*). Mikrokontroler pełni rolę nadzorującą natomiast serwomechanizm podrzędną. Kod obsługujący pracę serwomechanizmu jest umieszczony w plikach ST3020.h oraz ST3020.c.

Znacznymi problemami w opracowaniu obsługi serwa były: brak dokumentacji technicznej, opisu pracy urządzenia czy instrukcji użytkowania. Najbardziej wartościowym źródłem informacji były biblioteka oraz aplikacja przeznaczone na mikrokontrolery z rodziny AVR (Arduino lub ESP32). Na ich podstawie udało się w częściowo zdefiniować zasady działania serwa. Zadawanie oraz odczytywanie parametrów pracy odbywa się za pomocą operacji zapisu lub odczytu komórek pamięci SRAM lub EPROM. W pamięci trwałej serwomechanizm przechowuje przede wszystkim swój identyfikator, kod producenta oraz szybkość transmisji. W pamięci ulotnej znajdują się natomiast obecne parametry urządzenia takie jak zadany kąt, obecny kąt, temperatura, odczyt napięcia oraz prądu. W tabeli 5 pokazano mapę pamięci urządzenia.

EPROM (Wyłącznie odczyt)	
Nazwa rejestru	Adres rejestru [dec]
SCSCL_VERSION_L	3
SCSCL_VERSION_H	4
EPROM (Odczyt i zapis)	
Nazwa rejestru	Adres rejestru [dec]
SCSCL_ID	5

**Tabela 5.** Mapa pamięci serwa

SRAM (Odczyt i zapis)	
Nazwa rejestru	Adres rejestru [dec]
SCSCL_BAUD_RATE	6
SCSCL_MIN_ANGLE_LIMIT_L	9
SCSCL_MIN_ANGLE_LIMIT_H	10
SCSCL_MAX_ANGLE_LIMIT_L	11
SCSCL_MAX_ANGLE_LIMIT_H	12
SCSCL_CW_DEAD	26
SCSCL_CCW_DEAD	27
SCSCL_TORQUE_ENABLE	40
SCSCL_GOAL_POSITION_L	42
SCSCL_GOAL_POSITION_H	43
SCSCL_GOAL_TIME_L	44
SCSCL_GOAL_TIME_H	45
SCSCL_GOAL_SPEED_L	46
SCSCL_GOAL_SPEED_H	47
SCSCL_LOCK	48
SCSCL_PRESENT_POSITION_L	56
SCSCL_PRESENT_POSITION_H	57
SCSCL_PRESENT_SPEED_L	58
SCSCL_PRESENT_SPEED_H	59
SCSCL_PRESENT_LOAD_L	60
SCSCL_PRESENT_LOAD_H	61
SCSCL_PRESENT_VOLTAGE	62
SCSCL_PRESENT_TEMPERATURE	63
SCSCL_MOVING	66
SCSCL_PRESENT_CURRENT_L	69
SCSCL_PRESENT_CURRENT_H	70

Tabela 5. Mapa pamięci serwa

### 3.3.1 Zapis do pamięci serwa

Transmisja zapisu ma minimalną długość 6 ramek i składa się zawsze z dwóch transmisji, roboczo nazywanymi nagłówkiem oraz wiadomością właściwą. Ogólny format wiadomości widoczny jest na schemacie 3.9.

Bajty synchronizujące	Nagłówek	Wiadomość właściwa						
0xFF	0xFF	<table border="1"> <tr> <td>ID serwa</td> <td>Długość transmisji = N+3</td> <td>Komenda</td> <td>Adres pamięci serwa</td> <td>N danych</td> <td>Suma kontrolna</td> </tr> </table>	ID serwa	Długość transmisji = N+3	Komenda	Adres pamięci serwa	N danych	Suma kontrolna
ID serwa	Długość transmisji = N+3	Komenda	Adres pamięci serwa	N danych	Suma kontrolna			

**Ramka danych 3.9.** Ogólna struktura wiadomości serwomechanizmu

Pierwsza wiadomość musi zaczynać się od dwóch bajtów synchronizujących (o wartości 0xFF). Pozwalają one na jednoznaczne zidentyfikowanie początku wiadomości przez odbiorniki. Nie są one liczone jako faktyczna treść wymiany informacji – wyklucza je się z określania długości transmisji oraz sumy kontrolnej. Następnie podawany jest identyfikator serwa, który porównywany będzie z komórką pamięci EEPROM (SCSCL\_ID). Kolejny bajt mówi o długości transmisji równej N + 3, gdzie N to ilość wysyłanych bajtów danych do pamięci. Koniec treści nagłówka stanowią bajty kodu operacji (zgodnego z logiką serwa) oraz adresu pierwszego rejestru danych. Transmisja wiadomości właściwej składa się z N bajtów danych, które powinny zostać umieszczone w kolejnych komórkach pamięci serwomechanizmu. Koniec całej wymiany stanowi suma kontrolna będąca negacją sumy wszystkich bajtów obu transmisji.

Serwomechanizm przyjmuje trzy rodzaje komend:

- Zapis do pamięci = 0x03,
- Odczyt pamięci = 0x02,
- Ping = 0x01

Należy dodać, że w przypadku komendy ping występuje odstępstwo od formatu 3.9, bajt adresu pamięci serwa jest wówczas pomijany.

Funkcją odpowiadającą za transmisję wiadomości jest ServoWrite, której deklaracja przedstawiona jest w listingu 3.

```
1 static ReturnCode ServoWrite(UART_HandleTypeDef* uart, uint8_t cmd,
2                               uint8_t reg, uint8_t *data_buffer, uint8_t len)
```

**Listing 3.** Deklaracja funkcji ServoWrite

Zasięg funkcji ogranicza się do pliku ST3020\_servo.c, natomiast przyjmowane parametry to kolejno: wskaźnik do struktury interfejsu UART biblioteki HAL, kod operacji, pierwszy adres rejestru, do którego ma nastąpić zapis danych, wskaźnik do buforu zawierającego dane do transmisji, liczba danych w buforze.

```
1 HAL_StatusTypeDef status = HAL_HalfDuplex_EnableTransmitter(uart);
2 if (status != HAL_OK)
3 {
```

```
4     return C_UART_TRANSMIT;
5 }
6
7 const uint8_t buffer_size = (len > 5) ? len + 1 : 6;
8 uint8_t tx_buffer[buffer_size];
```

**Listing 4.** Początek funkcji ServoWrite

Jak widoczne w listingu 4, w pierwszej kolejności następuje zmiana roli pełnionej przez mikrokontroler na magistrali – wymuszane jest by stał się transmitem. Jest to realizowane za pomocą funkcji HAL\_HalfDuplex\_EnableTransmitter. Następnie obliczana jest potrzebna wielkość bufora danych i jest on inicjalizowany.

```
1 tx_buffer[0] = 0xFF;
2 tx_buffer[1] = 0xFF;
3 tx_buffer[2] = 0x01;
4 tx_buffer[3] = len + 0x03;
5 tx_buffer[4] = cmd;
6 tx_buffer[5] = reg;
```

**Listing 5.** Fragment ServoWrite odpowiadający za uzupełnienie treści nagłówka

W kolejnym kroku zapisywane są kolejne bajty nagłówka 5 zgodne z wcześniej przedstawionym formatem 3.9.

```
1 uint8_t checksum = 0;
2 for (uint8_t i = 0; i < len; i++)
3 {
4     checksum += *(data_buffer + i);
5 }
6
7 for (uint8_t i = 2; i < 6; i++)
8 {
9     checksum += tx_buffer[i];
10 }
11
12 status = HAL_UART_Transmit(uart, tx_buffer, 6, SERVO_TIMEOUT);
13 if (status != HAL_OK)
14     return C_UART_TRANSMIT;
```

**Listing 6.** Fragment ServoWrite odpowiadający za transmisję nagłówka

Przed wysłaniem nagłówka liczona jest suma kontrolna. Jest to realizowane poprzez dwie pętle `for` iterujące kolejne indeksy buforu nagłówka oraz treści właściwej, a następnie dodając ich zawartość do zmiennej `checksum`. By nie powodować niepotrzebnych opóźnień między transmisjami nagłówka a wiadomości właściwej suma kontrolna jest liczona przed rozpoczęciem pierwszej transmisji. W ostatnim kroku przeprowadza się pierwszą z dwóch transmisji, które realizowane są z użyciem funkcji `HAL_UART_Transmit`. Pełna implementacja tego etapu jest widoczna w listingu 6.

```

1   for (uint8_t i = 0; i < len; i++)
2   {
3       tx_buffer[i] = *(data_buffer + i);
4   }
5   tx_buffer[len] = (~checksum);
6
7   status = HAL_UART_Transmit(uart, tx_buffer, len + 1, SERVO_TIMEOUT);
8   if( status != HAL_OK)
9       return C_UART_TRANSMIT;
10
11 return G_SUCCESS;

```

**Listing 7.** Fragment ServoWrite odpowiadający za transmisję wiadomości właściwej

Jak widoczne w listingu 7, dane z buforu przepisywane są do buforu pomocniczego, a na jego końcu dodaje się pole stanowiące negację obliczonej sumy kontrolnej. Po tych operacjach otrzymuje się treść wiadomości właściwej. Jeśli w trakcie transmisji nie wystąpią błędy, całość funkcji zwraca kod sukcesu.

### 3.3.2 Odczyt pamięci serwa

Do odczytywania zawartości komórek pamięci serwa służy funkcja ServoRead, której deklaracja jest widoczna na listingu 8.

```

1   ReturnCode ServoRead(uint8_t servo_line, uint8_t reg, uint8_t*
2   data_buffer, uint8_t bytes);

```

**Listing 8.** Deklaracja funkcji ServoRead

Przyjmowane parametry to kolejno: indeks linii serwa definiujący magistralę UART, adres rejestru pierwszej komórki pamięci serwa do odczytu, wskaźnik do buforu w którym zapisane będą pobrane bajty, ilość bajtów do odczytu. Funkcja jest dostępna z poziomu programu głównego.

```

1   UART_HandleTypeDef* uart;
2   switch(servo_line)
3   {
4       case 0x00:
5           uart = uart_s1;
6           break;
7       case 0x01:
8           uart = uart_s2;
9           break;
10      default:
11          return C_UART_HANDLE;
12  }

```

**Listing 9.** Początek funkcji ServoRead

Proces zapisu zaczyna się od pobrania odpowiedniego wskaźnika na strukturę `UART_HandleTypeDef`, która jest kluczowa dla obsługi magistrali przez bibliotekę HAL. Interfejs jest wybierany w zależności od pobranego indeksu linii serwa, co jest widoczne w listingu 9.

```
1 if (ServoWrite(uart, SERVO_READ_INS, reg, &bytes, 1) != G_SUCCESS)
2     return G_SERVO_WRTIE;
```

---

**Listing 10.** Transmisja zażądająca odczytu danych

W pierwszej kolejności wysyłana jest wiadomość zażądająca N bajtów danych. Transmisja jest wykonywana przez wcześniej przygotowaną funkcję lokalną, która była opisana w poprzedniej sekcji. Jak widoczne w listingu 10 jest to operacja znacznie uproszczona dzięki zbudowanemu interfejsowi.

```
1 HAL_StatusTypeDef status = HAL_HalfDuplex_EnableReceiver(uart);
2 if (status != HAL_OK)
3     return C_UART_RECEIVE;
4
5 uint8_t header_buffer[5];
6 HAL_UART_Receive(uart, header_buffer, 1, SERVO_TIMEOUT);
7 if( header_buffer[0] != 0xFF)
8     HAL_UART_Receive(uart, header_buffer, 5, SERVO_TIMEOUT);
9 else
10    HAL_UART_Receive(uart, header_buffer + 1, 4, SERVO_TIMEOUT);
11
12 status = HAL_UART_Receive(uart, data_buffer, bytes, SERVO_TIMEOUT);
13 if (status != HAL_OK)
14     return C_UART_RECEIVE
15
16 if ( (header_buffer[2] != 0x01) || (header_buffer[3] != bytes + 0x02)
17 )
18     return C_UART_RECEIVE;
19 return G_SUCCESS;
```

---

**Listing 11.** Odczyt wiadomości zawierającej dane pobierane z pamięci serwa

Listing 11 przedstawia proces odbioru wiadomości od serwomechanizmu. Najpierw magistrala mikrokontrolera jest ustawiana w tryb odbiornika z użyciem funkcji `HAL_HalfDuplex_EnableReceiver` biblioteki HAL. W kolejnym kroku odczytywana jest pierwszy bajt transmisji i sprawdzane jest czy jest równy `0xFF`. Jeśli nie odebrane dane są odrzucane. Podczas pracy z serwomechanizmem nie zdarzyło się, by wymagane było dłuższe oczekiwanie na poprawne odebranie bajtów synchronizujących niż pojedyncza transmisja, dlatego nie ma potrzeby nasłuchiwanie wiadomości więcej niż raz. Finalnie otrzymuje się 5 bajtów stanowiących nagłówek odczytywanej wiadomości. Następnie magistrala jest ponownie nasłuchiwaną i odczytywane są bajty wiadomości właściwej. Na koniec weryfikowana jest zawartość nagłówka. Jeśli bajty długości transmisji oraz identyfikator serwomechanizmu są zgodne, zwracany jest kod sukcesu.

### 3.3.3 Polecenie ping serwomechanizmu

Ostatnią funkcją bezpośrednio obsługującą magistrale UART jest ServoPing. Służy ona do weryfikacji poprawności połączenia z serwomechanizmem. Deklaracja funkcji jest widoczna w listingu 12.

---

```
1 |     ReturnCode ServoPing(uint8_t servo_line, uint8_t id);
```

---

**Listing 12.** Deklaracja funkcji realizującej operację ping

Przyjmowane są dwa argumenty: indeks magistrali serwa oraz identyfikator serwa. Funkcja dostępna jest z poziomu programu głównego.

---

```
1 |     HAL_StatusTypeDef status = HAL_HalfDuplex_EnableTransmitter(uart);
2 |     if (status != HAL_OK)
3 |     {
4 |         return C_UART_TRANSMIT;
5 |     }
6 |
7 |     uint8_t buffer[6];
8 |     uint8_t checksum = ~(id + 0x02 + 0x01);
9 |     buffer[0] = 0xFF;
10 |    buffer[1] = 0xFF;
11 |    buffer[2] = id;
12 |    buffer[3] = 0x02;
13 |    buffer[4] = 0x01;
14 |
15 |    status = HAL_UART_Transmit(uart, buffer, 5, SERVO_TIMEOUT);
16 |    if (status != HAL_OK)
17 |        return C_UART_TRANSMIT;
18 |
19 |    status = HAL_UART_Transmit(uart, &checksum, 1, SERVO_TIMEOUT);
20 |    if (status != HAL_OK)
21 |        return C_UART_TRANSMIT;
```

---

**Listing 13.** Wiadomość zażądająca od serwomechanizmu operację ping

W pierwszej kolejności tak jak w wypadku ServoWrite mikrokontroler jest przełączony w tryb transmitemera za pomocą funkcji HAL\_HalfDuplex\_EnableTransmitter. Wiadomość nagłówka różni się od zwykłej transmisji tym, że nie posiada bajtu informującego o adresie rejestru serwa co jest widoczne w listingu 13. Reszta ramek danych pozostaje analogiczna. Widoczne jest że, treść wiadomości właściwej ogranicza się do sumy kontrolnej. Ponownie następują dwie transmisje.

---

```
1 |     status = HAL_HalfDuplex_EnableReceiver(uart);
2 |     if (status != HAL_OK)
3 |         return C_UART_RECEIVE;
4 |
5 |     status = HAL_UART_Receive(uart, buffer, 1, SERVO_TIMEOUT);
6 |     if (status != HAL_OK)
```

```
7     return C_UART_RECEIVE;  
8  
9     if(buffer[0] != 0xFF)  
10        status = HAL_UART_Receive(uart, buffer, 6, SERVO_TIMEOUT);  
11    else  
12        status = HAL_UART_Receive(uart, buffer + 1, 5, SERVO_TIMEOUT);  
13  
14    if (status != HAL_OK)  
15        return C_UART_RECEIVE;
```

**Listing 14.** Odbiór wiadomości ping

Listing 14 przedstawia część funkcji ServoPing odpowiedzialnej za odbiór wiadomości od serwomechanizmu. W pierwszej kolejności urządzenie wchodzi w tryb odbiornika i przeprowadza odczyt analogiczny do tego w funkcji ServoRead 11. Jednak, ponieważ jakość transmisji jest oceniana na podstawie treści treści wiadomości właściwej ale również nagłówka, funkcja ta nie jest tu wywoływana. Ponownie odczytywany jest pierwszy bajt, jeśli jest zgodny z oczekiwana wartością bajtów synchronizujących odczytywane jest kolejne 5 bajtów. W przeciwnym wypadku pierwsze dane są odrzucane i nasłuchiwanie jest kolejne 6 bajtów.

```
1   if ( buffer[2] != id )  
2       return G_SERVO_READ;  
3  
4   if ( buffer[3] != 0x02)  
5       return G_SERVO_READ;  
6  
7   checksum = buffer[2] + buffer[3] + buffer[4];  
8   checksum = ~checksum;  
9   if ( checksum != buffer[5] )  
10      return G_SERVO_READ;  
11  
12  return G_SUCCESS;
```

**Listing 15.** Weryfikacja wiadomości ping

Ostatni etap operacji ping przedstawia listing 15. Walidowane są pola bitowe związane z identyfikatorem serwomechanizmu, długością transmisji oraz sumą kontrolną. Jeśli nie wykryto żadnej niezgodności oraz wszystkie transmisje były przeprowadzone poprawnie, funkcja zwraca kod G\_SUCCESS. Jest to równoważne z dostarczeniem informacji zwrotnej o poprawnej pracy magistrali jak i samego urządzenia.

### 3.3.4 Funkcje pomocnicze

W celu ograniczenia redundancji kodu i uproszczeniu przepływu informacji między kolejnymi elementami programu głównego dodano kilka funkcji pomocniczych. Uporządkowują one również interfejs obsługi serwomechanizmu.

Pierwszą z funkcji pomocniczych jest ServoSetPos, która jak wskazuje nazwa pozwala na zdanie pozycji serwomechanizmowi, którą powinien osiągnąć z określona prędkością i przyśpieszeniem. Niestety producent nie zdefiniował, jak powyższe wartości są interpretowane przez sterownik układu. Na podstawie dokonanych obserwacji można przypuszczać, że podawane wartości są odpowiednio maksymalną prędkością oraz maksymalnym przyśpieszeniem.

```
1 ReturnCode ServoSetPos(uint8_t servo_line, int16_t pos, uint16_t
2   speed, uint8_t acc)
```

**Listing 16.** Deklaracja funkcji ServoSetPos label

Zgodnie z deklaracją ?? funkcja przyjmuje kolejno: indeks magistrali serwa, pozycję (mogącą być liczbą niedodatnią), prędkość oraz przyśpieszenie.

```
1 if(pos < 0)
2 {
3     pos = -pos;
4     pos |= (0x01 << 15);
5 }
6
7 uint8_t buff[7];
8 buff[0] = acc;
9 buff[1] = (uint8_t) (pos & 0x00FF);
10 buff[2] = (uint8_t) ((pos & 0xFF00) >> 8);
11 buff[3] = 0x00;
12 buff[4] = 0x00;
13 buff[5] = (uint8_t) (speed & 0x00FF);
14 buff[6] = (uint8_t) ((speed & 0xFF00) >> 8);
15
16 return ServoWrite(uart, SERVO_WRITE_INS, SERVO_ACC_REG, buff, 7);
```

**Listing 17.** Fragment kodu odpowiadający za zapis nowej pozycji do pamięci serwa

W listingu 17 przedstawiony jest odpowiedni zapis danych do transmisji oraz przekazanie wiadomości na magistralę. Kodowanie liczb niedodatnich jest inna niż w przypadku klasycznej zmiennej typu int – kod uzupełnień do dwóch. Logika serwa przyjmuje że znak liczby jest definowany przez najwyższy bit pozycji (0 dla liczb dodatnich oraz 1 dla niedodatnich). Bufor uzupełniany jest kolejno o: zadane przyśpieszenie, pozycję, czas (domyślnie wyzerowany) oraz prędkość. Wszystkie parametry oprócz przyśpieszenia są dwubajtowe. Kolejność i sposób zapisu danych do bufora transmisyjnego jest zdefiniowany przez odczytaną mapę pamięci serwomechanizmu 5.

Drugą funkcją pomocniczą jest ServoCurrentPosition, która pozwala na odczyt bieżącej pozycji serwomechanizmu.

```
1 ReturnCode ServoCurrentPosition(uint8_t servo_line, int16_t* result)
2 {
3     uint8_t pos_bytes[2];
```

```
4  ReturnCode status = ServoRead(servo_line, SERVO_POS_REG, pos_bytes, 2)
5  ;
6  *(result) = (((*(uint16_t) pos_bytes[1] & 0x7F) << 8) | pos_bytes[0]
7  );
8
9  if (pos_bytes[1] & 0x80 != 0x00)
10 *(result) = -*result;
11
12 return status;
13 }
```

**Listing 18.** Funkcja ServoCurrentPosition

Jak można odczytać z mapy pamięci serwa, informacja o jego bieżącej pozycji jest zapisana na przestrzeni dwóch rejestrów. Jak widoczne w listingu 18, po odczytce dane są łączone do jednej zmiennej – pierwszy bit wyższego bajtu jest maskowany, ponieważ zawiera on wyłącznie informację o znaku, tak jak wspomniano wcześniej.

Ostatnią funkcją pomocniczą jest ServoTemp. Odpowiada ona, jak nazwa wskazuje, za uzyskanie informacji o obecnej temperaturze wewnętrz obudowy serwomechanizmu. Całość kodu widoczna jest w listingu 19.

```
1  ReturnCode ServoTemp(uint8_t servo_line, uint8_t* temp)
2  {
3      return ServoRead(servo_line, SERVO_TEMP_REG, temp, 1);
4 }
```

**Listing 19.** Funkcja ServoTemp

Ponieważ funkcja polega wyłącznie na odczytaniu zawartości konkretnego adresu pamięci serwa można uznać ją za zbędną. Jednak została ona dodana dla zachowania jednolitości interfejsu obsługi serwomechanizmu.

### 3.4 Obsługa akcelerometru

Wybrany akcelerometr ADXL345 oferuje możliwość konfiguracji: wektora przerwań informującego o stanach wewnętrznych urządzenia oraz sposobu akwizycji danych i ich przechowywania. Na wybór tego elementu wpłynęła jego wszechstronność oraz jakość pomiarów. Możliwymi drogami komunikacji są protokoły SPI oraz I<sup>2</sup>C. Zdecydowano się na drugi z nich. Wynika to z faktu, że skomunikowanie peryferium wymagać będzie mniej przewodów (2 zamiast 4) oraz tego, że I<sup>2</sup>C daje pewność o odbiorze transmisji przez urządzenie na magistrali. Dzięki temu oprogramowanie będzie delikatnie odciążone, ponieważ nie zajdzie potrzeba implementacji kolejnego mechanizmu mitigującego ten problem.

### 3.4.1 Komunikacja z akcelerometrem

Zapis oraz odczyt danych akcelerometru opiera się o obpytywanie konkretnych rejestrów pamięci urządzenia, a ich adresy dane są w dokumentacji. Biblioteka HAL zapewnia "automatyczne" zarządzanie magistralą, interpretując bity konieczne dla potwierdzenia poprawności transmisji, ale nie będących "interesującymi" z punktu widzenia użytkownika.

```

1 static ReturnCode Acc_Cmd(I2C_HandleTypeDef* handler, uint8_t reg,
2                           uint8_t value)
3 {
4     uint8_t tx_buffer[] = {reg, value};
5     HAL_StatusTypeDef status = HAL_I2C_Master_Transmit(handler,
6               ACC_ALT_ADDRESS, tx_buffer, 2, 200);
7
8     if(status != HAL_OK)
9         return C_I2C_TRANSMIT;
10
11    return G_SUCCESS;
12 }
```

**Listing 20.** Zapis wartości do rejestru pamięci akcelerometru

Funkcją pozwalającą na zapis danych do rejestrów akcelerometru jest `Acc_Cmd`, a jej treść widoczna jest w listingu 20. Przyjmuje ona jako argumenty: wskaźnik struktury magistrali I<sup>2</sup>C biblioteki HAL, adres rejestru oraz wartość która powinna być do niego zapisana. Transmisja składa się z dwóch bajtów danych i jest realizowana przy pomocy funkcji `HAL_I2C_Master_Transmit`. Istotnym szczegółem jest fakt, że funkcja ta sama zarządza zapisywaniem poprawnego bitu kierunku danych [17] (będącym bitem o najmniejszym znaczeniu w adresie urządzenia), dlatego podawany jej adres peryferium powinien zostać przesunięty o jeden w lewo względem adresu, który można znaleźć w dokumentacji [8].

Odczyt danych z akcelerometru realizowany jest z użyciem funkcji `Acc_Read`, której treść przedstawia listing 21.

```

1 static ReturnCode Acc_Read(I2C_HandleTypeDef* handler, uint8_t* buffer,
2                            uint8_t reg, uint8_t bytes_number)
3 {
4     HAL_StatusTypeDef status = HAL_I2C_Mem_Read(handler, ACC_ALT_ADDRESS,
5           reg, 1, buffer, bytes_number, ACC_I2C_TIMEOUT);
6
7     if(status != HAL_OK)
8         return C_I2C_RECEIVE;
9
10    return G_SUCCESS;
11 }
```

**Listing 21.** Odczyt wartości z rejestru pamięci akcelerometru

Operacja odczytu jest analogiczna do operacji zapisu, tym razem jednak transmitowane są: pierwszy adres rejestru do odczytu oraz ilość rejestrów z których pobierane będą dane. Urządzenie umożliwia tzw. odczyt sekwencyjny (ang. *sequential read*), dzięki czemu wystarczy jedna transmisja zażądająca danych by pobrać zawartość N następujących sobie rejestrów pamięci. Tym razem za transmisję oraz odbiór nadchodzącej wiadomości odpowiada funkcja `HAL_I2C_Mem_Read`. Najbardziej enigmatycznym parametrem jest `MemAddSize` będący czwartym przyjmowanym argumentem. Informuje on o długości (w bajtach) pojedynczego rejestru pamięci urządzenia, z którego następuje odczyt danych [17] – w tym wypadku jest to 8 bitów zatem 1 bajt.

### 3.4.2 Konfiguracja akcelerometru

By zapewnić jak najszybszy odczyt danych z akcelerometru należy zmodyfikować jego tryb pracy. Pierwszą istotną zmianą jest zwiększenie częstotliwości próbkowania urządzenia – z domyślnej wartości 100 Hz na 3200 Hz. Następnie by upłynąć pobieranie próbek uruchomiono tryb strumienia (ang. *stream*) kolejki First In First Out (FIFO). Dzięki temu przetrzymywane są zawsze ostatnie 32 próbki [5], zamiast pojedynczego odczytu tak jak ma to miejsce w domyślnym trybie. By upewnić się, że każda odczytywana próbka jest poprawna wykorzystano opcję włączenia przerwań – konkretnie przerwania Data Ready. Jeśli akcelerometr gotowy jest na przesłanie nowej próbki, na wyjściu pinu `INT1` pojawi się stan wysoki.

```
1 ReturnCode Acc_Config(uint8_t line)
2 {
3     ReturnCode status = Acc_Cmd(acc_i2c, ACC_PWR_CTRL_REG, 0x00);
4     status = Acc_Cmd(acc_i2c, ACC_BW_RATE_REG, 0x0F);
5     status = Acc_Cmd(acc_i2c, ACC_FIFO_CTL_REG, 0x87);
6     status = Acc_Read(acc_i2c, &temp, 0x30, 1);
7     status = Acc_Cmd(acc_i2c, ACC_INT_ENB_REG, 0x80);
8     status = Acc_Cmd(acc_i2c, ACC_PWR_CTRL_REG, 0x08);
9     return G_SUCCESS;
10 }
```

**Listing 22.** Funkcja konfigurująca akcelerometr

Pełna konfiguracja jest widoczna na listingu 21. Na samym początku procedury rozpoczynającej konfigurację następuje uśpienie urządzenia poprzez zapis bajtu `0x00` do rejestru Power control. Następnie zmieniana jest częstotliwość próbkowania oraz zmieniany jest tryb pracy kolejki FIFO. Przed włączeniem wektora przerwań czyszczone są ich flagi poprzez odczytanie zawartości rejestru Interrupt source. Na końcu urządzenie jest wprowadzane w tryb pomiaru.

### 3.4.3 Pomiary i przetwarzanie danych

Pojedynczy odczyt z akcelerometru jest wykonywany przez wywołanie funkcji `Acc_RawMeasurment` widocznej na listingu 23.

```

1 ReturnCode Acc_RawMeasurment(int16_t* xyz_buffer, uint8_t line)
2 {
3     if(line > ACC1_LINE)
4         return C_ACC_HANDLE;
5
6     ReturnCode status = Acc_Read( (line == ACC0_LINE ? acc0.handle : acc1.
7         handle), data_buffer, ACC_X0_REG, 6);
8
9     if(status != G_SUCCESS)
10        return status;
11
12     *(xyz_buffer) = ((data_buffer[1] << 8) | data_buffer[0]);
13     *(xyz_buffer + 1) = ((data_buffer[3] << 8) | data_buffer[2]);
14     *(xyz_buffer + 2) = ((data_buffer[5] << 8) | data_buffer[4]);
15
16     return G_SUCCESS;
17 }
```

**Listing 23.** Funkcja realizująca pomiary

Ustawiona rozdzielcość pomiarów to 10-bitów zatem do przechowywania wyników potrzebna jest zmienna 16-bitowa. Zgodnie z dokumentacją należy odczytywać rejestyry pomiarowe wszystkich osi by zachować spójność danych, dlatego pobranie pomiarów składa się z pojedynczego odczytu sekwencyjnego.

Ze względu na czułość układu na m.in. drgania mechaniczne konieczne jest uśrednianie wyników. Za tę operację odpowiedzialna jest funkcja Acc\_RawMeasurment, której początek jest widoczny na listingu 24.

```

1 ReturnCode Acc_AvgMeasurment(int16_t *xyz_buffer, uint32_t samples,
2                               uint8_t line)
3 {
4     if(line > ACC1_LINE)
5         return C_ACC_HANDLE;
6
7     acc_config *acc = line == 0x00 ? &acc0 : &acc1;
8
9     int32_t avg_x = 0;
10    int32_t avg_y = 0;
11    int32_t avg_z = 0;
```

**Listing 24.** Początek funkcji Acc\_AvgMeasurment

Dość analogicznie do innych funkcji na podstawie linii magistrali przypisywany jest wskaźnik do obsługi interfejsu I<sup>2</sup>C Następnie zadeklarowane są pomocnicze zmienne, które będą odpowiedzialne za przechowywanie sum pomiarów.

```
1     uint32_t start_time;
```

```
2     for(uint16_t i = 1; i <= samples; i++)
3     {
4         start_time = HAL_GetTick();
5         while( HAL_GPIO_ReadPin(acc->port, acc->pin) != GPIO_PIN_SET
6     )
7         {
8             if (HAL_GetTick() - start_time > ACC_I2C_TIMEOUT)
9                 return G_ACC_READ;
10        }
11        ReturnCode status = Acc_RawMeasurment(read_buff, line);
12
13        if(status != G_SUCCESS)
14            return status;
15
16        avg_x += read_buff[0];
17        avg_y += read_buff[1];
18        avg_z += read_buff[2];
19    }
20
21    *(xyz_buffer) = avg_x / samples;
22    *(xyz_buffer + 1) = avg_y / samples;
23    *(xyz_buffer + 2) = avg_z / samples;
24
25    return G_SUCCESS;
}
```

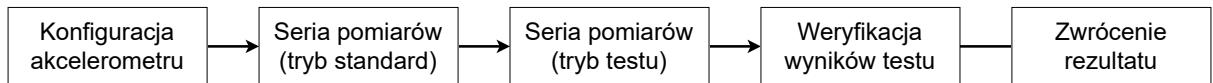
**Listing 25.** Fragment funkcji Acc\_AvgMeasurment odpowiadający za pomiary

Fragment funkcji odpowiedzialny za odczytywanie pomiarów oraz ich uśrednienie jest widoczny na listingu 25. Pętla `for` odtwarza się tyle razy ile zadanych jest próbek. Przed wysłaniem żądania danych sprawdzane jest czy wystąpiła flaga przerwania (Data ready), która objawia się stanem wysokim odczytywanym na pinie mikrokontrolera. By uniknąć nieskończonej pętli `while`, każda iteracja pętli `for` rozpoczyna się od sczytania cykli zegarowych. Jeśli ich ilość przekracza wartość `ACC_I2C_TIMEOUT` zwracany jest błąd. Po zakończeniu pętli `for` liczona jest średnia dla odczytów każdej osi oraz zapisywana pod adresem wskazywanym przez argument `xyz_buffer`.

#### 3.4.4 Test sprawności akcelerometru

Producent urządzenia uwzględnił procedurę umożliwiającą weryfikację jakości odczytów akcelerometru 13. Polega ona na porównaniu odczytów przed i po zadaniu dodatkowej siły elektrostatycznej, jeśli ich różnica mieści się w podanych w tabelach układ pracuje poprawnie. Podczas przeprowadzania tego testu akcelerometr nie może zmienić swojej pozycji.

Dodatkowo przed rozpoczęciem testu rozdzielcość oraz zakres pomiarów są ustawiane na odpowiednio  $\pm 16g$  oraz 13-bitów. Dokładny przebieg został zaimplementowany zgodnie z notą producenta [18].



Rysunek 13. Przebieg testu sprawności

### 3.4.5 Przetwarzanie odczytów

Dane otrzymywane z akcelerometru dostarczają informacji jedynie o przyśpieszeniach działających na każdą z osi. Zatem w nieprzetworzonej formie są one dość trudne w interpretacji – pozwalają one jedynie na bardzo szacunkowe określanie orientacji modułu. By uprościć kalkulacje pomiary powinny być wykonywane jedynie gdy robot jest statyczny, więc nie działają na niego przyśpieszenia inne niż grawitacyjne. Zgodnie z notą producenta [10] kąty nachylenia osi OX, OY oraz OZ względem kierunkiem wyznaczanym przez wektor grawitacji mogą zostać wyrażone przez następujące równania

$$\theta = \arctg \left( \frac{a_x}{\sqrt{a_y^2 + a_z^2}} \right) \quad (1)$$

$$\psi = \arctg \left( \frac{a_y}{\sqrt{a_x^2 + a_z^2}} \right) \quad (2)$$

$$\phi = \arctg \left( \frac{\sqrt{a_x^2 + a_y^2}}{a_z} \right) \quad (3)$$

Implementację funkcji pozwalającej na przeliczenie odczytów akcelerometru na kąty nachylenia osi można znaleźć w pliku trig.c, natomiast jej listing 26 przedstawiony jest poniżej.

```

1 void GetTiltAngles(float *abc_buffer, int16_t *xyz_acc)
2 {
3     float a_yz, a_xy, a_xz;
4     float x_2, y_2, z_2;
5
6     x_2 = (*(xyz_acc)) * (*(xyz_acc));
7     y_2 = (*(xyz_acc + 1)) * (*(xyz_acc + 1));
8     z_2 = (*(xyz_acc + 2)) * (*(xyz_acc + 2));
9
10    a_yz = sqrtf(y_2 + z_2);
11    a_xy = sqrtf(x_2 + y_2);
12    a_xz = sqrtf(x_2 + z_2);
13
14    *(abc_buffer) = atanf( ((*(xyz_acc)) / a_yz ) );
15    *(abc_buffer + 1) = atanf( (((xyz_acc + 1)) / a_xz ) );
16    *(abc_buffer + 2) = atanf( (a_xy / ((xyz_acc + 2))) );
17 }
  
```

Listing 26. Funkcja GetTiltAngles

Funkcja przyjmuje dwa wskaźniki: abc\_buffer, pod adresem którego zapisane będą obliczone kąty oraz xyz\_acc zawierający pomiary z akcelerometru. Na początku deklarowane są zmienne pomocnicze, a następnie w dwóch krokach wyliczane są kąty używając wcześniejszych wzorów 3. Do przeprowadzenia operacji arytmetycznych oraz pierwiastkowania wykorzystywano gotowe funkcje będące częścią biblioteki math.h.

### 3.5 Obsługa karty microSD

Obsługa karty microSD jest zrealizowana dzięki dodatkowej bibliotece *FATFS*. Pozwala ona na zapis oraz odczyt danych z nośnika zgodny z jego strukturą pamięci. Dodatkowo znacznie upraszcza obsługę bardziej zaawansowanych funkcji takich jak tworzenie folderów, plików czy partycji. Napisanie autorskiego kodu pozwalającego na osiągnięcie podobnych efektów wymagałoby bardzo dużego nakładu pracy, dlatego zdecydowano się na użycie gotowej aplikacji. Natynie biblioteka oferuje wsparcie dla komunikacji z kartą za pomocą protokołu SDIO. Został on wybrany również, dlatego że SPI będące dla niego alternatywą, nie jest wspierane przez wszystkie urządzenia.

Dodatkowym elementem kluczowym dla archiwizacji pracy robota jest wbudowany na płytce moduł RTC. Dzięki niemu informacje o pracy robota są powiązane z konkretną chwilą czasową, co znacznie ułatwia diagnozowanie problemów. Wówczas podczas serwisowania urządzenia będzie możliwe odtworzenie sekwencji mogących prowadzić do występowania błędów, ale również prowadzenie bardziej analitycznej diagnostyki dotyczącej, np.: częstotliwości pojawiania się konkretnych komunikatów.

By uniknąć sytuacji tworzenia się bardzo dużych plików trudnych do odtworzenia, dane grupowane są w oddzielne pliki .txt nazywane według daty. Format każdego z nich ma postać log\_DD.MM.RR.txt. Każda zapisana informacja również ma standardowy format utrzymany w całości kodu: DD.MM.RR GG:MM:SS [Peryferium] [Operacja]: [Dane]. W przypadku komunikatów błędów pole operacji nie jest uzupełniane. Dane będące w formacie zmienoprzecinkowym zapisywane są z dokładnością dwóch miejsc po przecinku.

By zapewnić poprawny zapis danych do pamięci stworzono niewielką maszynę stanów określona typem enumeracyjnym SD\_STATE. Jej wszystkie stany zawarte są w tabeli 6.

#### 3.5.1 Zapis danych na kartę

By móc zacząć zapis (czyli wprowadzić kartę w stan SD\_READY) należy wywołać funkcję InitLogging. Jej listing 27 jest widoczny poniżej,

 `ReturnCode InitLogging(RTC_HandleTypeDef *handler);`

**Listing 27.** Definicja funkcji InitLogging

Jedynym argumentem przekazywanym funkcji jest wskaźnik na strukturę zegara RTC, jest on zapisywany do tożsamej struktury zdefiniowanej lokalnie w pliku .c. Dzięki temu po inicjalizacji nie trzeba przekazywać go przy każdej operacji zapisu.

Wartość [hex]	Nazwa w kodzie	Nazwa użytkowa	Opis
0x00	SD_NOT_INIT	Niezainicjalizowanie	Urządzenie albo nigdy nie zostało poddane procesowi incjalizacji albo zakończyła się ona niepowodzeniem.
0x01	SD_READY	Gotowość	Mögliwe jest zapisywanie i odczytywanie danych
0x02	SD_UPDATE	Aktualizajca	Nastąpiła zmiana daty przechowywanej przez RTC, wymagana jest aktualizacja nazwy pliku przed kolejną operacją zapisu

**Tabela 6.** Stany maszyny stanów karty SD

```

1 if (f_mount(&fatfs, path, 0) != FR_OK)
2     return C_MOUNT_ERROR;
3
4 RTC_TimeTypeDef time;
5 RTC_DateTypeDef date;
6 HAL_StatusTypeDef HAL_status = HAL_RTC_GetTime(&hrtc, &time,
7 RTC_FORMAT_BIN);
8 if( HAL_status != HAL_OK)
9     return C_RTC_ERROR;
10
11 HAL_status = HAL_RTC_GetDate(&hrtc, &date, RTC_FORMAT_BIN);
12 if( HAL_status != HAL_OK)
13     return C_RTC_ERROR;

```

**Listing 28.** Odczytywanie czasu z modułu RTC

Najpierw karta microSD jest przygotowywana do dalszej pracy, co widoczne jest w listingu 27. W pierwszej kolejności dysk jest "montowany", odczytywane są jego parametry pracy takie jak wykryty system plików, rozmiar sektorów, nazwa czy numer identyfikacyjny. Następnie z zegara RTC odczytywane są data oraz czas, choć informacja o godzinie nie jest w tym przypadku potrzebna powinna zostać odczytana by zachować spójność danych. Wynika to z logiki układu RTC, informację o tym można znaleźć w nocy aplikacyjnej bibliotece HAL [17].

```

1 sprintf(file_name, sizeof(file_name), "log_%02d_%02d_%02d.txt",
2 date.Date, date.Month, date.Year);
3
4 FRESULT res = f_open(&file, file_name, FA_OPEN_EXISTING);
5 if ( res != FR_OK )
6 {
7     res = f_open(&file, file_name, FA_CREATE_NEW );
8     if ( res != FR_OK )
9         return G_FILE_READ;
}

```

```
10     f_close(&file);  
11  
12     state = SD_READY;  
13     return G_SUCCESS;
```

**Listing 29.** Utworzenie plików do archiwizacji danych

Końcowym etapem inicjalizacji jest utworzenie pliku 29. Na podstawie daty formatowana jest nazwa pliku przy użyciu funkcji `snprintf`. Warto dodać, że na ogół użycie funkcji `snprintf` jest korzystniejsze niż `sprintf` ze względu na ograniczenie ilości zapisywanych bajtów do buforu. W przypadku użycia funkcji tak jak w listingu, nie ma możliwości zapisu sformatowanych danych do komórek pamięci poza zdefiniowaną wielkością tablicy. Na końcu jest sprawdzane czy plik istnieje, jeśli nie podjęta jest próba jego utworzenia. Całość procedury kończy się zamknięciem pliku i zwróceniem kodu sukcesu.

Archiwizacja danych odbywa się wykorzystując funkcje `SD_LogStatus` oraz `SD_LogMsg`. Pierwsza z nich jako argument przyjmuje kod typu `ReturnCode` i pozwala na zapis powtarzalnych komunikatów błędów opatrzonych znacznikiem czasu. Druga funkcja natomiast pozwala na zapis ciągu znaków o dowolnej treści i długości.

### 3.5.2 Alarm RTC

Ponieważ dane archiwizowane są dziennie należy obsłużyć sytuację, w której zachodzi samoistna zmiana daty. Rozwiązaniem tego problemu jest włączenie przerwań od modułu RTC. Możliwe jest skonfigurowanie kilku budzików (ang. *alarms*) działających analogicznie do zwykłych przerwań od zegarów. W tej aplikacji wystarczy pojedynczy budzik uruchamiający się, gdy w rejestrach przechowujących czas wystąpi godzina 23:59:59.

Wówczas maszyna stanów karty SD wprowadzona jest w stan aktualizacji. Oznacza to, że przed wykonaniem kolejnej operacji zapisu zostanie utworzony nowy plik. Funkcja wywoływana przez przerwanie to `HAL_RTC_AlarmAEventCallback`

```
1 void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *handle)  
2 {  
3     SD_AlarmRoutine();  
4 }
```

**Listing 30.** Przerwanie od alarmu RTC

Listing 30 przedstawia funkcję obsługi przerwania znajdującą się w pliku `main.c`. W kodzie głównego programu obsługa przerwania sprowadza się do wywołania wcześniej przygotowanej funkcji `SD_AlarmRoutine`, która w obecnym stadium projektu ogranicza swoje działanie do zmiany stanu karty SD.

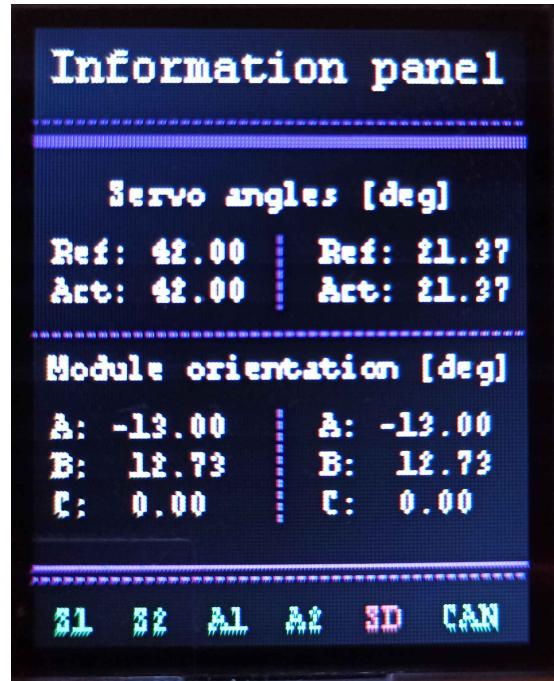
### 3.6 Obsługa wyświetlacza LCD

W projekcie wykorzystano wyświetlacz LCD o przekątnej 1,8 cala firmy Waveshare. Na pokładzie płytki jest sterownik ST7735S, który zarządza pracą wyświetlacza. Komunikacja z urządzeniem odbywa się z użyciem protokołu SPI, przy czym należy zaznaczyć, że linia Controller In Peripheral Out (CIPO) nie jest używana. Tak jak wcześniej wspomniano obsługa wyświetlacza została oparta o już istniejącą bibliotekę udostępnianą przez producenta. Dzięki temu oprogramowanie urządzenia jest możliwe bez zagłębiania się w jego dokumentację techniczną.

Wyświetlacz jest matrycją o wymiarach 160x128 pikseli, które przyjmują barwy zgodne z formatem RGB 16-bit (RGB565). Biblioteka pozwala na rysowanie podstawowych kształtów takich jak okręgi, koła, prostokąty, linie (ciągłe oraz przerwane). Dodatkowo dostępne są gotowe funkcje umożliwiające zapis tekstu w fontach o różnych rozmiarach. Pozwoliło to w pełni na zaprojektowanie dwóch prostych interfejsów graficznych: ekranu inicjalizacji oraz panelu informacyjnego.



(a) Wygląd ekranu inicjalizacji



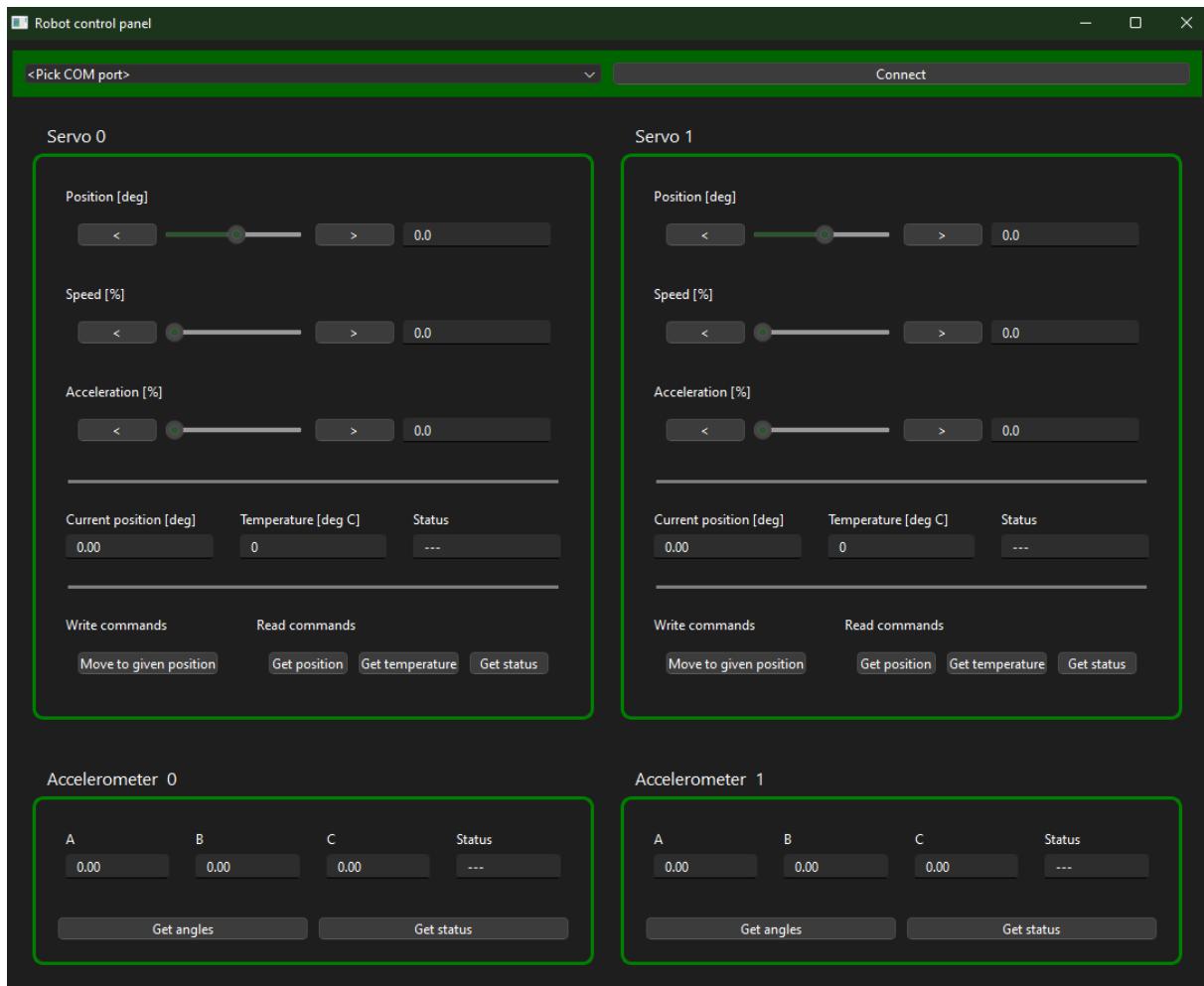
(b) Wygląd panelu informacyjnego

Ekran inicjalizacji 14a składa się z ozdobników w postaci nagłówka oraz wizerunku imiennika robota – gremlina *Gizmo* znanego z filmu *Gremliny rozbijają*. Ilustracja została stworzona na podstawie klatki filmowej, która została rozbita na grupy pikseli za pomocą strony *Pixel it*. Po drobnej obróbce graficznej stworzono tablice zawierające współrzędne każdego z pokolorowanych pikseli, dokonano tego za pomocą skryptu znajdującego się pod ścieżką *extras/LUT\_pixels.py*. Oprócz tego z informacji na wyświetlaczu można odczytać ostatnie zainicjalizowane peryferium (kolor zielony), obecnie inicjalizowane urządzenie (środkowy tekst o powiększonej czcionce) oraz następny podzespoł, który ulegnie uruchomieniu (dolny tekst).

Panel informacyjny 14b składa się z trzech segmentów grupujących dane związane kolejno z: pracą serwomechanizmu, kątami definiującymi orientację modułu oraz statusami podzespołów. Każda operacja zapisu/odczytu związana z danym urządzeniem skutkuje aktualizacją wyświetlanych parametrów.

### 3.7 Testy oprogramowania

Głównym narzędziem użyтыm do przeprowadzenia testów była aplikacja panelu sterowniczego 15 napisana w języku *Python* z wykorzystaniem bibliotek *Serial* oraz *PySide6*.



Rysunek 15. Interfejs graficzny panelu sterowniczego

Dzięki programowi udało się zweryfikować poprawność prac obu magistrów. Dokonano tego wysyłając rozkazy skierowane zarówno do modułu nadzorującego jak i podporządkowanego oraz porównując dane otrzymywane przez panel z tymi widocznymi z poziomu mikrokontrolera. Dodatkowo zestawiono ze sobą również dane zapisywane na kartę microSD oraz wyświetlane na wyświetlaczu LCD. Układ przeszedł poprawnie wszystkie testy.

### 3.7.1 Porównanie pomiarów kąta obrotu serwomechanizmu

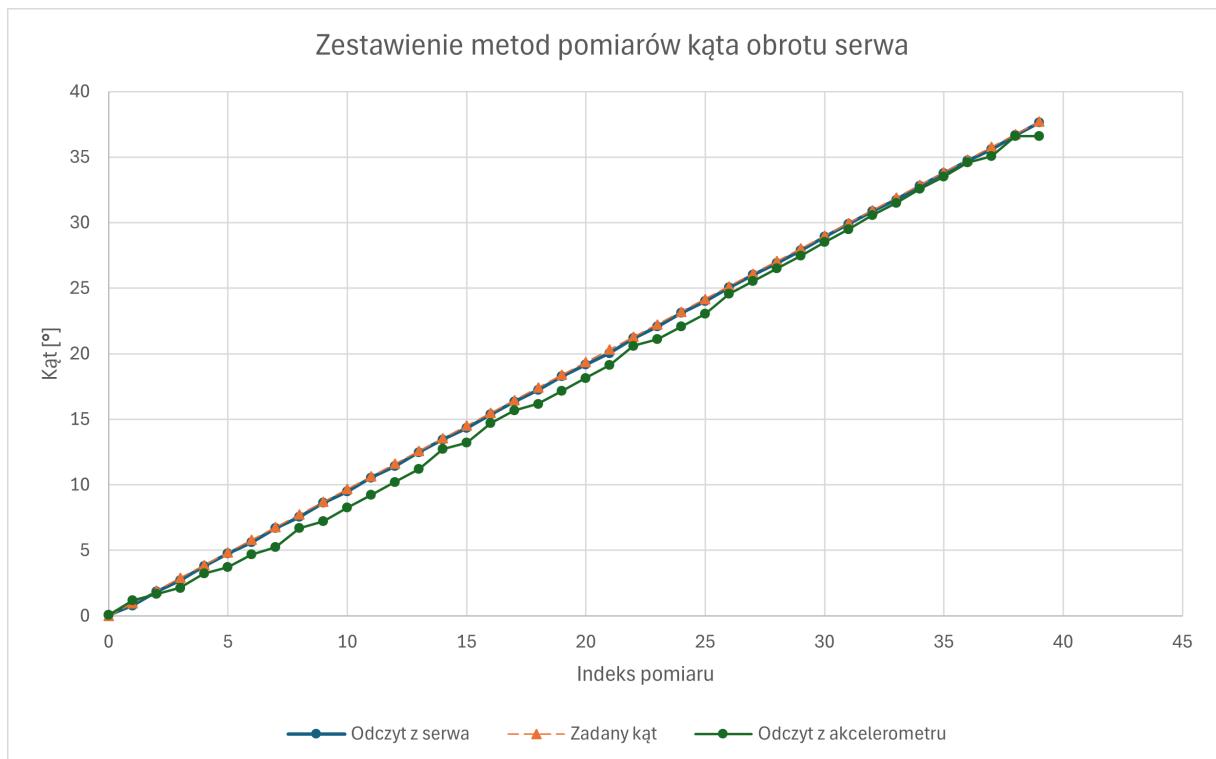
Tak jak wspomniano wcześniej pomiar kąta obrotu serwomechanizmu jest dokonywany na dwa sposoby. Pierwszy z nich jest niejawny, ponieważ ma miejsce wewnętrz serwomechanizmu – dostępny jest jedynie końcowy efekt zapisany w pamięci SRAM urządzenia. Wymaga on drobnego przekształcenia, ponieważ odczytywany rezultat jest wyrażany w ilości zliczeń wbudowanego enkodera. Na podstawie obserwacji pracy układu ustalono, że na pełen obrót serwomechanizmu przypada 4095 zliczeń. Zatem:

$$\alpha = \frac{n \cdot 360}{4095} [{}^{\circ}] \quad (4)$$

Gdzie  $\alpha$  to kąt obrotu serwa, a  $n$  to ilość zliczeń.

Drugi z nich polega na przetworzeniu danych z akcelerometru. Zgodnie z wzorami przytoczonymi [3].

By porównać ze sobą wyniki obu metod przeprowadzono prosty test. Na orczyku serwomechanizmu przyklejono akcelerometr, orientując jego osie tak by były równoległego do krawędzi elementu. Dzięki temu, jedna z osi pozostanie "nieruchoma" – odczyty akcelerometru powinny być dla niej stałe. Następnie wykonano trzy serie pomiarowe, w trakcie których serwomechanizmowi zadawane były kąty w zakresie  $0\text{--}39^{\circ}$  z krokiem jednego stopnia.



Rysunek 16. Przebieg jednej z serii pomiarowych

Wykres 16 przedstawia jedną ze zmierzonych serii pomiarowych. Uzyskanie wspólnego kąta zerowego dla obu układów zostało osiągnięte przez 'przesunięcie' całej serii pomiarowej akcelerometru

o kąt odnotowany dla pomiaru o indeksie zerowym. Wyniki wszystkich serii uśredniono uzyskując następujące dane:

Średni uchyb serwa [°]	Maksymalny uchyb serwa [°]	Średnia różnica pomiarów [°]	Maksymalna różnica pomiarów [°]
-0,14	0,26	1,47	2,31

**Tabela 7.** Zestawienie podstawowych danych statystycznych

Na podstawie tabeli 7 widoczne jest, że wysterowanie serwa daje dość satysfakcjonujące rezultaty dając uchyby rzędu  $0,1\text{--}0,2^\circ$ . Niestety jest to uchyb, który trudno wyeliminować ze względu na jego niejawnny sposób pracy oraz brak możliwości "ręcznego" wysterowania urządzenia. Różnica między kątem odczytywanym przez akcelerometr a serwem wynosiła średnio  $1,47^\circ$ . Jednak w ogólności widoczne jest, że pomiary są ze sobą zgodne. Prawdopodobną przyczyną takich różnic jest to jak zamocowano akcelerometr. Zarówno kable łączące urządzenie z mikrokontrolerem jak i niedostateczny sztywny montaż na orczyku sprawiły, że pomiary były zaburzone. Test należałoby powtórzyć, gdy elementy będą zamontowane w robocie. Wówczas akcelerometr będzie pracował we właściwych warunkach, dzięki czemu wyniki będą bardziej miarodajne

## Rozdział 4

### Podsumowanie

Celem pracy było opracowanie oprogramowania przeznaczonego dla robota modułarnego. W pierwszym kroku dokonano analizy literatury, dzięki czemu wyłoniono najważniejsze cechy i wymagania względem urządzenia. Dobór elementów elektronicznych powinien gwarantować pewien stopień autonomii każdego z segmentów – umożliwić robotowi ruch oraz zapewnić jego zdolność do wyznaczania swojej pozycji. Należało również znaleźć sposób na skomunikowanie modułów ze sobą oraz światem zewnętrznym. Ogólnym założeniem jest fakt, że oprogramowanie tak samo jak sam robot powinno być jak najbardziej uniwersalne. Ostatnim wnioskiem wyciągniętym na temat już istniejących konstrukcji jest potrzeba wprowadzenia narzędzi umożliwiających diagnostykę – ze względu na rozproszenie informacji w układzie konieczne jest archiwizowanie jego stanów, tak by dało je się później odtworzyć.

Przeanalizowano już gotową konstrukcję robota Gizmo i dobrano pod nią komponenty elektroniczne – serwomechanizmy ST3020, akcelerometry ADXL345, nadajniki-odbiorniki MCP2551, wyświetlacz LCD oraz kartę microSD. Elementy pozwalają na: sterowanie robotem, ustalenie kątów obrotu każdego z serwomechanizmów, bieżący odczyt parametrów pracy a także prowadzenie kompleksowego dziennika aktywności. Na podstawie porównania rodzin Arduino, ESP32 oraz STM32 dokonano doboru płytki deweloperskiej stanowiącej serce układu. Mikrokontroler Nucleo-446RE spełnia wymagania odnośnie ilości pinów, pamięci ROM oraz RAM, a także potrzebnych interfejsów komunikacyjnych.

Zdecydowano się na zaimplementowanie dwóch magistrali komunikacyjnych. Pierwsza z nich oparta na protokole UART służyła do wymiany danych między panelem sterowniczym a modułem nadrzędnym, co pozwoliło na wygodne zarządzanie pracą urządzenia oraz monitorowanie jego statusu. Druga magistrala była wykorzystywana jedynie do wewnętrznej komunikacji między modułami i została oparta na protokole CAN. Decyzja o tej metodzie komunikacji była oparta o trzy główne czynniki: wymaganie jedynie dwóch przewodów do zrealizowania połączenia, odporność na zakłócenia elektromagnetyczne, możliwość nadawania wiadomości przez każde urządzenie w sieci.

Oprogramowanie zostało napisane w języku C z użyciem *STM32CubeIDE*. Zastosowano wersjonowanie dzięki narzędziu *git*. Całość kodu znajduje się na publicznym repozytorium oraz jest udostępniona na licencji open-source (GPLv3).



**Rysunek 17.** Fizyczny model robota

Ze względu na niejawnosć oprogramowania robotów wspomnianych we wstępie, ciężko jest porównać metodykę pisania kodu. Widoczne jest, że zastosowanie dwóch magistral – wewnętrznej i zewnętrznej – nie jest nowym podejściem i wydaje się być dobrą drogą. Sam dobór elementów elektronicznych, w tym mikrokontrolera, został dokonany kierując się podobnymi motywacjami – dostosowując peryferia do warunków pracy robota. Elementem wspólnym jest również używanie akcelerometrów oraz enkoderów do wyznaczania pozycji urządzenia. Największą różnicą jest brak komunikacji bezprzewodowej, jednak jak wspomniano wcześniej jest ona możliwa do dodania w kolejnych iteracjach projektu.

Poprawność zaimplementowanych metod komunikacji zweryfikowano z użyciem aplikacji panelu sterowniczego. Natomiast jakość wysterowania serwomechanizmu oraz pomiarów jego kąta wyznaczono drogą eksperymentalną. Testy potwierdzają poprawność kodu, zatem wszystkie założenia pracy zostały zrealizowane zgodnie z oczekiwaniemi. Oprogramowanie mogłoby być, rozszerzone o szereg funkcjonalności, np.: związanych z wykrywaniem kolizji czy optymalizacji zużycia energii przez mikrokontroler oraz jego peryferia. Najbardziej korzystną zmianą, która powinna być wprowadzona

w kolejnych wersjach to wykorzystanie testów jednostkowych (ang. *Test Driven Development*) – znacznie poprawiłoby to jakość oraz niezawodność kodu.

## **4.1 Przyszłość projektu**

W ramach kolejnych etapów projektu należałyby stworzyć dedykowaną płytę PCB oraz przeprowadzić testy oprogramowania przy prawdziwej pracy robota składającego się z kilku modułów.

Dodatkowo należałyby rozpocząć pracę nad bardziej złożonym algorytmem sterowania robota, w połączeniu z interfejsem umożliwiającym zadawanie konkretnych pozycji układu narzędziowego robota.



# Bibliografia

- [1] Ahmadzadeh, H. i Masehian, E., „Modular robotic systems: Methods and algorithms for abstraction, planning, control, and synchronization,” *Artificial Intelligence*, t. 223, s. 27–64, 2015, ISSN: 0004-3702. DOI: <https://doi.org/10.1016/j.artint.2015.02.004>. adr.: <https://www.sciencedirect.com/science/article/pii/S0004370215000260>.
- [2] ARM, *Cortex-M4: Reference Manual*.
- [3] „CAN specification Version 2.0,” Robert Bosch GmbH, spraw. tech., 1991.
- [4] Chen, Y.-T., Chen, C.-L. i Hung, S.-K., „Orimo: Leg-Wheel Transformable Origami Modular Robots,” w 2024 IEEE International Conference on Advanced Intelligent Mechatronics (AIM), 2024, s. 892–897. DOI: [10.1109/AIM55361.2024.10637012](https://doi.org/10.1109/AIM55361.2024.10637012).
- [5] Christopher J. Fisher, J. L., *AN-1025: Utilization of the First In, First Out (FIFO) Buffer in Analog Devices, Inc. Digital Accelerometers*.
- [6] „CiA Draft Standard 102 Version 2.0,” CiA – CAN in Automaton, spraw. tech., 1994.
- [7] Corrigan, S., „SLOA101B: Introduction to the Controller Area Network (CAN),” TI – Texas Instruments, spraw. tech., 2016.
- [8] Devices, A., *ADXL345: datasheet*.
- [9] Fang, Z., Fu, Y. i Chai, T., „A low-cost modular robot for research and education of control systems, mechatronics and robotics,” w 2009 4th IEEE Conference on Industrial Electronics and Applications, 2009, s. 2828–2833. DOI: [10.1109/ICIEA.2009.5138725](https://doi.org/10.1109/ICIEA.2009.5138725).
- [10] Fisher, C. J., *AN-1057: Using an Accelerometer for Inclination Sensing*.
- [11] Guilin Yang, I.-M. C., *Modular Robots: Theory and practice*, Han Ding, R. S., red. Springer, 2022.
- [12] Kurokawa, H., Kamimura, A., Yoshida, E., Tomita, K., Kokaji, S. i Murata, S., „M-TRAN II: metamorphosis from a four-legged walker to a caterpillar,” w Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453), t. 3, 2003, 2454–2459 vol.3. DOI: [10.1109/IROS.2003.1249238](https://doi.org/10.1109/IROS.2003.1249238).
- [13] Lawrenz, W., *CAN System Engineering: From Theory to Practical Applications*, Lawrenz, W., red. Springer, 2013.
- [14] Lordos, G. i in., „WORMS: Field-Reconfigurable Robots for Extreme Lunar Terrain,” w 2023 IEEE Aerospace Conference, 2023, s. 1–21. DOI: [10.1109/AER055745.2023.10115833](https://doi.org/10.1109/AER055745.2023.10115833).

## Bibliografia

---

- [15] Olszewski, K., *Projekt mechaniczny robota modułarnego Gizmo*, 2025.
- [16] STMicroelectronics, *DS1093: STM32F446xC/E datasheet*.
- [17] STMicroelectronics, *UM1725: Description of STM32F4 HAL and low-layer drivers*.
- [18] Tusuzki, T., *AN-1077: ADXL345 Quick Start Guide*.
- [19] Vu, L. A. T., Bi, Z., Mueller, D. i Younis, N., „Modular Self-Configurable Robots—The State of the Art,” *Actuators*, t. 12, nr. 9, 2023, ISSN: 2076-0825. DOI: 10.3390/act12090361. adr.: <https://www.mdpi.com/2076-0825/12/9/361>.
- [20] Xuan, L., Minglu, Z. i Wei, L., „Design method to modular robot system,” w *2009 ASME/IFTOMM International Conference on Reconfigurable Mechanisms and Robots*, 2009, s. 521–528.
- [21] Yoshida, E., Murata, S., Kamimura, A., Tomita, K., Kurokawa, H. i Kokaji, S., „Self-reconfigurable modular robots – hardware and software development in AIST,” w *IEEE International Conference on Robotics, Intelligent Systems and Signal Processing, 2003. Proceedings*. 2003, t. 1, 2003, 339–346 vol.1. DOI: 10.1109/RISSP.2003.1285597.

# Spis rysunków

1	Robot M-TRAN II [12] . . . . .	10
2	Robot Orimo [4] . . . . .	11
3	Robot NEURobot [9] . . . . .	11
4	Robot WORMS [14] . . . . .	12
6	Interpretacja stanów magistrali CAN [7] . . . . .	17
7	Model serwomechanizmu ST3020 udostępniony przez producenta . . . . .	20
8	Płytki STM32 Nucleo-F446RE, zdjęcie ze strony producenta . . . . .	21
9	Kolejne etapy przebiegu programu głównego . . . . .	23
10	Schemat importowania plików nagłówkowych . . . . .	24
11	Legalne przejścia między stanami . . . . .	26
12	Generalny schemat wymiany wiadomości . . . . .	29
13	Przebieg testu sprawności . . . . .	45
15	Interfejs graficzny panelu sterowniczego . . . . .	50
16	Przebieg jednej z serii pomiarowych . . . . .	51
17	Fizyczny model robota . . . . .	54



# **Spis tabel**

1	Zestawienie wybranych cech robotów modularnych . . . . .	12
2	Spis możliwych stanów urządzeń . . . . .	25
3	Pierwsze dziesięć kodów błędów . . . . .	27
4	Spis dostępnych komend . . . . .	28
5	Mapa pamięci serwa . . . . .	31
5	Mapa pamięci serwa . . . . .	32
6	Stany maszyny stanów karty SD . . . . .	47
7	Zestawienie podstawowych danych statystycznych . . . . .	52