



Zajęcia Projektowe Podstawy Robotyki

*Platforma jeżdżąca automatycznie utrzymująca odległość od
ściany*

Autorzy:

Jakub Pająk
Łukasz Grabarski
Krzysztof Grądek
Piotr Legień
Bartosz Wuwer

AiR Grupa 5TI

Spis treści

1.	Wprowadzenie	2
1.1.	Cel projektu	2
1.2.	Założenia wstępne	2
2.	Realizacja projektu	2
2.1.	Panel sterowania w aplikacji	2
2.2.	Interfejs użytkownika aplikacji	3
2.3.	Implementacja połączenia Bluetooth	6
2.4.	Implementacja przesyłania danych	7
2.5.	Implementacja prostego skryptu w odczytującego dane z kanału BLE	8
2.6.	Proces wykonywania projektu obudowy robota	10
2.7.	Implementacja prostego kodu weryfikującego prawidłowe połączenie silników	14
2.8.	Podłączenie oraz weryfikacja poprawności działania czujników laserowych	15
2.9.	Montaż silników wewnątrz dolnej komory obudowy	16
2.10.	Montaż ogniw wraz z układem BMS oraz podłączenie czujników	16
2.11.	Integracja czujników z silnikami oraz implementacja logiki sterowania	17
2.11.1.	Master	17
2.11.2.	Slave	21
2.12.	Testy poprawności działania algorytmu sterującego	22
3.	Napotkane problemy	22
3.1.	Podłączenie laserowych czujników odległości	22
3.2.	Montaż silników oraz kół	22
3.3.	Implementacja logiki sterowania	23
4.	Podsumowanie	23

1. Wprowadzenie

1.1. Cel projektu

Celem niniejszego projektu jest opracowanie mobilnej platformy robotycznej, zdolnej do automatycznego utrzymywania określonej odległości od ściany. Platforma ta bazować będzie na mikrokontrolerze Arduino oraz laserowych czujnikach odległości typu ToF (ang. Time of Flight). Projekt ma na celu zbadanie i rozwinięcie zaawansowanych algorytmów sterowania i nawigacji, które umożliwią precyzyjne śledzenie ścian w zmiennych warunkach środowiskowych. Dodatkowym celem jest stworzenie wszechstronnego rozwiązania, które można łatwo dostosować do różnych zastosowań, takich jak roboty sprzątające, inspekcyjne czy systemy autonomiczne w logistyce.

1.2. Założenia wstępne

Robot został zaprojektowany w oparciu o mikrokontroler Arduino Uno R4, wybrany ze względu na wbudowany moduł Bluetooth LE (LE - Low Energy), co umożliwiło zdalne monitorowanie i kontrolę. Silniki zastosowane w projekcie są wyposażone w enkodery, co zapewni precyzyjne sterowanie. Czujniki odległości zostały podłączone do mikrokontrolera za pomocą magistrali I2C. Ze względu na planowaną liczbę czujników (osiem), połączenie ich innymi metodami nie byłoby możliwe.

Zasilanie platformy jest wystarczające do obsługi co najmniej jednego mikrokontrolera oraz czterech silników. Optymalnym rozwiązaniem jest zastosowanie czterech ogniw litowo-jonowych połączonych w konfiguracji 2S2P, co zapewni odpowiednią wydajność energetyczną. Dodatkowo, konieczne jest zastosowanie układu zarządzania baterią BMS (ang. Battery Management System), który zabezpieczy ogniwa przed nadmiernym rozładowaniem i przeładowaniem.

Całość konstrukcji została zamknięta w obudowie wykonanej techniką druku 3D. Obudowa jest podzielona na dwie główne sekcje: dolną, w której zostaną umieszczone silniki oraz ogniwa wraz z układem BMS, oraz górną, zawierającą płytę stykową, mikrokontroler oraz czujniki. Taka konstrukcja zapewnia łatwy dostęp do kluczowych komponentów i umożliwia ich sprawną wymianę w razie potrzeby.

2. Realizacja projektu

Projekt był realizowany etapami, jednak nie został zastosowany szczegółowy harmonogram. Przybliżone etapy rozwoju projektu:

1. Napisanie aplikacji w wersji dedykowanej systemowi Android w języku Dart,
2. Realizacja prostego skryptu w Arduino IDE w celu weryfikacji połączenia BLE z mikrokontrolerem,
3. Wykonanie projektu obudowy robota w 3D oraz przygotowanie do druku,
4. Wykonanie prostego kodu w celu weryfikacji poprawności połączenia silników do sterownika,
5. Podłączenie oraz weryfikacja poprawności działania laserowych czujników odległości,
6. Montaż silników wewnętrz dolnej komory obudowy,
7. Montaż ogniw wraz z układem BMS,
8. Integracja czujników z silnikami oraz implementacja logiki sterowania,
9. Testy poprawności działania algorytmu sterującego.

2.1. Panel sterowania w aplikacji

Poczas analizy możliwych rozwiązań problemu zdalnego sterowania robotem wybór padł na wykonanie aplikacji mobilnej oraz przesyłanie odpowiednich komend za pomocą protokołu BLE. Protokół BLE jest pewnym szczególnym przypadkiem ogólnego protokołu Bluetooth, jest on szczególnie często spotykany w przypadku mniej zaawansowanych mikrokontrolerów takich jak Arduino Uno R4. Dzięki odpowiedniej architekturze, protokół ten pozwala na bardziej efektywne zarządzanie pobieraną energią jednocześnie zachowując niezbędne funkcjonalności.

Przez wzgląd na wcześniej nabycie umiejętności tworzenia aplikacji za pomocą języka Dart przez jednego z członków sekcji, prace nad projektem rozpoczęto od implementacji podstawowej wersji aplikacji realizującej proste przesyłanie informacji w postaci całkowitoliczbowej w celu późniejszej interpretacji otrzymanych danych w środowisku Arduino.

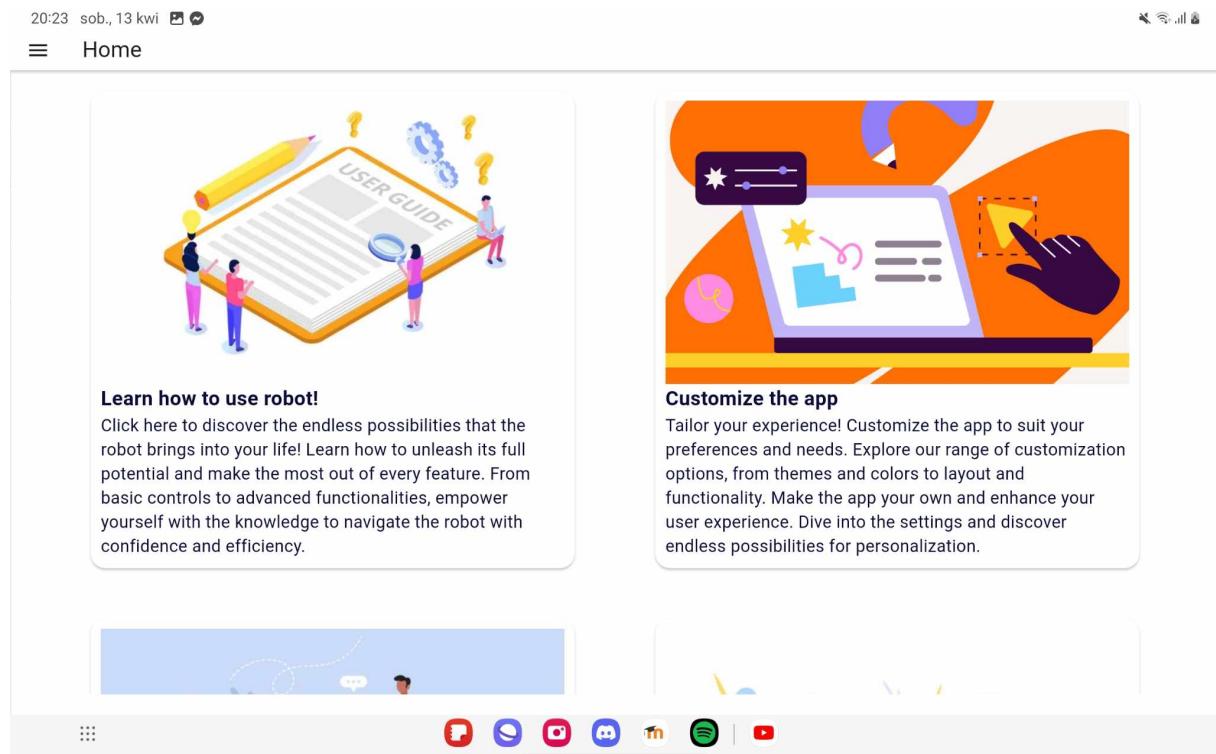
Przez wzgląd na tematykę projektu implementacja aplikacji nie zostanie w poniższym raporcie szczegółowo omówiona. Zostanie przedstawiony podstawowy interfejs graficzny oraz zasada działania kluczowych funkcji, takich jak połączenie lub realizacja przesyłu danych. Autor uważa, iż pewne wyszczególnienie części metod oraz zastosowanych bibliotek może pomóc niektórym osobom w prostrzym znalezieniu informacji na temat poprawnie działającego połączenia z mikrokontrolerem Arduino poprzez Bluetooth.

2.2. Interfejs użytkownika aplikacji

Interfejs użytkownika został zaprojektowany w celu umożliwienia podstawowej kontroli nad robotem. Z tego względu, nie poświęcono mu znacznej ilości czasu na poprawę estetyki.

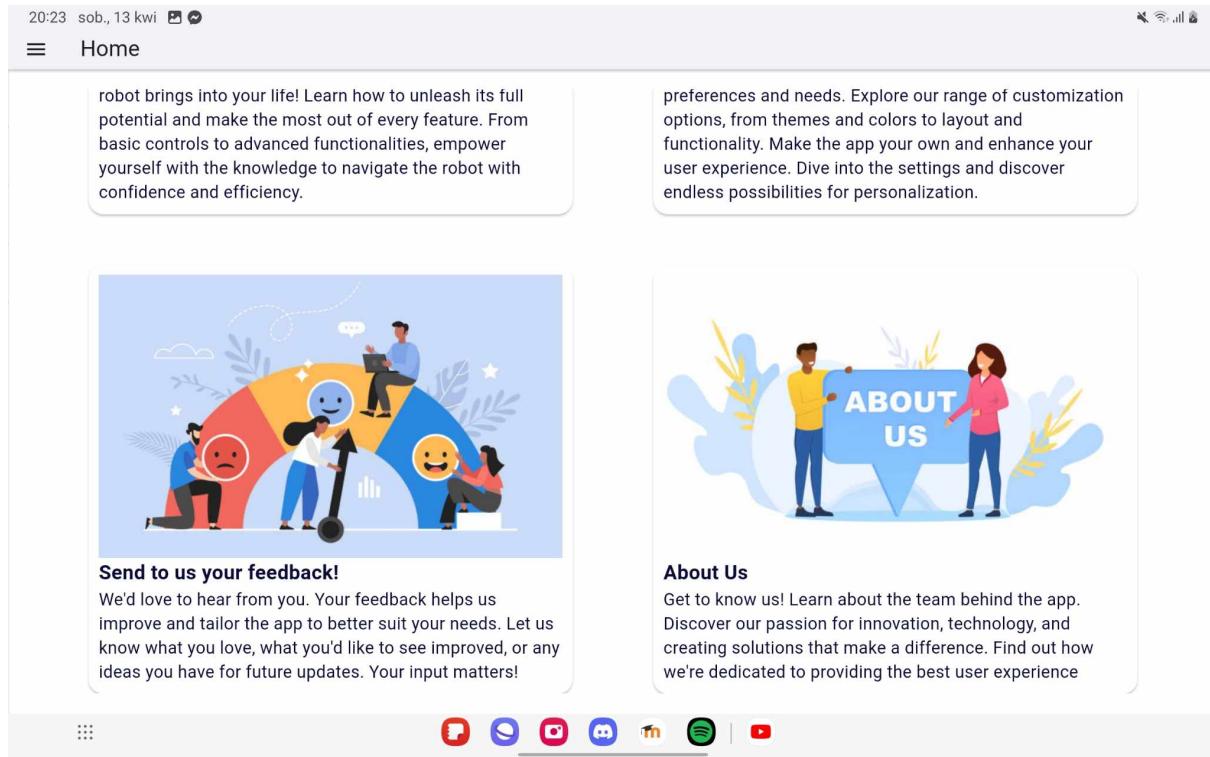
Po uruchomieniu aplikacji, użytkownik trafia na ekran początkowy (ang. Home), przedstawiony na Rysunku 1. Ekran ten zawiera cztery wyróżnione obszary, każdy z których pełni określona funkcję. Pierwsza sekcja ma na celu wprowadzenie nowego użytkownika w sposób bezpiecznego i poprawnego korzystania oraz sterowania robotem.

Druga sekcja prowadzi do ustawień wyglądu aplikacji, gdzie użytkownik może dostosować kolorystykę i inne elementy graficzne interfejsu.



Rysunek 1: Ekran początkowy aplikacji

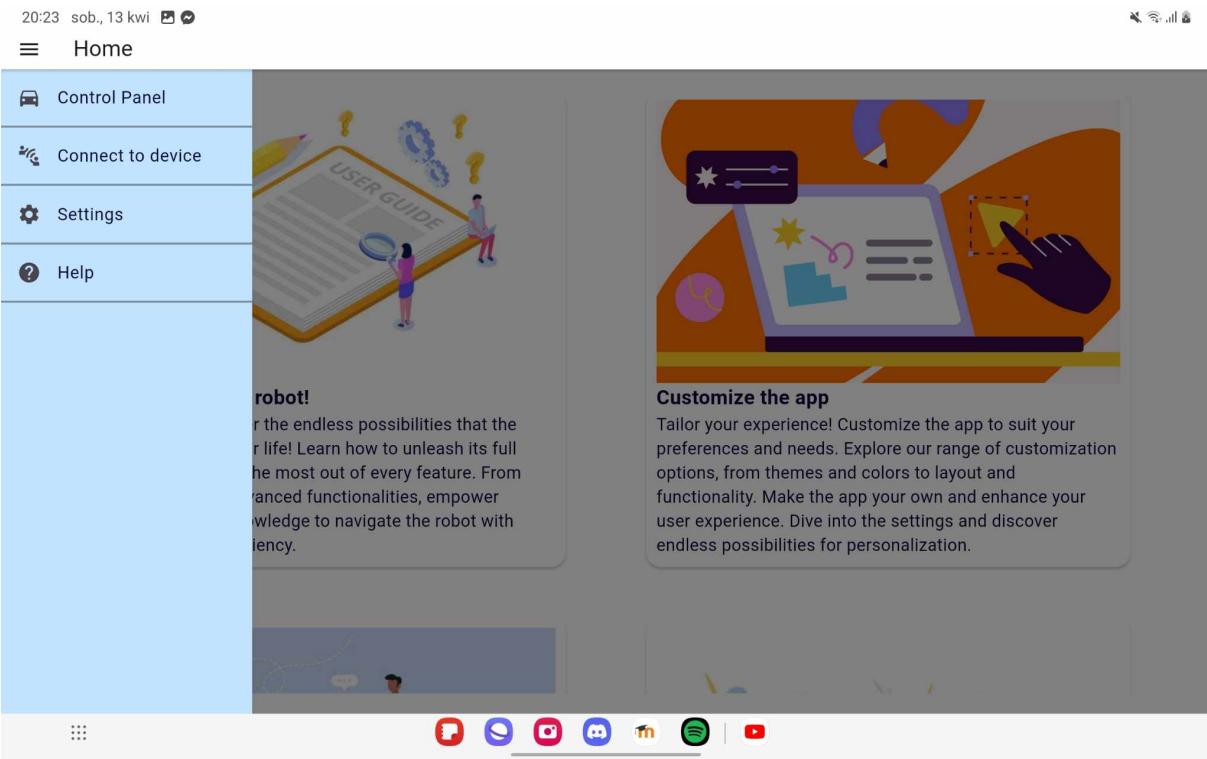
Trzecia i czwarta sekcja dotyczą genezy projektu oraz zespołu, który stworzył robota i aplikację. Ponadto, umożliwiają one użytkownikom przesyłanie opinii, co pozwala na wprowadzenie przyszłych aktualizacji mających na celu poprawę jakości korzystania z aplikacji.



Rysunek 2: Obraz prezentujący ekran początkowy aplikacji

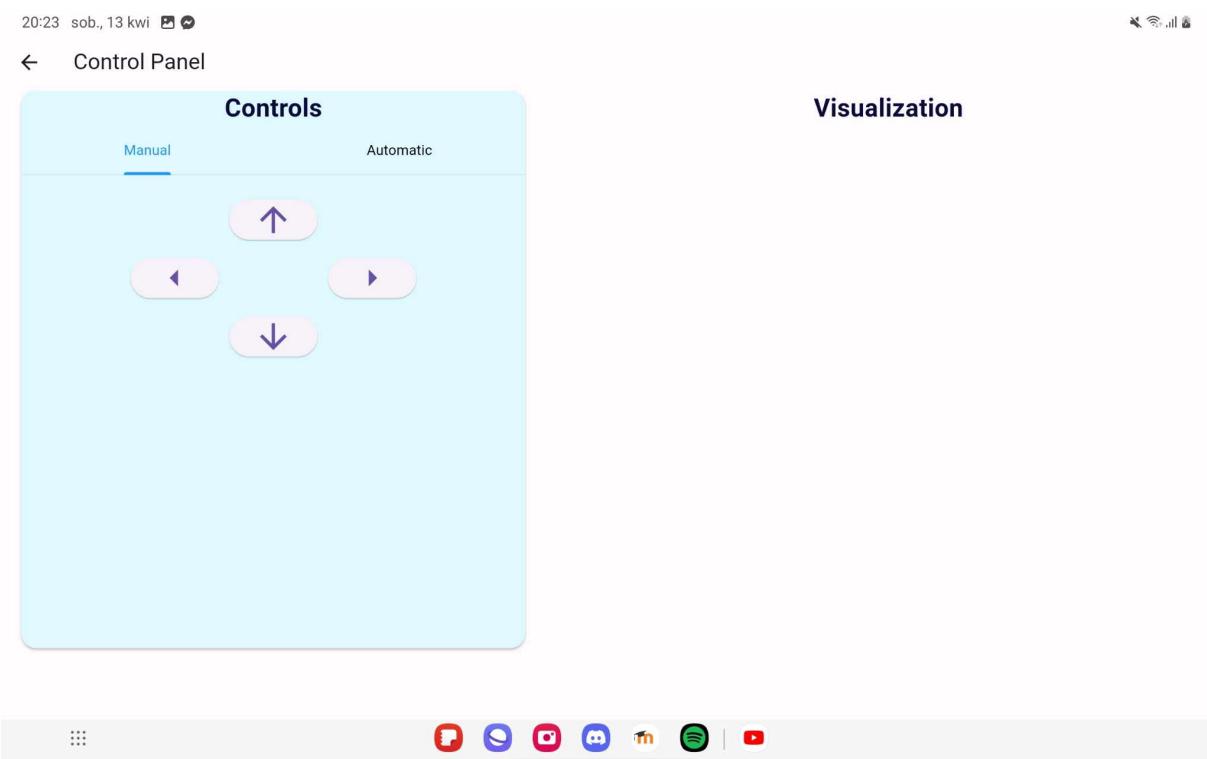
W lewym górnym rogu ekranu znajduje się standardowy dla aplikacji mobilnych przycisk rozwijający menu boczne. Menu zawiera wszystkie niezbędne zakładki, takie jak:

- *Control Panel* (Panel Kontrolny) – umożliwia sterowanie robotem,
- *Connect to device* (Połącz z urządzeniem) – pozwala na wyszukiwanie i łączenie z urządzeniem,
- *Settings* (Ustawienia) – umożliwia dostosowanie parametrów robota,
- *Help* (Pomoc) – zawiera informacje o korzystaniu z aplikacji oraz dane kontaktowe działu wsparcia.



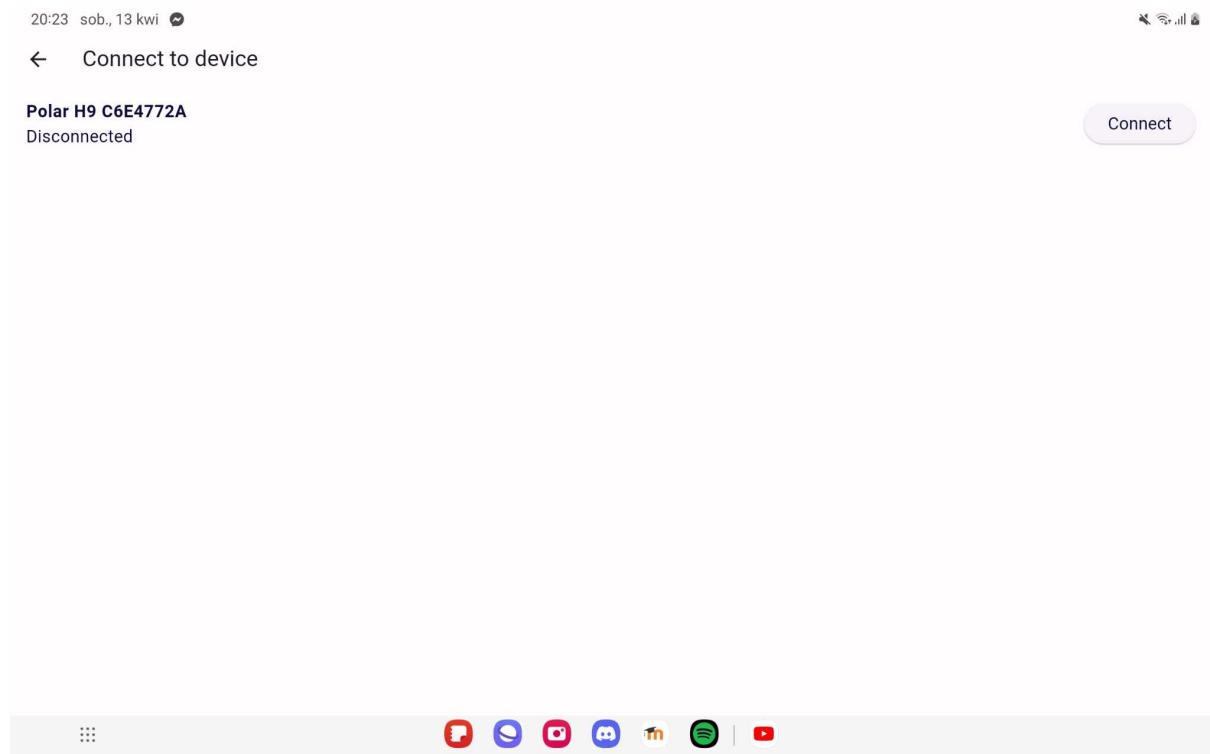
Rysunek 3: Menu boczne aplikacji

Po kliknięciu w pierwszą zakładkę użytkownik zostaje przekierowany do strony odpowiedzialnej za sterowanie robotem. Ekran ten podzielony jest na dwie sekcje. Pierwsza z nich, zatytułowana *Controls* (Kontrolki), umożliwia bezpośrednie sterowanie robotem poprzez odpowiednie przyciski. Po prawej stronie znajduje się sekcja, która w przyszłości ma wyświetlać animację przedstawiającą robota w ruchu.



Rysunek 4: Ekran kontroli robota

Kolejną zakładką jest ekran wyświetlający dostępne urządzenia Bluetooth. Użytkownik może połączyć aplikację z wybranym urządzeniem poprzez naciśnięcie przycisku *Connect* (Połącz). Zaleca się łączenie wyłącznie z urządzeniami będącymi częścią systemu robota.



Rysunek 5: Ekran połączenia z urządzeniami

Zakładki takie jak *Settings* (Ustawienia) i *Help* (Pomoc) nie zostały zaimplementowane z powodu zmiany priorytetów podczas rozwoju projektu. Niemniej jednak, aplikacja ma duży potencjał rozwoju i może zostać rozbudowana w przyszłości.

2.3. Implementacja połączenia Bluetooth

Kluczową informacją potrzebną do nawiązania połączenia z mikrokontrolerem Arduino UNO R4 jest fakt, iż mikrokontroler ten korzysta ze specyficznego rodzaju Bluetooth - Bluetooth Low Energy (BLE). W związku z tym konieczne było wykorzystanie odpowiedniej biblioteki, takiej jak *flutter_blue*. Bibliotekę tę można pobrać za pomocą narzędzia w terminalu, wpisując komendę:

```
flutter pub add flutter_blue
```

Drugą istotną kwestią jest konieczność wykorzystania fizycznego urządzenia do uruchamiania aplikacji. Jest to zalecane ze względu na brak możliwości wyszukiwania urządzeń Bluetooth w niektórych emulatorach.

Pierwszym krokiem do nawiązania połączenia jest wyświetlenie listy wszystkich dostępnych urządzeń Bluetooth. W tym celu wykonywana jest funkcja *_startScan()*, której zadaniem jest pobranie wszystkich dostępnych urządzeń i dodanie ich do listy w celu późniejszego wyświetlenia.

Funkcja *_getPairedDevices()* zwraca wszystkie urządzenia, które zostały pomyślnie sparowane.

Po zakończeniu skanowania, proces jest zamykany przy pomocy funkcji *_stopScan()*.



```
1 void _getPairedDevices() {
2     flutterBlue.connectedDevices.then((List<BluetoothDevice> devices) {
3         setState(() {
4             pairedDevices = devices;
5         });
6     });
7 }
8
9 void _startScan() {
10    flutterBlue.scanResults.listen((List<ScanResult> results) {
11        setState(() {
12            devices.clear();
13            for (ScanResult result in results) {
14                if (!devices.contains(result.device)) {
15                    devices.add(result.device);
16                }
17            }
18        });
19    });
20    flutterBlue.startScan();
21 }
22
23 void _stopScan() {
24     flutterBlue.stopScan();
25 }
```

Rysunek 6: Fragment kodu źródłowego zawierający implementację wyszukiwania dostępnych urządzeń

Kolejnym krokiem jest nawiązanie połączenia z wybranym urządzeniem. Proces ten jest realizowany przez funkcję `_connectToDevice()`, która jako parametr przyjmuje referencję do wybranego urządzenia z listy. Funkcja ta wywołuje metodę kończącą skanowanie, a następnie na instancji klasy `BluetoothDevice` wywołuje metodę `connect()`, która realizuje połączenie. Klasa `BluetoothDevice` jest dostępna dzięki użyciu biblioteki `flutter_blue`.



```
1 void _connectToDevice(BluetoothDevice device) async {
2     _stopScan();
3     await device.connect();
4     setState(() {
5         connectedDevice = device;
6     });
7     Provider.of<ConnectedDeviceProvider>(context, listen: false)
8         .setConnectedDevice(device);
9     PopupInfo.showInfo(
10         context,
11         'The connection have been established correctly with device: ${device}',
12         InfoType.success);
13 }
```

Rysunek 7: Fragment kodu źródłowego zawierający implementację nawiązywania połączenia BLE

2.4. Implementacja przesyłania danych

Proces przesyłania informacji do urządzenia Bluetooth opiera się na metodzie `sendValueToArduino()`. Ta metoda przyjmuje dwa parametry: referencję do połączonego urządzenia oraz wartość, która ma zostać przesłana.

Pierwszym krokiem w tej metodzie jest weryfikacja, czy urządzenie zostało poprawnie połączone oraz czy jest możliwe przesłanie danych. W przypadku niepowodzenia tej weryfikacji, w konsoli zostaje wyświetlona stosowna informacja diagnostyczna, co pozwala na szybkie zidentyfikowanie problemu.

Następnie metoda wyszukuje urządzenia o konkretnej charakterystyce, która jest niezbędna do prawidłowego funkcjonowania połączenia. Znalezienie urządzenia o odpowiedniej charakterystyce jest kluczowe, ponieważ zapewnia, że dane zostaną przesłane do właściwego odbiorcy.

Jeśli charakterystyka urządzenia zgadza się z oczekiwana, wartość jest przesyłana. Taki proces zapewnia nie tylko prawidłowość przesyłania danych, ale również bezpieczeństwo połączenia.

Poniżej znajduje się fragment kodu źródłowego, który ilustruje implementację logiki przesyłania informacji do urządzenia.



```
1 class SendingServices {
2     Future<void> sendValueToArduino(
3         BluetoothDevice? connectedDevice, int value) async {
4     if (connectedDevice != null) {
5         List<int> dataToSend = [value];
6         List<BluetoothService> services =
7             await connectedDevice.discoverServices();
8         BluetoothCharacteristic? characteristic;
9
10        services.forEach((service) {
11            service.characteristics.forEach((char) {
12                if (char.uuid.toString() == '19b10001-e8f2-537e-4f6c-d104768a1215') {
13                    characteristic = char;
14                }
15            });
16        });
17
18        if (characteristic != null) {
19            await characteristic!.write(dataToSend, withoutResponse: false);
20            print('Sent value $value to Arduino.');
21        } else {
22            print('Error: Characteristic not found.');
23        }
24    } else {
25        print('Error: Not connected to any device.');
26    }
27 }
28 }
```

Rysunek 8: Fragment kodu źródłowego zawierający implementację logiki przesyłania informacji do urządzenia

2.5. Implementacja prostego skryptu w odczytującego dane z kanału BLE

Fragment prezentowanego kodu rozpoczyna się od zadeklarowania zmiennych globalnych opisujących charakterystyki połączenia Bluetooth. Pierwsza charakterystyka określa identyfikator typu GUID serwisu, następna określa na jaki identyfikator przesyłane są zdarzenia typu "request". Ostatnia wartość definiuje identyfikator na jaki są przesyłane zdarzenia typu 'response'. W przypadku aktualnej wersji aplikacji istotna jest tylko wartość identyfikująca serwis oraz zdarzenia typu request, ponieważ mikrokontroler nie zwraca żadnej informacji do aplikacji. W razie dalszego rozwoju projektu zostanie zaimplementowanie wyświetlanie diagnostyki robota, stanu naładowania baterii oraz stanu pracy silników.

Funkcja `t5Callback()` ma za zadanie pobierać wartość która aktualnie została przesłana do kanału Bluetooth z poziomu aplikacji. Argumenty wejściowe funkcji to urządzenie z którym Arduino nawiązało połączenie, drugi argument to wcześniej wspomniany identyfikator zdarzenia "request".

Następnie wartość jest odczytywana z kanału za pomocą funkcji `readValue()`. Funkcja ta jest częścią biblioteki `ArduinoBLE.h`.

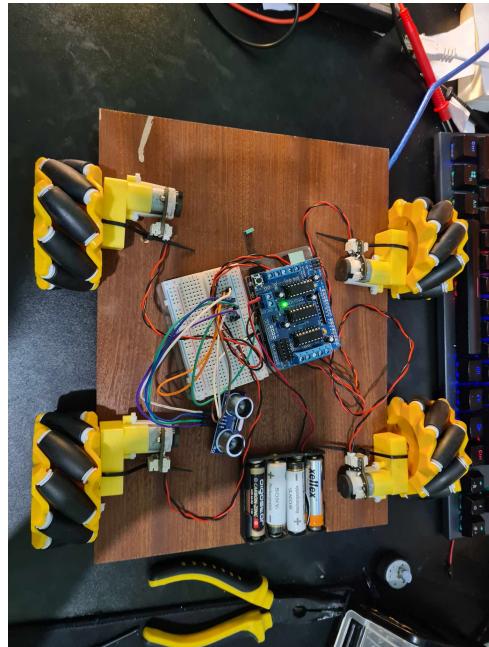
W funkcji `setup()` następuje inicjalizacja odpowiednich charakterystyk oraz za pomocą funkcji `setEventHandler()` zostaje określona sytuacja w której nastąpi wywołanie funkcji `t5Callback` oraz odczytanie danych z kanału.

```
 1 const char *deviceServiceUuid = "19b10000-e8f2-537e-4f6c-d104768a1214";
 2 const char *deviceServiceRequestCharacteristicUuid = "19b10001-e8f2-537e-4f6c-d104768a1215";
 3 const char *deviceServiceResponseCharacteristicUuid = "19b10001-e8f2-537e-4f6c-d104768a1216";
 4
 5 BLEService servoService(deviceServiceUuid);
 6 BLEStringCharacteristic servoRequestCharacteristic(
 7   deviceServiceRequestCharacteristicUuid, BLEWrite, 20); // Increased buffer size to 20
 8 BLEStringCharacteristic servoResponseCharacteristic(
 9   deviceServiceResponseCharacteristicUuid, BLENotify, 20); // Increased buffer size to 20
10
11 void t5Callback(BLEDevice central, BLECharacteristic characteristic) {
12   if (central.connected()) {
13     if (characteristic == servoRequestCharacteristic)
14     {
15       int16_t value = 0;
16       servoRequestCharacteristic.readValue(&value, 2);
17       Serial.println("Received value: ");
18       Serial.println(value);
19     }
20   }
21 }
22
23
24 void setup() {
25   Serial.begin(9600);
26
27   BLE.setDeviceName("Arduino APP");
28   BLE.setLocalName("Arduino APP");
29
30   if (!BLE.begin()) {
31     Serial.println("- Starting Bluetooth® Low Energy module failed!");
32     while (1);
33   }
34
35   BLE.setAdvertisedService(servoService);
36   servoService.addCharacteristic(servoRequestCharacteristic);
37   servoService.addCharacteristic(servoResponseCharacteristic);
38   servoRequestCharacteristic.setEventHandler(BLEWritten, t5Callback); // Set the callback function for when data is written to the characteristic
39   BLE.addService(servoService);
40   servoResponseCharacteristic.writeValue("0");
41
42   BLE.advertise();
43
44   Serial.println("Arduino BLE (Peripheral Device)");
45   Serial.println(" ");
46 }
```

Rysunek 9: Fragment kodu źródłowego przedstawiający implementację konfiguracji kanału Bluetooth na Arduino

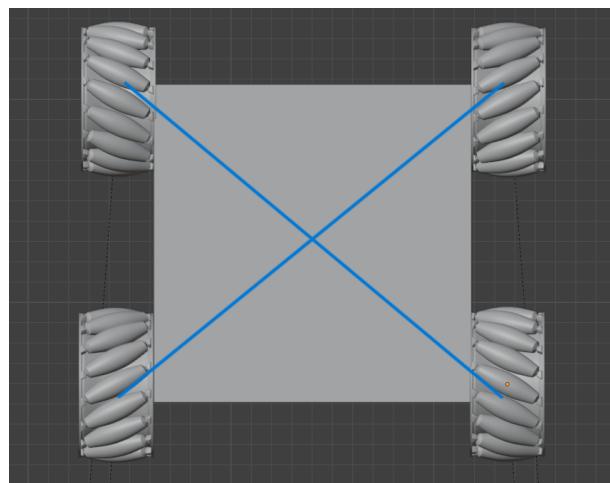
2.6. Proces wykonywania projektu obudowy robota

Proces projektowania obudowy dla platformy jeżdżącej stanowił wieloetapowe zadanie, które wymagało dokładnego zaplanowania i licznych poprawek. Pierwsza faza obejmowała stworzenie prototypu, który miał na celu optymalne rozmieszczenie komponentów oraz weryfikację ich poprawnego działania. Ten etap pozwalał zidentyfikować ewentualne problemy konstrukcyjne i funkcjonalne na wczesnym etapie prac.



Rysunek 10: Prototypowa konstrukcja mająca na celu sprawdzenie poprawności działania komponentów.

Kolejnym krokiem było opracowanie specjalnej obudowy, zdolnej pomieścić wszystkie wymagane elementy, przy jednoczesnym zachowaniu odpowiedniej geometrii kół szwedzkich (zwanych również kołami Mecanum). Aby robot działał zgodnie z założeniami, koła musiały być rozmieszczone w orientacji X Y równych odstępach od siebie, a ich rolki powinny być skierowane do środka. Konieczne jest aby z rzutu górnego posiadał przedłużenie osi kół w kształcie znaku X.

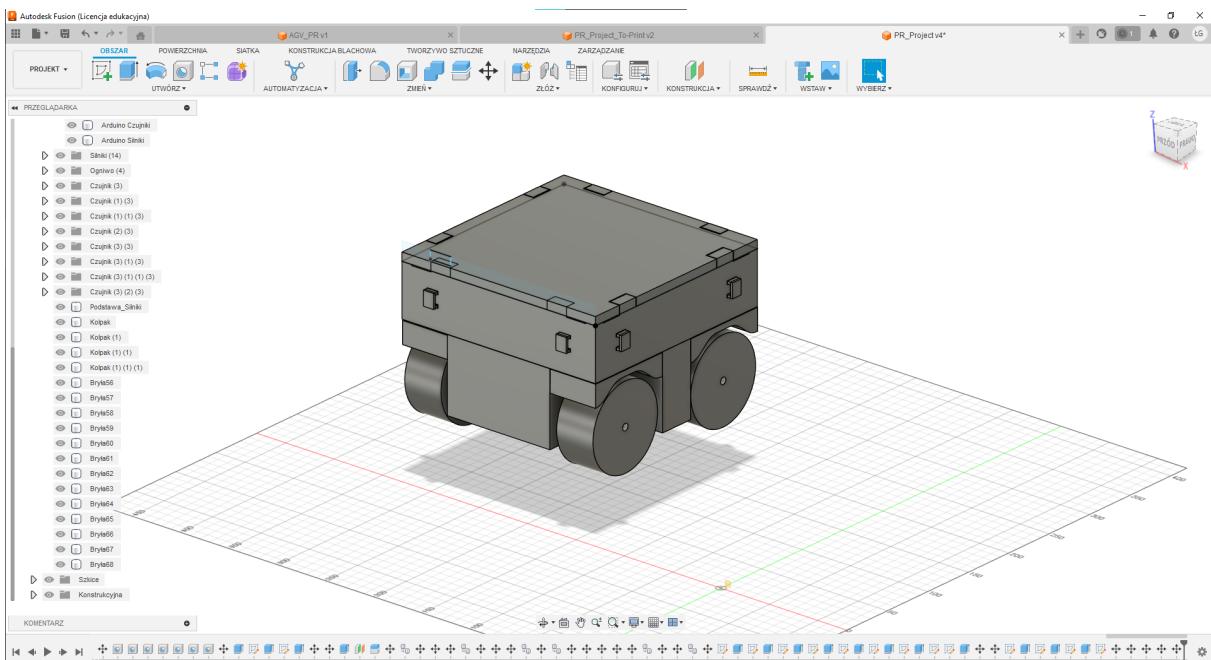


Rysunek 11: Plan rozmieszczenia kół szwedzkich. Kluczowe jest zachowanie kół w równych odstępach od siebie oraz utworzenie znaku X.



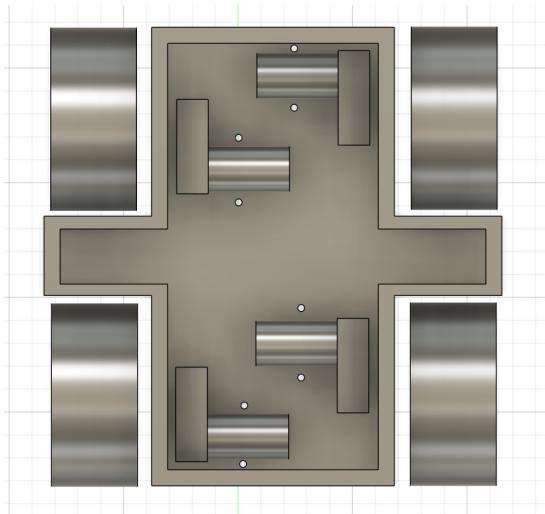
Rysunek 12: Pierwsza wersja projektu zamodelowana w programie Blender 4.1.

Jednakże podczas pierwszego etapu projektu nastąpiły problemy logiczne i konstrukcyjne. Pierwotnie używany darmowy program Blender 4.1 okazał się niewystarczający ze względu na niezgodność formatów eksportu z wymaganiami dostępnej drukarki 3D. W związku z tym, konieczne było przejście na bardziej zaawansowane narzędzie, jakim jest AutoDesrk Fusion. To oprogramowanie pozwoliło na dokładniejsze zaprojektowanie modelu, uwzględniając specyfikację drukarki, która posiadała obszar roboczy o wymiarach 25 cm x 25 cm. Z tego powodu, model musiał zostać zaprojektowany od nowa, z uwzględnieniem ograniczeń.

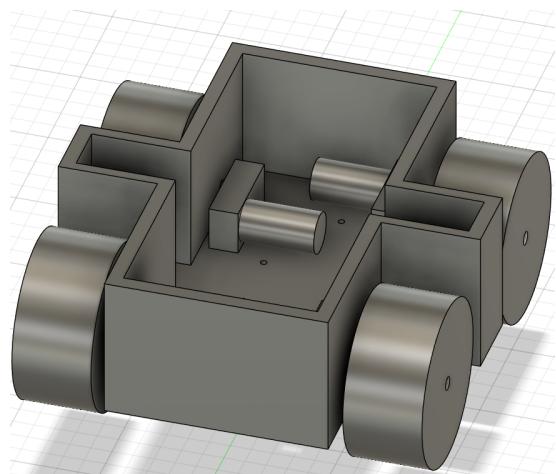


Rysunek 13: Wizualizacja gotowego modelu w programie AutoDesrk Fusion.

Gotowy produkt składa się z dwóch pięter, z których każde pełni inną funkcję. Pierwsze piętro to "piętro maszynowe", gdzie umieszczone zostały moduły napędowe, takie jak silniki kątowe z przekładnią oraz ogniva zasilające w ilości dwóch wraz ze sterownikiem BMS. Wszystkie istotne kable zostały przeprowadzone przez specjalnie przygotowany otwór do drugiego piętra.

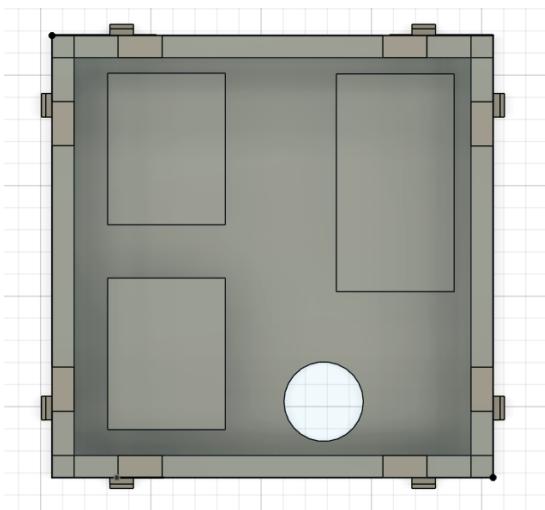


Rysunek 14: Widok rzutu górnego na pierwsze piętro modelu. Znajdują się w nim silniki kątowe z przekładnią oraz miejsce na ogniva zasilające wraz z BMS.

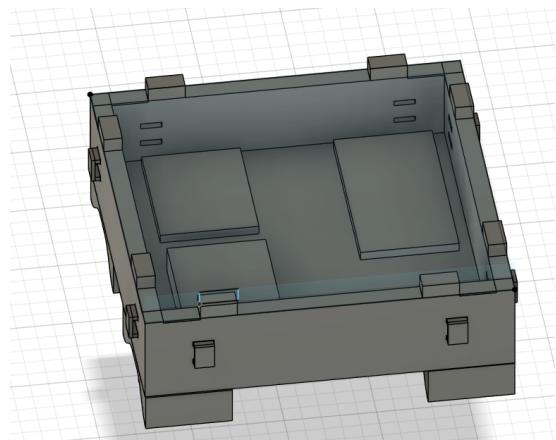


Rysunek 15: Widok rzutu bocznego na pierwsze piętro modelu.

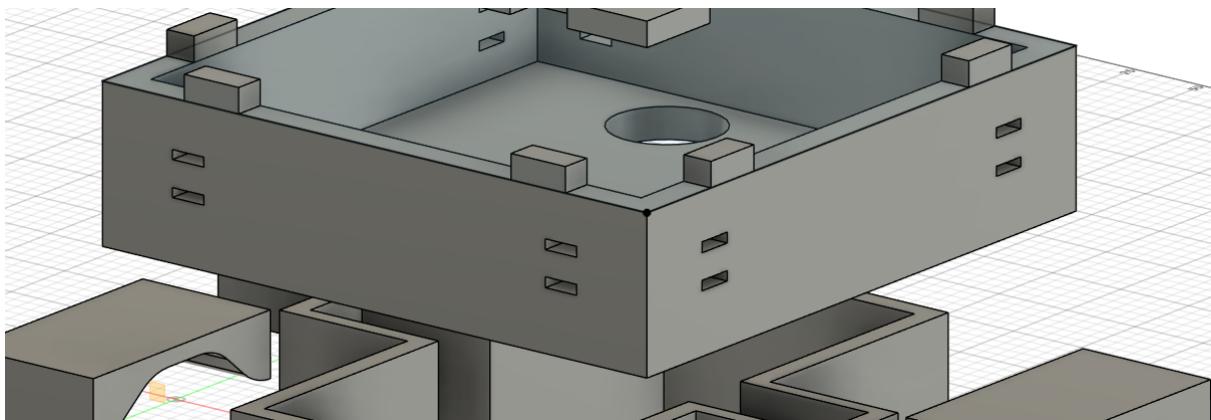
Drugie piętro to "mózg operacyjny" całego projektu. Zawiera ono kluczowe połączenia elektroniczne, w tym dwa mikrokontrolery Arduino UNO 4 i 3 w konfiguracji Master-Slave, oraz osiem czujników laserowych umieszczonych w przygotowanych otworach na ścianach bocznych. Takie rozmieszczenie i organizacja komponentów pozwalają na efektywne zarządzanie funkcjami robota.



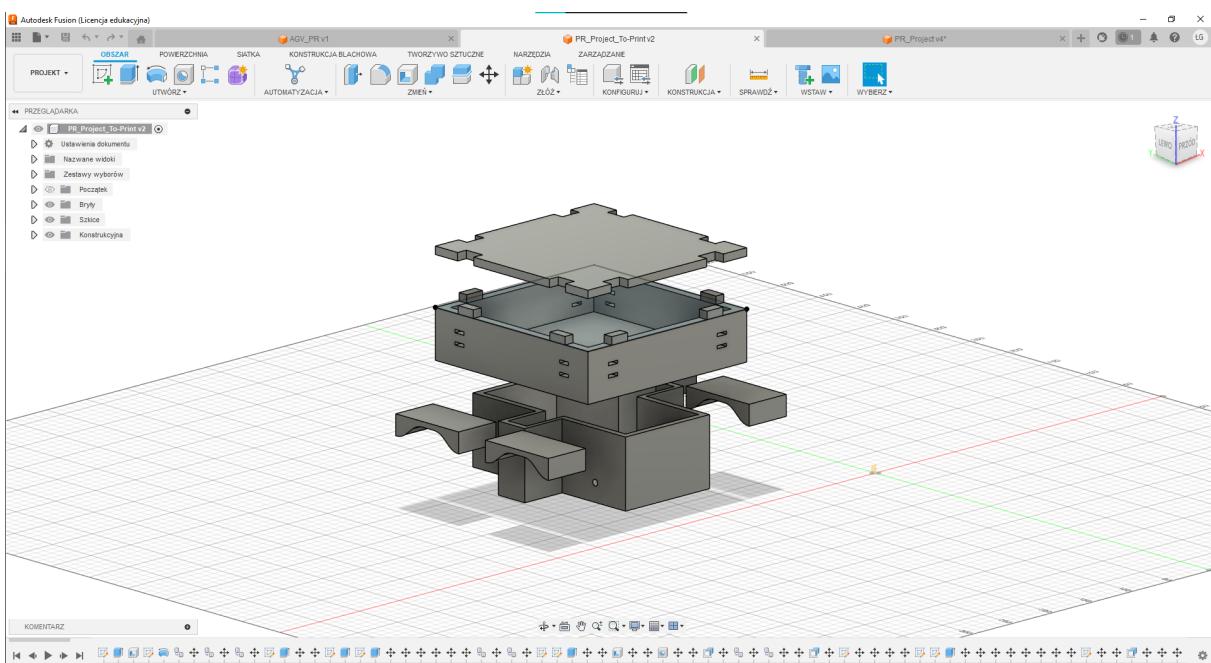
Rysunek 16: Widok rzutu górnego na drugie piętro modelu. Znajdują się w nim dwa mikrokontrolery Arduino UNO oraz płytka stykowa.



Rysunek 17: Widok rzutu bocznego na drugie piętro modelu.

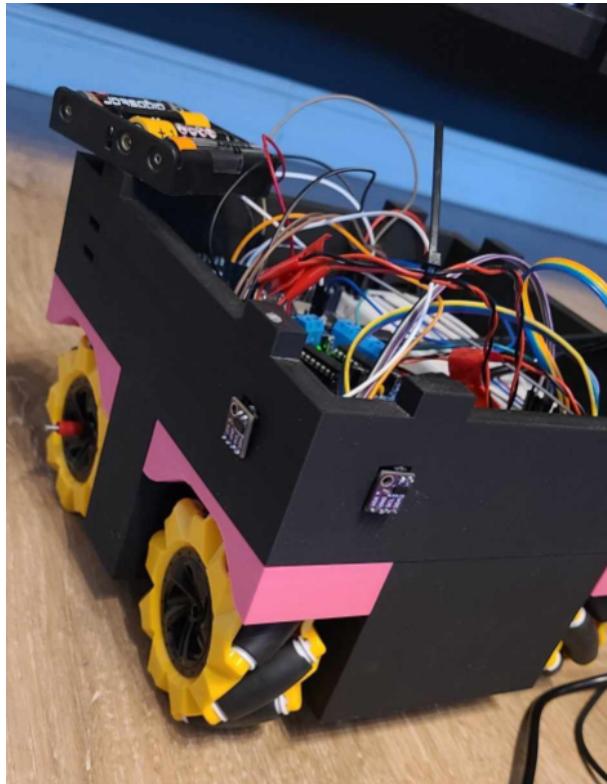


Rysunek 18: Widok rzutu bocznego na specjalnie przygotowane miejsca do wsunięcia czujników laserowych.



Rysunek 19: Gotowy model przesłany do wydruku. Został pozbawiony elementów stylistycznych takich jak koła, silniki kątowe, mikrokontrolerów Arduino UNO oraz czujników laserowych.

Gotowy model zawiera się w wymiarach 20 cm x 20 cm x 15 cm oraz wykonuje wszystkie zadane ruchy zdolne do wykonania tylko przy pomocy kół szwedzkich. Pojazd wykonuje jazdę w kierunku w przód i w tył oraz w lewo i w prawo jak i obrót w miejscu.



Rysunek 20: Gotowy model po złożeniu wszystkich elementów w całość.

2.7. Implementacja prostego kodu weryfikującego prawidłowe połączenie silników

Do połączenia silników z Arduino wykorzystano nakładkę Iduino ST1138. Umożliwia ona sterowanie czterema silnikami prądu stałego, o poborze prądu do 600 mA i napięciu zasilania do 36 V. Nakładka jest zasilana dwoma ogniwami 18650.

Weryfikacja połączenia silników została wykonana przy użyciu skryptu zamieszczonego poniżej. Na początku skryptu dodane są biblioteki Wire i AFMotor. Biblioteka Wire jest wykorzystywana do połączenia obu modułów Arduino w konfiguracji master-slave, natomiast biblioteka AFMotor służy do sterowania silnikami z nakładki Iduino.

W następnych liniach zdefiniowane są: adres urządzenia slave oraz globalna wartość bezpiecznego dystansu, który jest potrzebny, aby w przypadku wystającego gzymsu, który mógłby ominąć czujniki, pojazd nie zahaczył o niego.

Dalsza część kodu zawiera wykorzystanie biblioteki AFMotor i przypisanie konkretnych silników do zmiennych.

W ostatniej części, czyli setup, znajduje się połączenie przez bibliotekę Wire, rozpoczęcie komunikacji z prędkością 9600 bitów na sekundę (`Serial.begin(9600)`) oraz ustawienie prędkości wszystkich silników.

Do weryfikacji działania silników można zakomentować pozostałe ustawienia `setSpeed`, tak aby pozostało tylko jedno i sprawdzić, który silnik jest uruchomiony.

```

#include <Wire.h>
#include <AFMotor.h>

#define SLAVE_ADDR 9           // Address of the slave device
#define THRESHOLD_DISTANCE 20 // borderline for distance to the wall

// -----
// Creation of all necesary global instances
// -----

// Motor drivers
AF_DCMotor motor1(1);
AF_DCMotor motor2(2);
AF_DCMotor motor3(3);
AF_DCMotor motor4(4);

// -----
// Other global variables
// -----

int rd; // Global variable that stores value from I2C BUS

// -----
// -----


void setup() {

    Wire.begin(SLAVE_ADDR);
    Wire.onReceive(receiveEvent);
    Serial.begin(9600);
    motor1.setSpeed(250);
    motor2.setSpeed(250);
    motor3.setSpeed(250);
    motor4.setSpeed(250);
    Serial.println("Set up done.");
}

```

Rysunek 21: Fragment kodu źródłowego przedstawiający weryfikację prawidłowego połączenia silników

2.8. Podłączenie oraz weryfikacja poprawności działania czujników laserowych

We wczesnej fazie planowania zakładano wykorzystanie ultradźwiękowych czujników odległości. Pomyśl ten okazał się nietrafiony ze względu na sposób przekazywania danych przez te czujniki. Każdy czujnik posiada dwa wyprowadzenia, co przy użyciu ośmiu czujników wymagałoby wykorzystania co najmniej 16 pinów Arduino.

Następnie analiza skupiła się na wykorzystaniu magistrali I2C, która umożliwia podłączenie dużej

liczby czujników za pomocą niewielkiej liczby pinów. Kluczowym warunkiem jest poprawna komunikacja między czujnikiem a mikrokontrolerem.

Laserowe czujniki odległości VL53L0X okazały się trafnym wyborem ze względu na ich korzystny stosunek jakości do ceny. Pomiary odległości są wystarczająco dokładne, a biorąc pod uwagę charakter obiektu wykrywanego, wysoka precyzja nie jest wymagana.

Czujniki zostały podłączone zgodnie z przeznaczeniem poprzez magistralę I2C. Kluczową rolę w podłączaniu wielu identycznych czujników odegrało użycie dodatkowego pinu XSHUT oraz odpowiedniej sekwencji kodu, aby pomyślnie zmienić adres czujnika na magistrali I2C.

Pierwszym krokiem jest ustawienie odpowiedniego pinu GPIO (ang. General Purpose Input/Output) Arduino, podłączonego do pinu XSHUT czujnika, w trybie OUTPUT. Następnie należy skorzystać z funkcji `setAddress()`, aby ustawić odpowiedni adres dla wybranego czujnika. Kolejnym krokiem jest przełączenie stanu pinu Arduino na INPUT oraz odczekanie 10 milisekund.

Wykonanie powyższej sekwencji gwarantuje poprawną zmianę adresu czujnika na magistrali I2C. Użycie samej funkcji `setAddress()` bez zastosowania pinu XSHUT nie zmieni adresu czujnika.



```
1 #define Sensor2_newAddress 42
2
3 pinMode(XSHUT_pin2, OUTPUT);
4
5 Sensor2.setAddress(Sensor2_newAddress);
6 pinMode(XSHUT_pin2, INPUT);
7 delay(10);
```

Rysunek 22: Fragment kodu źródłowego przedstawiający sekwencję konieczną do zmiany adresu czujnika

2.9. Montaż silników wewnętrz dolnej komory obudowy

Silniki zostały przymocowane do dolnej części pojazdu za pomocą taśmy aluminiowej. Po wykonaniu tego kroku, przystąpiono do montażu kół na silnikach. Koła zostały zamontowane na wygwintowanej szprysze, która została przymocowana bezpośrednio do silników.

W celu wyeliminowania luzów na silnikach, podjęto dodatkowe działania. Wycięto dwie listwy aluminiowe, które zamontowano w taki sposób, aby przez nie przechodziły szprychy. Dzięki zastosowaniu tych listew, luz na kołach został znacznie zminimalizowany, co zapewniło stabilność całej konstrukcji.

Wykonanie tych czynności pozwoliło na usztywnienie wału silnika i zapewniło solidne połączenie kół z silnikami.



Rysunek 23: Listwa boczna usztywniająca mocowanie silnika i eliminująca luzy na kole.

2.10. Montaż ogniw wraz z układem BMS oraz podłączenie czujników

2.11. Integracja czujników z silnikami oraz implementacja logiki sterowania

W celu osiągnięcia optymalnej integracji modułu czujników oraz modułu sterowania silnikami zastosowano strukturę opartą na magistrali I2C. W omawianym układzie wykorzystano architekturę typu Master-Slave. Moduł odczytu danych z czujników oraz moduł odpowiedzialny za sterowanie silnikami są realizowane przez dwa oddzielne mikrokontrolery Arduino UNO. Mikrokontroler odpowiedzialny za odczyt wartości z czujników pełni rolę Master, przetwarzając informacje uzyskane z czujników i na ich podstawie przesyłając sygnały sterujące do mikrokontrolera pełniącego rolę Slave.

Zastosowanie tej specyficznej architektury było niezbędne z uwagi na ograniczenia w dostępności pinów GPIO. Wykorzystanie czterokanałowego sterownika silników w formie nakładki na mikrokontroler Arduino spowodowało zmniejszenie liczby dostępnych pinów do pinów analogowych o adresach od A0 do A5. Jak wspomniano w sekcji 2.8, wykorzystanie kilku czujników wymaga dostosowania ich adresów. W związku z tym konieczne było uzyskanie dostępu do co najmniej siedmiu pinów ogólnego przeznaczenia I/O, co zapewnia drugie Arduino pełniące rolę Master.

2.11.1. Master

Kod źródłowy rozpoczyna się od deklaracji oraz inicjalizacji zmiennych oraz stałych globalnych. Są to między innymi adresy czujników, typ wyliczeniowy zawierający informacje o aktualnej trajektorii robota oraz adresy pinów do których zostaną podłączone piny XSHUT.

Następnie w funkcji `setup()` wykonywana jest inicjalizacja czujników. Proces inicjalizacji rozpoczyna się od ustawienia odpowiedniego adresu dla każdego czujnika. Proces ten został opisany wcześniej. Kolejnym etapem jest właściwa inicjalizacja czujnika. Wykonywana jest ona za pomocą funkcji `init()` wypożyczanej na każdej instancji obiektu czujnika. W przypadku błędnej inicjalizacji wyświetlany jest stosowny komunikat. Następnie dla każdego czujnika ustawiany jest czas maksymalnego oczekiwania na wartość (ang. timeout) za pomocą funkcji `setTimeout()`. Jest on ustawiony na wartość 500ms, zatem po tym czasie zostanie zwrócona wartość maksymalna dla typu `UINT16`. Ostatnim zadaniem jest przełączenia czujnika w tryb odczytu ciągłego za pomocą funkcji `setContinuous()`.

Następna funkcja typowa dla środowiska Arduino IDE to `setup()`. Funkcja ta zawiera wywołania czterech dodatkowych funkcji:

- `searchForDevices()` - odpowiada za wyszukiwanie dostępnych urządzeń na magistrali I2C,
- `readFromSensor()` - odpowiada za odczytanie wartości z czujników,
- `processSensorInfo()` - odpowiada za interpretację otrzymanych wyników,
- `sendSteering()` - odpowiada za przesłanie instrukcji sterowania do urządzenia Slave.

Funkcja `searchForDevices()`

Funkcja ta jest funkcją pomocniczą, niekonieczną do poprawnego działania programu. Jest ona jedna przydatna w celu weryfikacji poprawności połączenia urządzeń do magistrali I2C. W przypadku braku dostępnych urządzeń lub braku zgodności ich liczby należy zwrócić uwagę, czy wszystkie urządzenia są poprawnie podłączone do zasilania oraz do magistrali. Jeśli pierwsze w kolejności urządzenie podłączone do magistrali jest wadliwe lub zostało niepoprawnie podłączone, żadne kolejne urządzenie nie zostanie wykryte.

Realizacja wyszukiwania dostępnych urządzeń (adresów) jest relatywnie prosta. W pętli `for` poprzez wywołanie funkcji `Wire.beginTransmission(int addr)` rozpoczynana jest próbna transmisja do potencjalnego urządzenia. Jeśli urządzenie odpowie bez błędu, jego adres jest wyświetlany w konsoli w postaci liczby w formacie szesnastkowym.

```
1 // This function is used to search for discovering accessible devices on the I2C BUS
2 void searchForDevices() {
3     byte error, address;
4     int nDevices;
5     nDevices = 0;
6     for(address = 1; address < 127; address++ )
7     {
8         Wire.beginTransmission(address);
9         error = Wire.endTransmission();
10
11        if (error == 0)
12        {
13            Serial.print("I2C device found at address 0x");
14            if (address<16)
15                Serial.print("0");
16            Serial.print(address,HEX);
17            Serial.println(" !");
18
19            nDevices++;
20        }
21        else if (error==4)
22        {
23            Serial.print("Unknown error at address 0x");
24            if (address<16)
25                Serial.print("0");
26            Serial.println(address,HEX);
27        }
28    }
29    if (nDevices == 0)
30        Serial.println("No I2C devices found");
31    else
32        Serial.println("done");
33 }
```

Rysunek 24: Kod źródłowy realizujący wyszukiwanie urządzeń dostępnych na magistrali I2C

Funkcja `readFromSensor()`

Implementacja funkcji czytania wartości z sensora jest bardzo prostolinienna. Za pomocą metody `readRangeContinuousMillimeters()` wywołanej na odpowiednim obiekcie klasy VL53L0X odczytywana jest wartość odległości wyrażona w milimetrach. Jednocześnie odczytana wartość jest przypisywana do odpowiedniej komórki w tablicy pomiarów.

We fragmentie kodu źródłowego można zauważyc wykomentowany fragment wyświetlający wartość odległości. Został on wykorzystany w celu walidacji danych odczytywanych z czujników.



```
1 // This function is used to read data from sensors
2 void readFromSensor() {
3     // Serial.print("Odczyt z pierwszego czujnika: ");
4     // Serial.println(Sensor1.readRangeContinuousMillimeters());
5     dist_array[0] = Sensor1.readRangeContinuousMillimeters();
6
7     // Serial.print("Odczyt z drugiego czujnika: ");
8     // Serial.println(Sensor2.readRangeContinuousMillimeters());
9     dist_array[1] = Sensor2.readRangeContinuousMillimeters();
```

Rysunek 25: Kod źródłowy realizujący odczytywanie odległości z czujnika

Funkcja processSensorInfo()

Powyższa funkcja ma wagę fundamentalną pod kątem poprawności działania robota. Jednocześnie funkcja ta jest najbardziej złożona, przez co zostanie dodany cały kod źródłowy.

Funkcja rozpoczyna się odinicjalizacji zmiennej `currentTime()`, jej sens wyjaśni się w miarę zgłębiania logiki wyznaczającej sterowania.

Pierwsza instrukcja warunkowa sprawdza czy robot znajduje się we wnęce. Stan domyślny jest ustawiony na `false`. Następnie, zgodnie z dodanymi komentarzami, jeśli przed robotem oraz z prawej strony strony odległość jest większa niż stała `SAFE_DIST`, wtedy robot dojedzie do ściany. Funkcja zwraca wartość na podstawie typu wyliczeniowego nadmienionego wcześniej. Dodatkowo w celach pomocniczych wyświetlany jest komunikat o aktualnej trajektorii.

Następne instrukcje warunkowe realizują sterowanie zgodnie z komentarzami. Warto zwrócić uwagę na instrukcję warunkową rozpoczynającą się w linii 15. Jeśli robot wykryje ścianę przed sobą, robot rozpoczyna ruch w lewo oraz przełącza flagę `wnęka` na wartość `true`.

Jeśli została wykryta wnęka wykonuje się alternatywna sekwencja instrukcji warunkowych. Jeśli przed robotem ciągle znajduje się przeszkoda, jego ruch pozostaje niezmienny w lewo. W przypadku, gdy czujniki wykryją wolną przestrzeń, włączany jest tryb wyjeżdżania z wnęki.

Opuszczanie wnęki zostało zrealizowane za pomocą licznika, stąd deklaracja zmiennej `currentTime` na początku funkcji. W linii nr 55 sprawdzany jest warunek, czy czas przeznaczony na wyjazd z wnęki minął. Jeśli tak, pojazd przełącza się w tryb normalny, jeśli nie, pojazd kontynuuje jazdę prosto.

Wyjaśnienie wykorzystania takiej struktury zostanie opisane w sekcji 3.3.

```

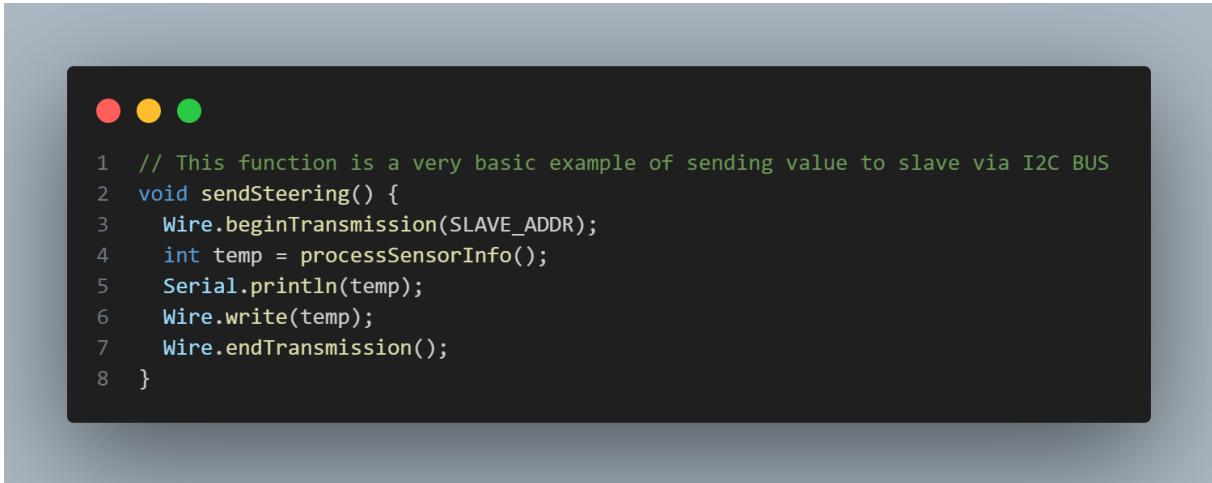
1 // This function will be used to process and prepare data to be sent to slave
2 int processSensorInfo() {
3     unsigned long currentTime = millis();
4
5     // Normal mode
6     if (!wneka) {
7         // If the robot is in an open space, it moves to the right
8         if ((dist_array[4] > SAFE_DIST && dist_array[4] > SAFE_DIST) &&
9             (dist_array[2] > SAFE_DIST && dist_array[1] > SAFE_DIST)) {
10            Serial.println("ENGINES TO RIGHT");
11            current_direction = CURRENT_DIR::MOVE_RIGHT;
12            return CURRENT_DIR::MOVE_RIGHT;
13        }
14        // If the robot encounters an obstacle in front, it enters recess mode
15        if ((dist_array[4] <= SAFE_DIST) || (dist_array[3] <= SAFE_DIST)) {
16            current_direction = CURRENT_DIR::MOVE_LEFT;
17            Serial.println("ENGINES TO LEFT");
18            wneka = true;
19            return CURRENT_DIR::MOVE_LEFT;
20        }
21        // If there are no obstacles in front of the robot, it moves forward
22        if (dist_array[4] > SAFE_DIST && dist_array[3] > SAFE_DIST) {
23            Serial.println("ENGINES FORWARD");
24            current_direction = CURRENT_DIR::MOVE_FORWARD;
25            return CURRENT_DIR::MOVE_FORWARD;
26        }
27        // If the distance from the wall is greater than the safe distance, it moves to the right
28        if ((current_direction != CURRENT_DIR::MOVE_LEFT) && dist_array[2] > SAFE_DIST && dist_array[1] > SAFE_DIST) {
29            Serial.println("ENGINES TO RIGHT");
30            current_direction = CURRENT_DIR::MOVE_RIGHT;
31            return CURRENT_DIR::MOVE_RIGHT;
32        }
33    }
34
35    // Alcove mode
36    if (wneka) {
37        // If there is an obstacle in front of the robot, it continues to move to the left
38        if ((dist_array[4] <= SAFE_DIST) || (dist_array[3] <= SAFE_DIST)) {
39            Serial.println("ENGINES TO LEFT - WNEKA");
40            current_direction = CURRENT_DIR::MOVE_LEFT;
41            return CURRENT_DIR::MOVE_LEFT;
42        }
43        // The condition of leaving the recess - if the robot has a safe distance in front and on the side
44        if(!exitAlcove && dist_array[4] > SAFE_DIST && dist_array[3] > SAFE_DIST) {
45            Serial.println("EXITING ALCOVE, SWITCHING TO FORWARD MODE");
46            exitAlcove = true;
47            alcoveExitTime = currentTime;
48            current_direction = CURRENT_DIR::MOVE_FORWARD;
49            return CURRENT_DIR::MOVE_FORWARD;
50        }
51    }
52
53    //After leaving the alcove, drive straight for a certain amount of time
54    if (exitAlcove) {
55        if (currentTime - alcoveExitTime < FORWARD_TIME_AFTER_ALCOVE) {
56            Serial.println("ENGINES FORWARD AFTER ALCOVE");
57            current_direction = CURRENT_DIR::MOVE_FORWARD;
58            return CURRENT_DIR::MOVE_FORWARD;
59        } else {
60            Serial.println("RESUMING NORMAL MODE");
61            wneka = false;
62            exitAlcove = false;
63        }
64    }
65 }

```

Rysunek 26: Kod źródłowy realizujący wyznaczenie odpowiedniego sterowania

Funkcja sendSteering()

Funkcja ta realizuje wysyłanie wartości odczytanej z funkcji przetwarzającej dane z sensorów. Pierwszym krokiem jest rozpoczęcie transmisji z urządzeniem o adresie 0x09 - taki adres jest typowo przydzielany dla drugiego mikrokontrolera Arduino. Następnie za pomocą funkcji `Wire.write()` odbywa się właściwy przesył danych poprzez magistralę I2C.

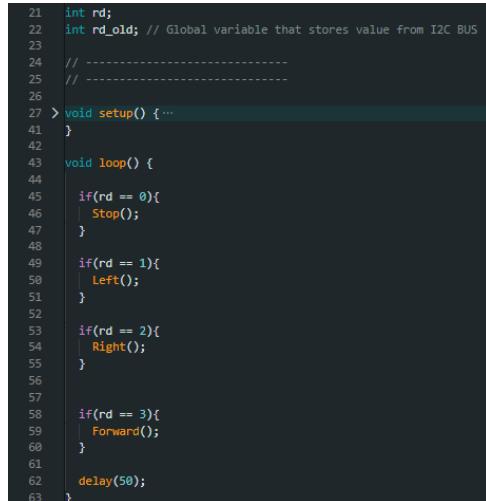


```
1 // This function is a very basic example of sending value to slave via I2C BUS
2 void sendSteering() {
3     Wire.beginTransmission(SLAVE_ADDR);
4     int temp = processSensorInfo();
5     Serial.println(temp);
6     Wire.write(temp);
7     Wire.endTransmission();
8 }
```

Rysunek 27: Kod źródłowy realizujący wysłanie instrukcji sterowania

2.11.2. Slave

Mikrokontroler Arduino UNO R3 skonfigurowany do pracy typu Slave odbiera paczkę danych przez zmienną "rd" wysyłane przez Master.



```
21 int rd;
22 int rd_old; // Global variable that stores value from I2C BUS
23 // -----
24 // -----
25 // -----
26
27 > void setup() { ...
41 }
42
43 void loop() {
44
45     if(rd == 0){
46         Stop();
47     }
48
49     if(rd == 1){
50         Left();
51     }
52
53     if(rd == 2){
54         Right();
55     }
56
57     if(rd == 3){
58         Forward();
59     }
60
61     delay(50);
62 }
```

Rysunek 28: Fragment kodu źródłowego przedstawiający odbiór komunikatów przesyłanych przez mikrokontroler Master.

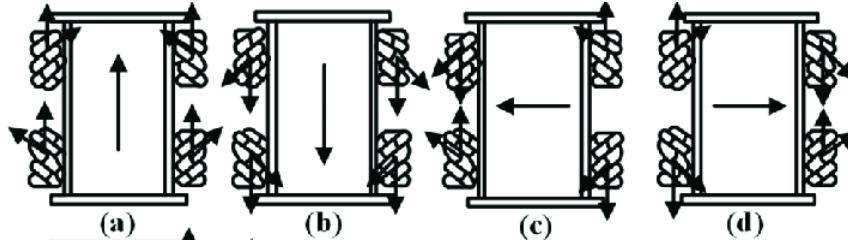
Zgodnie z odbieranym komunikatami o zmianie kierunku indeksowanymi od 0 do 3, program wywołuje odpowiednie funkcje.

```

88 void Right(){
89     |   motor1.run(BACKWARD); //LT
90     |   motor2.run(FORWARD); //PT
91     |   motor3.run(BACKWARD); //PP
92     |   motor4.run(FORWARD); //LP
93 }
94
95 void Left(){
96     |   motor1.run(FORWARD); //LT
97     |   motor2.run(BACKWARD); //PT
98     |   motor3.run(FORWARD); //PP
99     |   motor4.run(BACKWARD); //LP
100 }
101
102 void Forward(){
103     |   motor1.run(BACKWARD);
104     |   motor2.run(BACKWARD);
105     |   motor3.run(BACKWARD);
106     |   motor4.run(BACKWARD);
107 }
108
109 void Stop(){
110     |   motor1.run(RELEASE);
111     |   motor2.run(RELEASE);
112     |   motor3.run(RELEASE);
113     |   motor4.run(RELEASE);
114 }
115
116 // TODO:
117 // This function is responsible for calculating the steering
118 }
```

Rysunek 29: Fragment kodu źródłowego przedstawiający implementację ruchu pojazdu w odpowiednich kierunkach.

Funkcje określające kierunek jazdy zostały skonstruowane wedle logiki i geometrii działania kół szwedzkich.



Rysunek 30: Wizualizacja zależności kierunku jazdy od obrotu kół szwedzkich.

2.12. Testy poprawności działania algorytmu sterującego

3. Napotkane problemy

3.1. Podłączenie laserowych czujników odległości

Problematyczne okazało się użytkowanie więcej niż jednego czujnika laserowego przy wykorzystaniu magistrali I²C. Było to spowodowane domyślnym adresem, który był taki sam dla wszystkich czujników.

Naturalnym krokiem jest wykorzystanie funkcji `setAddress()`, ustawiającej inny niż domyślny adres dla czujnika. Niestety wykorzystanie tylko tej funkcji nie przyniosło oczekiwanych rezultatów, w wyniku czego należało szukać rozwiązań dalej.

Na podstawie dokumentacji oraz informacji znalezionych na forach internetowych, okazało się, że aby z sukcesem zmienić adres czujnika należy wykorzystać dodatkowy pin o nazwie XSHUT. Dzięki niemu czujnik zapamiętywał zmianę adresu i możliwe było wykorzystanie zamierzonej liczby sensorów.

Dokładny sposób podłączenia czujników oraz wykorzystania pinu XSHUT został opisany w sekcji nr 2.8.

3.2. Montaż silników oraz kół

Pierwszym problemem było zamontowanie samych silników. Okazało się, że dziury na mocowanie silników znajdująły się w innych miejscach niż było to potrzebne. W związku z tym, konieczne było wprowadzenie listwy aluminiowej, która pozwoliła na odpowiednie przymocowanie silników do dolnej części pojazdu.

Kolejnym wyzwaniem było mocowanie kół do silników. Początkowo używaliśmy długich śrub do przy-mocowania kół, jednak okazało się, że koła były bardzo niestabilnie zamocowane i miały spore luzy. Aby wyeliminować te luzy, wprowadziliśmy boczne listwy aluminiowe, przez które przechodzą długie gwintowane szprychy. To rozwiązanie znacznie poprawiło stabilność kół i zminimalizowało luzy, zapewniając lepsze działanie całego układu.

Dzięki tym działaniom udało się rozwiązać napotkane problemy i zakończyć montaż kół i silników w sposób zapewniający solidność i niezawodność konstrukcji.

3.3. Implementacja logiki sterowania

Realizacja logiki realizującej śledzenie ściany bez elementu opuszczania wnęki była realtywnie prosta. Jednak dodanie sekwencji wyjeżdżania z wnęki okazało się wyzwaniem.

Było tak ze względu na kolejność instrukcji warunkowych. Pierwotnie jeśli robot dojeżdzał do ściany jednocześnie mając z prawej swojej strony przeszkodę, poprawnie rozpoczynał ruch w lewo. Jednak po otrzymaniu informacji z sensora o pustej przestrzeni przed, robot od razu rozpoczynał ruch w prawo, ze względu na oddalenie się od prawej strony ściany. W wyniku czego pojazd jeździł tam i z powrotem w kółko.

Rozwiązaniem okazało się zastosowanie dodatkowego warunku sprawdzającego stan flagi `wneka` oraz wykorzystanie wymuszonego ruchu do przodu - opisany wcześniej licznik. Dzięki wykorzystaniu wymienionych ulepszeń, robot zarejestrowaniu pustej przestrzeni przed sobą, przez określony czas poruszał się przed siebie, w wyniku czego przynajmniej jeden z czujników z jego prawej strony otrzymywał otwartość uniemożliwiającą ruch w prawo.

4. Podsumowanie