# Painters

*czyli o dwóch takich, co się nie pozderzali*

Autorzy:
Bartłomiej Strózik
Adam Szreter

## 1. Założenia i cele projektu

Celem projektu było zaprogramowanie pary robotów mobilnych tak, aby zdołały przejechać narysowaną w programie graficznym i wgraną do ich pamięci trasę, skutecznie unikając kolizji. Podczas realizacji zadania roboty miały opierać się na znajomości własnej pozycji (na podstawie pomiaru obrotu kół za pomocą enkoderów) i komunikacji radiowej (w technologii Bluetooth Low Energy).

## 2. Zasoby

Nasze roboty oparliśmy o następujące zasoby sprzętowe:

- płytka ESP32 DevKitC V4 (z wbudowanym modułem bluetooth) (dokumentacja: ESP32, DevKitC-V4)
- podwozie Magician Chassis z silnikami Dagu DG01D-A130 GearMotor (dokumentacja: silniki)
- sterownik silników Pololu TB6612FNG (dokumentacja)
- moduł enkoderów z robota SparkFun RedBot (dokumentacja)

Do programowania robotów wykorzystaliśmy język C++ i środowisko Arduino, w tym Arduino IDE 1.8.9.

## 3. Repozytorium

### a) Adres

Projekt został upubliczniony przez umieszczenie go w publicznym repozytorium na GitHubie, dostępnym pod adresem: https://github.com/tyrrr-aj/Painters.git

### b) Struktura repozytorium

W głównym katalogu projektu znajdują się następujące pliki i katalogi:

- plik *Painters.ino* – jest to standardowy plik źródłowy platformy Arduino, zawierający główny kod wykonania programu (funkcje *setup* i *loop*).
- plik *Pins.h* – opisuje on odwzorowanie logicznych funkcji pinów w kodzie na ich fizyczne numery (zależne od zestawienia okablowania konkretnego egzemplarza robota)
- katalog *src* - zostały w nim umieszczone wszystkie biblioteki specyficzne dla projektu
- katalog *data* – przechowuje on pliki z trasami, jakie może przejechać robot

## 4. Fizyczna konstrukcja robotów

Każdy z robotów użytych w projekcie jest zbudowany na podwoziu Magician Chassis. Niezbędne połączenia elektryczne zrealizowaliśmy na płytkach stykowych (*breadboard*), choć docelowo najlepiej byłoby zastąpić je układami lutowanymi, np. na uniwersalnych płytkach [illegible]owniczych. Ograniczają się one de facto do połączenia odpowiednich pinów płytki ESP32 DevKitC z wejściami sterownika silników i rozprowadzenia zasilania, oprócz tego niezbędne jest podłączenie silników do wyjścia sterownika i enkoderów do pinów GPIO płytki. Przy podłączaniu silników, warto pamiętać, że w podwoziu Magician Chassis jeden z silników jest ustawiony „do góry nogami" – należy go wpiąć w sterownik na odwrót, niż wynikałoby to z dokumentacji (inaczej będzie się kręcił w przeciwną stronę, niż zakładamy).

Układ można zasilać na trzy sposoby – przez USB, stałym napięciem 5V lub stałym napięciem 3V. W praktyce oznacza to, że robot powinien być wyposażony albo w powerbank, albo koszyk na baterie – my wybraliśmy drugą z tych opcji, umieszczając w nim cztery akumulatory AA o napięciu 1.2V. Przy zasilaniu układu inaczej niż przez USB, warto pamiętać o dwóch rzeczach:

1. ESP32 DevKitC ma osobne piny do zasilania każdym z tych napięć
2. W każdym momencie, płytka może być zasilana z tylko jednego ze źródeł – w szczególności, przy programowaniu robota przez USB należy pamiętać o rozłączeniu zasilania bateryjnego (w naszym przypadku zapomnienie o tym nie spowodowało uszkodzenia płytki, ale nie znaczy to, że nie może)

Wybór metody zasilania jest dowolny, warto jednak pamiętać, że musi ono obsłużyć wykorzystane silniki – napięcie 3.3V może być do tego celu niewystarczające.

Przydatna dokumentacja:

- Omówienie, których pinów ESP32 można używać w jaki sposób, szczególnie użyteczna jest podsumowująca tabelka: https://randomnerdtutorials.com/esp32-pinout-reference-gpios/ (numery pinów na płytce DevKitC odpowiadają tym z tabelki/dokumentacji ESP32)
- Schemat wyprowadzeń sterownika silników TB6612FNG: https://botland.com.pl/pl/sterowniki-silnikow-moduly/32-pololu-tb6612fng-dwukanalowy-sterownik-silnikow-135v1a.html

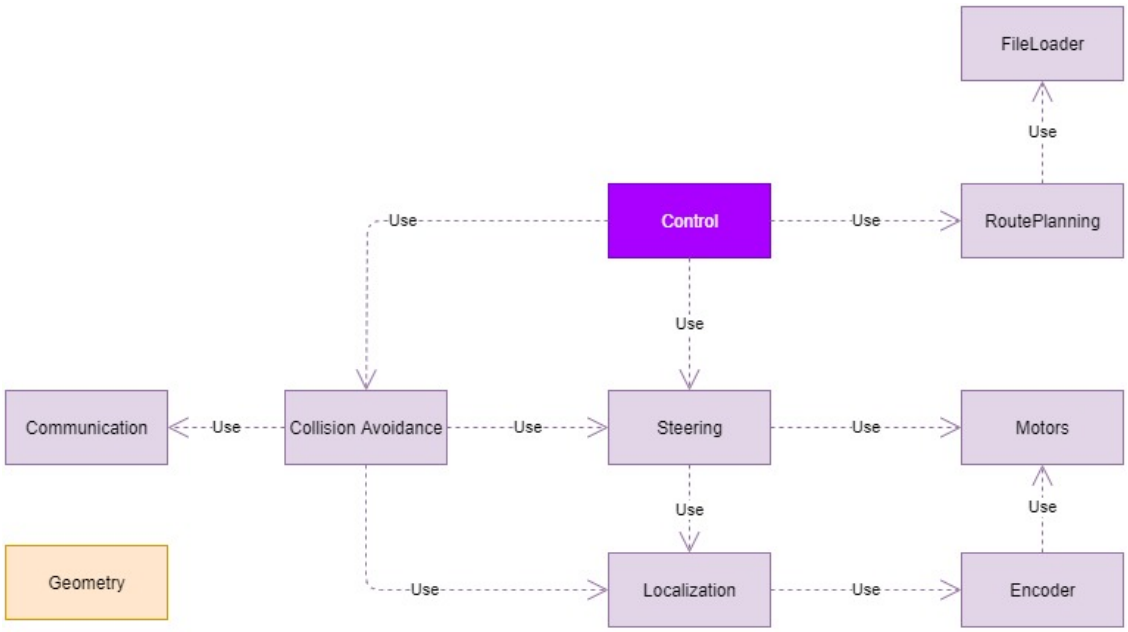
## 5. Konfiguracja środowiska

Skonfigurowanie środowiska Arduino IDE do pracy z ESP32 nie jest trudne – wystarczy dodać tę płytkę do Board Managera, co dobrze opisuje wykorzystany przez nas tutorial: https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/. Warto zwrócić uwagę na dwie rzeczy:

- Arduino IDE nie zapewnia żadnej obsługi bibliotek specyficznych dla projektu – w szczególności oznacza to, że wszelkie odnoszące się do nich *include'y* muszą uwzględniać przynajmniej ścieżkę względną
- Czas ładowania programu do pamięci mikrokontrolera jest przy domyślnych ustawieniach niemiłosiernie długi – pomaga zmiana wartości *baud* dla ładowania programu, co można

zrobić w pliku *preferences.txt* (aby go znaleźć, należy otworzyć Arduino IDE, i wybrać Plik->Preferencje-><hiperłącze do pliku, na dole okienka>). Źródło (dość oszczędne: https://github.com/nkolban/esp32-snippets/tree/master/Documentation/ideas/FastFlash

## 6. Struktura projektu




Projekt został zbudowany modułowo. Nadrzędnym komponentem jest klasa **Control**, pełniąca zasadniczy nadzór nad robotem. Jej zadaniem jest prowadzenie robota po zadanej trasie (otrzymanej z modułu **RoutePlanning**) – do czego wykorzystuje moduł **Steering**, i utrzymywanie poprawnego działania modułu **CollisionAvoidance**. Moduł ten, jak sama nazwa wskazuje, odpowiada za monitorowanie, czy robotowi nie grozi kolizja, i reagowanie, jeśli dojdzie do takiej sytuacji. Wykorzystuje w tym celu znajomość lokalizacji robota (**Localization**) i komunikację z partnerem (**Communication**).

Do kontroli nad ruchem robota zarówno klasa **Control**, jak i **CollisionAvoidance** wykorzystują moduł **Steering**. Udostępnia on funkcjonalności takie, jak jazda do określonego punktu, albo zatrzymanie robota i późniejsze wznowienie ruchu. Wykorzystuje on informacje o bieżącej lokalizacji i kursie (**Localization**) i wydaje polecenia silnikom (**Motors**).

Klasa **Localization** monitoruje aktualną względną pozycję i obrót robota – punkt odniesienia stanowi ustawienie początkowe. Posługuje się ona odczytami z enkoderów (**Encoder**). Moduł **Encoder** odpowiada za monitorowanie sygnałów z fizycznych enkoderów i udostępnianie liczby zliczonych ticków. Do realizacji tego zadania potrzebuje informacji o aktualnym kierunku ruchu, którą otrzymuje bezpośrednio od modułu **Motors** – odpowiedzialnego za kontrolę nad silnikami.

Zadaniem klasy **RoutePlanning** jest odczytanie trasy opisanej w pliku z programu graficznego i zbudowanie na jej podstawie uporządkowanej listy punktów, które robot musi odwiedzić. Do załadowania pliku z pamięci wykorzystuje klasę **FileLoader**.

Moduł **Geometry** jest użytkowym modułem, udostępniającym różnorodne funkcje związane z planimetrią, niecharakterystyczne dla projektu, ale niezbędne w wielu miejscach programu. Wiele z modułów projektu korzysta z różnych składowych modułu **Geometry** – powiązania te zostały pominięte na schemacie, aby nie zaciemniać obrazu.

# 7. Omówienie poszczególnych modułów

## a) Control

### Opis modułu

Klasa *Control* jest główną klasą, spaja wszystkie części projektu ze sobą, można powiedzieć, że łączy 3 składowe: składową rozplanowania trasy dla podwozia - *RoutePlanning*, składową *Steering* ze sterowaniem oraz nadzorem nad poruszaniem się robota oraz ostatnią składową: *CollisionAvoidance* unikania kolizji poprzez komunikację.

### Działanie modułu

Do modułu dostarczana jest trasa jako kolekcja punktów typu *Point* i wskaźnik do obiektu biblioteki z instrukcjami obsługi kolizji *Collision Avoidance*. Klasa *Control* na ich podstawie obsługuje sterowanie poprzez przekazywanie mu kolejnych destynacji. W programie jest wywoływana tylko jedna metoda tego modułu.

## b) Steering

### Działanie modułu

Odpowiedzialnością klasy *Steering* są matematyczne obliczenia służące do obliczenia kierunku i drogi jaką ma przebyć robot oraz wyznaczenia kierunku obrotu jaki ma on wykonać. Wykorzystuje się do tego własności wektorów i punktów matematycznych oraz geometrię analityczną, ściślej odejmowanie, dodawanie, tworzenie wektorów i iloczyn wektorowy. Posiadaną przez podwozie pozycję pobiera się używając klasy *Localization*.

Drugą odpowiedzialnością jest zmapowanie wyników powyższych obliczeń na odpowiednie instrukcje dla silników, gdyż moduł ten współpracuje z klasą *Motors*. Polega ono na uaktywnieniu odpowiedniej akcji silników i kontrolowania ruchu do momentu osiągnięcia celu. Podczas jazdy za każdym razem aktualizuje się pozycję w *Localization*, która służy potem do sprawdzenia osiągnięcia celu.

### Proces tworzenia

Według początkowej koncepcji klasa Steering miała przekazywać liczone kąty i odległości do agregowanej klasy Chassis, w której mielibyśmy zarówno lokalizację jak i silniki, a poruszanie się polegałoby na przeskalowaniu odległości lub kąta na czas obrotu. Oczywiście szybko się zreflektowaliśmy, ze jest to niewykonalne, a przede wszystkim – za dużo odpowiedzialności w jednej klasie.

Szybko zmieniliśmy klasę Chassis na klasę Localization, którą ograniczyliśmy do samego przechowywania pozycji. Klasa Steering stała się w pełni odpowiedzialna za sterowaniem silnikami – co jest dla niej bardzo naturalne, bo to ona wie dokładnie gdzie jest, dokąd zmierza i jak szybko się

porusza podwozie. Cecha ta umożliwia zatrzymywanie podwozia albo obracanie nim zależnie od sytuacji.

Obliczenia matematyczne jako pierwszy etap „działania" tej klasy były również przeniesione w inne miejsce (moduł „Angle") i to rozwiązanie było być może lepsze od obecnego. Pozwoliłoby na przeniesienie wszelkich wzorów matematycznych i uzyskiwanie wektorów przemieszczenia, czy kątów z jednego miejsca. Zrezygnowaliśmy z tej klasy, gdyż w naszym przypadku wzory były bardzo nieskomplikowane, a najwięcej pracy w tej klasie było poświęcone liczeniu kąta, które później okazało się nie być konieczne.

Ostatecznie klasa Steering przechowuje dwa wektory: wektor przemieszczenia (pomiędzy punktem źródłowym, a docelowym) wektor położenia celu (czyli zaczepiony w punkcie (0,0) ukł. wsp. i kończący się na punkcie docelowym). Dzięki pierwszemu ustala się zwrot podwozia, do którego się dąży, a dzięki drugiemu przeprowadza się jazdę do momentu osiągnięcia celu.

## c) Motors

### Działanie modułu

Klasa *Motors* tworzy warstwę abstrakcji nad silnikami. Udostępnia typowy zestaw metod, reprezentujących możliwe polecenia dla silnika – obrót do przodu/do tyłu z zadaną mocą i zatrzymanie (z aktywnym hamowaniem – *stop()* i bierne wyłączenie silnika – *coast()*). Detale obsługi silników są ukryte w prywatnych metodach, które można podmienić w razie użycia innego sprzętu.

Opcjonalną odpowiedzialnością obiektu *Motors* jest przekazywanie obiektowi *Encoder* informacji o tym, w którą stronę aktualnie kręci się dane koło – aby to robił, wystarczy wywołać na nim metodę *addEncoder()*.

### Proces tworzenia

Kluczem do łatwego napisania tej biblioteki okazał się wybór odpowiedniego fizycznego sterownika silników – użyty przez nas TB6612FNG pozwala sterować każdym z silników za pomocą trzech pinów. Dwa z nich (oznaczone tu jako IN1, IN2) należy ustawić w stan wysoki/niski, ustawiając w ten sposób kierunek obrotu silnika, a na trzeci podać sygnał PWM[1], sterujący mocą silnika. Dokładną tabelkę ze stanami pinów i reakcją silnika najłatwiej znaleźć na stronie jednego ze sprzedawców: https://botland.com.pl/pl/sterowniki-silnikow-moduly/32-pololu-tb6612fng-dwukanalowy-sterownik-silnikow-135v1a.html

Do generowania fali PWM wykorzystaliśmy wbudowane mechanizmy ESP32 – są one bardzo przystępnie opisane np. tutaj: https://randomnerdtutorials.com/esp32-pwm-arduino-ide/.

Naszą bibliotekę oparliśmy o oficjalną bibliotekę dla robota RedBot firmy SparkFun Electornics, opartego w znacznej mierze o podobne komponenty (w szczególności – ten sam sterownik silników). Można ją znaleźć tutaj: https://github.com/sparkfun/RedBot/archive/master.zip (po pobraniu i rozpakowaniu zipa, należy otworzyć plik *Libraries/Arduino/src/RedBotMotors.cpp.*

[1] PWM – fala prostokątna, w której informację niesie stopień wypełnienia fali (ang. *duty cycle*)

### Ogólna uwaga do obsługi pinów

Wygodną obsługę pinów zapewniła nam biblioteka *Arduino.h*. Wszystkie powiązania fizycznych pinów z nazwami używanymi w kodzie definiuje plik *pins.h* w katalogu głównym projektu

## d) Encoder

### Działanie modułu

Moduł enkodera ma za zadanie zliczać impulsy (*ticki*) przychodzące z fizycznych enkoderów, w ten sposób kontrolując, o ile obróciło się każde z kół robota (każdy z nich odpowiada przekręceniu się zębatki na wspólnym wale z kołem o jeden ząbek).

Zliczanie oparte jest o przerwania – za każdym razem, gdy w sygnale z enkodera pojawi się opadające zbocze, w mikroprocesorze generowane jest przerwanie (funkcjonalność tę osiągnęliśmy dzięki ustawieniu pinów, do których podłączone są enkodery, jako pinów zewnętrznych przerwań). W reakcji na takie zdarzenie, wywoływana jest funkcja callback, zwiększająca lub zmniejszająca (zależnie od kierunku obrotu koła) wartość odpowiedniego licznika o 1. Liczniki te są zadeklarowane z użyciem słowa kluczowego *volatile*[2]. Obiekt enkodera stale przechowuje aktualny kierunek ruchu każdego koła – informacja ta jest aktualizowana przez klasę *Motors*, kiedy tylko zostanie wydane polecenie zmiany kierunku (wywołuje ona udostępnianą metodę *setDirection()*).

W rzeczywistym świecie sygnał z enkodera nie jest gładki – szumy w okolicach wartości granicznej między logicznym zerem i jedynką tworzą wiele fałszywych zboczy, błędnie interpretowanych jako kolejne *ticki*. Aby zaradzić tej sytuacji, niezbędny jest *debouncing*[3] sygnału – zrealizowaliśmy go programowo, wprowadzając minimalne opóźnienie pomiędzy sygnałami (wszelie impulsy, które przyjdą przed upływem tego czasu od pierwszego impulsu, zostaną zignorowane).

### Przebieg tworzenia

Przy tworzeniu modułu ponownie oparliśmy się na oficjalnej bibliotece SparkFun'a stworzonej dla RedBota (ponownie, używamy tych samych enkoderów). Można ją odnaleźć w tym samym katalogu, co bibliotekę do obsługi silników – czyli, gwoli przypomnienia, w zipie możliwym do ściągnięcia pod adresem https://github.com/sparkfun/RedBot/archive/master.zip (plik *Libraries/Arduino/src/RedBotEncoder.cpp*).

Enkodery przysporzyły nam chyba najwięcej trudności w całym projekcie, a ich działanie do teraz nie jest zbyt dobre. Podstawowym problemem jest tutaj fizyczna rozdzielczość pomiaru – przy zębatkach o 16 ząbkach, umieszczonych na wspólnym wale z kołem (**za** przekładnią), otrzymana dokładność jest zwyczajnie niewystarczająca. Kompletując części do budowy robota, warto zaopatrzyć się w enkodery zakładane na wał silnika (i odpowiednie do tego silniki, z wyprowadzonym

---

[2] *volatile* – to słowo kluczowe przed nazwą zmiennej informuje, że jej wartość może nieoczekiwanie zmienić się, nawet jeśli nie wynika to z wykorzystującego je kodu, w związku z czym nie powinna nigdy być odczytywane z cache'a, a odwołania do jej wartości nie mogą być optymalizowane przez kompilator. W naszym przypadku, taka zmiana może mieć miejsce w funkcji obsługi przerwania – może ona zostać wywołana kiedykolwiek i jest to niemożliwe do przewidzenia przez „zwykły" kod korzystający ze zmiennej

[3] d*ebouncing* – filtrowanie z sygnału fałszywych odczytów, powstałych na skutek szumu w okolicy wartości granicznej (miedzy logicznym zerem i jedynką). Może być realizowany sprzętowo (obwód RC) lub programowo (tymczasowa blokada po pierwszym odczycie)

wystarczająco długim kawałkiem wału) – pomiar jest wówczas wykonywany **przed** przekładnią, co zwiększa rozdzielczość pomiaru kilkudziesięciokrotnie (np. przy przekładni w naszych silnikach – 1:48 - zwiększyłoby to ją 48 razy, co nawet przy założeniu mniejszej liczby *ticków* na pojedynczy obrót (enkodery montowane na silnikach są znacznie mniejsze) dałoby najpewniej ok. 10-krotny zysk rozdzielczości).

Kolejną kwestią jest jakość sygnału – ten pochodzący z naszych enkoderów jest bardzo silnie zaszumiony, co wymusiło na nas ustawienie bardzo długiego opóźnienia dla *debouncingu* – 50 ms, czyli czasu tego samego rzędu co częstotliwość odczytu wykonywanego przez moduł *Localization*. Tak kiepska jakość sygnału jest związana z samą zasadą działania naszych enkoderów – oparte są o czujnik optyczny, który powinien reagować na zmiany natężenia docierającego światła kiedy jest przysłaniany przez ząbek zębatki. Takie urządzenie do dokładnego działania wymagałoby bardzo dokładnego ustawienia szczeliny między czujnikiem a zębatką (musi być minimalna) i bardzo dobrych warunków oświetleniowych. Niewielką poprawę jakości sygnału można by prawdopodobnie osiągnąć, stosując zębatki o kontrastowych ząbkach i powierzchniach między nimi – użyte przez nas (dostarczone przez producenta podwozia) zębatki są monochromatyczne. Gdyby udało się uzyskać naprawdę czysty sygnał, rozdzielczość pomiaru można by dwukrotnie zwiększyć, reagując zarówno na wznoszące, jak i opadające zbocze sygnału (początek i koniec ząbka, a nie tylko fakt jego pojawienia się) – w naszym przypadku podobne próby nie przyniosły jednak żadnych rezultatów.

## e) Localization

### Działanie modułu

Moduł *Localization* odpowiada za śledzenie aktualnej pozycji robota na podstawie odczytów z enkoderów. Zapytany o aktualne położenie/obrót odpytuje on moduł *Encoder* o zmianę w liczbie *ticków* dla każdego z kół, i na tej podstawie wylicza zmianę odpowiedniej wielkości (ważne: tylko jednej z nich – tej, o którą nadeszło zapytanie). Rotacja jest przechowywana i zwracana w postaci znormalizowanego wektora wskazującego w odpowiednim kierunku. Punktem odniesienia dla określania lokalizacji jest początkowe ustawienie robota – moduł przyjmuje, że na początku znajduje się on w punkcie (0, 0), i jest zwrócony w kierunku dodatnim osi OX (rotacja równa (1, 0)).

### Przebieg tworzenia

Wobec słabej dokładności pomiaru z enkoderów (patrz wyżej) sposób obliczania lokalizacji musiał zostać uproszczony – zamiast częstego odpytywania enkoderów o niewielkie zmiany obrotu kół i jednoczesnej aktualizacji położenia i obrotu robota, w wyniku danego zapytania zmieniana jest tylko jedna z wielkości; pierwsze podejście prowadziło do tego, że obracając się w miejscu, robot gubił położenie, a jadąc po prostej, myślał, że jednocześnie się obraca. Moduł *Localization* został wobec tego dostosowany do uproszczonego sposobu działania modułu Steering (patrz wyżej), i w każdym momencie zakłada, że robot albo obraca się w miejscu, albo porusza do przodu po linii prostej; sytuacje te rozróżnia po tym, o jaką wielkość padło zapytanie uruchamiające obliczenia (odpowiednio położenie lub rotacja).

## f) FileLoader

### Działanie modułu

Moduł *FileLoader* udostępnia najbardziej podstawowy podzbiór operacji dostępu do pliku, tj. otwórz, odczytaj następną linię, zamknij. Pliki rezydują w pamięci flash mikrokontrolera, a dokładniej – są do niej wgrywane wraz z programem. W ESP32 dostęp do nich zapewnia SPIFFS – prosty system plików, pozwalający na operacje otwierania, zamykania, odczytu i usuwania plików. Pliki w tym systemie przechowywane są w płaskiej strukturze – (na ten moment) SPIFFS nie wspiera katalogów.

### Przebieg tworzenia

Moduł korzysta z biblioteki *SPIFFS.h*, zawartej w wtyczce ESP32 dla Arduino IDE (aby jej użyć, wystarczy umieścić w kodzie linijkę *#include „SPIFFS.h”*). Obsługa biblioteki jest prosta, i w zasadzie do jej poznania (przynajmniej w wykorzystywanym tu zakresie) wystarczają przykłady zawarte w tym tutorialu: https://randomnerdtutorials.com/install-esp32-filesystem-uploader-arduino-ide/ (oficjalna dokumentacja też oczywiście istnieje, ale nie jest tak przyjazna w poznaniu podstaw – opisuje za to znacznie więcej szczegółów. Można ją znaleźć tutaj: https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/storage/spiffs.html).

Zasadniczym tematem przytoczonego wyżej tutoriala jest wtyczka dla Arduino IDE, pozwalająca wrzucać pliki z komputera bezpośrednio do pamięci mikrokontrolera. Instalacja i użycie wtyczki zostały tam całkiem dobrze opisane. Wykorzystaliśmy ją w naszej pracy nad projektem. Podstawowe użycie wtyczki polega na umieszczeniu wszystkich plików, które chcemy wgrać do pamięci mikrokontrolera w podkatalogu */data* w folderze szkicu Arduino, a następnie uruchomieniu funkcji *Narzędzia/ESP32 Sketch* Data Upload.

**Ważne:** W naszym przypadku instalacja wtyczki sprawiła, że środowisko Arduino IDE przestało działać w pełni stabilnie – po (prawie) każdym podłączeniu płytki do komputera, pierwsza próba wgrania programu kończy się niezrozumiałym wyjątkiem. Prawdopodobnie jest to błąd we wtyczce. W naszym przypadku wystarcza ponownie spróbować wgrać kod – druga próba za każdym razem kończyła się powodzeniem.

## g) RoutePlanning

### Opis modułu

Moduł *RoutePlanning* odpowiada za przetworzenie zbioru punktów wczytanych z pliku na uporządkowaną kolekcję (reprezentowanej przez std::vector z C++) tych samych punktów. Ma to na celu ułożenie trasy, po której będzie poruszał się robot.

### Działanie modułu

Klasa *RoutePlanning* agreguje obiekt FileLoadera, który umożliwia mu operacje na plikach. Jedną z odpowiedzialności jest tutaj sparsowanie pliku tekstowego wygenerowanego przez program graficzny do zaznaczania punktów w układzie współrzędnych do listy obiektów klasy *RpPoint*.

Do generowania wykorzystaliśmy darmową, wieloplatformą aplikację Wykresy z pakietu dynamicznego oprogramowania GeoGebra. Można je pobrać ze strony https://www.geogebra.org/download?lang=pl.

W planowaniu trasy korzysta się z metody *getPath()*, która tworzy obiekt klasy *Solver* i przekazuje mu kolekcję punktów.

**Działania Solvera**

*Solver* wybiera punkt, od którego ma zacząć, a następnie szuka punktów najbliższych mu w metryce euklidesowej.

Algorytm Solvera jest zoptymalizowany o obszary zagęszczenia punktów. Nie pozwala on na pozostawienie obszarów gęstych punktów, do których „kiedyś się wróci". Z tego powodu wybór punktów zaczynany jest od wykrojenia obszaru o promieniu w odległości od obecnego punktu.

W wykrojonym obszarze buduje się trasę. Każdy następny punkt trasy to najbliższy w metryce euklidesowej punkt w stosunku do punktu obecnego. Z obszaru wybierane są wszystkie punkty, czyli zanim zakończy się etap związany z obszarem musi on być w całości pokryty.

Gdy wykrojony obszar zostanie pokryty, tworzy się kolejny obszar. Startujemy z ostatniego punktu z właśnie pokrytego obszaru.

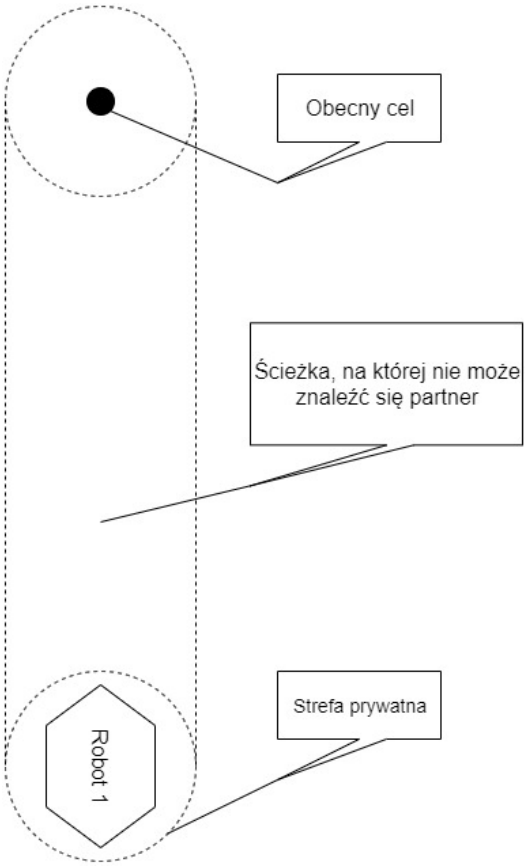
## h) CollisionAvoidance

### Działanie modułu

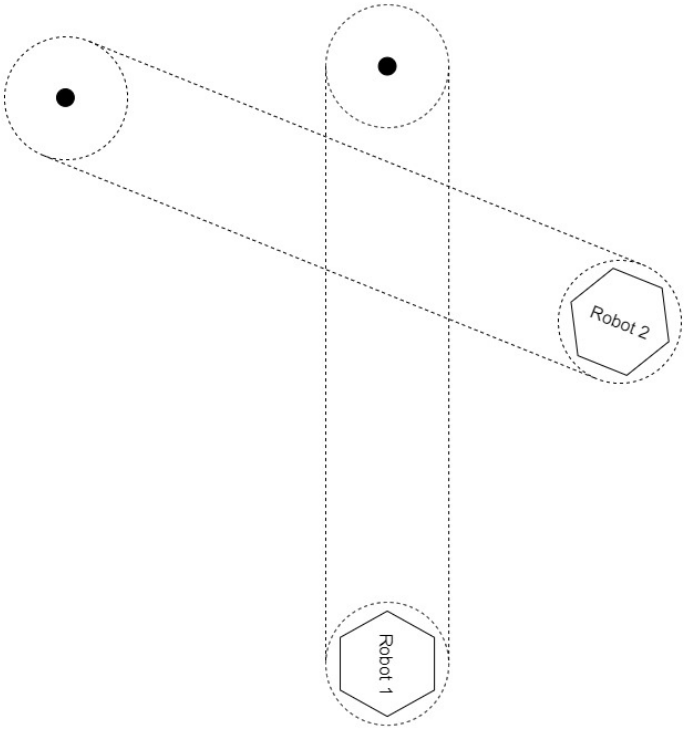
Zadaniem modułu *CollisionAvoidance* jest zapewnienie, że jeżdżące roboty nie zderzą się ze sobą. Czyni to w sposób aktywny, komunikując się z partnerem i ustalając, czy obrane ścieżki robotów nie doprowadzą do kolizji.

Z punktu widzenia tego modułu robota otacza pewna strefa prywatna – jest to okrąg o środku w obecnej lokalizacji robota, w którym mieści się cały pojazd. Za kolizję uznawany jest stan, w którym strefy prywatne dwóch robotów najdą na siebie (będą miały punkty wspólne).

Za każdym razem, gdy robot ruszy w stronę kolejnego punktu na ścieżce, *CollisionAvoidance* ogłasza jego aktualne położenie i nowy cel partnerowi. Ten, znając położenie i cel własnego robota, może określić, czy na badanym odcinku trasy (pomiędzy obecnym położeniem każdego z robotów i jego obecnym celem) wyznaczone ścieżki będą miały jakieś punkty wspólne (z dokładnością do promienia strefy prywatnej – można go traktować jako szerokość ścieżki). Jeżeli tak, uruchamiany jest algorytm reakcji na zagrożenie kolizją.

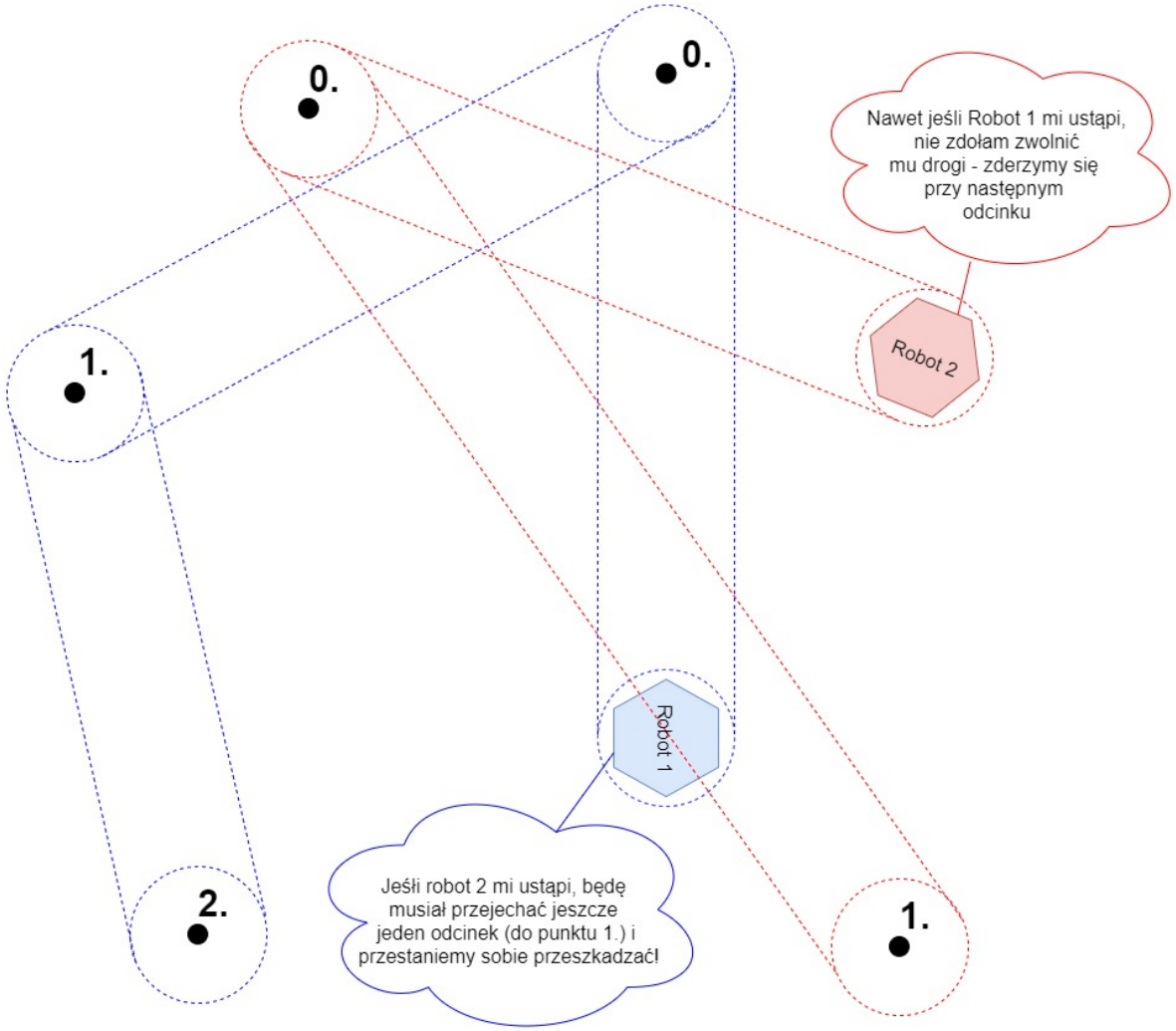



Rysunek 1 Podstawowe koncepcje



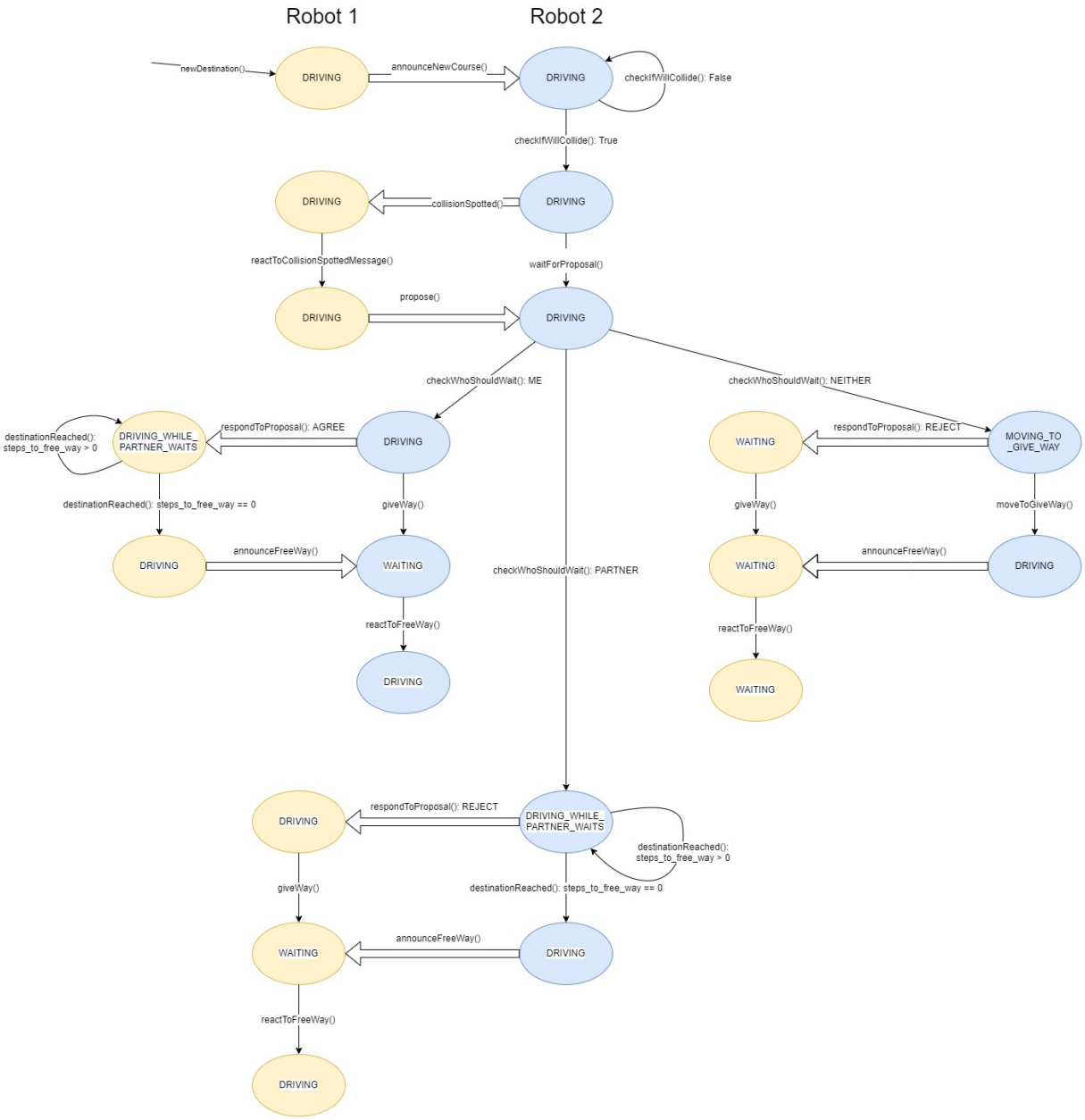

Rysunek 2 Przykład kolizji

Gdy robot wykryje potencjalną kolizję, po pierwsze ostrzega o tym partnera. Następnie każdy z robotów, znając swoją zaplanowaną dalszą ścieżkę, sprawdza, czy jeśli jego partner mu ustąpi (zatrzyma się), to jego ruch doprowadzi do rozwiązania sytuacji, a jeśli tak, to w ilu krokach (krok jest tutaj rozumiany jako przejazd do kolejnego punktu na trasie, a rozwiązanie – za sytuację, kiedy na trasie do kolejnego punktu swoich tras roboty nie będą zagrożone kolizją). Sprawdzanie jest natychmiast przerywane z wynikiem negatywnym, jeżeli na kolejnym sprawdzanym odcinku nastąpi kolizja z aktualnym położeniem partnera, a nie tylko z jego ścieżką (oznacza to, że robot nie będzie w stanie przejechać do następnego punktu, i ustąpienie przez partnera nic nie da).




**Rysunek 3 Przykład rozstrzygania kolizji**

Po zakończeniu sprawdzania, jeden z robotów (ten, który zainicjował całą wymianę komunikatów) przesyła drugiemu swój wynik (liczbę kroków), którą ten porównuje z własnym wyliczeniem i odpowiada decyzją co do dalszego postępowania pary. Prawo do kontynuowania ruchu zachowuje robot, który zaproponował niższą wartość liczby kroków do rozwiązania konfliktu – jego partner musi zatrzymać się i poczekać, aż tamten oznajmi o przejechaniu zadeklarowanej liczby punktów. Jeżeli obydwa roboty zgłoszą, że ustąpienie przez partnera nie rozwiąże sprawy, zatrzymają się i zakończą swoje działanie – ten sposób postępowania jest jednak tymczasowy, w ostatecznej implementacji jeden z nich powinien wyznaczyć najbliższy punkt pozwalający objechać partnera i wyminąć się z nim, możliwie jak najmniej zbaczając przy tym ze swojej ścieżki.




**Rysunek 4 Schemat wysyłanych komunikatów i zmian stanu robotów w czasie pojedynczej sesji komunikacyjnej**

# i) Communication

## Działanie modułu

Moduł *Communication* działa w oparciu o architekturę klient-serwer. Udostępnia metodę *setUpCommunication()*, która ustawia komunikację i tworzy sesję połączenia pomiędzy klientem, a serwerem. Za każdym razem tworzony jest serwer, a po nim przeprowadza się skanowanie w poszukiwaniu innego serwera. Jeżeli zostanie wykryty serwer o odpowiednim identyfikatorze oznaczającym drugiego robota – drugie podwozie, tworzony jest klient, który podłącza się do tego serwera.

Serwer i klient mają zawsze uwspólniony identyfikator charakterystyki, nazwę serwera oraz posiadają wskaźnik do charakterystyki, która przechowuje przesyłaną wartość między nimi.

Do utworzenia serwera i klienta w technologii Bluetooth Low Energy wykorzystaliśmy wolną bibliotekę zamieszczoną na githubie: https://github.com/espressif/arduino-esp32.

W samym zrozumieniu architektury i konstrukcji obiektów serwera i klienta pomógł nam tutorial https://randomnerdtutorials.com/esp32-bluetooth-low-energy-ble-arduino-ide/. Odkrytym przez nas nieco później, ale bezcennym źródłem wiedzy w tej materii jest oficjalna dokumentacja biblioteki BLE dla ESP32: https://github.com/nkolban/esp32-snippets/blob/master/Documentation/BLE%20C%2B%2B%20Guide.pdf (**uwaga**: wykorzystanie biblioteki jest tutaj wyjaśnione bardzo dobrze, ale w nazwach metod w przykładach są błędy – warto je konfrontować z plikami nagłówkowymi biblioteki, które można znaleźć tutaj: https://github.com/espressif/arduino-esp32/tree/master/libraries/BLE/src. Generalnie warto zapoznać się z tymi plikami, klasy i metody są bardzo dobrze ponazywane i można z nich wyciągnąć wiele informacji jak rzeczywiście używać biblioteki, trudnych do znalezienia gdzie indziej).

Poza tym klasa *Communication* jest odpowiedzialna za udostępnienie funkcji związanych z przesyłaniem konkretnych komunikatów. Każde przesłanie komunikatu odbywa się na zasadzie umieszczenie wartości w charakterystyce po stronie serwera, a następnie odczytania tej wartości po stronie klienta. Wykorzystaliśmy prostą aplikację, która tworzy serwer i wywołuje funkcję notify(), dzięki czemu może nastąpić pewne przerwanie lub po prostu powiadomienie u klienta. Pomógł nam w tym drugi tutorial: https://www.esp32.com/viewtopic.php?t=4181.

**Funkcje służące do przesyłania komunikatów:**

- *announceNewCourse(*Point*,* Point*)* – funkcja przesyłająca dwa punkty partnerowi wskazujące aktualną pozycję oraz cel, do którego zmierza robot
- *signalCollision(*Point*,* Point*)* – funkcja oddająca robotowi swój własny kierunek (czyli punkt startowy oraz cel) oraz sygnalizująca kolizję – sygnalizacja kolizji odbywa się po tym jak zostanie sprawdzone, czy obszary bliskie robotom w jakiś sposób się przecinają
- *propose(*int*)* – funkcja przesyłająca liczbę kroków, w którym robot mógłby ominąć drugiego robota (to znaczy, jeśli partner się zatrzyma to ile punktów musi przejechać, aby zwolnić partnerowi drogę). Jest ona jednocześnie propozycją, żeby partner zatrzymał się
- *respondToProposal(*RespondToProposal*)* – funkcja przesyłająca odpowiedź na propozycję do ustąpienia partnerowi. Najpierw wykonuje obliczenia w ilu krokach przeciwny robot mógłby ominąć partnera. Po obliczeniu liczby kroków odsyła zgodę na propozycję jak jest ona większa albo odrzucenie i jej i rozkazanie partnerowi zaczekać jak jest mniejsza
- *announceFreeWay()* – funkcja przesyłająca komunikat o zwolnieniu drogi przez jedno z podwozi. Przesyłana po tym jak robot przejedzie tyle punktów ile mu było potrzeba do rozłączenia obszarów intymnych, które roboty miały dookoła siebie