Painters

czyli o dwóch takich, co się nie pozderzali

Autorzy:  
Bartłomiej Strózik  
Adam Szreter

# 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).

# Zasoby

Nasze roboty oparliśmy o następujące zasoby sprzętowe:

* płytka ESP32 DevKitC V4 (z wbudowanym modułem bluetooth) (dokumentacja: [ESP32](https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf), [DevKitC-V4](https://www.espressif.com/sites/default/files/documentation/esp32-devkitc-v4_reference_design.zip))
* podwozie Magician Chassis z silnikami Dagu DG01D-A130 GearMotor (dokumentacja: [silniki](https://botland.com.pl/pl/index.php?controller=attachment&id_attachment=363))
* sterownik silników Pololu TB6612FNG ([dokumentacja](https://botland.com.pl/pl/index.php?controller=attachment&id_attachment=83))
* moduł enkoderów z robota SparkFun RedBot ([dokumentacja](https://botland.com.pl/pl/index.php?controller=attachment&id_attachment=520))

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

# Repozytorium

## Adres

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

## 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

# 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 lutowniczych. 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:

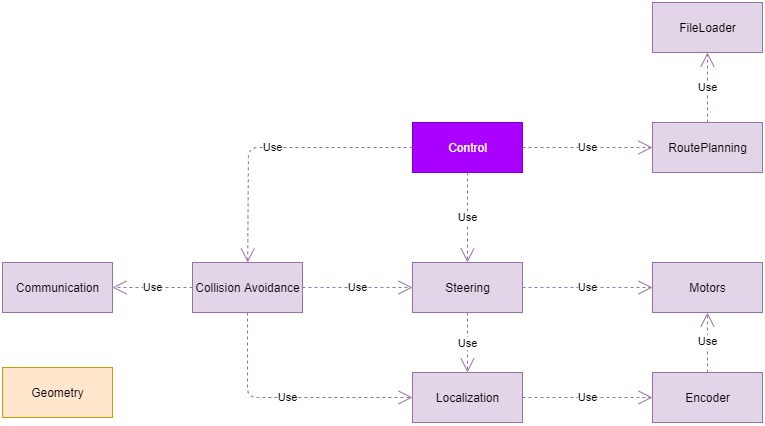
1. ESP32 DevKitC ma osobne piny do zasilania każdym z tych napięć
2. W każdym momencie, płytka może być zasilana z tylko jednego ze źródeł – w szczególności, przy programowaniu robota przez USB należy pamiętać o rozłączeniu zasilania bateryjnego (w naszym przypadku zapomnienie o tym nie spowodowało uszkodzenia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

# 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.

# Omówienie poszczególnych modułów

## Control

## Steering

## 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]](#footnote-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.*

### 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

## 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]](#footnote-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]](#footnote-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 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.

## 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).

## 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](https://randomnerdtutorials.com/install-esp32-filesystem-uploader-arduino-ide/) 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.

## RoutePlanning

## CollisionAvoidance

## Communication

1. PWM – fala prostokątna, w której informację niesie stopień wypełnienia fali (ang. *duty cycle*) [↑](#footnote-ref-1)
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 [↑](#footnote-ref-2)
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) [↑](#footnote-ref-3)