

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: AUTOMATYKA I ROBOTYKA (AIR)
SPECJALNOŚĆ: TECHNOLOGIE INFORMACYJNE W SYSTEMACH
AUTOMATYKI (ART)

PRACA DYPLOMOWA
INŻYNIERSKA

Aplikacja mobilna do sterowania
robotem minisumo

Mobile application for controlling
a minisumo robot

AUTOR:

Łukasz Miłaszewski

PROWADZĄCY PRACĘ:

dr inż. Łukasz Jeleń

OCENA PRACY:

Spis treści

1. Wstęp	7
1.1. Cel i założenia	8
2. Wykorzystane technologie	9
2.1. C	9
2.1.1. HAL	9
2.2. Swift	10
2.2.1. UIKit	11
2.2.2. CoreBluetooth	11
2.2.3. CoreGraphics	11
2.2.4. CoreMotion	12
2.3. Arduino	12
3. Komunikacja	14
3.1. Moduł bluetooth	14
3.2. Realizacja komunikacji	15
4. Robot minisumo	18
4.1. Konstrukcja	19
4.1.1. Założenia	19
4.1.2. Nadwozie	19
4.1.3. Podwozie	19
4.1.4. Napęd	20
4.2. Elektronika	21
4.2.1. Układ zasilania	22
4.2.2. Procesor	22
4.2.3. Sensoryka	23
4.2.4. Sterownik silników	23
4.2.5. Schemat elektroniki	23
4.3. Oprogramowanie	25
4.3.1. Transmisja danych	25
4.3.2. Sterowanie silnikami	27

4.3.3. Obsługa czujników	27
5. Aplikacja mobilna	28
5.1. Kompatybilność	28
5.2. Wzorzec MVC	28
5.3. Komunikacja	29
5.4. Struktura aplikacji	31
5.4.1. Panel powitalny	31
5.4.2. Panel główny	34
5.4.3. Panel sterowania automatycznego	34
5.4.4. Panel sterowania zdalnego	36
5.4.5. Panel diagnostyki	41
6. Implementacja	47
6.1. Kompilacja projektu	47
7. Podsumowanie	48
7.1. Zrealizowane założenia	48
7.2. Dalszy rozwój projektu	48
7.3. Uwagi	48
Bibliografia	49

Spis rysunków

1.1.	Zawody sumo [4]	7
2.1.	Konfiguracja peryferiów użytego procesora STM32F100C8T6B – LQFP48.	10
2.2.	Środowisko Xcode.	10
2.3.	Układ służący do konfiguracji modułu bluetooth.	12
3.1.	Schemat komunikacji między urządzeniami.	14
3.2.	Moduły Bluetooth.	15
3.3.	Schemat wiadomości zawierającej informacje odnośnie stanu sensorów.	16
3.4.	Schemat wiadomości w której zawarta jest konfiguracja robota.	16
3.5.	Schemat wiadomości zdalnego sterowania przy użyciu akcelerometru.	17
4.1.	Wykonany robot minisumo.	18
4.2.	Projekt nadwozia w środowisku Autodesk Inventor 2017.	19
4.3.	Projekt podwozia w środowisku Autodesk Inventor 2017.	20
4.4.	Napęd robota minisumo.	21
4.5.	Porównanie wykonanych płyt drukowanych.	22
4.6.	Schemat elektroniki.	24
5.1.	Podział ról we wzorcu MVC.	29
5.2.	Struktura plików aplikacji mobilnej.	31
5.3.	Widok połączenia.	32
5.4.	Widok dostępnych urządzeń.	32
5.5.	Panel główny aplikacji mobilnej.	34
5.6.	Panel sterowania automatycznego aplikacji mobilnej.	35
5.7.	Panel sterowania zdalnego akcelerometrem.	36
5.8.	Osie X, Y oraz Z określające orientację urządzenia [5].	37
5.9.	Panel sterowania zdalnego dżojstikiem.	38
5.10.	Wirtualny dżojstik.	39
5.11.	Widok zakładek w panelu diagnostyki.	41
5.12.	Widok diagnostyki czujników.	42
5.13.	Poprawność działania diagnostyki czujników.	43
5.14.	Widok diagnostyki silników.	44

5.15. Widok funkcji czyszczenia opon.	46
---	----

Spis listingów

2.1.	Kod programu umożliwiającego konfigurację modułu Bluetooth HM-10.	13
4.1.	Funkcja obsługująca przerwanie.	26
4.2.	Sterowanie mocą silników.	27
4.3.	Obsługa czujników przeciwnika.	27
5.1.	Protokół odpowiedzialny za komunikację między urządzeniami.	30
5.2.	Implementacja protokołu.	33
5.3.	Klasa Algorithm.	35
5.4.	Implementacja metody startAcc.	37
5.5.	Metoda klasy JoystickViewController.	40
5.6.	Parsowanie wiadomości zawierającej stan czujników.	43
5.7.	Nasłuchiwanie zmiany położenia suwaka.	45
5.8.	Nasłuchiwanie zmiany położenia przełącznika.	46

Rozdział 1

Wstęp

Minisumo jest jedną z kategorii walk robotów wzorowanych na popularnym japońskim sporcie – zapasach sumo. Tak samo jak i w prawdziwym sporcie, starcie odbywa się na okrągłym ringu. Wygrywa ten robot, który jako pierwszy wypchnie rywala z areny. Obowiązujące zasady są takie same dla każdej z kategorii (sumo, minisumo, nanosumo, pentosumo) z wyjątkiem dopuszczalnej masy oraz rozmiaru. Dla minisumo maksymalna masa to 500 gramów, a szerokość oraz długość nie mogą przekroczyć 100 milimetrów. Dodatkowo każdy z robotów musi spełniać poniższe wymagania:

- musi być w pełni autonomiczny,
- nie może być przytwierdzony do areny,
- nie może zakłócać sterowania przeciwnika,
- musi posiadać na wyposażeniu moduł startowy, dający możliwość zdalnego uruchomienia robota przez sędziego,
- nie może emitować cieczy, gazów oraz nadmiernego ciepła.

Na rysunku 1.1 przedstawiono przykładową walkę robotów klasy sumo. Warto zauważyć, iż wnętrze areny jest czarne, natomiast obwód biały. Dzięki zastosowanemu kontrastowi robot wyposażony w odpowiednie czujniki jest w stanie wykryć brzeg areny.



Rys. 1.1: Zawody sumo [4].

1.1. Cel i założenia

Celem niniejszej pracy jest implementacja aplikacji mobilnej służącej do sterowania robotem minisumo. W ramach pracy dyplomowej powstał samodzielnie wykonany dwukołowy robot w pełni spełniający wymagania do startu w zawodach minisumo. Dodatkowo powstała aplikacja mobilna na platformę iOS, która daje możliwość obsługi oraz konfiguracji wyżej wspomnianego robota. Dzięki niej użytkownik może wybrać jedną z wielu strategii walki, ustalić maksymalną moc silników oraz uwzględnić oczekiwanie na start za pomocą odbiornika fal podczerwonych. Dodatkowo aplikacja oferuje możliwość zdalnego sterowania robotem za pomocą akcelerometru lub wirtualnego dżojstiku oraz sprawdzenia poprawności działania sensorów i silników.

Główne założenia realizowanej pracy:

- stworzenie robota spełniającego wymagania kategorii minisumo,
- sprawna sensoryka pozwalająca na wykrycie przeciwnika oraz końca ringu,
- w pełni działająca komunikacja między robotem a aplikacją,
- aplikacja mobilna pozwalająca na konfigurację wyżej wspomnianego robota.

Etapy realizacji wyżej wymienionych założeń zostały opisane w dalszej części pracy z podziałem na tworzonego robota minisumo oraz aplikację mobilną.

Rozdział 2

Wykorzystane technologie

Projekt został zrealizowany pod systemem Windows 10 oraz Mac OS X El Capitan.

2.1. C

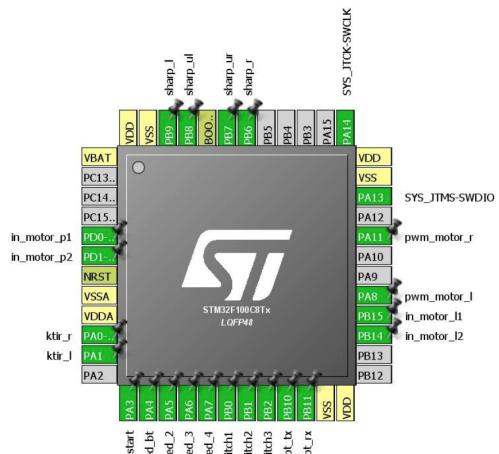
Z racji, iż sercem robota jest procesor z rodziny STM, wybór technologii został ograniczony do języka C lub C++. Wybrano język C z powodów optymalizacyjnych oraz małego stopnia skomplikowania programu.

Oprogramowanie zostało stworzone w środowisku Eclipse z dodatkiem AC6 wspierającym platformę STM32.

2.1.1. HAL

HAL (Hardware Abstraction Layer) jest biblioteką będącą wysokopoziomowym interfejsem służącym do konfiguracji peryferiów mikrokontrolera. Zdecydowano się na wyżej wspomnianą bibliotekę z powodu bardzo przejrzystej dokumentacji oraz łatwości użytkowania. Dodatkowo użyto środowiska CubeMX, które udostępnia graficzny interfejs, który pozwala na stosunkowo łatwą oraz intuicyjną konfigurację procesora oraz wygenerowanie projektu w języku C wraz z użyciem bibliotek HAL.

Rysunek 2.1 przedstawia konfigurację peryferiów użytego procesora w środowisku CubeMX.



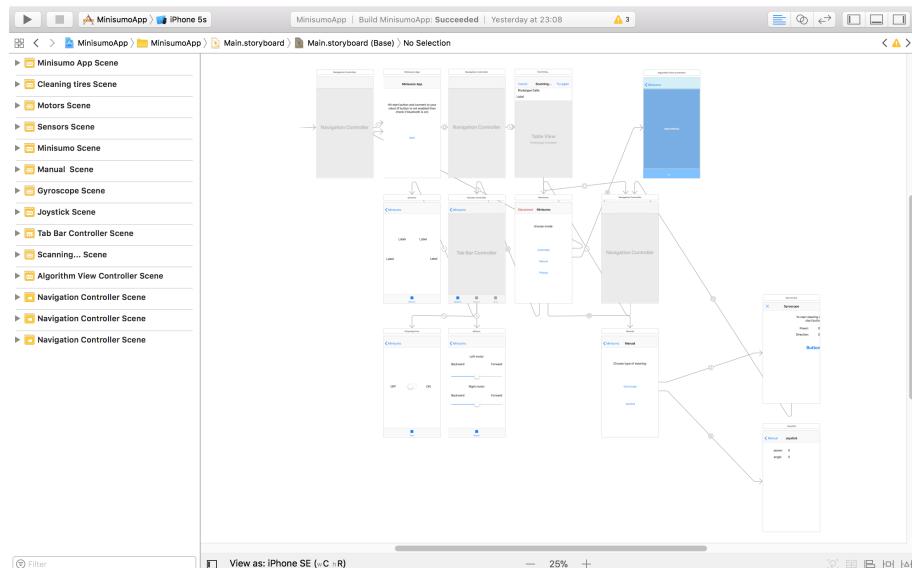
Rys. 2.1: Konfiguracja peryferiów użytego procesora STM32F100C8T6B – LQFP48.

2.2. Swift

Swift jest językiem natywnym (następcą języka *Objective-C*) zaprezentowanym przez *Apple Inc.* w 2014 roku. Wykorzystywany jest do tworzenia oprogramowania na platformy *macOS*, *iOS*, *watchOS*. W pracy dyplomowej użyto wersji języka 3.0, ponieważ była to najnowsza wersja wspierana przez docelowe urządzenie, którym był telefon iPhone 5.

Środowiskiem użytym do tworzenia aplikacji mobilnej w technologii *Swift* był *Xcode*, którego dużym atutem jest występowanie graficznego interfejsu umożliwiającego tworzenie widoków aplikacji. Dzięki temu tworzenie aplikacji jest bardziej intuicyjne oraz pozwala na sprawne wprowadzanie zmian w tworzonych widokach.

Ilustracja 2.2 ukazuje środowisko Xcode wraz z widokami aplikacji oraz zależnościami między nimi.



Rys. 2.2: Środowisko Xcode.

2.2.1. UIKit

UIKit jest platformą, która zapewnia infrastrukturę dla aplikacji. Udostępnia architekturę widoku do implementacji interfejsu, obsługę zdarzeń, obsługę wielopunktowego dotyku, zarządza zasobami oraz interakcjami pomiędzy aplikacją a użytkownikiem. Dodatkowymi funkcjami są obsługa dokumentów, aplikacji oraz jej rozszerzeń, zarządzanie tekstem i wyświetlaniem.

UIView

Widoki są podstawowymi elementami składowymi interfejsu użytkownika aplikacji, a klasa `UIView` definiuje zachowania wspólne dla wszystkich widoków. Obiekt widoku renderuje zawartość w obrębie prostokąta obwiedni i obsługuje wszelkie interakcje z tą zawartością.

UITableView

`UITableView` wyświetla listę pozycji w pojedynczej kolumnie. Jest podklassą `UIScrollView`, która pozwala użytkownikowi przewijać tabelę, z tą różnicą, że przewijanie dozwolone jest wyłącznie w pionie. Komórki zawierające poszczególne elementy tabeli są obiektami `UITableViewCell`. `UITableView` używa wspomnianych obiektów do rysowania widocznych wierszy tabeli. Komórki mogą posiadać tytuły treści, obrazy oraz widoki akcesoriów, które mogą pełnić rolę elementów sterujących, takich jak przełączniki i suwaki.

2.2.2. CoreBluetooth

Platforma `CoreBluetooth` dostarcza klasy niezbędne do komunikacji aplikacji z urządzeniami wyposażonymi w bezprzewodową komunikację Bluetooth.

CBPeripheral

Klasa `CBPeripheral` reprezentuje zdalne urządzenia peryferyjne, których połączenie aplikacja wykrywa za pośrednictwem instancji `CBCentralManager`. Urządzenia peryferyjne są identyfikowane za pomocą unikalnych identyfikatorów (UUID) reprezentowanych przez obiekty NSUUID.

CBCentralManager

Obiekty `CBCentralManager` służą do zarządzania podłączonymi urządzeniami peryferyjnymi (reprezentowanymi przez obiekty `CBPeripheral`). Pozwalają na skanowanie oraz wykrywanie zdalnych urządzeń.

2.2.3. CoreGraphics

Platforma `CoreGraphics` oparta jest na zaawansowanym mechanizmie rysowania `Quartz`. Zapewnia niskopoziomowe renderowanie 2D, zarządzanie wzorcami, gradientami oraz danymi

obrazu. Dodatkowo pozwala na tworzenie oraz maskowanie obrazu, a także wyświetlanie i analizowanie dokumentów PDF. W pracy dyplomowej *CoreGraphics* zostało wykorzystane do stworzenia wirtualnego dżojstiku służącego zdalnemu sterowaniu robotem minisumo.

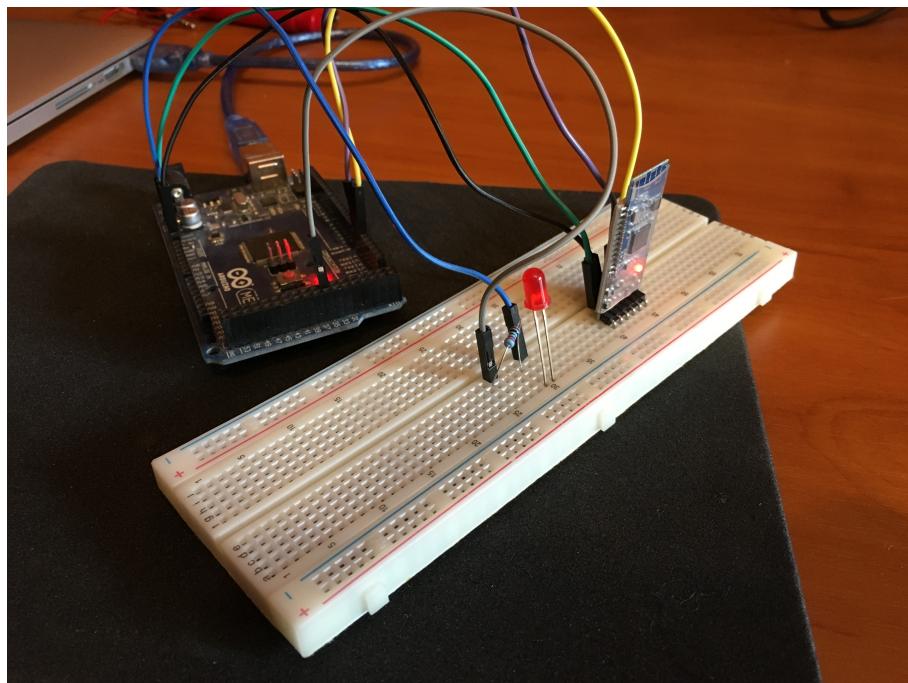
2.2.4. CoreMotion

Dostarcza dane dotyczące ruchu urządzenia na podstawie wbudowanego akcelerometru, żyroskopu, krokomierza, magnetometru, barometru itp. W aplikacji użyto wspomnianej platformy do sterowania ruchem robota za pomocą akcelerometru [2].

2.3. Arduino

Arduino jest platformą programistyczną przeznaczoną dla mikrokontrolerów z wbudowaną obsługą układów wejścia oraz wyjścia [3]. Język programowania *Arduino* oparty jest na języku C/C++. Głównym atutem omawianej platformy jest szeroka społeczność, dokładna dokumentacja oraz łatwość prototypowania prostych układów. Jednakże omawiania platformy nie jest zalecana w przypadku złożonych projektów ze względu na wysokopoziomowość oraz niestabilność pracy.

Platforma została użyta do konfiguracji modułu Bluetooth za pomocą komend AT. Skonfigurowano takie parametry jak nazwa urządzenia oraz liczbę symboli na sekundę (ang. *baud rate*). Do tego celu użyto mikrokontrolera *Arduino Mega 2560* oraz płytka stykowej wraz z przewodami połączeniowymi. Konfiguracja modułu była jednorazową czynnością, dlatego zdecydowano się na platformę *Arduino* ze względu na wyżej wspomnianą szybkość oraz prostotę obsługi.



Rys. 2.3: Układ służący do konfiguracji modułu bluetooth.

Ilustracja 2.3 obrazuje układ Arduino Mega 2560 wraz z płytą stykową oraz podłączonym modułem bluetooth. Dodatkowo układ wyposażono w diodę mającą na celu sygnalizację poprawności wgrywania komend.

```
1 #include <SoftwareSerial.h>
2
3 int LEDPIN = 13;
4
5 void setup() {
6     pinMode(LEDPIN, OUTPUT);
7     Serial.begin(9600); // communication with computer
8     Serial3.begin(9600); // communication with HM-10
9 }
10
11 void loop() {
12     if (Serial3.available() ) {
13         char dioda = Serial3.read();
14         Serial.write(dioda);
15         if (atoi(&dioda)==1) {
16             digitalWrite(LEDPIN, HIGH);
17         }
18
19         if (atoi(&dioda)==0) {
20             digitalWrite(LEDPIN, LOW);
21         }
22     }
23
24     if (Serial.available() ) { Serial3.write( Serial.read() ); }
25 }
```

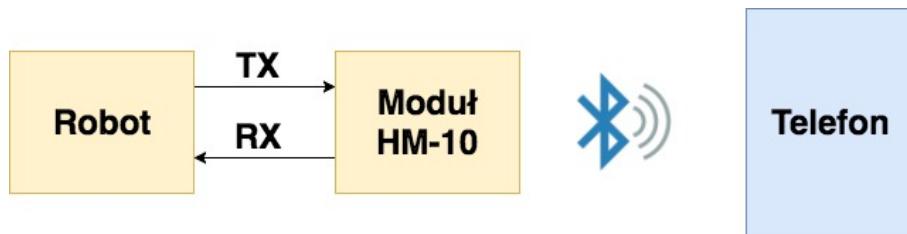
Listing 2.1: Kod programu umożliwiającego konfigurację modułu Bluetooth HM-10.

Listing 2.1 przedstawia kod programu zapewniającego komunikację między komputerem a modułem Bluetooth przy użyciu mikrokontrolera *Arduino Mega 2560*. Za pomocą terminalu udostępnionego przez środowisko *Arduino* możliwa jest komunikacja oraz konfiguracja modułu HM-10. Dodatkowo program został rozszerzony o zapalanie oraz gaszenie diody LED w celu sprawdzenia poprawności połączenia. Jeżeli przy użyciu wyżej wspomnianego terminala zostanie wysłana wartość 1, powinna zapalić się dioda. W przypadku wysłania wartości 0, dioda powinna zgasnąć.

Rozdział 3

Komunikacja

Komunikacja między urządzeniami (robotem a aplikacją mobilną) została zrealizowana przy użyciu łączności bezprzewodowej Bluetooth.

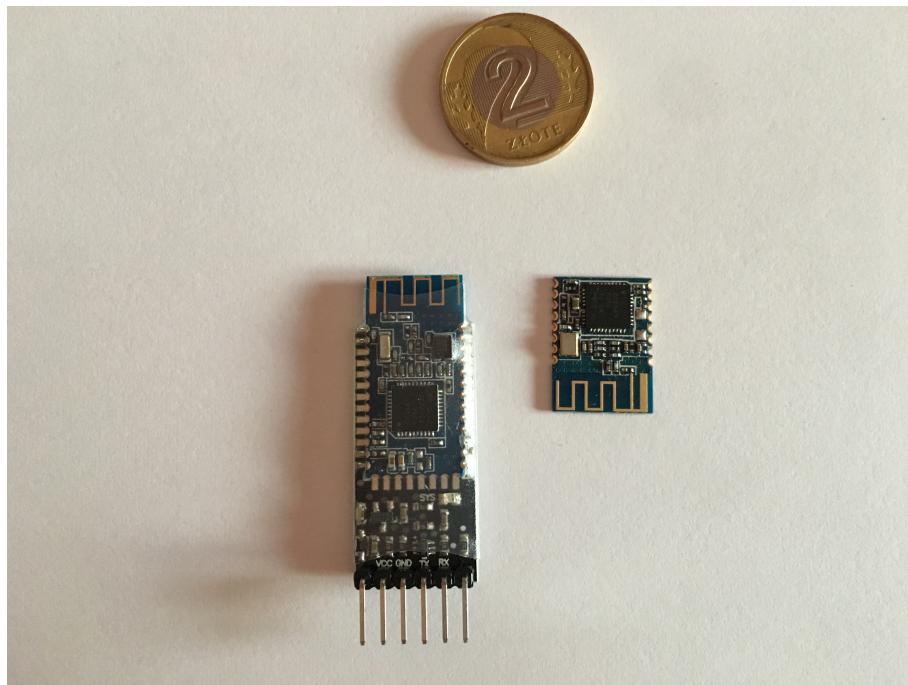


Rys. 3.1: Schemat komunikacji między urządzeniami.

Na rysunku 3.1 przedstawiono schemat komunikacji między robotem a aplikacją mobilną. Aplikacja mobilna komunikuje się radiowo z modułem Bluetooth, który następnie za pomocą przewodów podłączonych do linii TX oraz RX procesora przesyła wyslaną wiadomość.

3.1. Moduł bluetooth

Z racji, iż docelowym urządzeniem komunikującym się z robotem był telefon iPhone, który obsługuje wyłącznie technologię BLE (Bluetooth Low Energy), wybór docelowego modułu komunikacyjnego został mocno ograniczony. Z początku zdecydowano się na moduł Bluetooth HM-11 ze względu na bardzo małe rozmiary. Niestety okazało się, iż owy moduł posiada nietypowy raster, przez co niemożliwym było przylutowanie jakiegokolwiek złącza. Jedynym rozwiązaniem byłoby przylutowanie modułu do płytki elektronicznej robota, aczkolwiek projekt płytki nie przewidywał dodatkowych elementów. W związku z czym zdecydowano się na starszy model HM-10, który dostarczał taką samą funkcjonalność, lecz posiadał gotowe złącze goldpin, które znaczco ułatwiało podłączenie do wyżej wspomnianej płytki. Jedyną wadą zastosowanego modułu był rozmiar.



Rys. 3.2: Moduły Bluetooth.

Na rysunku 3.2 po lewej stronie widoczny jest docelowo użyty moduł HM-10, natomiast po prawej wcześniej wspomniany HM-11. U góry ilustracji znajduje się moneta dla odwzorowania rozmiarów omawianych układów.

3.2. Realizacja komunikacji

Komunikacja między urządzeniami bazuje na wysyłaniu ciągu dziesięciu znaków, a dokładniej nieujemnych liczb całkowitych, które następnie są parsowane oraz interpretowane w odpowiedni sposób [1]. Domyślnie wiadomość składa się z dziesięciu zer, a następnie wypełniana jest odpowiednimi wartościami. Poniżej przedstawiono przykładowe formaty wiadomości przesyłane między urządzeniami.

Sensory

Komunikacja między urządzeniami rozpoczyna się od wysłania wiadomości przez aplikację mobilną w której pierwsza cyfra wiadomości ma wartość równą 4. Następnie co określony czas robot wysyła wiadomość w której zawarty jest aktualny stan czujników. Komunikacja kończy się w momencie, gdy aplikacja mobilna wyśle wiadomość w której pierwsza cyfra jest różna od 4.

4 A B C D XX YY 0

Rys. 3.3: Schemat wiadomości zawierającej informacje odnośnie stanu sensorów.

Na rysunku 3.3 przedstawiono strukturę wiadomości wysłanej przez robota do aplikacji mobilnej. Pierwsza cyfra (równa 4) sygnalizuje aplikacji, iż przesyłana wiadomość zawiera informacje na temat stanu czujników. Cyfry A , B , C , D sygnalizują stan cyfrowych czujników wykrywających przeciwnika w sposób binarny (0 – nie wykryto przeciwnika, 1 – wykryto przeciwnika). Cyfry XX oraz YY informują o stanie analogowych czujników wykrywających koniec ringu (00 – wykryto czarny kolor, 99 – wykryto biały kolor).

Automatyczne Sterowanie

Komunikacja bazuje na jednorazowym wysłaniu wiadomości do robota z poziomu aplikacji mobilnej. Treść wiadomości określa czy robot powinien wstrzymać start do momentu otrzymania komunikatu od sędziego, rodzaj algorytmu walki oraz maksymalną moc silników.

1 A B XXX 0000

Rys. 3.4: Schemat wiadomości w której zawarta jest konfiguracja robota.

Rysunek 3.4 obrazuje schemat wiadomości wysłanej z aplikacji mobilnej do robota. Pierwsza cyfra (równa 1) informuje, iż robot przechodzi w stan automatycznej jazdy. Cyfra A określa numer algorytmu walki (na chwilę obecną występują trzy algorytmy, numerowane 1,2 oraz 3). Kolejna cyfra B dostarcza informację odnośnie rodzaju startu (0 – ruch robota rozpoczyna się w momencie wysłania wiadomości, 1 – robot oczekuje na sygnał sędziego). Ostatnia wartość XXX określa procentowo maksymalną moc jaką mogą osiągnąć obydwa silniki (przyjmuje wartości 000 – 100, gdzie 100 oznacza maksymalną moc, a 000 minimalną).

Sterowanie zdalne - akcelerometr

Aplikacja mobilna jednorazowo wysyła komunikat, którego pierwsza cyfra oznacza wejście w tryb zdalnego sterowania. Następnie wspomniana aplikacja rozpoczyna nasłuchiwanie wiadomości zawierających kierunki ruchu oraz moce poszczególnych silników wysyłanych przez robota.

6 A B XXX YYY 0

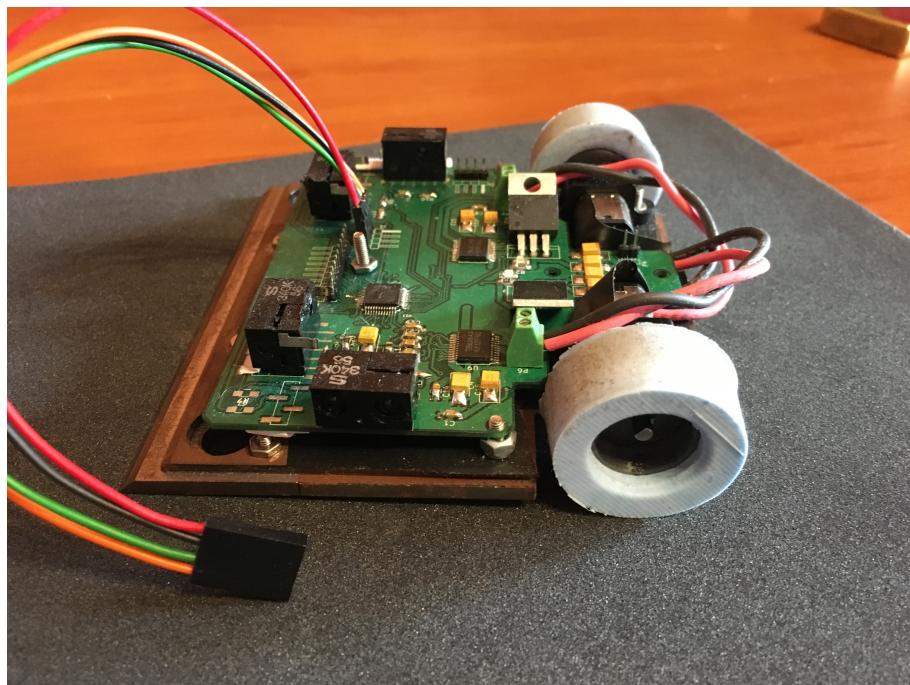
Rys. 3.5: Schemat wiadomości zdalnego sterowania przy użyciu akcelerometru.

Ilustracja 3.5 przedstawia budowę wiadomości wysłanej do aplikacji mobilnej przez robota. Pierwsza cyfra (równa 6) wskazuje, iż robot jest w trybie zdalnego sterowania za pomocą akcelerometru. Kolejne cyfry *A* oraz *B* oznaczają kierunki odpowiednio lewego oraz prawego silnika (1 – obrót w tył, 2 – obrót w przód). *XXX* oraz *YYY* podobnie jak w przypadku trybu automatycznej jazdy określają procentowe moce silników, lewego oraz prawego (przyjmują wartości 000 – 100, gdzie 100 oznacza maksymalną moc, a 000 minimalną).

Rozdział 4

Robot minisumo

W niniejszym rozdziale opisany został proces tworzenia robota minisumo. Wyróżniono w nim trzy główne sekcje: *Konstrukcja*, *Elektronika* oraz *Oprogramowanie*.



Rys. 4.1: Wykonany robot minisumo.

Zdjęcie 4.1 przedstawia omawianego robota minisumo. W celu ukazania elektroniki ściągnięto nadwozie.

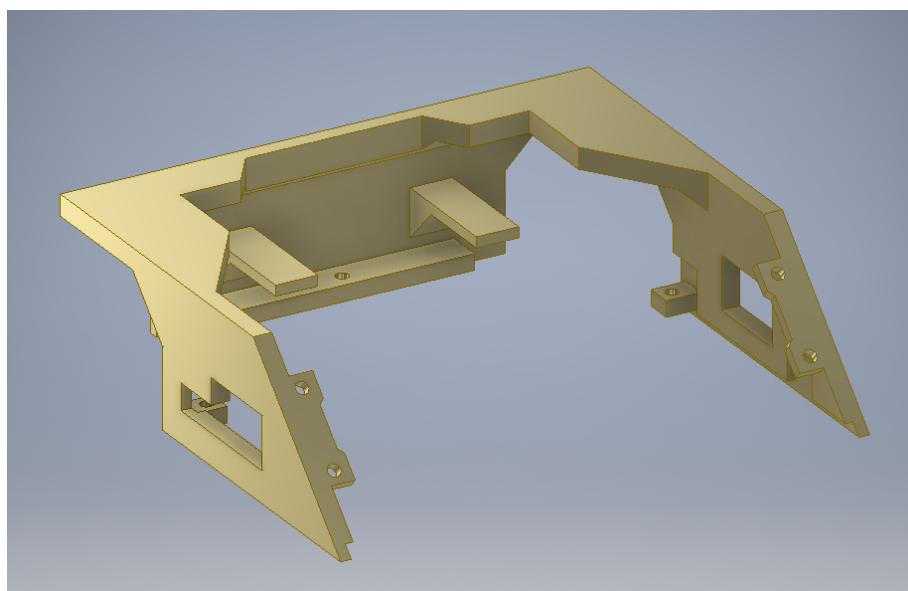
4.1. Konstrukcja

4.1.1. Założenia

Główym założeniem konstrukcyjnym było zminimalizowanie szansy podbicia robota przez przeciwnika. W związku z czym napęd robota składa się z dwóch kół umiejscowionych w tylnej części konstrukcji. Przednia część robota zakończona jest pługiem, który bezpośrednio dotyka podłożu. Dodatkowo starano się, aby środek ciężkości całej konstrukcji znajdował się jak najbliżej podłożu.

4.1.2. Nadwozie

Konstrukcja nadwozia została zaprojektowana przy użyciu środowiska *Autodesk Inventor 2017*. Wybór został podykowany łatwością obsługi narzędzi oraz darmową licencją studencką. Obudowa została wykonana w technologii druku 3D, ze względu na małą masę, koszt produkcji oraz możliwość łatwego dostosowania do potrzeb projektu (otwory na śruby oraz czujniki przeciwnika, przestrzeń na koła).



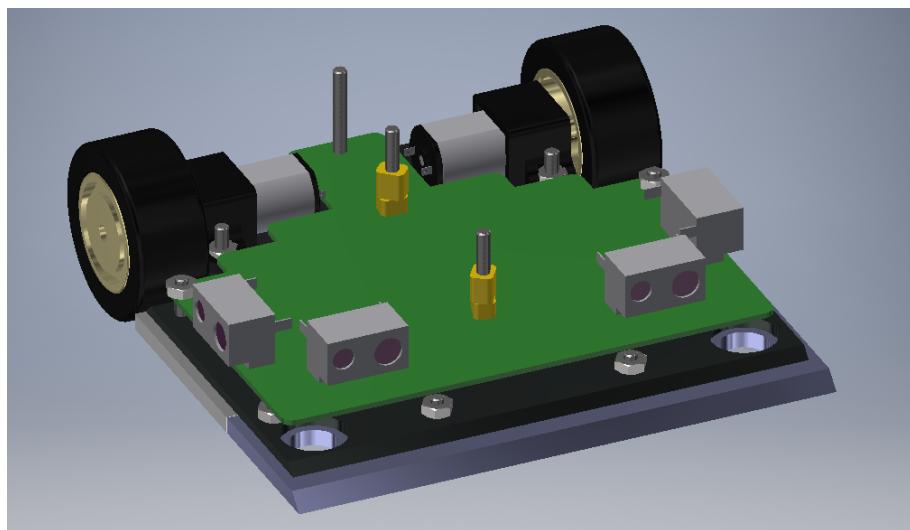
Rys. 4.2: Projekt nadwozia w środowisku Autodesk Inventor 2017.

Na ilustracji 4.2 znajduje się projekt nadwozia robota. Ze względu na niewielką wytrzymałość plastikowego wydruku przywiązano dużą wagę do szerokości obudowy. Starano się, aby była możliwie jak najszerzsza, co skutkowało wzrostem trwałości całej konstrukcji. Dodatkowo front robota został wykonany ze stali, ponieważ w przypadku podważenia przeciwnika jest obszarem najbardziej podatnym na zniszczenia.

4.1.3. Podwozie

Całość podwozia wykonana została ze stali ze względu na wytrzymałość oraz dużą masę, która przyczyniła się do znacznego obniżenia środka ciężkości konstrukcji. Podobnie jak w przypadku

nadwozia projekt został stworzony w środowisku *Autodesk Inventor 2017*. Konstrukcja składa się z trzech płyt, z czego jedna zawiera ostrze od strugarki, wykonane z węglika spiekanego. Materiał ten cechuje się wysoką wytrzymałością oraz większą odpornością na kruszenie w porównaniu do zwykłej stali.

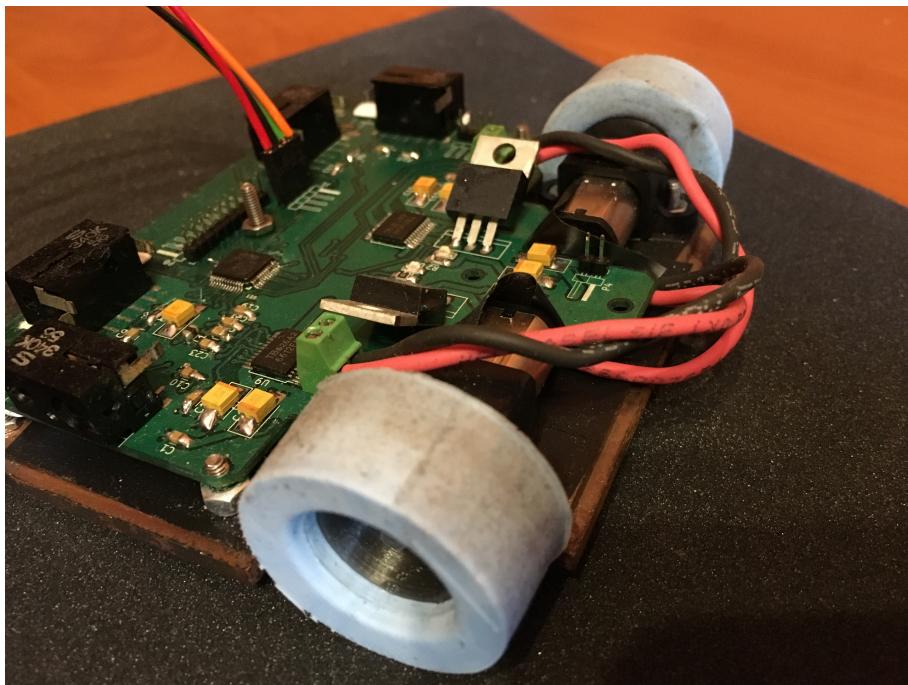


Rys. 4.3: Projekt podwozia w środowisku Autodesk Inventor 2017.

Ilustracja 4.3 ukazuje wizualizację podwozia wraz z silnikami oraz kołami. Projekt wyeksportowano do formatu 2D, a następnie wykonano przy użyciu techniki wypalania laserem.

4.1.4. Napęd

Jak wspomniano wcześniej, napęd robota składa się z dwóch kół, które sterowane są niezależnie. Ruch robota wzorowany jest na zasadzie działania czołgu. W przypadku skrętu, do jednego z silników dostarczana jest większa moc, co skutkuje wzrostem prędkości kątowej napędzanego koła. Z powodu różnicy prędkości kół robot zaczyna skracać. Zdecydowano się na takie rozwiązanie ze względu na dozwoloną masę (większa ilość kół oraz silników znacząco by ją zwiększyła) oraz manewrowość – dzięki zastosowaniu omawianego rozwiązania możliwy jest obrót robota w miejscu (w przypadku, gdy koła kręczą się w przeciwnie strony), dzięki czemu pojazd jest szybki oraz zwinny.



Rys. 4.4: Napęd robota minisumo.

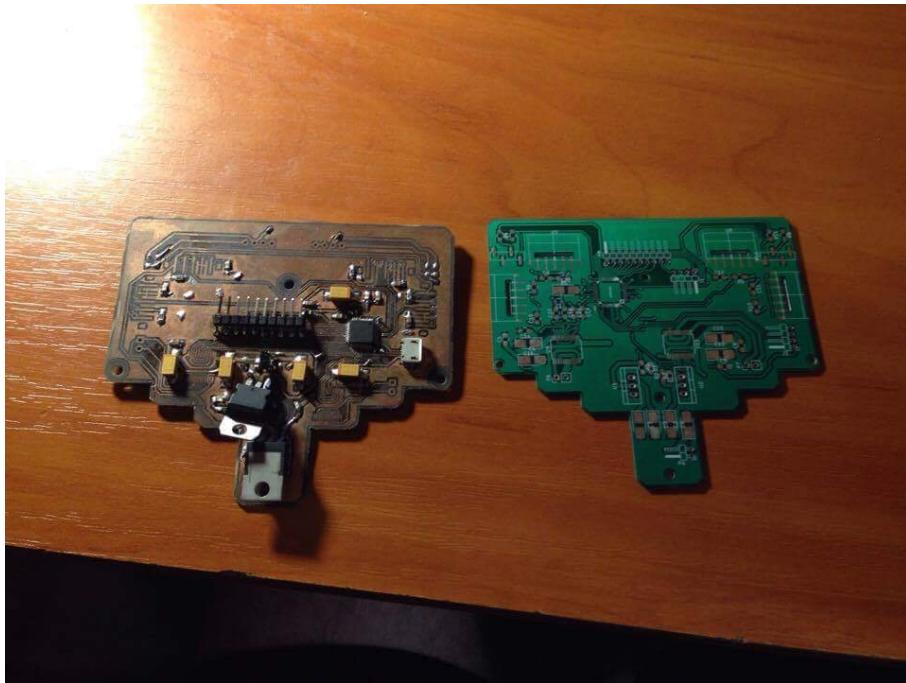
Na zdjęciu 4.4 przedstawiono napęd robota na który składają się dwa silniki oraz dwie felgi z oponami. Jako jednostkę napędową użyto silników *Pololu HPCB* ze względu na rozmiary, dopuszczalne napięcie zasilania oraz korzystny stosunek ceny do jakości. Felgi zostały wykonane przy pomocy wydruku 3D, a opony były gotowym komponentem wykonanym z poliuretanu.

4.2. Elektronika

Projekt elektroniki robota, będącego tematem pracy dyplomowej, został stworzony przy użyciu darmowego środowiska *KiCad*. Z początku płytki zostały wykonane własnoręcznie za pomocą metody termotransferu. Aczkolwiek z powodu dużej ilości przelotek, braku odpowiednich narzędzi oraz kilku błędów popełnionych na etapie projektowania, stwierdzono, iż lepszym rozwiązaniem będzie przeprojektowanie płytki oraz zlecenie wydruku firmie specjalizującej się w tej dziedzinie. Nowa płytka była zdecydowanie trwalsza od poprzedniej wersji. Ponadto dzięki zastosowaniu soldermaski oraz cynowania HAL lutowanie elementów okazało się znacznie łatwiejsze.

Docelowo robot posiada dwie płytki:

- główną – zawierającą całą logikę robota,
- interfejs – pełniący rolę interfejsu użytkownika, posiadający przyciski oraz diody LED.



Rys. 4.5: Porównanie wykonanych płyt drukowanych.

Na zdjęciu 4.5 przedstawiono pierwszą (wykonaną metodą termotransferu) oraz finalną (wykonaną przez firmę) wersję górnej płytka.

4.2.1. Układ zasilania

Ze względu na ograniczone miejsce oraz dozwoloną masę za źródło zasilania posłużył akumulator litowo–polimerowy (Li–Pol) posiadający dwie cele oraz 400mAh pojemności. Bateria cechowała się wystarczającą wydajnością prądową, a pojemność pozwalała na stoczenie kilku pojedynków bez potrzeby ładowania.

Z racji iż nie wszystkie elementy elektroniki tolerują napięcie 7.4V, układ zasilania został wyposażony w stabilizatory 3.3V oraz 5V.

4.2.2. Procesor

Poniżej wymieniono niezbędne funkcjonalności jakie musiała spełniać docelowa jednostka obliczeniowa:

- przetwornik analogowo – cyfrowy (*ang. ADC – analog to digital converter*) potrzebny do obsługi czujników końca ringu,
- odpowiednia ilość portów wejścia/wyjścia dla przycisków oraz diod,
- możliwość zmiany wypełnienia sygnału prądowego lub napięciowego (*ang. PWM – Pulse–Width Modulation*) w celu odpowiedniego sterowania mocą silników,
- interfejs UART (*ang. Universal Synchronous and Asynchronous Reciever and Transmitter*) niezbędny przy komunikacji z modułem Bluetooth,
- obsługa przerwań wykorzystywana w odbiorze wiadomości.

Powyższe wymagania spełniał procesor *STM32F100C8T6B - LQFP48*. Dodatkowymi atutami była niska cena, ogólnodostępność oraz małe rozmiary.

4.2.3. Sensoryka

Konstrukcja została wyposażona w cztery czujniki *SHARP Gp2y0d340k* wykrywające obecność przeciwnika do 40 centymetrów. Umieszczono je z przodu robota – dwa po bokach oraz dwa od frontu. Dodatkowo przy pługu znajdują się dwa czujniki sygnalizujące opuszczenie pola walki.

4.2.4. Sterownik silników

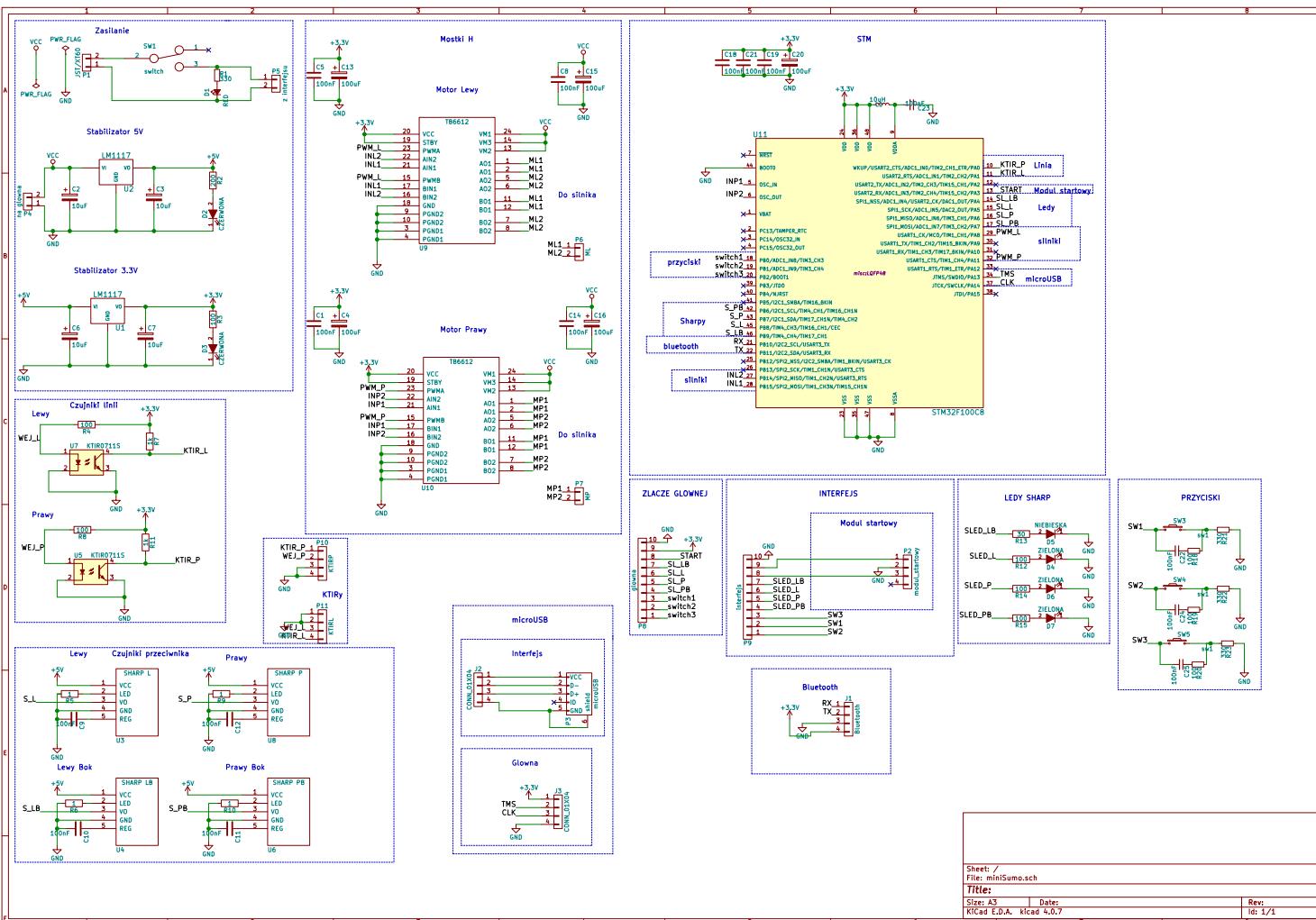
Jako sterownik posłużył dwukanałowy, 1.2 amperowy mostek *H TB6612*. Wybór został podkotwany dostępnością oraz ceną. Aczkolwiek z czasem okazało się, iż omawiany układ jest bardzo awaryjny, szczególnie nie nadaje się do walki w zwarciu [6].

4.2.5. Schemat elektroniki

W celu zwiększenia czytelności schemat podzielono na moduły, które podpisano oraz obramowano niebieską linią.

Poniżej na zdjęciu 4.6 przedstawiono schemat zaprojektowanej elektroniki (zarówno płytki górnej, jak i interfejsu).

4.2. Elektronika



Rys. 4.6: Schemat elektroniki.

4.3. Oprogramowanie

Oprogramowanie sterujące robotem zostało napisane przy użyciu środowiska *Eclipse*. Zawiera obsługę przychodzących/nadawanych wiadomości, zarządza sensoryką oraz motoryką konstrukcji.

4.3.1. Transmisja danych

Transmisja danych opiera się na przesyłaniu ciągu liczb tak jak to opisano w 3 rozdziale. Przechwytywanie wiadomości odbywa się przy użyciu przerwania, które zostaje obsłużone w momencie zmiany stanu logicznego na pinie RX (połączonego z modułem Bluetooth) procesora. Następnie zostaje wywołana funkcja obsługująca przerwanie, przerywając tym samym aktualnie wykonywany kod.

Listing 4.1 przedstawia funkcję wywoływaną w momencie przerwania. Pierwszą czynnością było zapisanie odebranej wiadomości do zmiennej tablicowej *received_message*, Linia 6 przedstawia przypisanie pierwszej liczby otrzymanej wiadomości do zmiennej *option*, która kolejno wykorzystywana jest w warunku *switch* i w zależności od jej wartości ustawiane są odpowiednie flagi, będące zmiennymi globalnymi. Po zakończeniu działania opisywanej funkcji, program zostaje wznowiony w miejscu w którym został przerwany.

Działanie głównej funkcji programu *main()* wykorzystuje pętle, które przy każdej iteracji sprawdzają stan wyżej wspomnianych flag. Na ich podstawie określana jest obecnie wykonywana funkcjonalność robota. Reszta wiadomości parsowana jest przy użyciu metody *parseValueFromMessage*, a następnie interpretowana w odpowiedni sposób, zależny od obecnie wykonywanej funkcjonalności.

```
1 void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
2
3     sscanf((char*)receive_buffer, "%s", received_message);
4     HAL_GPIO_TogglePin(GPIOA, led_bt_Pin);
5     char option[1];
6     option[0] = received_message[0];
7
8     switch(atoi(option)) {
9         case 1:
10            automatic = true;
11            HAL_UART_Receive_IT(&huart3, receive_buffer, 10);
12            break;
13        case 2:
14            debug_motors = true;
15            break;
16        case 3:
17            debug_tires = true;
18            HAL_UART_Receive_IT(&huart3, receive_buffer, 10);
19            break;
20        case 4:
21            debug_sensors = true;
22            break;
23        case 5:
24            remote_joystick = true;
25            break;
26        case 6:
27            remote_gyro = true;
28            break;
29        default:
30            debug_tires = false;
31            automatic = false;
32            TIM1->CCR1 = 0;
33            TIM1->CCR4 = 0;
34            HAL_UART_Receive_IT(&huart3, receive_buffer, 10);
35            break;
36    }
37 }
```

Listing 4.1: Funkcja obsługująca przerwanie.

4.3.2. Sterowanie silnikami

Sterowanie mocą silników odbywa się za pomocą modyfikacji wypełnienia sygnału przesyłanego do sterowników silników. Zakres wartości mieści się w przedziale 0 – 100 i określa procentowy czas trwania stanu wysokiego impulsu w pojedynczym cyklu generowanym przez *timer*.

Poniższy listing 4.2 przedstawia fragment kodu, który modyfikuje moc silników. *TIM1* oznacza *timer* generujący impuls, a *CCR1* oraz *CCR4* określają kanały, czyli docelowo silniki do których zostanie przesłany impuls. Wartość wypełnienia zwracana jest przez metodę *parseValueFromMessage*, która parsuje odpowiedni fragment wiadomości nadesłanej z aplikacji mobilnej.

```
1 TIM1->CCR1 = parseValueFromMessage(3,4);
2 TIM1->CCR4 = parseValueFromMessage(5,6);
```

Listing 4.2: Sterowanie mocą silników.

4.3.3. Obsługa czujników

Obsługa stanu czujników została zrealizowana w bardzo prosty sposób wynikający z charakterystyki pracy sensorów. W przypadku czujników wykrywających przeciwnika sprawdzany jest stan pinu procesora do którego podłączono wyjście sensoru. Sam czujnik działa na zasadzie diody podczerwonej generującej sygnał, który zostaje odbity od przeciwnika i trafia do odbiornika. Następnie na wyjściu czujnika występuje stan wysoki (logiczna 1) w przypadku braku przeciwnika lub stan niski (logiczne 0), gdy wykryto przeciwnika.

```
1 if(!HAL_GPIO_ReadPin(GPIOB, sharp_l_Pin))
2     transmit_message[0] = 1;
3
4 if(!HAL_GPIO_ReadPin(GPIOB, sharp_ul_Pin))
5     transmit_message[1] = 1;
6
7 if(!HAL_GPIO_ReadPin(GPIOB, sharp_ur_Pin))
8     transmit_message[2] = 1;
9
10 if(!HAL_GPIO_ReadPin(GPIOB, sharp_r_Pin))
11     transmit_message[3] = 1;
```

Listing 4.3: Obsługa czujników przeciwnika.

Listing 4.3 przedstawia kod, który sprawdza stan wyjścia czujników, a następnie w przypadku wykrycia przeciwnika zmienia wartość odpowiednich pól wiadomości, która zostanie przesłana do aplikacji mobilnej.

Rozdział 5

Aplikacja mobilna

Aplikacja stworzona w ramach pracy pozwala na konfigurację oraz komunikację z wcześniej opisany robotem minisumo. Z racji, iż została napisana w natywnym języku *Swift* wspieranym systemem operacyjnym jest system iOS.

5.1. Kompatybilność

Jedynym wymogiem, który musi zostać spełniony w celu uruchomienia aplikacji jest wersja systemu – nie starsza niż *iOS 10*. W związku z czym aplikacja z powodzeniem powinna działać zarówno na tabletach oraz telefonach marki *Apple* posiadających wspomnianą lub nowszą wersję systemu.

Docelowo aplikacja była uruchamiana oraz testowana na urządzeniu *iPhone 5*, który jest najstarszym telefonem wspierającym system iOS 10. Powodem wyboru urządzenia była optymalizacja, a mianowicie poprawność działania aplikacji na najsłabszych podzespołach gwarantowała odpowiednią pracę na mocniejszych jednostkach.

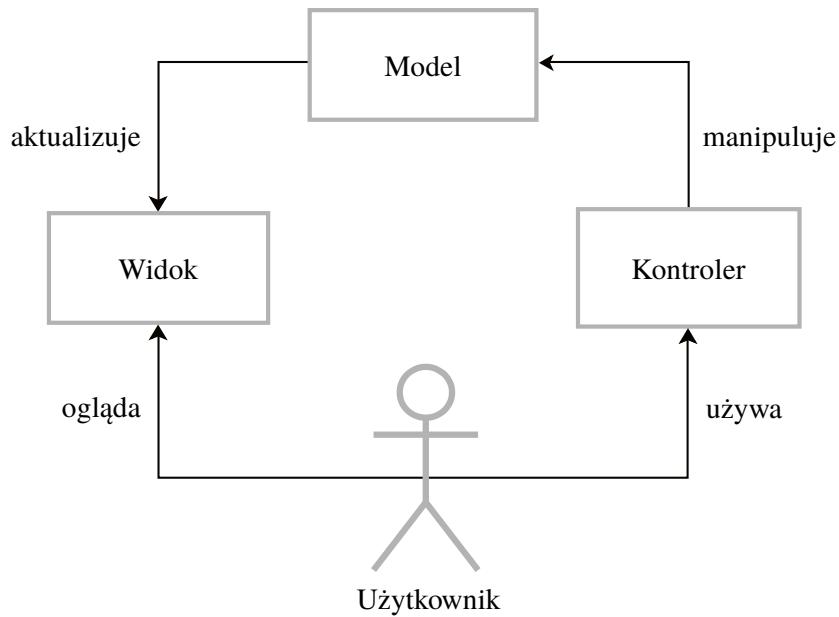
Dodatkowo poprawność wyświetlania oraz skalowania została przetestowana w symulatorze dostarczonym wraz z środowiskiem *Xcode* na urządzeniach takich jak *iPhone 6 Plus* oraz *iPad Pro*.

5.2. Wzorzec MVC

Aplikacja mobilna została zaprojektowana została według wzorca architektonicznego MVC (z ang. *Model–View–Controller*), który często wykorzystywany jest do tworzenia interfejsów użytkownika. W omawianym wzorcu można wyróżnić trzy obiekty:

- model – odpowiada za serwowanie danych,
- widok – odpowiada za wizualizację danych,
- kontroler – reaguje na poczynania użytkownika, zarządza odświeżeniem widoków oraz aktualizacją modelu.

Podział opisanych ról przedstawiono na rysunku 5.1.



Rys. 5.1: Podział ról we wzorcu MVC.

Dzięki wykorzystaniu tego wzorca uniezależniono przechowywane dane od sposobu w jaki są one przedstawiane użytkownikowi.

5.3. Komunikacja

Komunikacja z modułem Bluetooth została zrealizowana przy użyciu wcześniej wspomnianej platformy *CoreBluetooth*. Utworzono klasę *BluetoothSerial*, która była odpowiedzialna za całą logikę potrzebną do poprawnej wymiany danych między urządzeniami. Z racji, iż komunikacja jest nieodłącznym elementem każdej z funkcjonalności aplikacji skorzystano z protokołów. Protokoły są czymś na wzór interfejsów używanych w językach takich jak C++ bądź Java. Deklarują metody, lecz ich nie implementują. Każda klasa, która korzysta z komunikacji *Bluetooth* musi zaimplementować metody zawarte w protokołach. Dzięki takiemu podejściu uniezależniamy logikę od kontrolerów. Klasa *BluetoothSerial* nie jest w żaden sposób zależna od innych klas. Natomiast klasy, które wykorzystują komunikację, implementują odpowiednie metody w zależności od indywidualnych konkretnych potrzeb.

```
1 protocol BluetoothSerialDelegate {  
2     func serialDidChangeState()  
3     func serialDidDisconnect(_ peripheral: CBPeripheral, error: NSError?)  
4     func serialDidReceiveString(_ message: String)  
5     func serialDidReadRSSI(_ rssi: NSNumber)  
6     func serialDidDiscoverPeripheral(_ peripheral: CBPeripheral, RSSI:  
7                                     ↴ NSNumber?)  
8     func serialDidConnect(_ peripheral: CBPeripheral)  
9     func serialDidFailToConnect(_ peripheral: CBPeripheral, error: NSError?)  
10    func serialIsReady(_ peripheral: CBPeripheral)
```

Listing 5.1: Protokół odpowiedzialny za komunikację między urządzeniami.

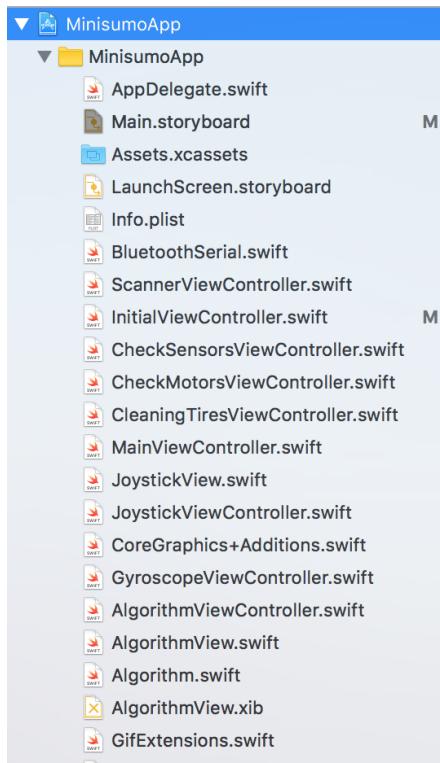
Listing 5.1 przedstawia zestaw metod zawartych w protokole komunikacyjnym. Poniżej przedstawiono przypadki w których zostają wywołane:

- *serialDidChangeState* – zmiana stanu modułu *Bluetooth* (zostanie wyłączony lub włączony),
- *serialDidDisconnect* – brak połączenia z modułem komunikacyjnym,
- *serialDidReceiveString* – została odebrana wiadomość typu tekstowego *String*,
- *serialDidReadRSSI* – gdy otrzymano sygnał RSSI (z ang. *Received Signal Strength Indication*),
- *serialDidDiscoverPeripheral* – wykryto urządzenie podczas skanowania,
- *serialDidConnect* – połączono z urządzeniem, aczkolwiek urządzenie nie jest jeszcze gotowe na komunikację,
- *serialDidFailToConnect* – nie udało się nawiązać połączenia,
- *serialIsReady* – moduł jest gotowy na komunikację z aplikacją mobilną.

Tak jak wcześniej wspomniano, klasy korzystające z komunikacji mogą implementować do-wolne z wyżej wymienionych metod w zależności od potrzeb.

5.4. Struktura aplikacji

Tak jak wcześniej wspomniano, aplikacja została stworzona przy użyciu wzorca *MVC*. Poniżej przedstawiono strukturę plików aplikacji.



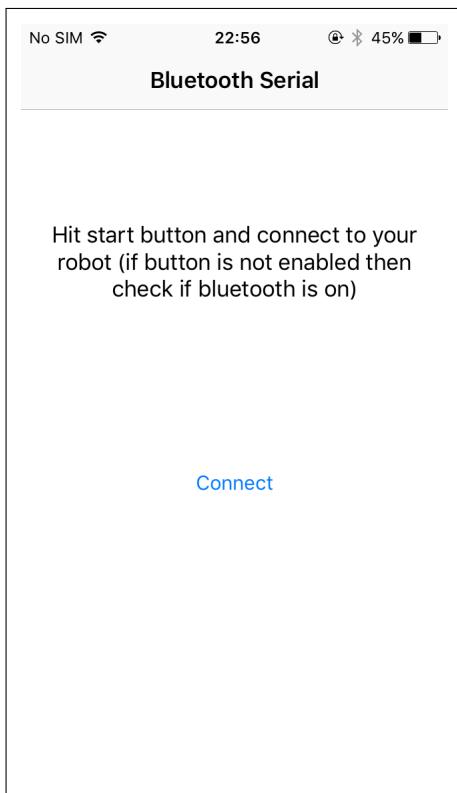
Rys. 5.2: Struktura plików aplikacji mobilnej.

Na rysunku 5.2 pliki o rozszerzeniu *swift*, których nazwy kończą się na *Controller* pełnią rolę kontrolerów, a *View* widoków przedstawianych użytkownikowi. Reszta plików odpowiedzialna jest za logikę aplikacji. Dodatkowo plik *Main.storyboard* zawiera konfiguracje większości widoków stworzonych przy użyciu graficznego narzędzia dostarczonego przez środowisko *Xcode*.

5.4.1. Panel powitalny

Panel powitalny jest pierwszym widokiem z którym użytkownik ma do czynienia po uruchomieniu aplikacji mobilnej. Odpowiada on za poprawne wykrycie urządzenia *Bluetooth* oraz dalszą komunikację. Składa się z elementów takich jak krótka instrukcja obsługi oraz przycisk *Connect*, który po wcisnięciu wyświetla listę urządzeń gotowych na połączenie. Dodatkowo w przypadku wyłączonej komunikacji *Bluetooth* zostanie wyświetlony komunikat o konieczności jej włączenia.

Po lewej stronie na obrazku 5.3 przedstawiono omawiany panel, natomiast zdjęcie 5.4 po prawej stronie przedstawia listę dostępnych urządzeń.



Rys. 5.3: Widok połączenia.



Rys. 5.4: Widok dostępnych urządzeń.

Za obsługę panelu powitalnego odpowiedzialna jest klasa *InitialViewController*, która inicjalizuje globalny obiekt *serial* , wykorzystywany do komunikacji *Bluetooth* w obrębie całej aplikacji. Zdecydowano się na taki krok, ponieważ transmisja danych jest nieodłącznym elementem każdej funkcjonalności aplikacji, przez co uniknięto problemu z przekazywaniem obiektu pomiędzy kontrolerami.

```

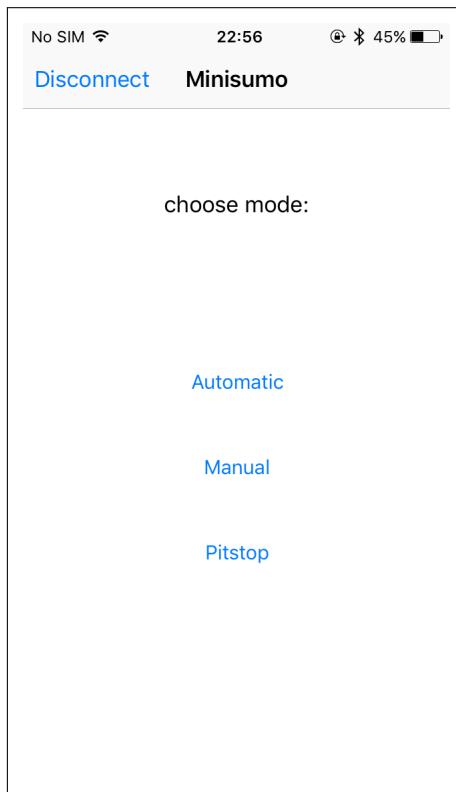
1 func serialDidDiscoverPeripheral(_ peripheral: CBPeripheral, RSSI:
        ↴ NSNumber?) {
2     for exisiting in peripherals {
3         if exisiting.peripheral.identifier == peripheral.identifier { return }
4     }
5     let theRSSI = RSSI?.floatValue ?? 0.0
6     peripherals.append(peripheral: peripheral, RSSI: theRSSI)
7     peripherals.sort { $0.RSSI < $1.RSSI }
8     tableView.reloadData()
9 }
10
11 func serialDidFailToConnect(_ peripheral: CBPeripheral, error: NSError?) {
12     tryAgainButton.isEnabled = true
13 }
14
15 func serialDidDisconnect(_ peripheral: CBPeripheral, error: NSError?) {
16     tryAgainButton.isEnabled = true
17 }
18
19 func serialIsReady(_ peripheral: CBPeripheral) {
20     performSegue(withIdentifier: "ShowMainViewController", sender: nil)
21 }
22
23 func serialDidChangeState() {
24     if serial.centralManager.state != .poweredOn {
25         dismiss(animated: true, completion: nil)
26     }
27 }
```

Listing 5.2: Implementacja protokołu.

Listing 5.2 przedstawia implementację metod zawartych w protokole *BluetoothSerialDelegate* (omówionym w rozdziale 5.3) dla klasy *ScannerViewController* odpowiedzialnej za wyświetlenie wykrytych urządzeń oraz poprawne połączenie z wybranym urządzeniem. Metoda *serialDidDiscoverPeripheral* agreguje wykryte urządzenia, a następnie wysyła do widoku żądanie o wyświetleniu posortowanej listy ze znalezionymi urządzeniami. Funkcje *serialDidFailToConnect* oraz *serialDidDisconnect* odpowiedzialne są za przypadek, gdy nie udało się ustawić połączenia z modułem *Bluetooth* – wyświetlają przycisk dzięki któremu możliwa jest ponowna próba połączenia. Gdy aplikacja mobilna poprawnie ustanowiła połączenie z którymś z urządzeń uruchamiana jest funkcja *serialIsReady*, której wynikiem jest przejście do panelu głównego. Natomiast w przypadku rozłączenia wywoływana jest metoda *serialDidChangeState* która wywołuje powrót do poprzedniego widoku jakim jest widok powitalny.

5.4.2. Panel główny

Panel główny odpowiedzialny jest tylko i wyłącznie za nawigację do poszczególnych funkcjonalności aplikacji. Składa się z prostego kontrolera *MainViewController*, który nawiguje do odpowiednich paneli w zależności od wciśniętego przycisku.



Rys. 5.5: Panel główny aplikacji mobilnej.

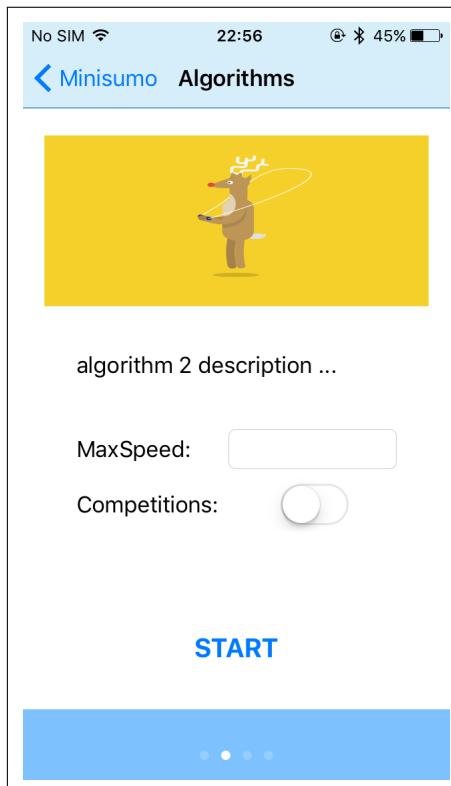
Na obrazku 5.5 przedstawiono wygląd panelu głównego aplikacji. Można zauważyć trzy przyciski *Automatic*, *Manual*, *Pitstop*, które nawigują odpowiednio do panelu sterowania automatycznego, zdalnego oraz diagnostyki.

5.4.3. Panel sterowania automatycznego

Panel sterowania automatycznego został stworzony w celu testowania algorytmów walki oraz konfigurowania robota do zawodów. Zaprojektowano go tak, aby był jak najbardziej intuicyjny w użytkowaniu. W górnej części widoku umieszczone zostało zdjęcie formatu *.gif* przedstawiające zachowanie robota na ringu dla danego algorytmu. Poniżej wizualizacji znajduje się krótki opis działania strategii, okno do ustawienia maksymalnej procentowej mocy silników oraz przełącznik, który określa czy robot powinien czekać na sygnał startowy (opcja wymagana podczas większości zawodów). Jej celem jest zapewnienie równego startu walczących ze sobą robotów. Na dole znajduje się przycisk *START*, który po wciśnięciu wysyła do robota wiadomość z konfiguracją. Przerwanie walki następuje po wciśnięciu przycisku *STOP* (który jest widoczny gdy wciśnięto przycisk *START*). W celu zmiany algorytmu należy przesunąć ekran od prawej do

lewej lub na odwrót. Na samym dole znajdują się kropki odpowiadające każdemu z algorytmów, a najjaśniejsza z nich sygnalizuje, która strategia jest obecnie konfigurowana.

Zdjęcie 5.6 ukazuje wygląd opisywanego panelu sterowania automatycznego.



Rys. 5.6: Panel sterowania automatycznego aplikacji mobilnej.

Za obsługę panelu sterowania automatycznego odpowiada klasa *AlgorithmViewController*, która wszystkie algorytmy walki przechowuje w tablicy *algorithms* w postaci obiektów klasy *Algorithm*.

```

1 class Algorithm {
2     var description = ""
3     var videoName = ""
4     var isCompetition = false
5 }
```

Listing 5.3: Klasa Algorithm.

Listing 5.3 przedstawia klasę *Algorithm*, która zawiera pola typu tekstowego *description*, *videoName* oraz logicznego *isCompetition* odpowiadające odpowiednio opisowi działania algorytmu walki, ścieżce zdjęcia *.gif* obrazującego zachowanie robota oraz informacji odnośnie tego czy robot powinien oczekiwąć na sygnał startowy.

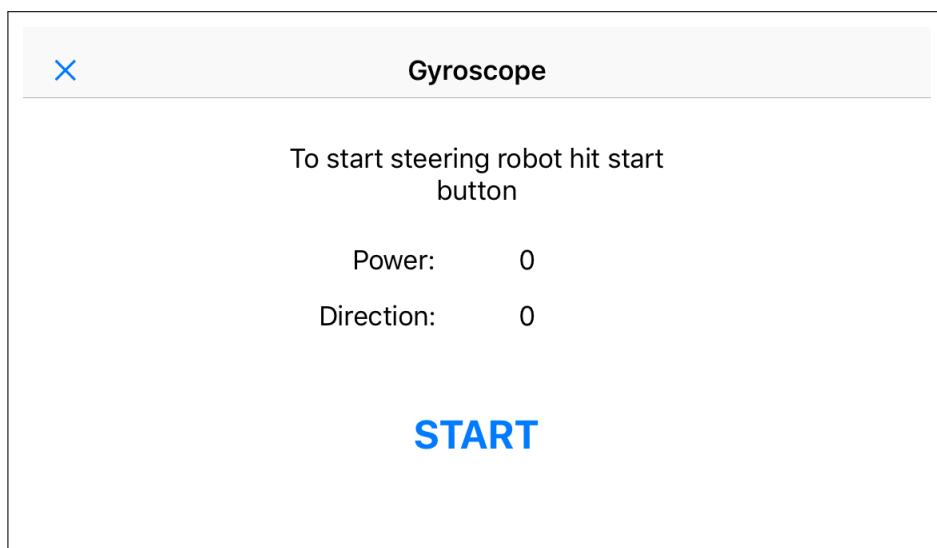
5.4.4. Panel sterowania zdalnego

Panel sterowania zdalnego pozwala na zdalne sterowanie silnikami robota za pomocą akcelerometru bądź wirtualnego dżojstiku. podobnie jak panel główny, posiada widok z dwoma przyciskami, który nawiguje do poszczególnych funkcjonalności:

Sterowanie akcelerometrem

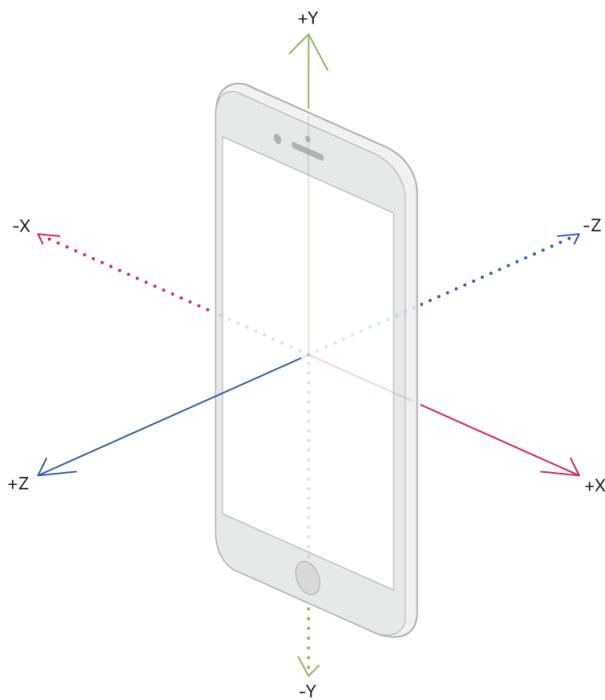
Sterowanie akcelerometrem pozwala na zdalne sterowaniem ruchem robota poprzez odpowiednie przechylenie telefonem. Sam panel składa się z krótkiej instrukcji obsługi, informacji o mocy oraz kierunku jazdy oraz przycisku *START*, który służy do uruchomienia omawianej funkcjonalności.

Zdjęcie 5.7 przedstawia wygląd omawianego panelu. Warto zauważyć, iż dla wygody został on zaprojektowany w orientacji poziomej.



Rys. 5.7: Panel sterowania zdalnego akcelerometrem.

W celu realizacji wyżej wspomnianej funkcjonalności wykorzystano obrót urządzenia wokół osi Y oraz Z. Obrót wokół osi Y odzwierciedla ruch robota w przód oraz w tył, natomiast obrót wokół osi Z kierunek ruchu – lewo lub prawo. Im większe wychylenie tym większa prędkość lub mocniejszy skręt. Dla lepszego zobrazowania problemu na rysunku 5.8 przedstawiono orientację urządzenia.



Rys. 5.8: Osie X, Y oraz Z określające orientację urządzenia [5].

Listing 5.4 przedstawia fragment kodu, należący do klasy *GyroscopeViewController*, a dokładniej metodę *startAcc*, która zostaje wywołana w momencie wcisnięcia przycisku *START*. Linia 2 ustawia czas, co który będzie zczytywany pomiar wychylenia urządzenia (w tym przypadku pomiar dokonywany jest co 200 milisekund). Następnie zostaje uruchomiony akcelerometr oraz przechwytywane są wartości obrotów wokół wspomnianych wcześniej osi. Ich wartości zawierają się w zakresie od -1 do 1. Na ich podstawie określany jest kierunek oraz prędkość, które zostaną wysłane do robota.

```

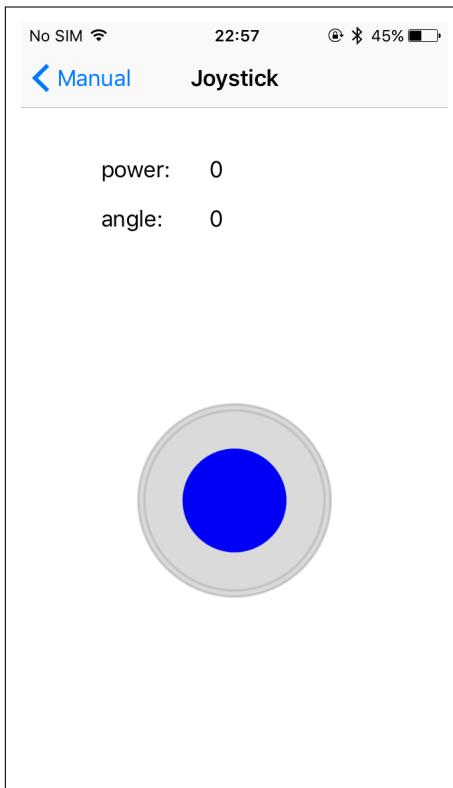
1 func startAcc() {
2     motionManager.deviceMotionUpdateInterval = 0.2
3     motionManager.startDeviceMotionUpdates(to: OperationQueue.current!) {
4         ↴ (data, error) in
5         if let motion = data {
6             var turn = Int(100 * motion.attitude.pitch)
7             var power = Int(100 * motion.attitude.roll)
8             .
9             .
10        }
11        .
12    }

```

Listing 5.4: Implementacja metody *startAcc*.

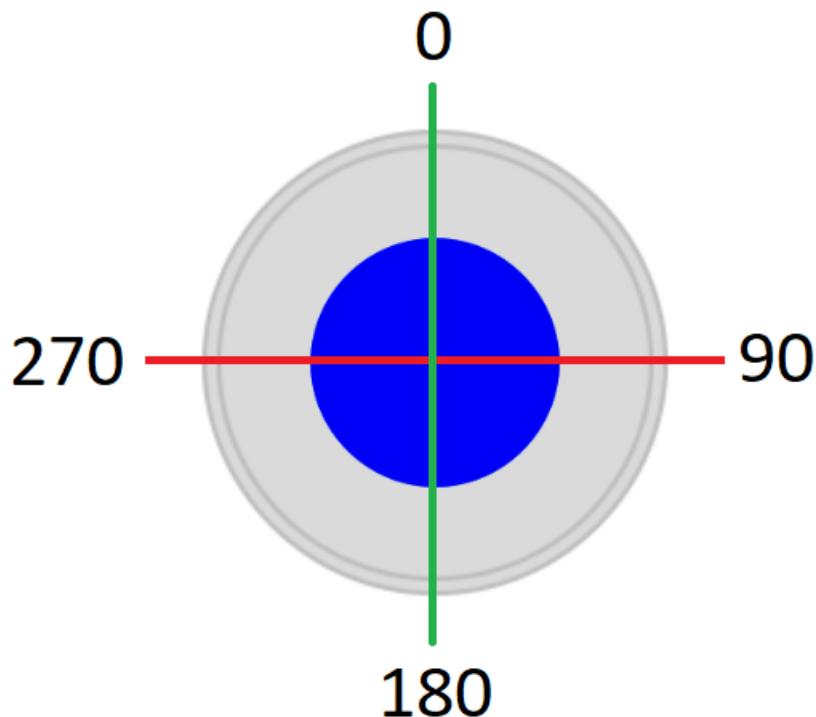
Sterowanie wirtualnym dżojstikiem

Sterowanie odbywa się za pomocą dwuwymiarowego drążka w postaci niebieskiego koła. Wykrycie drążka wpływa na kierunek jazdy oraz prędkość kół konstrukcji (im większe wykrycie tym większa prędkość). Dzięki zastosowaniu takiego rozwiązania możliwa jest jazda w przód oraz w tył. Dodatkowo w górnej części widoku znajduje się informacja odnośnie kierunku jazdy, wyrażonej kątowo w stopniach oraz mocy silników wyrażonej procentowo. Warto dodać, iż po puszczeniu drążka wraca on do początkowego położenia, które jest jednoznaczne z brakiem ruchu robota.



Rys. 5.9: Panel sterowania zdalnego dżojstikiem.

Powyższe zdjęcie 5.9 przedstawia wygląd panelu służącego do zdalnego sterowania robotem za pomocą dżojstiku.



Rys. 5.10: Wirtualny dżojstik.

Ilustracja 5.10 przedstawia zależność kierunku jazdy konstrukcji od kąta odchylenia drążka. Jak łatwo zauważyć, górną półkulą odpowiedzialna jest za jazdę w przód, natomiast dolna w tył (półkule zostały oddzielone czerwoną kreską). Analogicznie lewa półkula pozwala na jazdę w lewo, a prawa w prawo (półkule zostały oddzielone zieloną kreską). Dodatkowo w celu ułatwienia oszacowania żądanej mocy silników utworzono szary okrąg, który wyznacza granicę maksymalnego wychylenia drążka.

Za zdalne sterowanie wirtualnym dżojstikiem odpowiedzialna jest klasa *JoystickView*, (implementuje wygląd oraz położenie drążka) oraz klasa *JoystickViewController*, która nasłuchuje zmian położenia drążka oraz wysyła odpowiednią informację do robota.

```

1 override func viewWillAppear(_ animated: Bool) {
2
3     super.viewWillAppear(animated)
4     serial.delegate = self
5     msg[0] = "5"
6     let rect = view.frame
7     let size = CGSize(width: 150.0, height: 150.0)
8     let joystick1Frame = CGRect(origin: CGPoint(x: rect.width/2 -
9         size.width/2, y: rect.height/2), size: size)
10    let joystick1 = JoyStickView(frame: joystick1Frame)
11
12    view.addSubview(joystick1)
13    joystick1.movable = false
14    joystick1.alpha = 1.0
15    joystick1.baseAlpha = 0.5
16    joystick1.handleTintColor = UIColor.blue
17
18    joystick1.monitor = { angle, displacement in
19        let power = Int(100 * displacement)
20        let direction = Int(angle)
21
22        self.leftTheta.text = "\((direction)"
23        self.leftMagnitude.text = "\((power) %"
24
25        . . .
26
27        usleep(useconds_t(20 * self.ms))
28        serial.sendMessageToDevice(self.msg.joined())
29    }
29 }

```

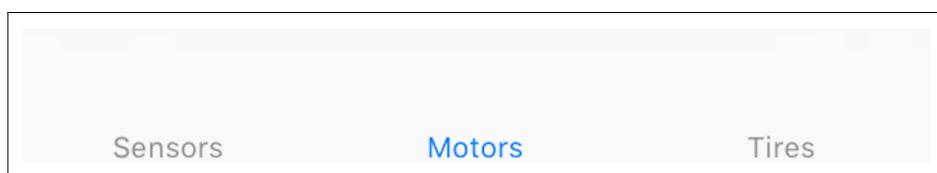
Listing 5.5: Metoda klasy JoystickViewController.

Powyższy listing 5.5 przedstawia implementację najważniejszej funkcji klasy *JoystickViewController*, którą jest *viewWillAppear*. Jest to funkcja dostarczona przez platformę *UIViewController* automatycznie wywoływana po załadowaniu widoku. Linia 5 przedstawia ustawienie pierwszego znaku wiadomości, która informuje robota jaka funkcjonalność powinna zostać obsłużona. Linie 6 – 15 ukazują konfigurację wyglądu dżojstiku, natomiast linia 17 deklaruje nasłuchiwanie zmian położenia drążka, które zwraca zmienne typu *float*: *angle* oraz *displacement*, określające kąt (zakres 0 – 360 stopni) oraz wysunięcie (wartość od 0 do 1) drążka. Linia 18 skaliuje wartość wysunięcia tak, aby wartość mieściła się w przedziale 0 – 100, co jest podyktowane wymogiem struktury wysyłanej wiadomości. Następnie w liniach 21 – 22 aktualizowane są informacje dostarczane użytkownikowi. Po odpowiednim przeliczeniu otrzymanych wartości na kierunki oraz moce silników następuje wysłanie wiadomości do robota (linia 27) poprzedzone

20 milisekundowym opóźnieniem (linia 26). Zastosowane opóźnienie spowodowane było ograniczoną przepustowością modułu *Bluetooth* oraz niską wydajnością aplikacji. Dzięki opóźnieniu uzyskano pewność obsługi każdej wysłanej wiadomości kosztem niezauważalnego opóźnienia aktualizacji widoku przesunięcia drążka.

5.4.5. Panel diagnostyki

Panel diagnostyki powstał z myślą o sprawdzeniu poprawności działania poszczególnych komponentów robota. Niejednokrotnie zdarzało się, iż robot zachowywał się w nieoczekiwany sposób. Wtedy pojawiało się pytanie, czy wina leży po stronie oprogramowania, czy któryś z komponentów nie działa prawidłowo. Dzięki panelowi diagnostyki uzyskano możliwość szybszego zdiagnozowania problemu. Stworzony widok składa się z trzech zakładek: . Nawigowanie między funkcjonalnościami odbywa się za pomocą kliknięcia jednej z zakładek widocznych u dołu ekranu.

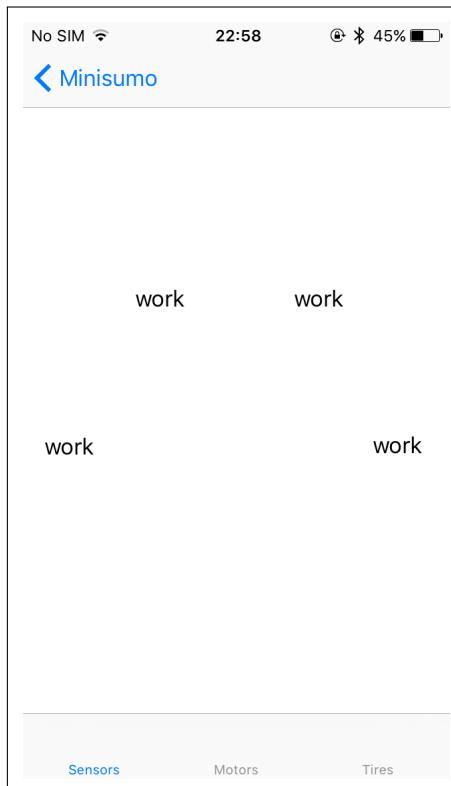


Rys. 5.11: Widok zakładek w panelu diagnostyki.

Rysunek 5.11 przedstawia zakładki nawigujące do poszczególnych opcji diagnostycznych.

Diagnostyka czujników

Panel diagnostyki czujników powstał z myślą o sprawdzeniu poprawności działania czterech czujników wykrywających przeciwnika. Po włączeniu tego widoku zostaje wysłana wiadomość do robota z żądaniem obsłużenia odpowiedniej funkcjonalności. Następnie co 20 milisekund robot wysyła informację zwrotną, zawierającą stan czujników. W przypadku, gdy któryś z czujników wykrył obiekt znajdujący się przed nim, na ekranie pojawia się etykieta *work* w odpowiednim miejscu. Klasą odpowiedzialną za obsługę funkcjonalności jest *CheckSensorsViewController*.



Rys. 5.12: Widok diagnostyki czujników.

Rysunek 5.12 ukazuje widok diagnostyki czujników. Przedstawiono na nim przypadek, gdzie każdy z czujników wykrył przeciwnika. Górnne etykiety odpowiadają czujnikom znajdującym się w frontalnej części robota, natomiast boczne odpowiadają odpowiednio lewemu oraz prawemu czujnikowi.



Rys. 5.13: Poprawność działania diagnostyki czujników.

Natomiast na ilustracji 5.13 przedstawiono działanie omawianego panelu. Po lewej stronie znajduje się robot z zasłoniętym lewym oraz prawym górnym czujnikiem (oznacza to wykrycie przeciwnika), a po prawej stronie telefon z wizualizacją działania aplikacji. Jak widać wiadomość nadesłana z robota została odpowiednio zinterpretowana, ponieważ pojawiła się lewa oraz prawa górną etykietą *work*.

```

1 func parseSensors(state: String, label: UILabel) {
2     if state == "1" {
3         label.text = "work"
4     } else {
5         label.text = " "
6     }
7 }
```

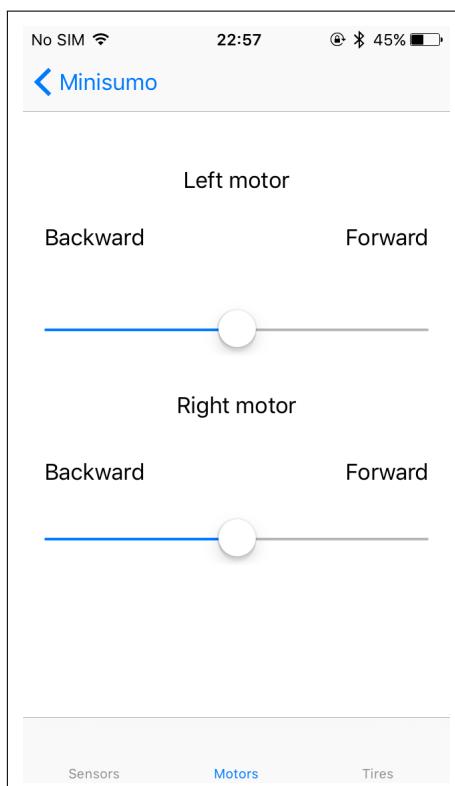
Listing 5.6: Parsowanie wiadomości zawierającej stan czujników.

Powyższy listing 5.6 przedstawia funkcję *parseSensors*, która ma na celu interpretację fragmentu wiadomości odnośnie stanu czujników nadesłanej przez robota. W przypadku otrzymania wartości *1* oznaczającej wykrycie przeciwnika pojawia się etykieta *work*. W innych przypadkach etykieta jest niewidoczna.

Diagnostyka silników

Panel diagnostyki silników powstał w celu sprawdzenia zachowania poszczególnego motoru. Jego widok składa się z dwóch niezależnych suwaków, które służą do ustawienia odpowiedniej mocy oraz kierunku obrotu każdego z silników.

Obrazek 5.14 przedstawia wygląd omawianego panelu. Domyślnie suwaki są wyśrodkowane, co oznacza brak obrotu. Przesunięcie suwaka w lewo powoduje obrót silnika w tył, natomiast w prawo obrót w przód.



Rys. 5.14: Widok diagnostyki silników.

Listing 5.7 zawiera implementację metody *RightMotor* (należącej do klasy *CheckMotorsViewController*), która nasłuchuje zmian położenia suwaka. Zakres wartości mieści się w przedziale od -100 do 100, gdzie wartości ujemne interpretowane są jako obrót silnika w tył (przeciwnie do ruchu wskazówek zegara). Linia 3 przedstawia pobranie wartości położenia suwaka. Linie 4 – 9 ukazują fragment kodu, który parsuje uzyskaną wartość i określa kierunek obrotu. Następnie wartość położenia suwaka jest parsowana na pojedyncze znaki oraz przypisywana do odpowiednich indeksów tablicy *msg*, która zostaje wysłana do robota. Wiadomość z żądaną mocą oraz kierunkiem obrotu silnika wysyłana jest co 200 milisekund z powodów opisanych w poprzednich podrozdziałach pracy. Analogicznie dla lewego silnika powstała metoda *LeftMotor*.

```

1 IBAction func RightMotor(_ sender: UISlider) {
2
3     rightMotorPower = Int(sender.value)
4     if rightMotorPower < 0 {
5         rightMotorDirection = 1
6         rightMotorPower *= -1
7     } else {
8         rightMotorDirection = 2
9     }
10
11    msg[2] = String(rightMotorDirection)
12
13    if rightMotorPower < 10 {
14        msg[5] = "0"
15        msg[6] = String(rightMotorPower)
16    } else {
17        msg[5] = String(rightMotorPower / 10)
18        msg[6] = String(rightMotorPower % 10)
19    }
20
21    usleep(useconds_t(20 * ms))
22    print(msg.joined())
23    serial.sendMessageToDevice(msg.joined())
24 }

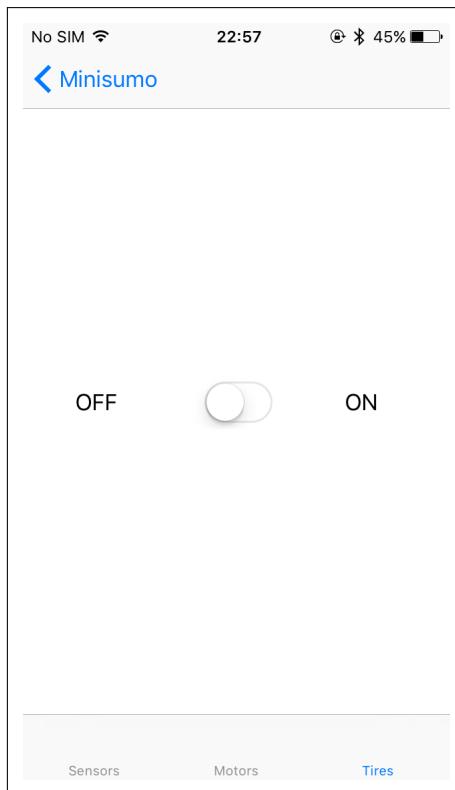
```

Listing 5.7: Nasłuchiwanie zmiany położenia suwaka.

Czyszczenie opon

Panel czyszczenia opon jest uboższą wersją panelu diagnostyki silników. Składa się z jednego przełącznika, który ustala moc obu silników na 20% dostępnej mocy. Funkcjonalność powstała z myślą o czyszczeniu opon robota, które wykonane zostały z poliuretanu przez co bardzo szybko tracą przyczepność z powodu dużej podatności na zbieranie brudu.

Poniższe zdjęcie 5.15 przedstawia wygląd panelu służącemu czyszczeniu opon konstrukcji.



Rys. 5.15: Widok funkcji czyszczenia opon.

Na listingu 5.8 przedstawiono metodę *changedState* będącą członkiem klasy *CleaningTiresViewController*. Wywoływana jest w momencie zmiany położenia przełącznika, a następnie w zależności od jego stanu wysyła odpowiednio skonstruowaną wiadomość do robota.

```
1 IBAction func changedState(_ sender: UISwitch) {  
2     if sender.isOn {  
3         msg[0] = "3"  
4         serial.sendMessageToDevice(msg.joined())  
5     } else {  
6         msg[0] = "0"  
7         serial.sendMessageToDevice(msg.joined())  
8     }  
9 }
```

Listing 5.8: Nasłuchiwanie zmiany położenia przełącznika.

Rozdział 6

Implementacja

6.1. Kompilacja projektu

Kompilacja projektu składa się z 2 etapów – kompilacji programu uruchamianego w robocie oraz aplikacji mobilnej. W celu wgrania oprogramowania robota należy skompilować kod źródłowy w środowisku *Eclipse*, a następnie utworzony plik binarny przesłać do procesora pojazdu przy użyciu środowiska *STMStudio*. Rozwiążanie nie należy do najwygodniejszych, szczególnie na etapie testowania. Niestety jest to jedyny sposób, wynikający z błędów popełnionych na etapie projektowania, które zostaną opisane w podsumowaniu. Drugi etap jest mniej problematyczny, ponieważ nie wymaga integracji oprogramowania z niededykowanym urządzeniem. Kompilacja kodu aplikacji mobilnej odbywa się w obrębie jednego narzędzia jakim jest *Xcode*.

Rozdział 7

Podsumowanie

7.1. Zrealizowane założenia

7.2. Dalszy rozwój projektu

7.3. Uwagi

Bibliografia

- [1] *Kurs HAL* <https://forbot.pl/blog/kurs-stm32-f4-1-czas-poznac-hal-spis-tresci-kursu-id14114> (dostęp 09.11.2017).
- [2] *Dokumentacja Apple* https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/index.html (dostęp 09.11.2017).
- [3] *Arduino* <https://www.arduino.cc/> (dostęp 09.11.2017).
- [4] *Zdjęcie zawodów sumo* <https://cnet4.cbsistatic.com/img/gH1BrN3y1qHjwlB6gZuFLHFXu9g=/670x503/2017/06/20/017bb355-4376-4e00-a0c5-8b2db95e08c6/sumorobots1.jpg> (dostęp 09.11.2017).
- [5] *Dokumentacja akcelerometru* https://developer.apple.com/documentation/coremotion/getting_raw_accelerometer_events (dostęp 09.11.2017).
- [6] Elliot Williams *Make: AVR programming* Wydawnictwo Maker Media, 2014.