POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka (AIR)
SPECJALNOŚĆ: Systemy informatyczne w automatyce (ASI)

PRACA DYPLOMOWA INŻYNIERSKA

Budowa i oprogramowanie robota klasy (2,0) ze zdalnym interfejsem sterowania

Hardware and software design of remotely controlled (2,0) class robot

AUTOR:
Kornel Mrozek

PROWADZĄCY PRACĘ:
Dr inż. Janusz Jakubiak

OCENA PRACY:

WROCŁAW 2015

Spis treści

Rozdział 1

Wprowadzenie

Robot mobilny klasy 2,0 jest robotem posiadającym dwukołowy napęd różnicowy. Przemieszczania robota umożliwia sterowanie prędkością obrotu koła przymocowanego do danego napędu bez możliwości zmiany kierunku ułożenia koła w stosunku do platformy robota.

1.1 Cel projektu

Celem projektu było zbudowanie fizycznego robota mobilnego 2.0 wraz z napisaniem oprogramowania na mikrokontroler oraz aplikacji, która umożliwia zdalne sterowanie robotem. Komunikacja pomiędzy platformą mobilną a programem sterującym oparta jest na standardzie Bluetooth. Celem pracy było umożliwianie jak największej interakcji z robotem zarówno w czasie rzeczywistym, jak i za pomocą komend tekstowych, z możliwością pisania skryptów. Główny nacisk położony został na rozwinięcie niezawodnego i rozszerzalnego oprogramowania zarówna na platformie mobilnej, jak i aplikacji desktopowej.

1.2 Założenie projektowe

Projekt zakłada przygotowanie modelu mechanicznego zawierającego dwukołowy napęd różnicowy wraz z układem elektronicznym umożliwiającym sterowanie silnikiem prądu stałego, jak również odczytem prędkości obrotu wału silnika. W układzie elektronicznym konieczny jest moduł Bluetooth umożliwiający komunikację szeregową z mikrokontrolerem. Oprogramowanie mikrokontrolera osadzonego na robocie zawiera:

- komunikację dwukierunkową interfejsu szeregowego
- asynchroniczną reakcję na otrzymane wiadomości z zewnątrz
- obsługę zliczania impulsów (wymagane przy odczycie prędkości)
- układ regulacji prędkości
- zabezpieczenie przed utratą transmisji

Aby możliwa była komunikacja między robotem a otoczeniem z zewnątrz wymagane jest przygotowanie protokołu komunikacyjnego dającego aplikacji interfejs na którym może operować. Protokół zawiera wiadomości typu pobierz/zapisz które pozwalają na

pobranie z robota wartości i ich nadpisanie. Aplikacja sterująca, zgodnie z celem projektu, ma umożliwiać szeroki zakres form sterowania, opierając się na wyżej wymienionym protokole. Oprogramowanie sterujące zakłada:

- możliwość zapisu i odczytu danych w standardzie Bluetooth poprzez wirtualny port COM
- możliwość sterowania robotem za pomocą komend tekstowych. Punkt ten zakłada przygotowanie modułu interpretera do wprowadzania komend/skryptów oraz wykorzystanie silnika języka JavaScript
- możliwość sterowania w trybie graficznym za pomocą myszki, wizualizacja prędkości oraz konfiguracja robota

Rozdział 2

Konstrukcja sprzętowa

2.1 Platforma mechaniczna

Platforma mechaniczna robota oparta jest o model Zumo Chassis Kit firmy Pololu. Podstawa mechaniczna zaprojektowana została do konstrukcji robotów o napędzie różnicowym wyposażonym w miniaturowe silniki elektryczne prądu stałego tej samej firmy. Model wyposażony jest w gąsienicowy układ bieżny. Główną zaletą tego rozwiązania jest zwiększenia powierzchni styku z podłożem, co zwiększa przyczepność i zmniejsza prawdopodobieństwo poślizgu poprzecznego. Jest szczególnie istotne w przypadku użycia odometrii przy pomiarze prędkości robota. Platforma skonstruowana została z myślą o używaniu baterii AA, do zasilania robota. Projekt zakładał jednak zastosowanie akumulatora Li-Pol jako źródła energii. Konstrukcja napędu, jak również rozmiar i cena były przyczyną wykorzystania właśnie tego modelu.

(a) Platforma mechaniczna

(b) SLinik wrazz enkoderem

Rysunek 2.1: Gotowy moduł nadajnika

Wymiary modelu mechanicznego:

- wymiary pełnego modelu: 86[mm] x 98[mm]
- odległość pomiędzy kołami (pomiar od środka gąsienicy): 7.5[6mm]

- przestrzeń na podstawę elektroniczną:71[mm] x 80[mm]
- średnica koła: 1.56[mm]

Napędami są miniaturowe silniki Pololu HP z dwustronną osią. Silnik posiada przekładnie 50:1 co ogranicza prędkość obrotową wału silnika zwiększając jednocześnie jego moment. Obustronny wał pozwala na wygodny montaż enkoderów, czyli czujników wykorzystywanych przy pomiarze prędkości obrotowej silnika. Parametry silnika:

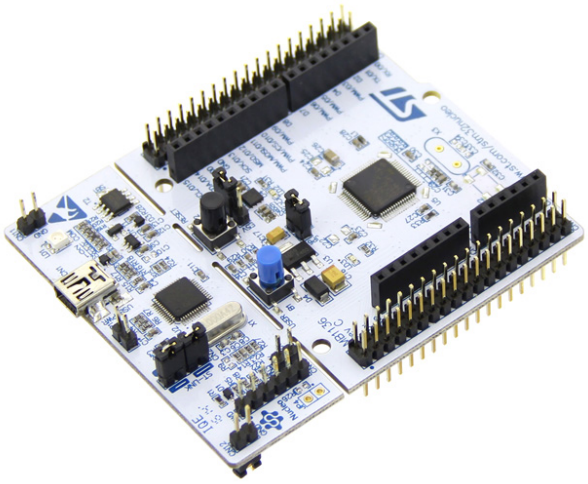
- Maksymalne napięcie pracy: 9[V]
- Prędkość obrotowa dla napięcia 6[V]: 625 obrotów na minutę
- Moment obrotowy dla napięcia 6[V]: 1.1 [kg*cm]
- Maksymalny prąd dla napięcia 6[V]: 1600[mA]

Wybór silnika podyktowany był nie tylko możliwościami jakie oferuje, ale przed wszystkim pełną iteracją z modelem mechanicznym oraz enkoderami (montaż zaprezentowany jest na rysunku powyżej). Dzięki temu konstrukcja była przebiegła szybciej i w dużym stopniu zapobiegła wszelkim błędom montażom.

2.2 Konstrukcja elektroniczna

Na konstrukcje elektroniczną robota składają się 4 moduły:

- układ Nucleo F401RE z mikrokontrolerem STM32F401
- moduł Bluetooth HC-05
- zestaw enkoderów magnetycznych
- bazowy obwód łączący układ Nucleo z silnikami
- akumulator LiPol Redox 7.4 V 500mAh



Rysunek 2.2: Wizualizacja zasady wyznaczania pozycji

Elektronika robota opera się o układ Nucleo F401RE.Sercem układu jest mikrokontroler STM32F401.Posiada on 32-bitową architekturę opartą o rdzeń ARM Cortex M-4 i może być taktowany do 84 MHz. Wydajna jednostka obliczeniowa wraz z sprzętową obsługą operacji na liczbach zmiennoprzecinkowych była kluczowa przy realizacji projektu. Oprócz szybkiego przetwarzania danych od mikrokontrolera wymagane była sprzętowa realizacja komunikacji szeregowej oraz duża liczba timerów. Obecny mikrokontroler posiada kontroler DMA (ang. Direct Memory Access),pozwalający przeprowadzać transmisję danych bez obciążania procesora, jak również 10 sprzętowych timerów, co pokrywa wymagania projektowanego systemu. Parametry STM32F401:
Parametry STM32F401:

- Częstotliwość taktowania: 84[MHz]
- Pamięć trwała Flash: 512kB
- Pamięć Static RAM: 96kB
- Ilość programowalnych wejść/wyjść: 81
- Interfejsy: 3x I2C, 3x USART, 4x SPI, USB 2.0 Full Speed

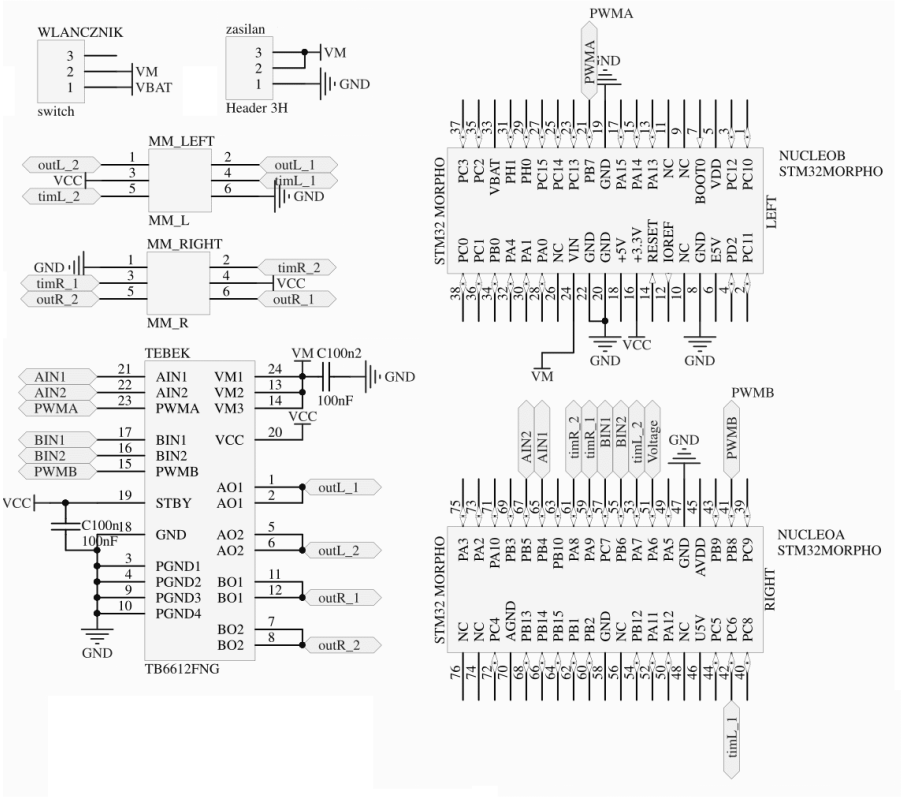
Nie licząc wypowadzeńpinów mikrokontrolera, istotnymi elementami w Nucleo F401RE są stabilizatorynapięcia na 3.3V i 5V z możliwością zasilania bateryjnego od 9V do 12V oraz progrmator/debuger ST-Link v2 .
Komunikacja bezprzewodowa umożliwia moduł HC-05. Techniczne szczegóły opisane są w rozdziale 3.3. Najważniejsze parametry HC-05:

- Napięcie zasilania: 3.3[V]
- Zasięg: do 10[m]
- Komunikacja: UART
- Standard: Bluetooth 2.0 + EDR

Odczyt prędkości realizowany jest przez enkodery magnetyczneMagnetic Encoder Pair Kit for Micro Metal Gearmotors formy Pololu. Wybór jest wynikał głównie z wyżej wspomnianej integracji z elementami mechanicznymi robota. Podstawą fizyczną działania magnetycznych enkoderów inkrementacyjnych jest efekt Halla.

2.2.1 Bazowy obwód

Układ elektroniczny zaprojektowany został w programie CircuitMaker firmy Altium. Zadaniem układu było połączenie modułu Nucleo z wyprowadzeniami enkoderami i umieszczenie sterownika silnika. Powszechnym sposobem kontroli silnika prądu stałego jest zastosowanie mostka H. Układ ten pozwala na sterowanie kierunkiem przepływu prądu przez silnik. Funkcja realizowana jest przez układ scalony TB6612FNG, który pozwala na kontrole kierunku obrotów dwóch silników oraz regulacje ich prędkości sygnałem PWM (ang. Pulse Width Modulation). Sygnał zasilający silniki oraz wyprowadzenia enkoderów połączone z układem jest za pośrednictwem złącza micromatch . Wszystkie sygnały podłączone są do mikrokontrolera przez wyprowadzenia Nucelo.

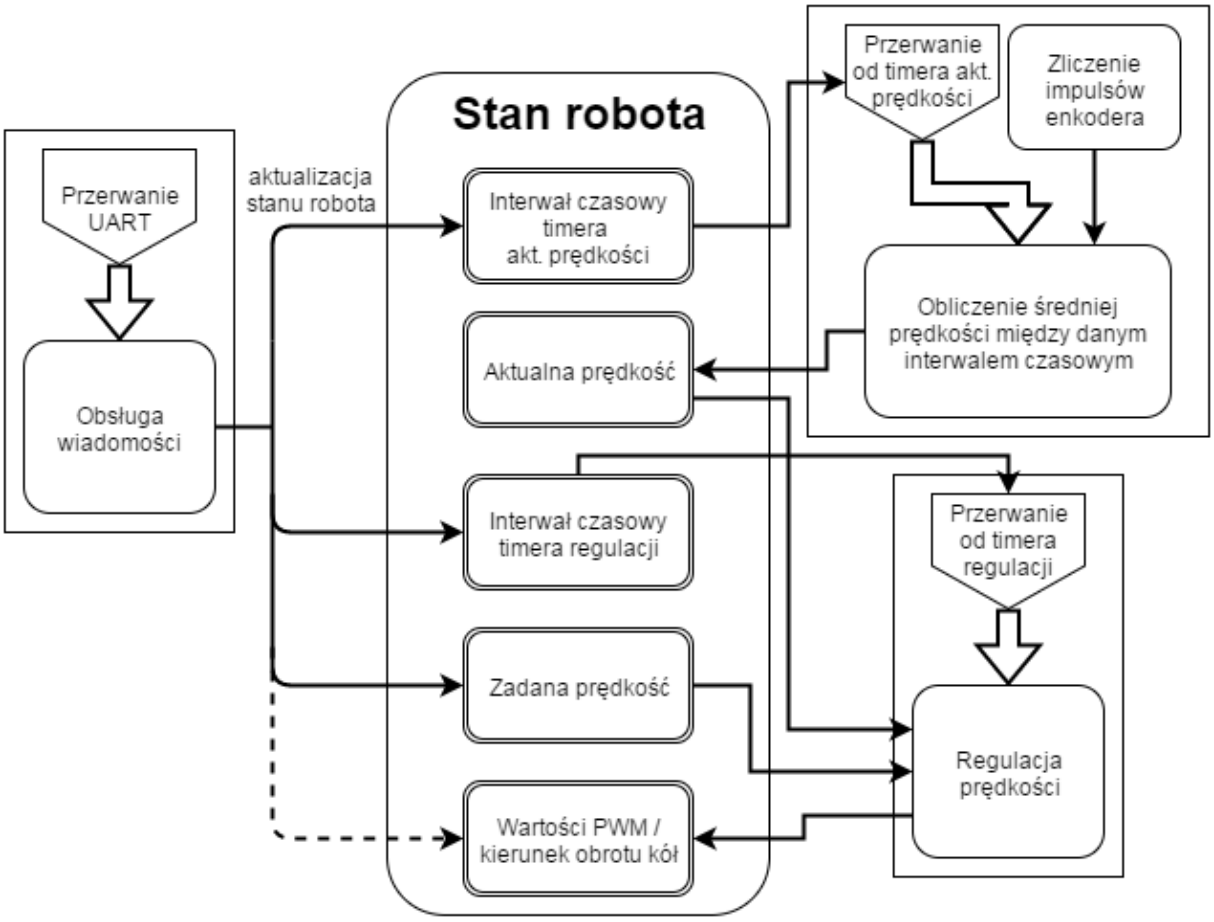


Rysunek 2.3: Schemat układu elektronicznego

Rozdział 3

Oprogramowanie robota mobilnego

3.1 Struktura oprogramowania



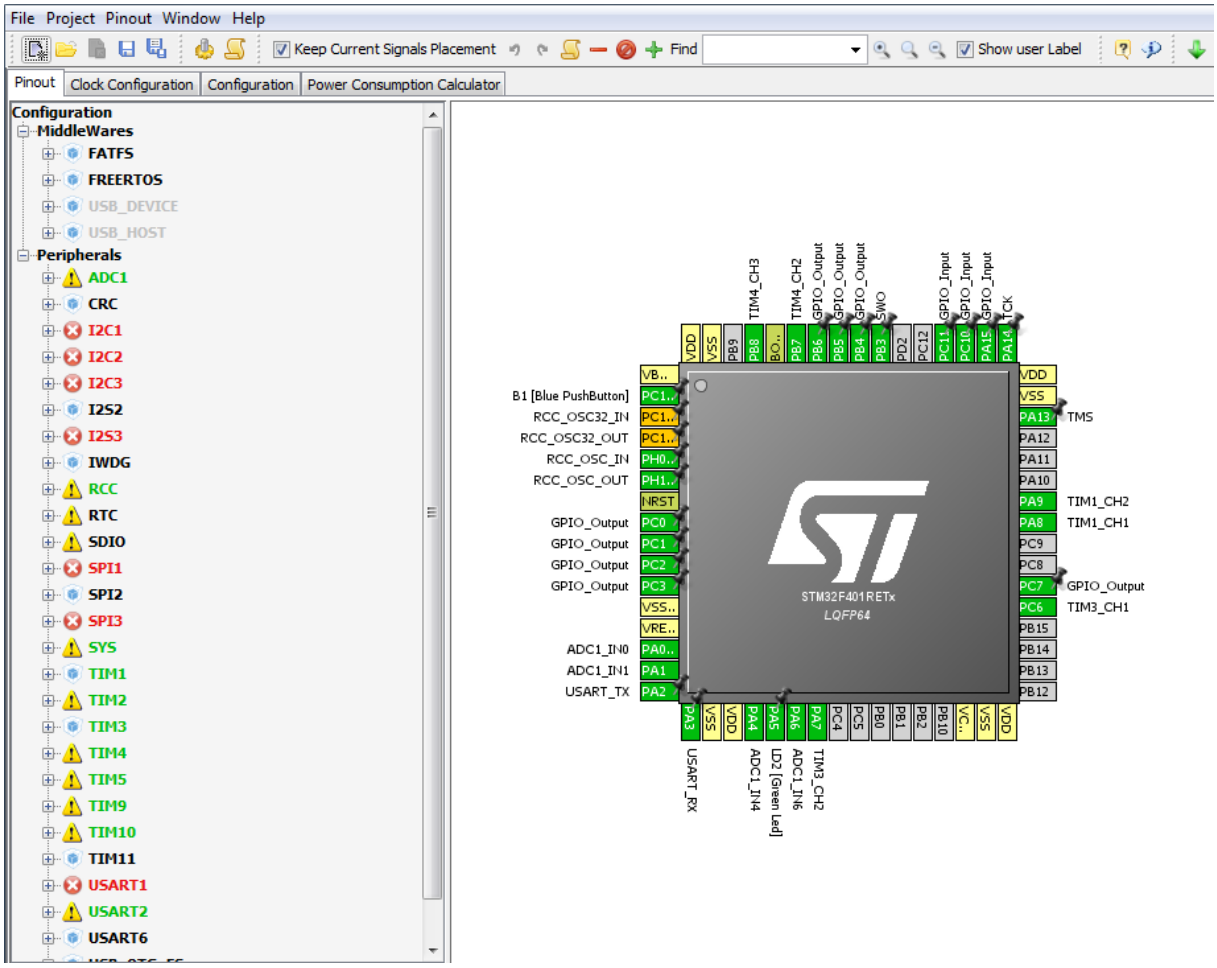
Rysunek 3.1: Schemat blokowy głównego algorytmu robota

Oprogramowanie napisane jest w języku C korzystając z bibliotek HAL (Hardware Abstraction Layer) udostępnionych przez firmę STMicroelectronics. Powyższy schemat pokazuje rozwiązanie podstawowej funkcjonalności robota, czyli regulacji prędkości. Stan robota wyrażonych jest zmiennych globalnych, które poszczególne procedury modyfikują wartość. Cienkie strzałki wskazują kierunek przepływu danych, natomiast grube wskazują na przerwania sprzętowa wyzwalające dane funkcję obsługi. Bezpośrednia zmiana wypełnienia sygnału PWM oraz kierunek obrotu silników za pomocą wiadomości została

zaznaczona linią przerywaną. Modyfikacja tych wartości nie jest możliwa podczas trybu regulacji prędkości i wymaga wyłączenia tej procedury(Zagadnienie jest omówione w rozdziale piątym). Przedstawione na schemacie procedury opisane są w dalszej części tego rozdziału.

3.2 Konfiguracja mikrokontrolera

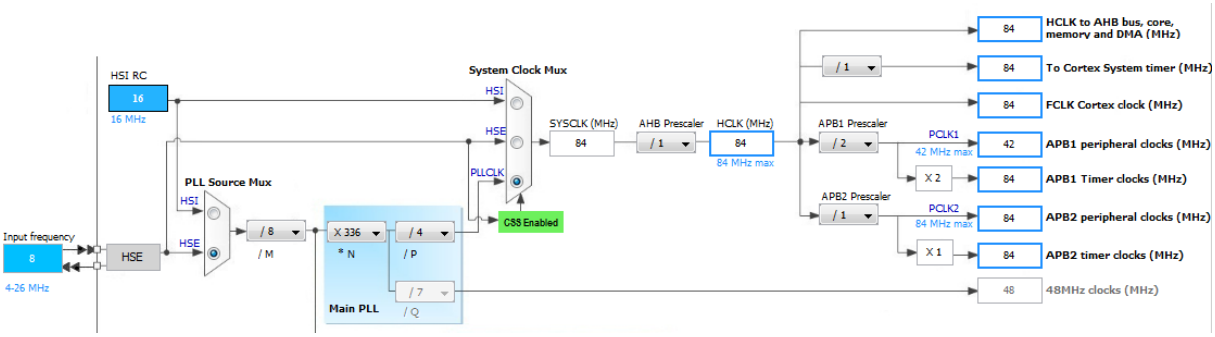
Konfiguracja mikrokontrolera wykonana jest przy użyciu oprogramowania STMCubeMX. Pozwala one w bezpieczny i wygony sposób dokonać konfiguracji.



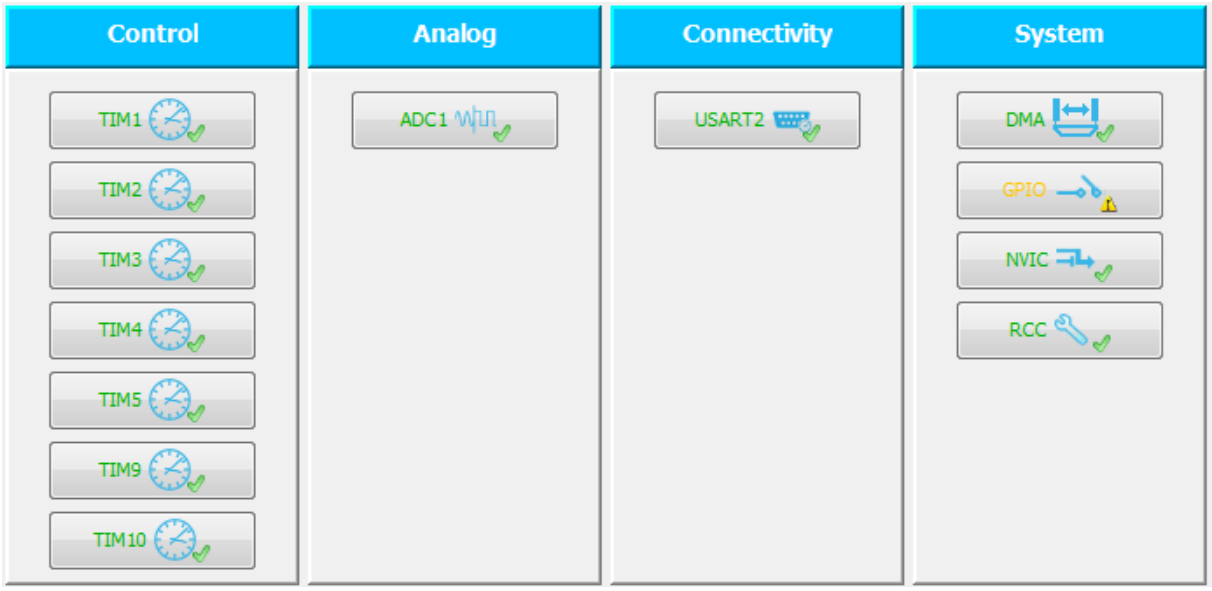
Rysunek 3.2: Konfiguracja pinów mikrokontrolera

Mikrokontroler jest taktowany z częstotliwością 84 MHz. Źródłem taktowania jest rezonator kwarcowy 8 MHz, którego częstotliwość jest powielana przez wewnętrzną pętlę PLL.

Oprogramowanie wykorzystuje 7 sprzętowych timerów, obsługę transmisji szeregowej UART, magistrale DMA oraz kontroler przerwań NVIC. Oprogramowanie nie zawiera żadnej instrukcji w pętli głównej programu, zachowanie robota jest całkowicie sterowane sprzętowymi przerwaniami.



Rysunek 3.3: Konfiguaracja taktowania



Rysunek 3.4: Wykorzystane komponenty STM32F401

3.3 Realizacja komunikacji Bluetooth

Komunikacja Bluetooth wykorzystuje obsługę dwóch komponentów mikrokontrolera, interfejsu szeregowo UART oraz technika DMA. UART (Universal Asynchronous Receiver and Transmitter) umożliwia asynchroniczne wysyłanie i odbieranie danych przez port szeregowy. Wykorzystanie modułu Bluetooth ogranicza się do połączenia linii danych TX (wysyłanie) mikrokontrolera z linią danych RX (odbiór) układu HC-06 oraz analogicznie linii RX mikrokontrolera z linią TX modułu. Transmisja bezprzewodowa jest przezroczysta z poziomu mikrokontrolera. Interfejs UART konfigurowany jest przed podanie 4 wartości określających transmisję:

- prędkość transmisji (ang. Baud rate)
- długość słowa
- ilość bitów stopu
- bit parzystości (kontrola błędów odbiory danych)

Zdecydowałem się na użycie standardowej konfiguracji prędkość: 9600, 8-bitowe słowo, jeden bit stopu, brak bitów parzystości.

DMA jest modułem który umożliwia bezpośredni dostęp do pamięci RAM i układów peryferyjnych. Układ DMA umożliwia transfer danych pomiędzy urządzeniem peryferyjnym a pamięcią RAM bez angażowania procesora, którego zadanie ogranicza się do konfiguracji urządzenia. Komponent został użyty do przesyłu i odbioru danych pomiędzy układem UART i dziesięciobajtowym obszarem pamięci RAM. Włączenie odbioru danych szeregowych z użyciem DMA:

```
1 int8_t command[10];
2 HAL_UART_Receive_DMA(&huart2, command, sizeof(command));
```

Argumentami tej funkcji są adres obiektu obsługującego interfejs UART, dziesięciobajtowy obszar pamięci (tablica dziesięciu bajtów utworzona w globalnej przestrzeni pamięci) oraz długość tego obszaru.
Asynchroniczność transmisji możliwa jest poprzez obsługę przerwania które generuje UART po zakończeniu transmisji. Biblioteki HAL zapewniają niskopoziomową obsługę przerwań oraz deklaracje zestawu funkcji zwrotnych które zostają wykonane w odpowiedzi na zarejestrowane przerwanie. Obsługa zakończenia transmisji:

```
1 voidHAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
2 {
3         int8_t response[10];
4         int8_t currentCommand[10];
5         uint8_t index = 0;
6         for(;index < 10; ++index)
7                 currentCommand[index] = command;
8         if(getTimerTimeout(&htim9))
9                 stopBluetoothTimer();
10        commandHandler(currentCommand,response);
11        HAL_UART_Transmit_DMA(&huart2,response,10);
12        if(getTimerTimeout(&htim9))
13                startBluetoothTimer();
14 }
```

Funkcja jest przez bibliotekę HAL wywoływana dla zarejestrowanego przerwania od dowolnego interfejsu UART. Obiekt UART komponentu który wywołał przerwanie jest przekazywany jako argument funkcji. Z uwagi na to, że używany jest jeden interfejs szeregowy nie ma potrzeby sprawdzania czy to on jest źródłem przerwania. Funkcja tworzy tablicę bajtów - response, która wraz z kopią odebranej komendy przekazywana jest do funkcji obsługującej daną wiadomość. Wypełniona odpowiedź wysyłana jest tą samą techniką. Obsługa wiadomości oraz zastosowany timer opisany jest w dalszej części tego rozdziału. Problem, jaki należy rozważyć, w przypadku komunikacji asynchronicznej, jest sytuacja, w której wiadomość jest odbierana zanim wcześniejsza zostanie obsłużona. Dodatkową komplikacja, w tej sytuacji jest użycie DMA, które wypełnia wartości w pamięci z pominięcie pracy, a tym samym kontroli procesora. Praca, na tym samym obszarze pamięci, które wypełnia DMA, podczas obsługi, mogłaby skutkować zmianą wartości tablicy podczas odczytu z niej wiadomości. Z tego powodu, pierwszym etapem w obsłudze przerwania, jest utworzenie kopii tablicy commandi praca na duplikacie. To rozwiązanie, zwiększa stabilność przetwarzania wiadomości, ale nie rozwiązuje problemu całkowicie. Nowa wiadomość mogłaby pojawić się podczas tworzenia duplikatu, co znów generuje ten sam problem. Z pomocą przychodzi tutaj sprzętowa obsługa przerwańw mikrokontrolerach STM. Przetworzenie wiadomości wywoływana jest w obsłudze przerwania, które kończy się dopiero po zaaplikowaniu komendy. Kontroler przerwań NVIC, w przypadku takiego samego priorytetu, tworzy kolejkę LIFO przerwań i a następnie kolejno przekazuje sterowanie, po zakończeniu aktualnie wywołanej funkcji obsługi. Z tego powodu handler nie wywoła się

przed zakończeniem przetwarzania bieżącej komendy. Poza wyżej wymienionymi zabezpieczeniami omawiany błąd eliminowany jest przez konstrukcję protokołu komunikacyjnego z aplikacją sterującą. Zgodnie z nim, aplikacja nie wysyła, następnej wiadomości bez otrzymania odpowiedzi na poprzednią, w przypadku prawidłowej wymiany danych, rozwiązuje ten problem.

3.4 Proces przetwarzania wiadomosci

W poprzednim rozdziale zaprezentowana jest obsługa przerwania od zakończenia transmisji szeregowej. Obecna jest funkcja „commandHandler" odpowiedzialna za obsługę wiadomości i uformowanie odpowiedzi. Struktura funkcji „commandHandler" (dla czytelności pominąłem fragment funkcji,n gdyż nie wpływa to na zrozumienie jej działania):

```
1
2 int8_t* commandHandler(int8_t* command,int8_t* response)
3 {
4 if(command[0] != 'B' || command[4] != 'M' || command[9] !='E')
5 return invalidMessage(response);
6 if(command[1] == 1)
7 {
8 //set speed
9 returnsetVelocityResponse(command,response);
10 }
11 if(command[1] == 2)
12 {
13 //get speed
14 returngetVelocityResponse(command,response);
15 }
16 //(...)
17 if(command[1] == 14)
18 {
19 //set direction pins
20 returnsetMotorsDirection(command,response);
21 }
22 if(command[1] == 15)
23 {
24 //get direction pins
25 returngetMotorsDirection(command,response);
26 }
27 returninvalidMessage(response);
28 }
```

Działanie prezentowanej funkcji jest ściśle związane z protokołem komunikacyjnym pomiędzy robotem a aplikacją sterującą. Zagadnienie to jest szczegółowo opisane w rozdziale piątym. Początek funkcji sprawdza czy na odpowiednich pozycjach występują wartości stałe i charakterystyczne dla każdej wiadomości, w przeciwnym wypadku wysyłana jest wiadomość informująca o błędzie. Następnie na podstawie identyfikatora wiadomości wywoływana jest konkretna funkcja obsługująca daną wiadomość, a jeżeli podany identyfikator nie jest znany zwracany jest błąd.
Obsługa wiadomości zaprezentowana będzie na przykładzie komendy ustalające prędkość zadaną w układzie regulacji:

```
1
2 int8_t* setVelocityResponse(int8_t* command,int8_t* response)
3 {
4 uint8_tindex = 0;
```

```
 5 for(;index<10; ++index)
 6 response[index] = 0;
 7 response[0] = 'B';
 8 response[1] = command[1];
 9 response[2] = command[2];
10 response[3] = command[3];
11 response[4] = command[5];
12 response[5] = command[6];
13 if(command[2] == 0 && command[3] == 0 && command[5] == 0 && command
       [6] == 0)
14 {
15 stopMotors();
16 return response;
17 }
18 int8_t data[2];
19
20 data[0] = command[2];
21 data[1] = command[3];
22 setLeftSpeed = byte2Float(data);
23
24 data[0] = command[5];
25 data[1] = command[6];
26 setRightSpeed = byte2Float(data);
27
28 if(command[7] != 0&& command[8] != 0)
29 {
30 data[0] = command[7];
31 data[1] = command[8];
32 uint32_t timer = byte2int(data);
33
34 startSpeedTimer(timer);
35 }
36
37 return response;
```

Działanie prezentowanej funkcji jest ściśle związane z protokołem komunikacyjnym pomiędzy robotem a aplikacją sterującą. Zagadnienie to jest szczegółowo opisane w rozdziale piątym. Początek funkcji sprawdza czy na odpowiednich pozycjach występują wartości stałe i charakterystyczne dla każdej wiadomości, w przeciwnym wypadku wysyłana jest wiadomość informująca o błędzie. Następnie na podstawie identyfikatora wiadomości wywoływana jest konkretna funkcja obsługująca daną wiadomość, a jeżeli podany identyfikator nie jest znany zwracany jest błąd.
Obsługa wiadomości zaprezentowana będzie na przykładzie komendy ustalające prędkość zadaną w układzie regulacji:

```
 1
 2 int8_t* setVelocityResponse(int8_t* command,int8_t* response)
 3 {
 4 uint8_tindex = 0;
 5 for(;index<10; ++index)
 6 response[index] = 0;
 7 response[0] = 'B';
 8 response[1] = command[1];
 9 response[2] = command[2];
10 response[3] = command[3];
11 response[4] = command[5];
12 response[5] = command[6];
13 if(command[2] == 0&& command[3] == 0&& command[5] == 0&& command[6]
```

```
      == 0)
14 {
15 stopMotors();
16 return response;
17 }
18 int8_t data[2];
19
20 data[0] = command[2];
21 data[1] = command[3];
22 setLeftSpeed = byte2Float(data);
23
24 data[0] = command[5];
25 data[1] = command[6];
26 setRightSpeed = byte2Float(data);
27
28 if(command[7] != 0&& command[8] != 0)
29 {
30 data[0] = command[7];
31 data[1] = command[8];
32 uint32_t timer = byte2int(data);
33
34 startSpeedTimer(timer);
35 }
36
37 return response;
```

Każda funkcja początkowo wypełnia odpowiedź zerami i wartościami stałymi. Dla wiadomości typu „ustaw" odpowiedź powinna być identyczna jak komenda przychodząca. Dzięki temu aplikacja sterująca może mieć pewność, że ustawione zostały podane wartości. Pobór fatycznych wartości ustawionych na robocie odbywa się za pomocą wiadomości typu „pobierz". Zaprezentowana wiadomość ustawia 3 pozycje: prędkość lewego koła, prędkość prawego koła (w cm/s) oraz opcjonalnie czas trwania zadanego ruchu. Wiadomości ustawiające prędkość charakteryzują się jeszcze jedną właściwością. Mając wszystkie pola wartości wypełnione zerami wywołuje się funkcja zatrzymania silników robota. Jest to jedna z form zabezpieczenia. Warunek ten jest sprawdzany od razu po uformowaniu odpowiedzi. Kolejnym etapem jest konwersja odpowiednich bajtów na liczbę zmiennoprzecinkową i zapis wartości do globalnej zmiennej określającej zadaną prędkość dla danego koła. Ostatnim etapem obsługi wiadomości jest ustawienie czasu trwania długości danego ruchu. Jeżeli pola nie są wypełnione zerami wystartuje timer z przesłanymi danymi ilościami milisekund, i po tym czasie zostaną zatrzymane silniki. W przeciwnym razie zadana prędkość będzie się utrzymywała aż do jej zmiany. Po zakończeniu procedury zwracany jest adres z odpowiedzią.

3.5 Zabezpieczenia robota

Podstawową funkcją wykorzystywaną we każdej formie zabezpieczeń robota jest stopMotors:

```
1 voidstopMotors(void)
2 {
3         setLeftSpeed = 0.0;
4         setRightSpeed = 0.0;
5
6         LeftPWM(0);
7         RightPWM(0);
```

```
8         stopSpeedTimer();
9 }
```

Funkcja wyzerowuje wartość zadaną oraz ustawiony PWM, jak również zatrzymuje timer odpowiedzialny za czas trwania ustawionej prędkości. Robot posiada 3 mechanizmy zabezpieczeń. Pierwszym jest wymienione w poprzednim rozdziale wysłanie wiadomości zadającej prędkość z wartościami równymi zero. Kolejnym mechanizmem jest wysłanie niepoprawnej wiadomości. Funkcja przygotowująca odpowiedź informującą o błędzie również zatrzymuje silniki. Ostatnim zabezpieczeniem jest możliwość ustawienia timera Bluetooth (widoczny jest we fragmencie kodu w podrozdziale 3.2). Timer zaczyna odliczanie po otrzymaniu dowolnej wiadomości i gdy następna wiadomość nie pojawi się przed zakończeniem odliczania, silniki są zatrzymywane. Jest to forma zabezpieczenia przed utratą łączności. Domyślnie timer ten jest wyłączony. Włączenie oraz ustalenie czasu odliczania jest możliwe za pomocą wiadomości kontrolnej.

3.6 Realizacja odczytu prędkości

Aby możliwa była regulacja prędkości wymagane jest sprzężenie zwrotne od faktycznej prędkości osiągniętej przez robota. W tym celu kluczowe jest niezawodne przetwarzanie informacji uzyskanych za pomocą enkoderów. Enkodery zamontowane na wale silników podłączone są to pinów skonfigurowanych jako wejścia kanałów timerów obsługujących ten czujniki. Mikrokontroler STM umożliwia skonfigurowanie timera do trybu Encoder Mode, który wiąże wejście sygnału taktującego licznika z impulsem generowanym przez enkoder. Tryb ten również pozwala na sprzętową analizę kolejności generowanych impulsów wyjściowych, co w wygody sposób pozwala na detekcje kierunku obrotu koła. (...). Do obliczania prędkości użyty został jeszcze jeden niezależny timer, który o ustawiony interwał aktualizuje globalne zmienne zawierające aktualne prędkości kół robota.

```
1 voidHAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
2 {
3         if(htim == &htim2)
4         {
5                 leftVelocity = LeftVelocity();
6                 rightVelocity = RightVelocity();
7                 htim2.Instance->CNT = encoderTimer - 1;
8                 HAL_TIM_Base_Start_IT(&htim2);
9         }
10        //(...)
11        else if(htim == &htim3)
12        {
13                onEncoderOverload(&htim3,&leftTotalTicks);
14        }
15        Else if(htim == &htim1)
16        {
17                onEncoderOverload(&htim1,&rightTotalTicks);
18        }
19 }
```

Jest to uniwersalny handler dla wszystkich timerów zarejestrowanych na przerwanie od końca odliczania. Biblioteka HAL obsługuje niskopoziomowo przerwanie timera i wywołuje funkcje obsługi przekazują w argumencie adres obiektu kontrolującego dany licznik. Timer (htim2) odpowiedzialny za aktualizację prędkości jest obsługiwany na początku zapisuje wyniki funkcji obliczających prędkość oraz zapisuje je zmiennych globalnych. Interwał czasowy dla licznika może być ustalony przez wiadomość kontrolną. Obliczona wartość

jest więc średnią prędkością w ustawionej ramie czasowej. Z powodu, że interwał aktualizacji prędkości może być ustalany dowolną szesnastobitową wartością oraz maksymalna prędkość jaką mogą uzyskać koła nie jest stała (zależy od zużycia silników, poziomu naładowania baterii, użytej przekładni i innych czynników) wymagane było zabezpieczenie przed przekroczeniem maksymalnej wartości timera zanim nastąpi aktualizacja prędkości. Obsługę tej sytuacji zapewnia funkcja:

```
1 void onEncoderOverload(TIM_HandleTypeDef *htim,int32_t *
    totalTickCounter)
2 {
3 if(htim->Instance->CNT >64000&& htim->Instance->CNT <65000)
4         *totalTickCounter += (ENCODER_INITIAL + 65000 - htim->
              Instance->CNT);
5 else if (htim->Instance->CNT >0&& htim->Instance->CNT <1000)
6         *totalTickCounter -= (ENCODER_INITIAL + htim->Instance->CNT);
7
8 htim->Instance->CNT = ENCODER_INITIAL;
9 HAL_TIM_Base_Start_IT(htim);
10 }
```

Argumentami jest adres danego timera oraz adres zmiennej pomocniczej określającej ilość impulsów wysłanych przez enkoder od ostatniej aktualizacji prędkości. Zmienna „totalTickCounter" jest ze znakiem co umożliwia określenie zwrotu prędkości. Prędkość zgodna z przyjętą przednią częścią robota przekracza licznik zmniejszając wartość rejestru przy wartości 0, w związku z czym pod rejestrem, podczas obsługi przerwania, jest zapisana wartość mniejsza od 65000. Dolne ograniczenie zostało dobrane empirycznie. Analogicznie sytuacja wygląda przy przepełnieniu wartości timera odwrotną prędkością. Zmienna pomocnicza jest inkrementowana lub dekrementowana o obliczoną wartość impulsów.
Obliczenie prędkości podczas aktualizacji odbywa się za pomocą funkcji „LeftVelocity" i „RightVelocity". Obliczanie prędkości koła na przykładzie funkcji „LeftVelocity":

```
1 floatLeftVelocity(void)
2 {
3 leftTotalTicks += (ENCODER_INITIAL -htim3.Instance->CNT);
4 htim3.Instance->CNT = ENCODER_INITIAL;
5
6 float vel = 0.0f;
7 vel = calcVelocity(leftTotalTicks);
8 leftTotalTicks = 0;
9 return vel;
10 }
```

Funkcja dodaje do zmiennej „leftTotalTicks" ilość impulsów wygenerowanych przez enkoder od ostatniej aktualizacji prędkości. Następnie zeruje ilość impulsów, oblicza prędkość liniową w cm/s i zwraca ją. Funkcja obliczająca prędkość liniową:

```
1 floatcalcVelocity(int8_t encoderTicks)
2 {
3 float tmp = encoderTicks/TICK_PER_ROUND;
4 tmp = tmp*2.0*SHORT_PI;
5 tmp = tmp /(getTimerTimeout(&htim2)*0.0001);
6 tmp = tmp*RADIUS;
7 return tmp;
8 }
```

W pierwszym kroku obliczana jest ilość obrotów jakie wykonało koło biorąc po uwagę przekładnie silnika i rozdzielczość samego czujnika . Następnym krokiem jest zamiana ilości obrotów na prędkość kątową wyrażoną w radianach na sekundę. Ostatnim etapem

jest uzyskanie prędkości liniowej mnożąc prędkość kątową przez promień koła i zwrócenie wartości.

Przedstawione rozwiązanie jest odporne na duże prędkości, niemniej jednak może się nie sprawdzać w przypadku małych. Korzystając z enkoderów inkrementalnych należy być świadomym niepewności pomiarowych w odczycie ilości impulsów w stałej ramie czasowej, wywołanej brakiem synchronizacji generowanego przez czujniki sygnału i timera wyzwalającego aktualizację. Problem pogłębia fakt, że sama rama czasowa może być ustalona z zewnątrz podczas pracy robota. Bezpieczniejsze byłoby zablokowanie możliwości zmiany ustawień timera, jednakże byłoby to sprzeczne z ideą projektu. Dobrana empirycznie wartość 60 [ms] jest wartością nominalną i zalecaną podczas korzystania z robota, a za zamianę tej wartości przejmuję pełną odpowiedzialność użytkownik.

W stałym interwale czasowym problem pojawia wariancja odczytu impulsów, która rośnie wraz ze zmniejszaniem prędkości. Dla stosunkowo dużych prędkości wariancja odczytanych impulsów jest na tyle niewielka, że można ją pominąć, natomiast przy niewielkich potrafi mocno zaburzyć pomiar. W celu wyznaczenia progu prędkości, dla którego pomiar jest stabilny, przygotowany został eksperyment, którego wynika przedstawiono poniżej. Doświadczenie polegało na 30 krotnym pomiarze prędkości oraz ilości impulsów w ramie czasowej dla ustalonego PWM, który następnie inkrementowany był do nowej wartości. W ten sposób wyeliminowano problem błędów wywołanych samym procesem regulacji.

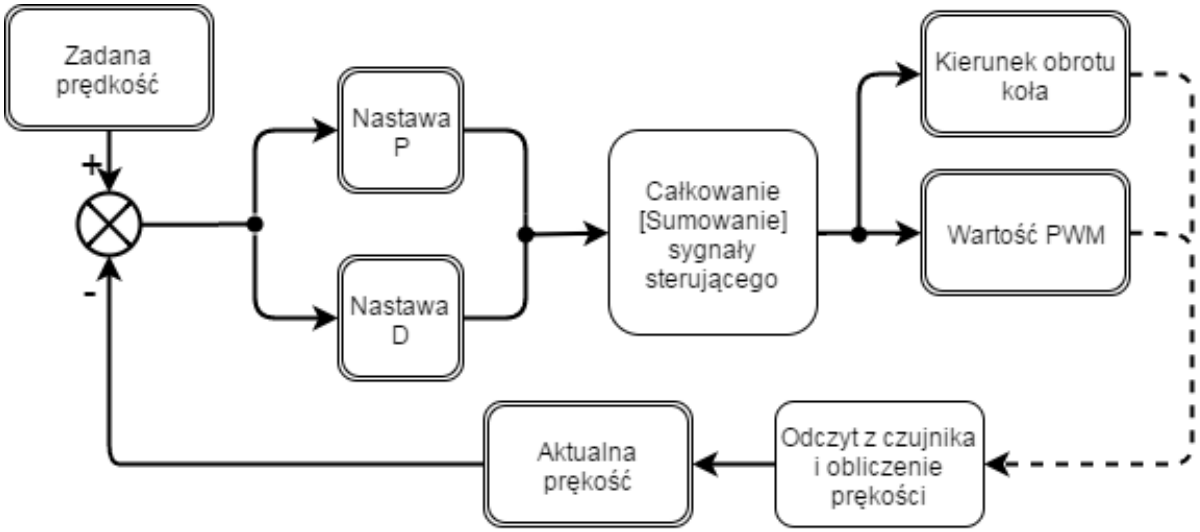
Pomiar odbył się przy pełnym naładowaniu baterii (8.2[V]). Całe doświadczenie było w pełni zautomatyzowane poprzez skrypt JavaScript. W tabeli przedstawione są kolejno: wypełnienie PWM, średnia zliczonych impulsów w ramce 60 ms, ich odchylenie standardowe, procent odcylenia w stosunku do średniej oraz uśredniona prędkość.

PWM	śr. impulsów	Odchyl. standardowe	Stosunek odch do śr [%]	Prędkość [cm/s]
100	6.83	0.9	13.1412	0.7
150	13.57	0.5	3.65	2.17
200	19.67	0.47	2.395	3.21
250	25.3	0.86	3.407	4.26
300	30.87	0.67	2.171	5.24
350	36.5	0.81	2.208	6.44
400	44.33	1.08	2.425	7.82
450	50.4	1.28	2.542	8.8
500	56.33	1.01	1.795	9.94
550	61.93	0.57	0.927	11.07
650	73.6	0.66	0.901	13.39
700	80.17	0.64	0.795	14.66
800	93.6	0.5	0.534	17.15
900	107.23	0.45	0.42	19.63
1000	120.93	0.47	0.389	22.12

Tabela. 3.1: Wyniki doświadczenia

Na podstawie widzimy, że odczyt przy prędkości 0,7 cm/s odczyt jest w bardzo dużym stopniu niestabilny o czym świadczą trzynastoprocentowe wahanie w zliczaniu impulsów. Kolejne odczyty odchylenie od średniej, jednakże błąd na poziome 2.5

3.7 Układ regulacji



Rysunek 3.5: Schemat układu regulacji

Bloki o podwójnym obramowaniu oznaczają stan robota, natomiast o pojedynczym reprezentują operację. Strzałki pokazują przepływ danych podczas regulacji prędkości.Uchyb regulacji jest różnicą pomiędzy prędkością zadaną, a aktualną, uzyskaną w trakcie procedury opisanej w poprzedniej sekcji. Wartość uchybu podlega wzmocnieniu i całkowaniu z ustawionymi nastawami, a następnie podlega sumowani ze stanem poprzednim. Operacja ta jest wymagana, ponieważ sterowniki silników kontrolowane są przez sygnał PWM, nie jest w stanie utrzymać stałej wartości sterującej. To działanie mogłoby być realizowana przez całkujący człon regulatora, jednakże wykorzystane rozwiązanie upraszcza implementacje i dobrze wpasowuje się w dyskretną naturę regulatora. W implementacji członuróżniczkującego na mikrokontrolerze występuje problem przy dyskretyzacji sygnału. Poniższy wzór aproksymuje operacje różniczkowania na dziedzinie dyskretnej:

$$\frac{de(t)}{d} \sim \frac{[e(n) - e(n-1)]}{T} \tag{3.1}$$

Pochodna uchybu w systemie dyskretnym przybliżona jest przez iloraz różnicowy. Dokładność wynika przybliżenia jest tym większa, im mniejszy jest okres T. Projekt zakładał wyzwalanie procedury regulacji prędkości poprzez timer, którego interwał ustalany jest wiadomością kontrolną przez użytkownika. Podobnie jak w przypadku aktualizacji prędkości, kosztem poniesionym za swobodę ustawień jest bezpieczeństwo działania układu regulacji. Wartością nominalną timera regulacji jest 150[ms] co jest jednocześnie okresem dyskretyzacji członu różniczkującego. Pomimo, że aproksymacja w te sposób w znaczny sposób od różniczkowania w dziedzinie ciągłej, wykorzystałem to rozwiązanie ze względu na łatwość implementacji oraz jakość regulacji na oczekiwanym poziomie.

```
1 void regulation(void)
2 {
3         eLeft = setLeftSpeed - leftVelocity;
4         eRight = setRightSpeed - rightVelocity;
5         newPWMLeft = pVal * eLeft + dVal * (eLeft - peLeft);
6         newPWMRight = pVal * eRight + dVal * (eRight - peRight) / (
              getTimerTimeout(&htim10) * 0.0001);
7         peLeft = eLeft;
```

```
 8          peRight = eRight;
 9          addNewPWM(newPWMLeft,newPWMRight);
10 }
```

Algorytm regulacji zaimplementowany została według wyżej wymienionego sposobu. Okres dyskretyzacji członu różniczkującego przeskalowany został do sekund. Aby regulator działął prawidłowo ważne jest spełnienie warunku:

$$T_{regulacji} \geqslant T_{akt.predkosci} \tag{3.2}$$

Aktualizacja prędkości musi być odbywać się przynajmniej tak często jak regulacja. W przeciwnym razie, regulator będzie wyliczał uchyb względem błędnej prędkości, co może całkowicie zaburzy pracę robota.

Rozdział 4

Oprogramowanie aplikacji sterującej

4.1 Struktura oprogramowania

Aplikacja do sterowania zdalnego robotem została napisana w języka Java. Kod źródłowy jest kompilowany do kodu bajtowego (forma pośrednia pomiędzy kodem źródłowym, a instrukcjami maszynowymi) i może być uruchamiany na każdej platformie posiadającej Maszynę Wirtualną Javy. Wybór języka wynika, z wieloplatformowości powstałego programu oraz osobistym doświadczeniu w tworzeniu graficznego interfejsu użytkownika.

Aplikacja składa się z trzech podstawowych komponentów:

1. modułu kontroli komunikacji Bluetooth
2. modułu sterowania tekstowego
3. modułu sterowania graficznego

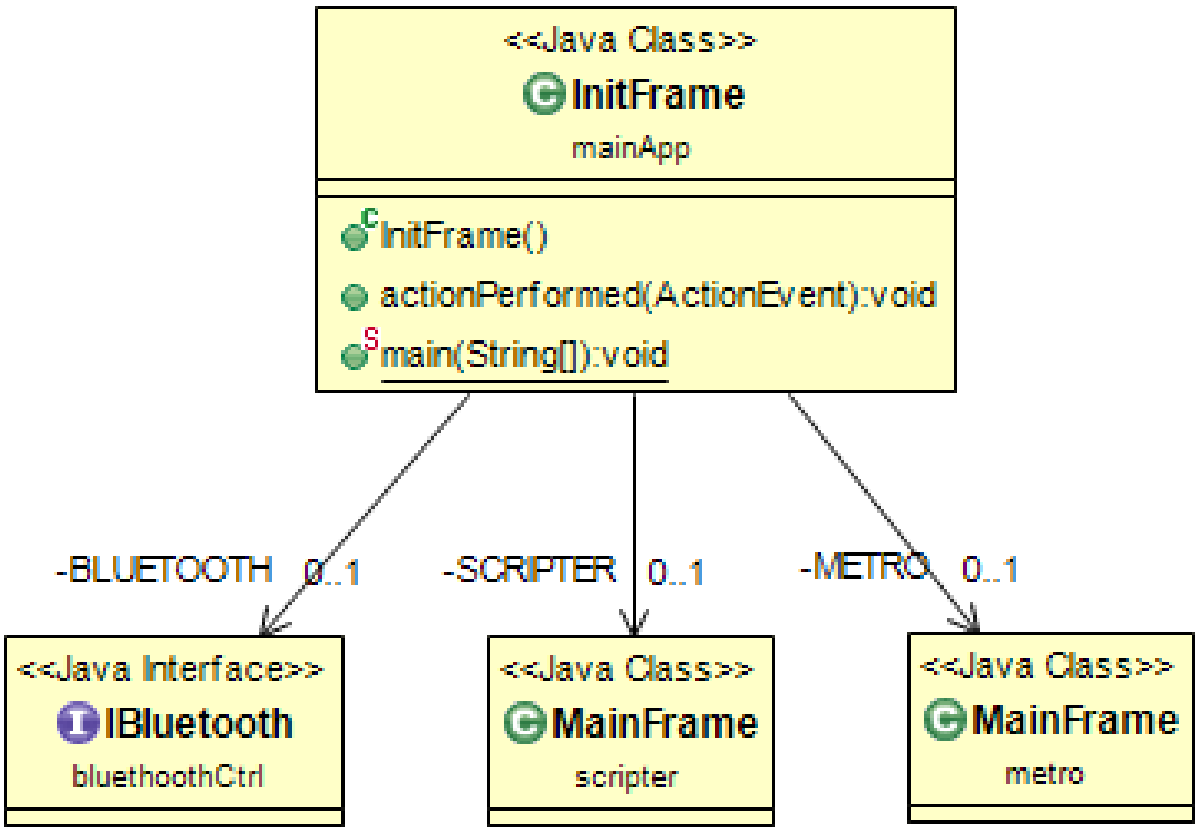
Poza tymi, istnieje jeszcze jeden komponent odpowiedzialny ze utworzenie wyżej wymienionych modułów oraz posiada zbiór funkcji używanych przez wszystkie moduły. Wszystkie komponenty w kodzie źródłowym zorganizowane są w formie pakietów Java. Pakiet Java jest mechanizmem języka, który pozwala grupować powiązane ze sobą klasy, nadając im jednocześnie wspólną przestrzeń nazw.

Program staruje w klasie głównej InitFrame. Obiekt tej klasy zawiera w sobie egzemplarze klas zarządzających oknami modułów sterujących oraz implementacje interfejsu IBluetooth. Wszystkie przedstawione moduły oraz ich struktura opisane są w dalszej części tego rozdziału.

4.2 Moduł kontroli komunikacji Bluetooth

Moduł ten jest komponentem, którego zadaniem jest obsługiwanie transmisji szeregowej Bluetooth z robotem. Moduł definiuje abstrakcyjny interfejs który pozwala na wymianę informacji z robotem, jednocześnie ukrywając techniczne szczegóły realizacji transmisji danych.

Podstawą modułu jest interfejs IBluetooth. Udostępnia szereg funkcji, do wymiany danych z robotem. Argumenty funkcji posiadają typ danych, który w naturalny sposób odzwierciedla wartość jaką wyraża. Przykładem może być, zadana prędkość, która posiada typ double, czy czas ekspiracji timera, wyrażony w liczbach całkowitych. Odpowiedzialność za przygotowanie odpowiedniej wiadomości, zgodnej z zaprojektowanym protokołem



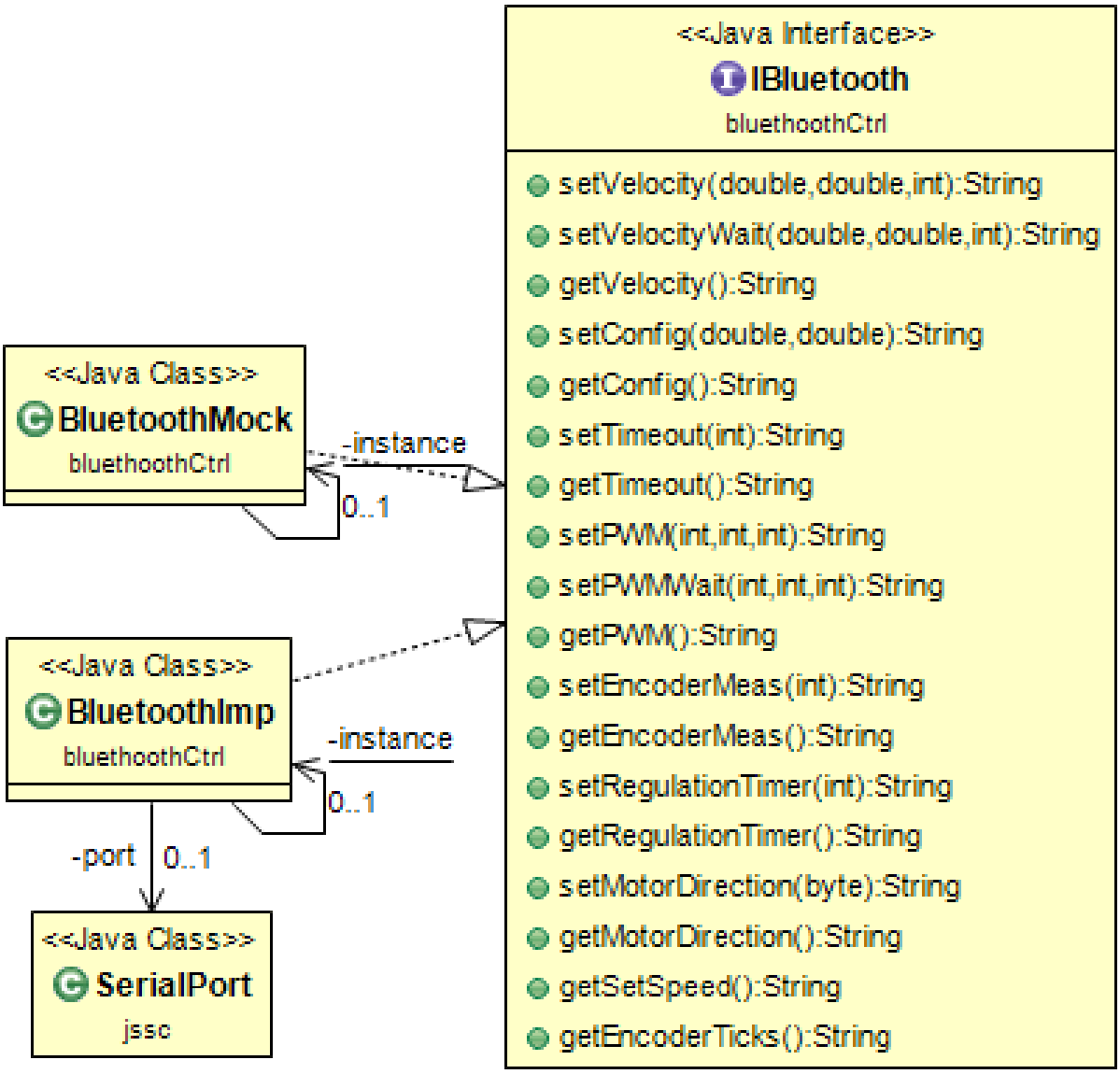
Rysunek 4.1: Diagram klas głównego modułu

przeniesiona jest na ten komponent. Warto również zauważyć, że funkcja zadająca prędkość (setVelocity) posługuje się argumentami : Translacja, Rotacja. Z poziomu aplikacji użytkownik korzysta z zadawania prędkości translacji (ruchu postępowego robota w dwuwymiarowym układzie współrzędnych) i rotacji (ruchu obrotowego) , a nie prędkościami poszczególnych kół. Z punktu widzenia użytkownika jest to bardziej intuicyjne rozwiązanie. Pobieranie wartości również jest przekształcane z prędkości lewego i prawego koła na prędkość translacji i rotacji. Transformacje za pomocą funkcji:

```
1  public static double[] TranRot2LeftRight(double translation, double rotation)
2  {
3      double[] result = new double[2] ;
4      result[0] = translation + rotation;
5      result[1] = translation - rotation;
6      return result;
7  }
8  public static double[] LertRight2TranRot(double left,double right)
9  {
10     double[] result =new double[2] ;
11     result[0] = (left + right)/2;
12     result[1] = (left - right)/2;
13     return result;
14 }
```

Każda funkcja zwraca łańcuch znaków zawierająca sformatowaną odpowiedź robota. Wybór tego typu jest omówiony w podrozdziale 4.3. Interfejs posiada dwie implementacje:

- klasa, która symuluje działanie robota, używana we wstępnej fazie implementacji oraz do testowania działania aplikacji bez połączenia z robotem



Rysunek 4.2: Diagram klas modułu Bluetooth

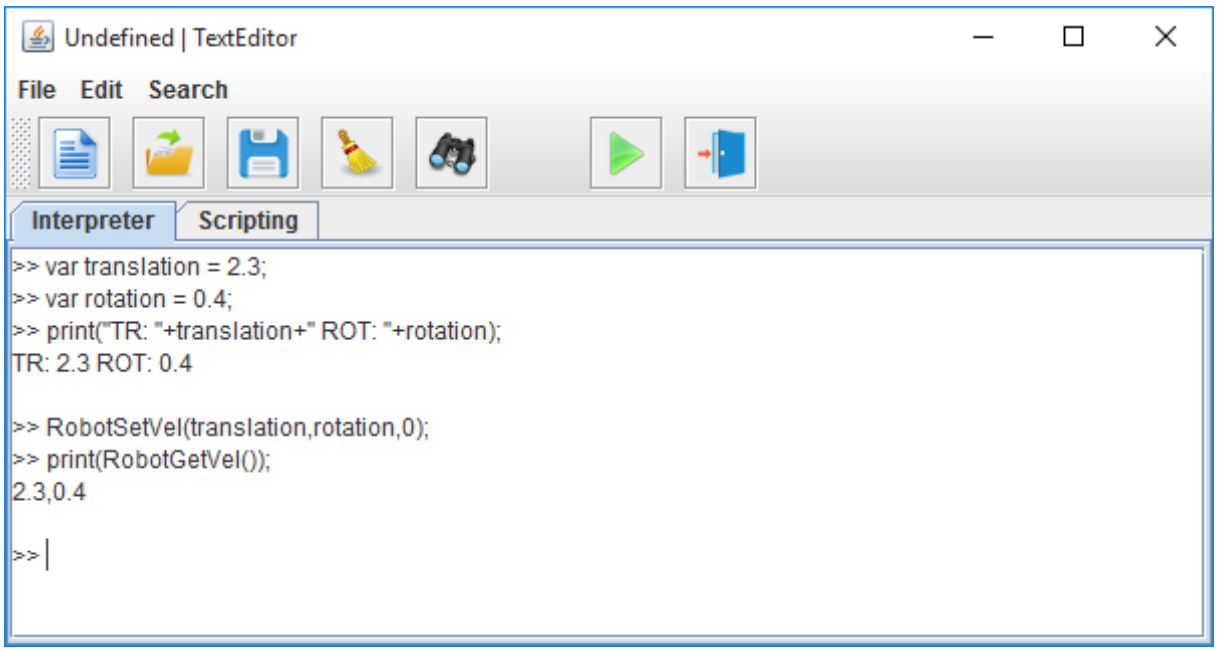
- klasa, która obsługuje połączenie szeregowe poprzez wirtualny port COM z modułem Bluetooth na robocie oraz tworzy właściwe wiadomości gotowe do transmisji

Istotną koncepcją przy tworzeniu było zapewnianie, aby tylko jedna instancja obiektu była tworzona podczas pracy programu. Jak w przypadku klasy symulującej nie ma to większego znaczenia, jest niezbędne we właściwej implementacji. Wynika to z faktu, że nawiązywana jest transmisja szeregowa, która blokuje dany port COM. Próba utworzenia kolejnej instancji zakończy się błędem. Rozwiązane to jest poprzez zastosowanie wzorca projektowego singleton.

Sama obsługa portu szeregowego wykorzystuje bibliotekę jSSC udostępnioną na otwartej licencji. Obsługa portów szeregowych jest ściśle związana z systemem operacyjnym i wymagana jest ich natywna implementacja, specyficzna dla danego systemu. Zastosowana biblioteka posiada zaimplementowaną natywną obsługę transmisji szeregowej dla większości znanych platform oraz udostępnia abstrakcyjny interfejs w języku Java, który pozwala wykorzystanie go w programie.

4.3 Moduł sterowania tekstowego

Głównym założeniem przy projektowaniu aplikacji sterującej było umożliwienie użytkownikowi tworzenia skryptów pozwalający implementować dowolny algorytm oraz interakcję z robotem za pomocą komend tekstowych. Wymagało to stworzenie modułu o podstawowej funkcjonalności edytora testowego. Założeniem była również interakcja z poziomu konsoli interpretera, który swoją funkcjonalnością będzie przypominał inne znane interpretery języków skryptowych.



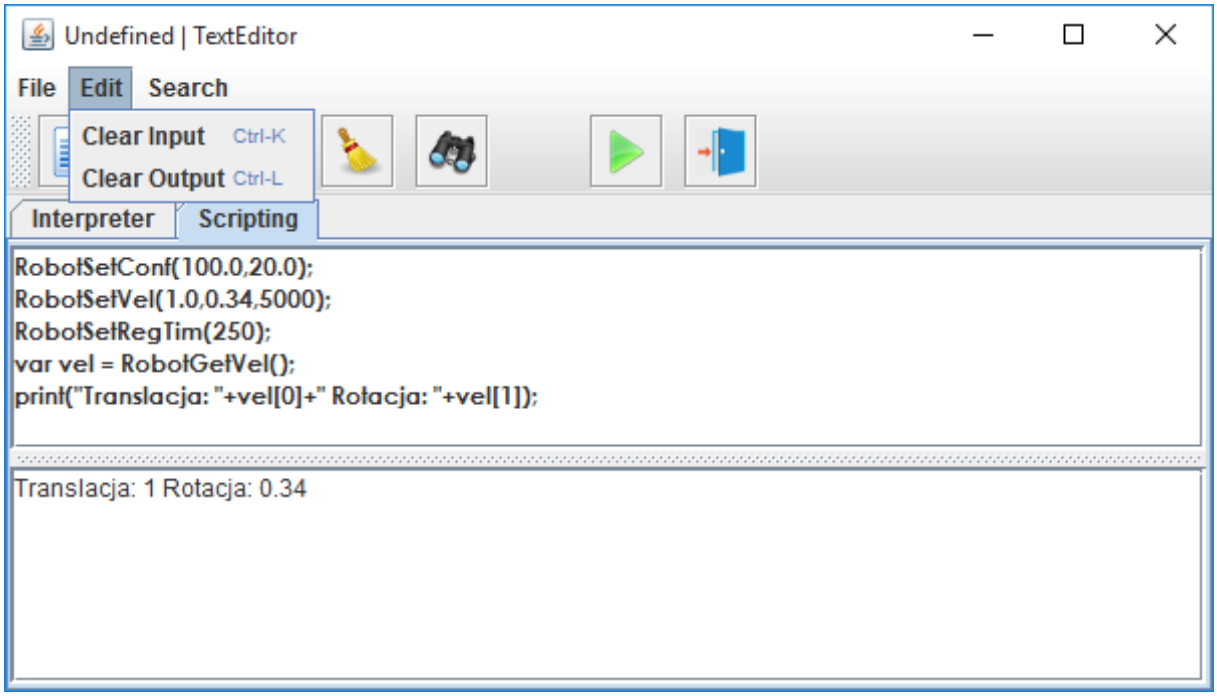
Rysunek 4.3: Okno panelem interpretera z interpreterem

Interpreter wykonuje pojedynczą instrukcję, zakończoną średnikiem. Klawisz ENTER rozpoczyna wykonywanie komendy, natomiast klawiszami strzałek możemy przeglądać historie wykonanych instrukcji. Interpreter nie jest związany z panelem sterowania umieszczonym powyżej, interakcja odbywa się tylko za pomocą klawiszy.

Panel edycji skryptów posiada funkcjonalność podstawowego edytora testu. Górne pole tekstowe służy do tworzenia kodu, natomiast dolne jest przekierowanym standardowym wyjściem silnika JavaScript. Powyżej pól tekstowych znajduję się pasek narzędzi oraz menu. Wszystkie funkcje wspomagania edycji dostępne są z poziomu menu, paska narzędzi oraz skrótów klawiszowych. Edytor umożliwia:

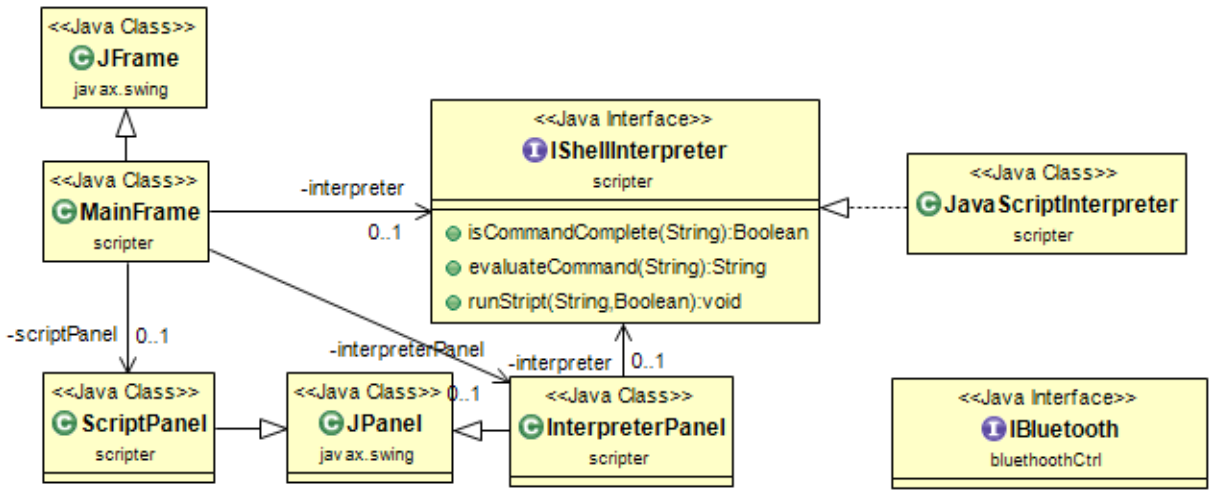
- uruchomianie skryptu
- zapis i odczyt plików z dysku
- czyszczenie pola wprowadzania testu i pola wyjściowego
- prosty mechanizm przeszukiwania tekstu
- zakończenie pracy edytora

Ważnym aspektem modułu jest fakt, że zarówno interpreter, jak i edytor korzystają z tej samej instancji silnika JavaScript. Oznacza to, że działania wykonane na jednym panelu widoczne są w drugim. Nie stoi nic na przeszkodzi, aby konfiguracje robota przeprowadzić



Rysunek 4.4: Okno z panelem skryptu

za pomocą edytora skryptów, natomiast same sterowanie odbyło się za pomocą panelu interpretera.



Rysunek 4.5: Diagram klas modułu sterowania tekstowego

Wykorzystanie silnika języka interpretowanego umożliwię interfejs IShellInterpreter. Zastosowanie interfejsu nie wiąże programu z konkretnym silnikiem, umożliwia to w podłączenie dowolnego interpretera bez wprowadzania zmian w programie. Interfejs IBluetooth wywoływany jest poziomu samego JavaScriptu, w związku z czym nie jest powiązany z żadnym z obiektów w module.

4.3.1 Użycie silnika JavaScript w aplikacji

Możliwość pisania skryptów wymagała użycia silnika jednego z języków skryptowych. Wybór padł na JavaScript, który główne zastosowanie ma w stronach internetowych. Ję-

zyk ten pozwala na pisanie pełnoprawnych aplikacji oraz realizacji dowolnego algorytmu. Głównym powodem była obecność silnika Nashorn w Java 8 . Jest on wydajną implementacją JavaScript pozwalającą łączyć funkcjonalność obu języków.

```
public class JavaScriptInterpreter implements IShellInterpreter {
private final String InitialScript="src\\scripter\\cmd.js";
// (...)
private ScriptEngine engine;
// (...)
public JavaScriptInterpreter()
 {
    engine =new ScriptEngineManager().getEngineByName("nashorn");
    stringWriterInit();
    engine.getContext().setWriter(stringWriter);

    loadInitScript();
}
//(...)
```

Java 8 udostępnia możliwość stworzenia obiektu klasy ScriptEngine, który zarządza silnikiem JavaScript. Konstruktor obiektu ustawia jako strumień wyjściowy obiekt tekstowy stringWriter powiązany z oknem aplikacji oraz uruchamia skrypt początkowy, umieszczając tym samym w przestrzeni JavaScript definicję funkcji używane przy komunikacji z robotem. Uruchamianie kodu JavaScript

```
@Override
public Stringe valuateCommand(Stringcommand) throws ScriptException {
    engine.eval(command);
    String res = stringWriter.toString();
    stringWriter.getBuffer().setLength(0);
    return res;
}
(...)
@Override
public void runStript(Stringtext, Boolean isFromFile) throws Exception{
    stringWriter.setIsEvent(true);
    if(isFromFile)
        engine.eval(newFileReader(text));
    else
        engine.eval(text);
    stringWriter.setIsEvent(false);

}
```

Przedstawione metody używane są przy wywoływaniu skryptów. Pierwsza funkcja używana jest przez panel interpretera. Po wykonaniu fragmentu kodu („engine.eval(command);”)czyści bufor a następnie umieszcza w nim dane odebrane z standardowego wyjścia silnika. Druga funkcja używana jest w edytorze skryptów. Pozwala na uruchamianie zarówno plików tekstowych jak i ciągów znaków. W tej metodzie przekierowanie strumienia wyjściowego opiera się na obsłudze generowanych prze obiekt stringWriter zdarzeń.

Fragment inicjalnego skryptu umożliwiającego kontakt z robotem:

```
var BluethoothClass = Java.type('bluethoothCtrl.BluetoothImp');
var btObject = BluethoothClass.getInstance();
//(...)
function RobotSetVel(translate,rotate,time)
{
    var result =btObject.setVelocity(translate,rotate,time);
    returndecode(result);
}
function RobotGetVel()
{
    var result = btObject.getVelocity();
```

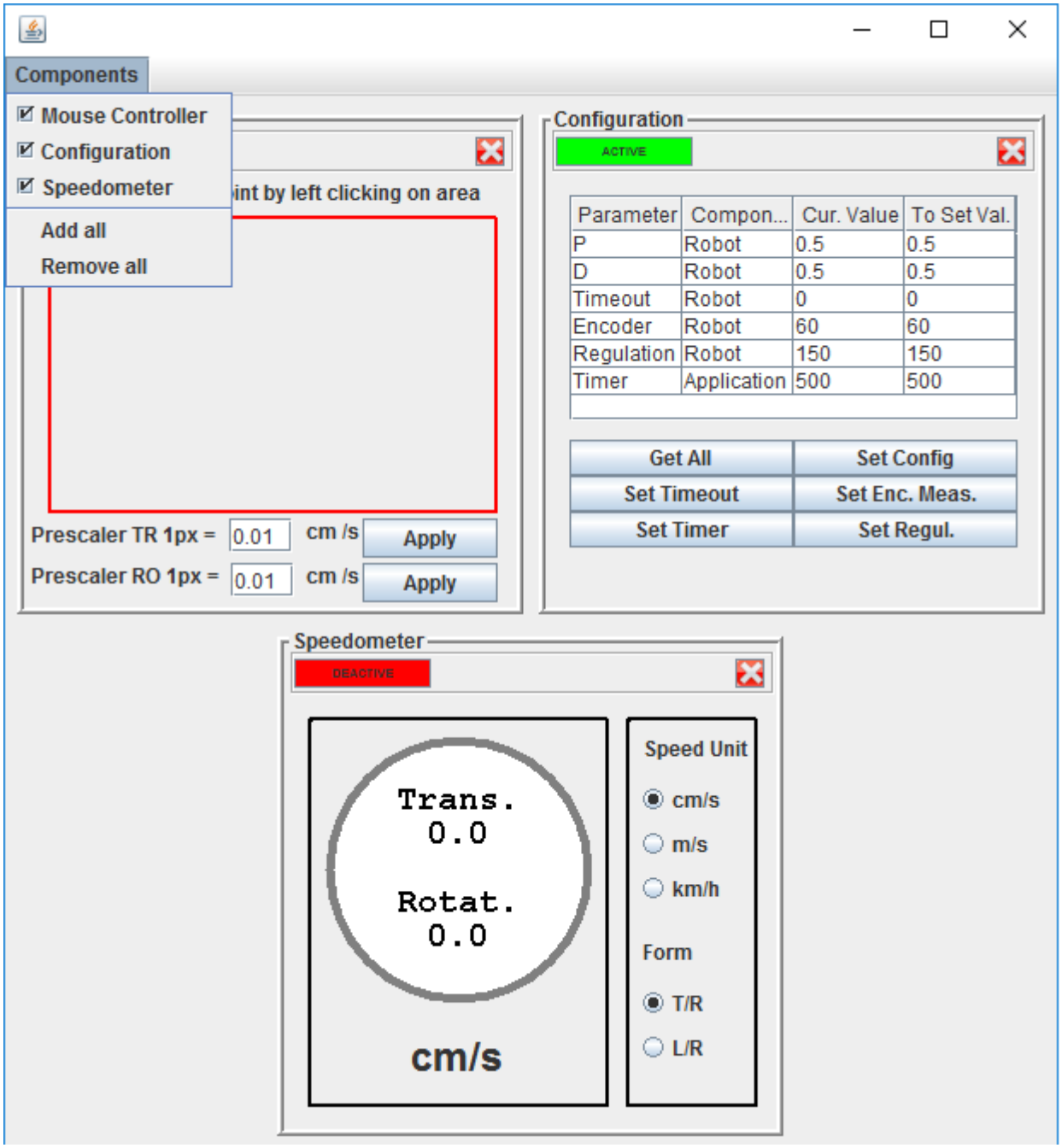
```
12      returndecode(result);
13  }
14  //(...)
```

Sam inicjalny skrypt ładowany podczas konstruktora klasy opakowuje wszystkie metody interfejsu IBluethooth. Skrypt uzyskuję dostęp do klasy Java (jedna z wspomnianych wcześniej możliwości silnika Nashorn) oraz pobiera statyczny obiekt tej klasy. Każda funkcja wywołuje metodę z interfejsu IBluethoot, a następnie zwraca zdekodowany do tablicy łańcuch znaków. Powodem, dla którego w interfejsie IBluethooth każda metoda zwraca ten typ jest fakt, że z poziomu JavaScript pobieranie zmiennej tekstowej działa zawsze zgodnie z oczekiwaniami, czego nie można powiedzieć o innych typach danych.

4.4 Moduł sterowania graficznego

Wykorzystanie potencjału sterowanie tekstowego wymaga podstawowej wiedzy programistycznej i poświęcenia czasu aby w pełni poprawnie przygotować skrypt. Poza tym moduł ten nadaje się do implementacji algorytmu poruszania się robota, niekoniecznie natomiast sprawdza się w sytuacji gdy chcemy , aby robot reagował na nasze działania w czasie rzeczywistym. Odpowiedzią na ten problem jest moduł sterowania graficznego. Idea tego komponentu polega na wykorzystaniu bloków sterujących, które pozwalająca komunikację z robotem za pomocą interfejsu graficznego. Bloki sterujące pozwalają na zmianę konfiguracji robota, kontrolę i wizualizację prędkości bez konieczności używanie komend tekstowych.

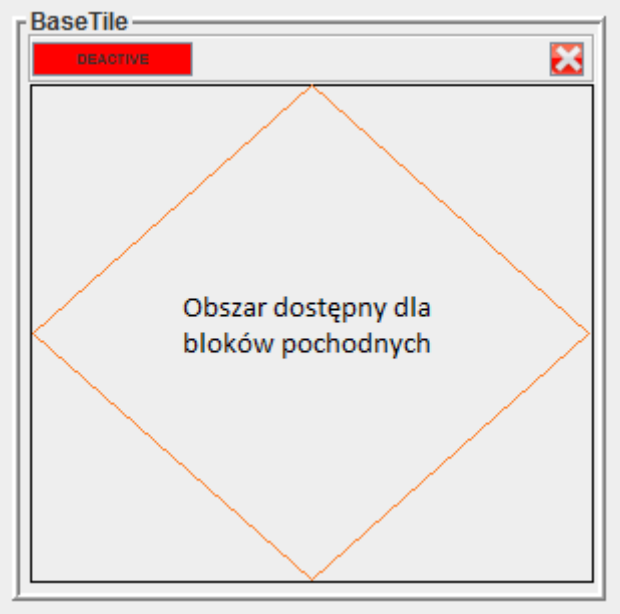
Główne okno modułu graficznego pozwala wybierać bloki, które chcemy aktualnie użyć przy sterowaniu robotem, spośród wszystkich dostępnych. Dodatkowo w menu umieszczone są dwie pomocnicze opcje pozwalające na dodawanie bądź usunięcie wszystkich bloków. Bloki sterujące są szczegółowo opisane w następnych podrozdziałach.



Rysunek 4.6: Okno modułu graficznego

Samo okno aplikacji pozwala dynamicznie zarządzać blokami na ekranie. W zależności od rozmiaru okna bloki układają się w sposób, aby jak najefektywniej wykorzystać dostępną przestrzeń. Rozkład elementów można kontrolować poprzez kolejność dodawania bloków do głównego okna. Minimalizacja i zamknięcia okna wysyłają komendę stop do robota.

4.4.1 Bazowy blok sterujący

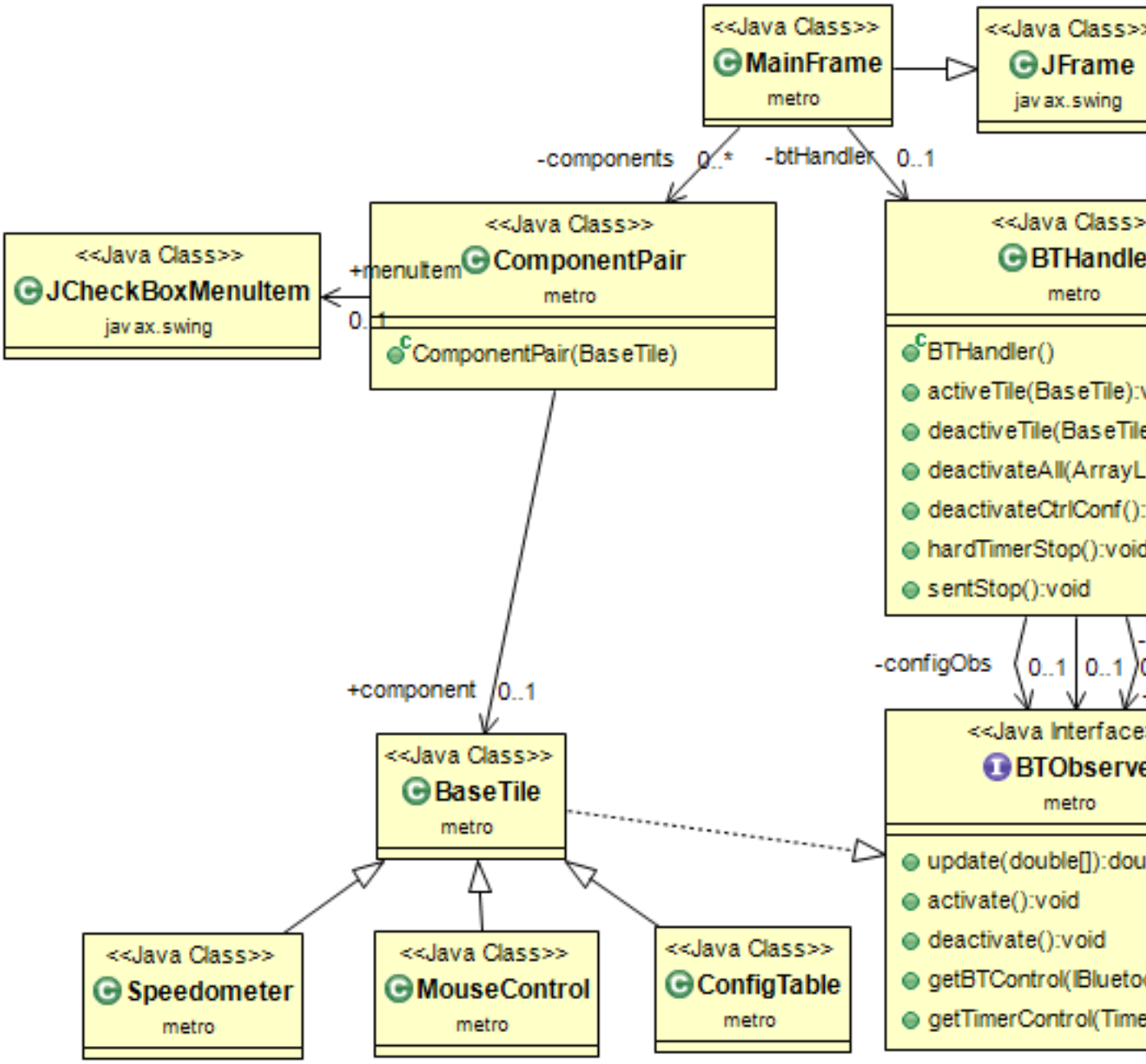


Rysunek 4.7: Bazowy blok sterujący

Przycisk z krzyżykiem odpowiada za wyłączenie elementu i jest jednoznaczny z odznaczeniem elementu w głównym oknie modułowym. Przycisk aktywacji określa włączenie działania bloku i jest podstawowym mechanizmem w synchronizacji pracy. Współpraca bloków opisana jest w podrozdziale 4.4.3. Ich wymiary są ustalone i nie podlegają modyfikacji. Blok może modyfikować udostępnione przez klasę BaseTile ciało elementu.

Właściwości wspólne dla każdego elementu :

- nazwa – każdy blok musi wprowadzić swoją unikalną nazwę
- typ - blok może by jednego z trzech typów
  - STEROWANIE – element kontrolujący (zadający) prędkość robota
  - WIZUALIZACJA – element wizualizujący faktyczną prędkość robota
  - KONFIGURACJA – element, który nie steruje prędkością, może natomiast zmieniać konfiguracje robota np. nastawy regulatora
- definiuje podstawową funkcję aktywacji i dezaktywacji elementu. Element może rozszerzyć tą funkcjonalność, niemniej jednak musi wtedy wywołać funkcje klasy bazowej
- definiuje funkcje wyłączenia elementu



Rysunek 4.8: Diagram klas modułu sterowania graficznego

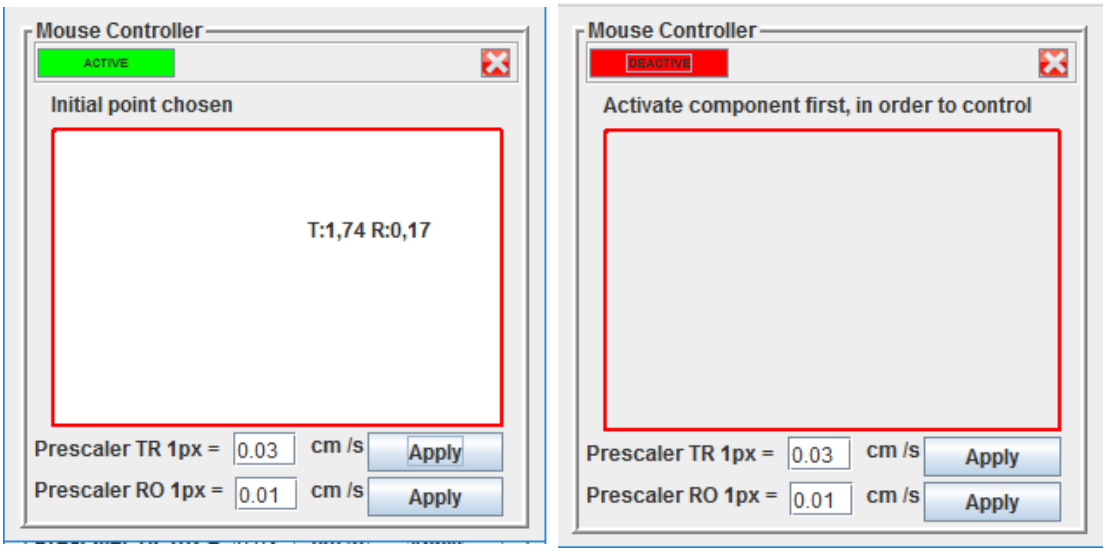
Obiekt skojarzony z głównym oknem modułu zawiera listę obiektów ComponentPair łączących bloki sterujące oraz pola wyboru. Życiem obiektów implementujących zarządza obiekt głównej klasy. Kolejnym komponentem jest obiekt klasy BTHandler kieruje komunikacją pomiędzy modułem Bluetooth a blokami sterującymi oraz jest odpowiedzialny za synchronizację pracy bloków sterujących. Klasa ta uzyskuje referencję do obiektów sterujących i komunikuje się z nimi przez interfejs BTObserver. Działanie poszczególnych bloków oraz synchronizacja ich pracy opisana jest w dalszej części rozdziału.

4.4.2 Bloki sterujące dostępne w aplikacji

W aplikacji stworzone zostały 3 bloki sterujące, po jednym w każdym typie.

Nazwa Sterowanie Myszką

Typ STEROWANIE

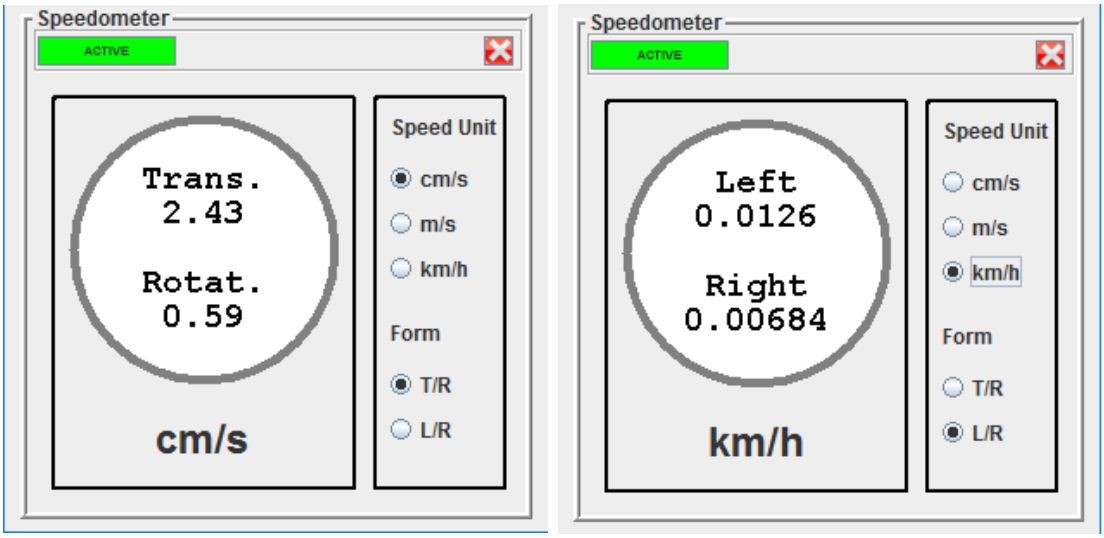


Rysunek 4.9: Sterowanie myszką

Blok umożliwia zadawanie prędkości za pomocą myszki. W centralnej części elementu znajduje się obszar obramowany czarowną linią i główny panel sterowania. Naprowadzenie kursora spowoduje zmianę koloru z szarego na biały, natomiast kliknięcie myszką rozpocznie sterowanie. Po rozpoczęciu sterowania nad kursorem pojawi się napis określający zadaną prędkość translacji i rotacji robota wyrażony w cm/s. Punkt zerowy jest w miejscu kliknięcia, które rozpoczyna sterowanie robotem. Na głównym panelem znajduje się informator aktualnego stanu bloku. Pod głównym panelem jest możliwość ustawienia przeskalowania ruchu myszki (zmiana o ilości pikseli) na zadaną prędkość. Praktyka pokazuje, że lepiej steruje się robotem, gdy przeskalowanie translacji jest większe niż rotacji. Przesunięcie myszki w pionie zmienia rotację, natomiast przesunięcie w poziomie zmienia zadaną rotację. Można zadawać prędkości zarówno dodatnie jak i ujemne. Ponowne kliknięcie, bądź wyjście kursora poza panel główny powoduje zakończenie sterowania.

Nazwa Prękościomierz

Typ WIZUALIZACJA



Rysunek 4.10: Blok prędkościomierza

Prędkościomierz pokazuje aktualną prędkość robota. Lewy panel odpowiada za wyświetlenie prędkości natomiast prawy za formę w jakiej prędkość ma być wyświetlana. Za pomocą prędkościomierza może ustawić jednostkę w jakiej chcemy prezentować prędkość:

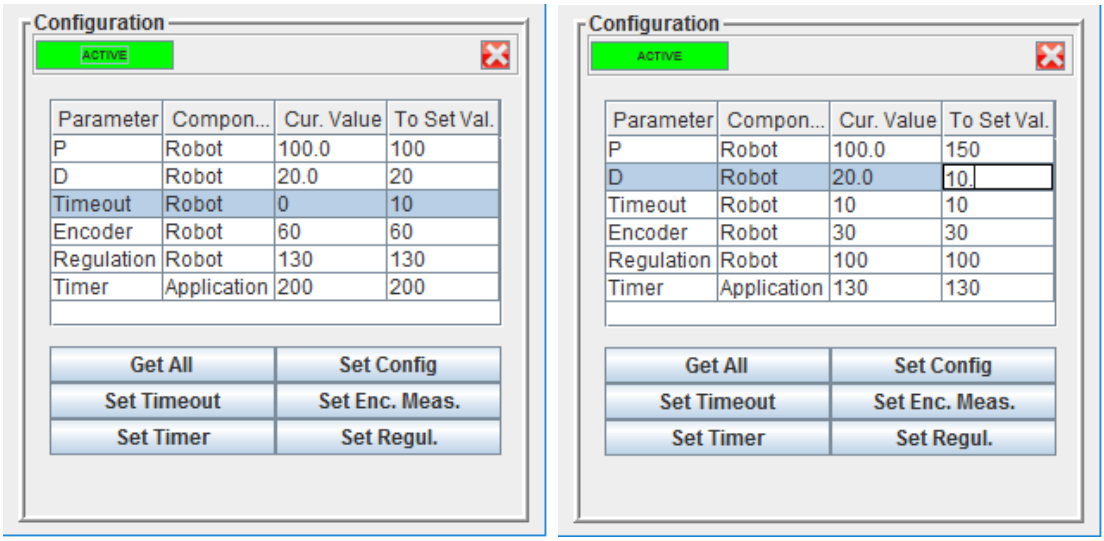
- centymetry na sekundę(domyślnie)
- metry na sekundę
- kilometry na godzinę

Kolejną opcją jest ustawienie formy wyświetlenia prędkości:

- prędkość translacji i rotacji robota (domyślnie)
- prędkość lewego i prawego koła robota

Nazwa Konfiguracja

Typ KONFIGURACJA



Rysunek 4.11: Konfiguracja robota

Prędkościomierz pokazuje aktualną prędkość robota. Lewy panel odpowiada za wyświetlenie prędkości natomiast prawy za formę w jakiej prędkość ma być wyświetlana. Za pomocą prędkościomierza może ustawić jednostkę w jakiej chcemy prezentować prędkość:

- parametr (pierwsza kolumna) : określa ustawionyparametr
- komponent (druga kolumna) : określa czy dany parametr dotyczy konfiguracji robota czy aplikacji
- aktualna wartość (trzecia kolumna):pokazuje aktualnie ustawiony parametr
- wartość do ustawienie(czwarta kolumna):jedyna gdzie pola można modyfikować. Pozwala na wpisanie wartości, którą będziemy mieli możliwość ustawić

Kolejną opcją jest ustawienie formy wyświetlenia prędkości:

- ustawianie timera zabezpieczającego przed utratą łączności Bluetooth (Timeout)
- ustawianie timera aktualizacji prędkości(Encoder) oraz interwału regulatora(Regulation)
- ustawienie timera modułu graficznego (Timer) , omówiony szczegółowo w podrozdziale 4.4.3

4.4.3 Synchronizacja aktywności bloków

Architektura blokowa modułu graficznego pozwala na tworzenie nieograniczonej ilości bloków sterujących dowolnie wybranego typu. Kontrolowanie prędkości robota może być, oprócz sterowania myszką realizowane przez klawiaturę, joystick czy nawet komendy głosowe. Konfigurowanie robota również może przyjmować wiele różnych form. Z tego powodu wymagane było wprowadzenie mechanizmu synchronizacji pracy bloków. Klasa, która odpowiada za zarządzanie blokami sterującymi jest wspomniany już BTHandler. Oprócz synchronizacji klasa ta bierze odpowiedzialność za kontrolowanie dostępu do komponentu Bluetooth dla bloków funkcyjnych.

Podstawowym elementem, służącym synchronizacji jest przytoczony wcześniej mechanizm aktywacji. Aby działanie komponentu było możliwe, musi być on aktywowany. Uruchomienie elementu następuje w dwóch etapach. Pierwszy jest w obszarze obiektu klasy BTHandler. Obiekt posiada informacje o wszystkich blokach w aplikacji i aktywowanie komponentów podlega niżej wymienionym zasadom:

- aktywny może być tylko jeden komponent typu STEROWANIE lub KONFIGURACJA. Uruchomienie więcej niż jednego komponentu tego typu powoduje dezaktywacje bloku aktualnie pracującego
- aktywnych może być dowolna ilość bloków typu WIZUALIZACJA
- timer modułu graficznego rozpoczyna odliczanie gdy przynajmniej jeden komponent typu STEROWANIE lub WIZUALIZACJA jest aktywny, a wyłącza się gdy nie jest aktywny żaden z tych elementów
- wyłączenie elementu powoduje jego automatyczną dezaktywację
- minimalizacja lub wyłączenie okna powoduje dezaktywację wszystkich elementów

Drugi etap aktywacji realizowany jest w przestrzeni danego bloku i przygotowuje go do pracy. Wymiana danych bloków typu STEROWANIE i WIZUALIZACJE jest cyklicznie uruchamiana przez, wymieniony już wcześniej, timer aplikacyjny. Licznik ten co podany okres uruchamia swoją funkcję obsługi:

```
1  Private class Task implements ActionListener
2  {
3  private int currentVal =0;
4  private const int typeNumber = 1;
5  double[] velocity;
6
7  private void iterate CurrentVal()
8  {
9      if(currentVal != typeNumber)
10         currentVal++;
11     else
12         currentVal =0;
13 }
14
15 @Override
16 public void actionPerformed(ActionEvent e) {
17 try{
18     if(setSpeedObs !=null && currentVal ==0)
19     {
20         velocity = setSpeedObs.update(null);
21         bluethooth.setVelocity(velocity[0], velocity[1], 0);
22         System.out.println("SET Tr: "+velocity[0]+" Rot: "+ velocity[1]);
23     }
24     if(!getSpeedObs.isEmpty() && currentVal ==1)
```

```
25      {
26          String tmp = bluethooth.getVelocity();
27          velocity =GeneralConverter.deserializeStr2Dbl(tmp);
28          System.out.println("GET Tr: "+velocity[0]+" Rot: "+ velocity[1]);
29          for(BTObserver obs : getSpeedObs)
30              obs.update(velocity);
31      }
32
33      }catch(SerialPortException serial E) {}
34          System.err.println("Bluetooth transmission error!!!");
35      }
36  }
```

Obsługa przepełnienia timera naprzemienne aktualizuje włączony blok sterujący i wszystkie bloki pobierające aktualną prędkość.

Rozdział 5

Protokół komunikacyjny

Protokół komunikacyjny używany do wymiany danych pomiędzy robotem, a aplikacją składa się z 15 wiadomości kontrolnych. Wiadomości pozwalają pobierać i ustawiać dane na robocie. W większości występują w parach ustaw/pobierz. Protokół nie zakłada istnienia wiadomości jednocześnie zmieniających stan robota i pobierających aktualny stan. Prawidłowa odpowiedź na komendę typu ustaw jest identyczna do odebranej komendy, natomiast obsługa wiadomości typu pobierz nie ujmuje pod uwagę pól danych odebranej komendy.

Wiadomość składa się z 10 bajtów:

Nr bajtu	0	1	2	3	4	5	6	7	8	9
Rodzaj wiadomości	"B"	ID	Dane	Dane	"M"	Dane	Dane	Dane	Dane	"E"

Bajty 0, 4 i 9 w każdej wiadomości są stałe i są równe wartości danej litery w kodzie ASCII. Służą one do kontroli poprawności przesłanej wiadomości.

Bajt 1 określa identyfikator danej wiadomości i na jego podstawie oprogramowanie robota jest w stanie uruchomić odpowiednią funkcję obsługi przesłanej wiadomości. Zostało to zaprezentowane w rozdziale 3.4.

Reszta bajtów przenosi dane konkretnej wiadomości. Wiadomości pozwalają na kodowanie liczby zmiennoprzecinkowej, jak również większych niż 1 bajt liczb całkowitych.

5.1 Spis wiadomości kontrolnych

Każda wiadomość kontrolna przedstawiona jest według poniższego schematu:

Nazwa Wiadomości				Komenda Javascript				Typ	
BAJT 0	BAJT 1	BAJT 2	BAJT 3	BAJT 4	BAJT 5	BAJT 6	BAJT 7	BAJT 8	BAJT 9
CHARAKTERYSTKA WIADMOSCI									

Dla wielkości USTAW w drugim wierszu zaprezentowana jest komenda którą odbiera robot, dla wiadomości typu POBIERZ jest to odpowiedź otrzymywana przez aplikację sterującą.

Ustawienie prędkości zadaniej				RobotSetVel(float TR,float ROT,float time)				USTAW	
"B"	1	L1	L2	"M"	R1	R2	T1	T2	"E"

Wiadomość ustawia prędkość zadaną dla regulatora. Wartości L1 i L2 są zserializowaną zmienną zmiennoprzecinkową oznaczającą zadaną prędkość dla lewego koła Wartości R1

i R2 są zserializowaną zmienną zmiennoprzecinkową oznaczającą zadaną prędkość dla prawego koła Wartości T1 i T2 są rozłożoną 16-bitową zmienną całkowitą oznaczającą ilość milisekund po jakim zadana prędkość" będzie wyzerowana. Jeżeli wartość T1 i T2 wynoszą zero timer odpowiedzialny za zatrzymanie silników po ustalonej wartości nie będzie uruchomiony a zadana prędkość nie będzie zmieniana.

Dane zawierające prędkości są deserializowane i zapisywane do odpowiadających im globalnych zmiennych robota. Jeżeli te dane zawierają wszystkie wartości równe zero silniki są zatrzymywane Wywołując komendę z poziomu JavaScript prędkoć translacji i rotacji zamieniana jest na lewego i prawego koła przed wysłaniem. Komenda zwraca sterowanie od razu po wywołaniu

Ustawienie prędkości zadaniej z blokadą				RobotSetVelWait(float TR,float ROT,float time)				USTAW	
"B"	17	L1	L2	"M"	R1	R2	T1	T2	"E"

Komenda różni się od poprzedniej tym, że zwraca sterowanie po ekspiracji timera

Pobranie faktycznej prędkości				RobotGetVel()				POBIERZ	
"B"	2	L1	L2	"M"	R1	R2	0	0	"E"

Wiadomość zwraca faktyczną prędkość, jaką posiada robot. Wartości L1 i L2 są zserializowaną zmienną zmiennoprzecinkową oznaczającą faktyczną prędkość dla lewego koła Wartości R1 i R2 są zserializowaną zmienną zmiennoprzecinkową oznaczającą faktyczną prędkość dla lewego koła

Odsyłane prędkości są zserializowanymi zmiennymi globalnymi, których wartość obliczana jest w cyklu aktualizacji prędkości. Opis procedury jest w rozdziale 3.5

Ustawienie nastaw regulatora				setConfig(double P, double D)				USTAW	
"B"	3	P1	P2	"M"	D1	D2	0	0	"E"

Wiadomość ustawia nastawy regulatora. Wartości P1 i P2 są zserializowaną zmienną zmiennoprzecinkową oznaczającą wartość wzmocnienia członu proporcjonalnego regulatora PD Wartości P1 i P2 są zserializowaną zmienną zmiennoprzecinkową oznaczającą wartość wzmocnienia członu różniczkującego regulatora PD

Dane są deserializowane i zapisywane do odpowiadających im globalnych zmiennych robota.

Pobranie nastaw regulatora				getConfig()				POBIERZ	
"B"	4	P1	P2	"M"	D1	D2	0	0	"E"

Wiadomość zwraca aktualne nastawy regulatora. Wartości P1 i P2 są zserializowaną zmienną zmiennoprzecinkową oznaczającą wartość wzmocnienia członu proporcjonalnego regulatora PD Wartości P1 i P2 są zserializowaną zmienną zmiennoprzecinkową oznaczającą wartość wzmocnienia członu różniczkującego regulatora PD

Odsyłane nastawy są zserializowanymi zmiennymi globalnymi.

Ustawienie timera bezpieczeństwa				setTimeout(int timeout)				USTAW	
"B"	5	T1	T2	"M"	0	0	0	0	"E"

Wiadomość ustawia timer bezpieczeństwa. Timer jest startuje wraz z odebraniem wiadomości przez robota. Jeżeli nie otrzyma następnej wiadomości przed końcem odliczania

Pobranie interwału timera bezpieczeństwa				getTimeout()				POBIERZ	
"B"	6	T1	T2	"M"	0	0	0	0	"E"

robot zatrzyma silniki Wartości T1 i T2 są rozłożoną 16-bitową zmienną całkowitą oznaczającą ilość milisekund co ile generowane jest przerwanie.

Wiadomość zwraca ustawioną wartość milisekund po którym uruchamiamy jest timer bezpieczeństwa. Wartości T1 i T2 są rozłożoną 16-bitową zmienną całkowitą oznaczającą ilość milisekund co ile generowane jest przerwanie.

Dane są składane w 16-bitową wartość całkowitą, która jest wykorzystywana jest w ustawieniach timera bezpieczeństwa. Timer został opisany w rozdziale 3.5.

Ustawienie wartosci wypelnienia PWM				setPWM(int left, int right, int time)				USTAW	
"B"	7	L1	L2	"M"	R1	R2	T1	T2	"E"

Wiadomość ustawia wartość sygnału PWM dla silników Wartości L1 i L2 są rozłożoną 16-bitową zmienną całkowitą oznaczającą wartość sygnału PWM, który steruje lewym silnikiem. Wartości R1 i R2 są rozłożoną 16-bitową zmienną całkowitą oznaczającą PWM który, steruje prawym silnikiem. Wartości T1 i T2 są rozłożoną 16-bitową zmienną całkowitą oznaczającą ilość milisekund po jakim zadana prędkość będzie wyzerowana. Jeżeli wartość T1 i T2 wynoszą zero timer odpowiedzialny za zatrzymanie silników po ustalonej wartości nie będzie uruchomiony i a zadana prędkość nie będzie zmieniana.

Dane zawierające PWM są składane do zmiennych 16-bitowych i zapisywane ustawiana na timery generujące sygnały PWM Maksymalne wypełnienie wynosi 4200 i wysłanie prędkości większej spowoduję ustawienie tej wartości. Jeżeli te dane zawierają wszystkie wartości równe zero silniki są zatrzymywane

Ustawienie wartości sygnałów PWM jest niemożliwe, gdy uruchomiony jest timer odpowiedzialny za działanie regulatora prędkości. Nie można nastawiać bezpośrednio wartości wypełnienia sygnału PWM, gdy kontroluje tę wartość regulator. Komenda zwraca sterowanie od razu po wywołaniu

Ustawienie wartosci wypelnienia PWM				setPWM(int left, int right, int time)				USTAW	
"B"	17	L1	L2	"M"	R1	R2	T1	T2	"E"

Komenda różni się od poprzedniej tym, że zwraca sterowanie po ekspiracji timera

Pobranie wartości wypełnienia PWM				getPWM()				POBIERZ	
"B"	8	L1	L2	"M"	R1	R2	0	0	"E"

Wiadomość zwraca wartość sygnału PWM dla silników Wartości L1 i L2 są rozłożoną 16-bitową zmienną całkowitą oznaczającą wartość sygnału PWM, który steruje lewym silnikiem. Wartości R1 i R2 są rozłożoną 16-bitową zmienną całkowitą oznaczającą PWM który, steruje prawym silnikiem.

Ustawienie timera aktualizacji prędkości				setEncoderMeas(int time)				USTAW	
"B"	9	T1	T2	"M"	0	0	0	0	"E"

Wiadomość ustawia timer który, co określony interwał czasu aktualizuje prędkość robota Wartości T1 i T2 są rozłożoną 16-bitową zmienną całkowitą oznaczającą ilość milisekund co ile generowane jest przerwanie. Działanie timera i opis aktualizacji prędkości opisany został w rozdziale 3.6

Pobranie nastawionej prędkości regulatora				getEncoderMeas()				POBIERZ	
"B"	10	T1	T2	"M"	0	0	0	0	"E"

Wiadomość zwraca ustawioną wartość milisekund interwału aktualizacji prędkości robota Wartości T1 i T2 są rozłożoną 16-bitową zmienną całkowitą oznaczającą ilość milisekund interwału czasowego timera

Pobranie nastawionej prędkości regulatora				RobotGetSetVel()				POBIERZ	
"B"	11	L1	L2	"M"	R1	R2	0	0	"E"

Wiadomość zwraca prędkość która jest zadana na wejście regulatora. Wartości L1 i L2 są zserializowaną zmienną zmiennoprzecinkową oznaczającą zadaną prędkość dla lewego koła Wartości R1 i R2 są zserializowaną zmienną zmiennoprzecinkową oznaczającą zadaną prędkość dla lewego koła

Ustawienie timera aktywacji regulatora				setRegulationTimer(int time))				USTAW	
"B"	12	T1	T2	"M"	0	0	0	0	"E"

Wiadomość ustawia timer który, co określony interwał czasu uruchamia regulator. Wartości T1 i T2 są rozłożoną 16-bitową zmienną całkowitą oznaczającą ilość milisekund co ile generowane jest przerwanie. Ustawienie wartości T1 i T2 na zero oznacza wyłączenia regulatora i umożliwia nastawianie wartości wypełnienia PWM bezpośrednio na silniki. W przypadku wyłączonego timera, ustawienie wartości większej od zera włącza regulację prędkości. Działanie timera i opis regulacji prędkości opisany został w rozdziale 3.7

Pobranie nastawionej prędkości regulatora				getRegulationTimer()				POBIERZ	
"B"	13	T1	T2	"M"	0	0	0	0	"E"

Wiadomość zwraca ustawioną wartość milisekund interwału regulatora prędkości Wartości T1 i T2 są rozłożoną 16-bitową zmienną całkowitą oznaczającą ilość milisekund interwału czasowego timera

Rozdział 6

Prezentacja możliwości wykonanego projektu

6.1 Dobór nastaw regulator

6.2 Ustawianie trasy robota za pomocą skrypu

6.3 Prezentacja graficznego sterowania

Rozdział 7

Podsumowanie

Bibliografia