Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Informatyki

Praca dyplomowa magisterska

na kierunku Informatyka
w specjalności Inżynieria systemów informatycznych

Opracowanie i weryfikacja systemu sterowania manipulatora o 6 stopniach swobody

Marcin Baran
Numer albumu 259804

promotor
dr inż. Tomasz Winiarski

Warszawa 2019

Opracowanie i weryfikacja systemu sterowania manipulatora o 6 stopniach swobody

Streszczenie

Celem pracy było opracowanie, implementacja i weryfikacja działania systemu sterującego manipulatorem antropomorficznym. Opisane zostały istniejące rozwiązania oraz narzędzia wykorzystane podczas tworzenia projektu. Sterownik robota został wykonany wykorzystując koncepcję agenta upostaciowionego. System składa się z dwóch części. Pierwszą z nich jest oprogramowanie kontrolujące wszystkie peryferia robota i jest oparte na systemie czasu rzeczywistego. Komendy sterujące ruchem manipulatora przekazywane są ze sterownika monitorującego jego stan za pomocą warstwy pośredniczącej (interfejsu sprzętowego). Dodatkowo sterownik połączony został z oprogramowaniem symulacyjnym, co pozwala na sprawdzenie jego działania w wirtualnym środowisku, bez uruchamiania rzeczywistego manipulatora. Główny kontroler robota został oparty o mikrokontroler STM32F407-VET, a program sterujący został napisany wykorzystując biblioteki Standard Peripheral Libraries (STDPeriph) oraz system czasu rzeczywistego FreeRTOS. Sterownik został przygotowany z zastosowaniem pakietu Robot Operating System (ROS) oraz środowiska symulacji Gazebo. Weryfikację systemu wykonano poprzez analizę testów ruchu robota po zadanej trajektorii do określonej pozycji.

Słowa kluczowe:*manipulator, system czasu rzeczywistego, ROS, FreeRTOS, symulacja*

Implementation and analysis of 6 degrees of freedom robotic manipulator control system

Abstract

The aim of this thesis was to invent, implement and verify the operation of antropomorphic manipulator's control system. State of the art and technologies used in system development were discussed. The control driver was designed using an embodied agent theory. The system consists of two parts. First of them is a software managing all robot's peripherals and it is based on a real time operating system (RTOS). The manipulator's movement control commands are transfered from the driver which is monitoring the manipulator's state using hardware interface. Additionally the driver was created along with simulation software. That makes it possible to check it's performance in virtual environment, with no necessity to use actual robot. The main manipulator's controller is STM32F407-VET microchip and it's program was written using Standard Peripheral Libraries (STDPeriph) and FreeRTOS real time operating system. The control driver was prepared based on Robot Operating System (ROS) software and Gazebo simulation environment. The system's verification was done by analysing robot's tests of movement along specified trajectory to given position.

Keywords: *manipulator, real time operating system, ROS, FreeRTOS, simulation*

załącznik nr 10 do zarządzenia
nr 46 /2016 Rektora PW

Politechnika Warszawska
Warsaw University of Technology

Warszawa, 16.09.2019 r.
miejscowość i data
place and date

Marcin Baran
imię i nazwisko studenta
name and surname of the student
259804
numer albumu
student record book number
Informatyka
kierunek studiów
field of study

OŚWIADCZENIE

DECLARATION

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.
Under the penalty of perjury, I hereby certify that I wrote my diploma thesis on my own, under the guidance of the thesis supervisor.

Jednocześnie oświadczam, że:
I also declare that:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- *this diploma thesis does not constitute infringement of copyright following the act of 4 February 1994 on copyright and related rights (Journal of Acts of 2006 no. 90, item 631 with further amendments) or personal rights protected under the civil law,*
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- *the diploma thesis does not contain data or information acquired in an illegal way,*
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- *the diploma thesis has never been the basis of any other official proceedings leading to the award of diplomas or professional degrees,*
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- *all information included in the diploma thesis, derived from printed and electronic sources, has been documented with relevant references in the literature section,*
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.
- *I am aware of the regulations at Warsaw University of Technology on management of copyright and related rights, industrial property rights and commercialisation.*

Politechnika Warszawska
Warsaw University of Technology

załącznik nr 10 do zarządzenia
nr /2016 Rektora PW

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.
I certify that the content of the printed version of the diploma thesis, the content of the electronic version of the diploma thesis (on a CD) and the content of the diploma thesis in the Archive of Diploma Theses (APD module) of the USOS system are identical.

.....
czytelny podpis studenta
legible signature of the student

Spis treści

1 Wprowadzenie **9**
- 1.1 Motywacja . . . 9
- 1.2 Plan pracy . . . 10
- 1.3 Analiza trendów . . . 10

2 Kinematyka mechanizmów wieloczłonowych **15**
- 2.1 Notacja Denavita-Hartenberga . . . 16
- 2.2 Zadanie proste i odwrotne kinematyki . . . 17

3 Opis narzędzi **19**
- 3.1 Sprzęt . . . 19
  - 3.1.1 Manipulator . . . 19
  - 3.1.2 Mysz 3D . . . 21
- 3.2 Oprogramowanie . . . 21
  - 3.2.1 SysML . . . 21
  - 3.2.2 Koncepcja agenta upostaciowionego . . . 22
  - 3.2.3 ROS . . . 23
  - 3.2.4 Gazebo . . . 23
  - 3.2.5 ros_control . . . 24
  - 3.2.6 Spacenav . . . 24
  - 3.2.7 STDPeriph . . . 25
  - 3.2.8 FreeRTOS . . . 25

4 Specyfikacja systemu **27**
- 4.1 Założenia i wymagania . . . 27
  - 4.1.1 Założenia sprzętowe . . . 27
  - 4.1.2 Założenia dotyczące systemu . . . 27
- 4.2 Przypadki użycia systemu . . . 28
- 4.3 Struktura systemu . . . 29
  - 4.3.1 Podsystem sterowania c_{robot} . . . 30
  - 4.3.2 Rzeczywiste efektory $E_{robot1..7}$ . . . 35
  - 4.3.3 Wirtualne efektory $e_{robot1..6}$ . . . 36
  - 4.3.4 Podsystem sterowania c_{sim} . . . 36
  - 4.3.5 Rzeczywisty receptor R_{sim1} . . . 39
  - 4.3.6 Wirtualny receptor r_{sim1} . . . 39
  - 4.3.7 Rzeczywisty receptor R_{sim2} . . . 39
  - 4.3.8 Wirtualny receptor r_{sim2} . . . 39

5 Implementacja systemu **40**
- 5.1 Oprogramowanie kontrolera . . . 40
  - 5.1.1 Wirtualne efektory $e_{robot1..6}$ . . . 40
  - 5.1.2 Podsystem sterowania c_{robot} . . . 43
- 5.2 Sterownik i symulacja . . . 47

6 Weryfikacja **51**

7 Podsumowanie **52**
7.1 Wyniki działania systemu 52
7.2 Wnioski 52
7.3 Perspektywy rozwoju 52

DODATEK A. ZAWARTOŚĆ PŁYTY CD **53**

BIBLIOGRAFIA **54**

WYKAZ SYMBOLI I SKRÓTÓW **56**

SPIS RYSUNKÓW **57**

SPIS TABLIC **58**

1 Wprowadzenie

W latach 1960-1990 robotyka skupiona była głównie na wykorzystaniu robotów do pracy w przemyśle i automatyzacji procesów produkcyjnych. Dzięki ciągłemu rozwojowi w tym polu roboty stały się wszechobecne. W ostatnich kilkudziesięciu latach można zaobserwować rosnące zastosowanie robotów w domenach wykraczających poza ramy samego przemysłu. Międzynarodowa Organizacja Normalizacyjna (ISO) definiuje roboty usługowe jako roboty wykonujące autonomicznie lub półautonomicznie użyteczne zadania dla ludzi lub sprzętu z wyłączeniem zastosowań do automatyzacji procesów [9]. Przykładami takiego wykorzystania mogą być roboty medyczne, sprzątające, czy pomagające osobom starszym. Wiele z tych zastosowań wymaga bezpośredniej interakcji z ludźmi i otoczeniem w precyzyjny i bezpieczny sposób. Stąd przed wprowadzeniem takich rozwiązań do użytku i kontaktu z otoczeniem powinny one być dokładnie sprawdzone i przetestowane w środowisku symulacyjnym.

1.1 Motywacja

Celem pracy magisterskiej jest opracowanie systemu sterowania do autorskiego projektu manipulatora o 6 stopniach swobody. Stworzenie systemu do kontroli ruchu takiego robota wiąże się z kilkoma wymaganiami. Głównym z nich jest bezpieczeństwo użytkownika, a także otoczenia. Z tego względu użytkownik powinien móc sprawdzić poprawność zadanego ruchu przed wykonaniem go na urządzeniu. Ewentualnie występujące błędy mogą generować duże koszty i stanowić zagrożenie. Ponadto oprogramowanie samego robota, które bezpośrednio kontroluje jego napędami, a także odbiera komendy ruchu i wysyła dane o aktualnym stanie powinno być zabezpieczone w przypadku pojawienia się sytuacji zapobiegającej poprawnemu działaniu urządzenia.

Ze względu na wyżej wymienione wymagania opracowany system sterujący powinien pozwalać na weryfikację sterownika poza urządzeniem, a także na możliwie szybkie przeniesienie jego działania na urządzenie. Symulacja rzeczywistego robota pozwala na testy i analizę jakości sterowania i poprawności wykonywanych zadań w środowisku wirtualnym. Następnie zadania te mogą zostać wykonane bezpośrednio na robocie dzięki połączeniu programu symulacyjnego ze sterownikiem urządzenia wykorzystując w tym celu odpowiednio przygotowany interfejs sprzętowy. Zastosowanie systemu czasu rzeczywistego do kontrolowania napędów manipulatora, odczytywania danych z sensorów i komunikowania się z interfejsem sprzętowym sterownika pozwala na wykonanie każdego z tych zadań w zdefiniowanym i najkrótszym czasie, a w przypadku wystąpienia opóźnienia lub błędu zapewnia bezpieczne jego zatrzymanie. Opisany system w znaczącym stopniu ułatwia programowanie, testowanie i wykonywanie ruchów robota. Składa się on z zadajnika (myszki 3D pozwalającej na ręczne zadawanie ruchu robota), aplikacji symulacyjnej i sterującej na komputer PC oraz oprogramowania na mikrokontroler z serii STM32F4.

Konstrukcja i oprogramowanie manipulatora szeregowego jest częścią projektu łazika marsjańskiego Koła Naukowego Robotyków działającego na Wydziale Mechanicznym Energetyki i Lotnictwa Politechniki Warszawskiej.

1.2 Plan pracy

Dokument ten stanowi raport z przeprowadzonej pracy magisterskiej. Składa się on z następujących części:

- rozdział 2 zawiera wprowadzenie teoretyczne do dziedziny kinematyki mechanizmów wieloczłonowych; zostały w nim opisane zadania proste i odwrotne kinematyki manipulatorów oraz notacja Denavita-Hartenberga użyta do opisu konfiguracji manipulatora,
- rozdział 3 opisuje konstrukcje i technologie użyte w projekcie autorskiego manipulatora szeregowego,
- rozdział 4 przedstawia architekturę oprogramowania wykorzystując koncepcję agenta upostaciowionego i język opisu sprzętowego SysML,
- rozdział 5 przedstawia implementację systemu: działanie oprogramowania kontrolera robota opartego na systemie czasu rzeczywistego FreeRTOS oraz sterownika i oprogramowania symulacyjnego stworzonego z pomocą pakietu ROS oraz środowiska Gazebo,
- rozdział 6 stanowi szczegółowy opis przeprowadzonych testów ruchu manipulatora w środowisku symulacyjnym Gazebo oraz uzyskane wyniki,
- rozdział 7 poświęcony jest podsumowaniu pracy.

1.3 Analiza trendów

Oprogramowanie sterujące każdego robota jest rozwijane mając na uwadze to, że powinno być w stanie zapewnić jego użytkownikowi zaimplementowanie wykonywanego przez robota zadania w możliwie prosty sposób. Aby było to możliwe system sterujący powinien być przygotowany tak, aby pozwalał na:

- zdefiniowanie prędkości poszczególnych napędów
- zdefiniowanie pozycji chwytaka/narzędzia
- zaprogramowanie robota, by podążał wyznaczoną ścieżką
- zaprogramowanie robota, by wykonywał określone zadanie

Zieliński w swoim artykule [2] wskazuje na ewolucje podejścia przy tworzeniu oprogramowania sterującego robotami jakie nadeszło wraz z rosnącym rozwojem robotyki i tym samym zwiększaniem liczby rodzajów stosowanych konstrukcji. W rozwiązaniach przemysłowych głównymi metodami programowania ruchu są:

- programowanie online (nietekstowe)- polega na tym, że operator przesuwa robota do wymaganych pozycji, które są zapamiętywane, a później odtwarzane
- programowanie offline (tekstowe)- polega na wykorzystaniu specjalnie przygotowanego języka komend i struktur do definiowania rodzaju i punktów docelowych ruchu robota

Rozszerzenie programowania nietekstowego o formę programowania offline spowodowało powstanie hybrydowej metody programowania robotów, która jest dziś szeroko stosowana w przemyśle. Jednym z przykładów takiego rozwiązania jest KUKA Robot Language (KRL) dedykowany do robotów firmy KUKA. Jednakże ze względu na pojawiające się nowe rodzaje robotów ta metoda wymaga ciągłej zmiany języków programowania, a także ich interpreterów.

W książce Kozłowskiego i innych [11] w rozdziale poświęconym systemom programowania robotów autorzy wskazują uwagę na fakt, że w pracach badawczych dotyczących algorytmów sterowania i planowania trajektorii często używane są roboty przemysłowe. Dużą przeszkodą w prowadzeniu takich prac jest zamknięta konstrukcja sterownika robota przemysłowego. W dodatku projektowane są one do wykonywania określonych i powtarzalnych zadań, stąd są mało przydatne w celach badawczych. Z tego względu autorzy zaproponowali połączenie robota posiadającego własny sterownik z komputerem PC stosując jedno z dwóch rozwiązań:

- wykorzystując robota z interfejsem sieciowym do komunikacji
- własnoręcznie definiując i implementując interfejs komunikacji z urządzeniem

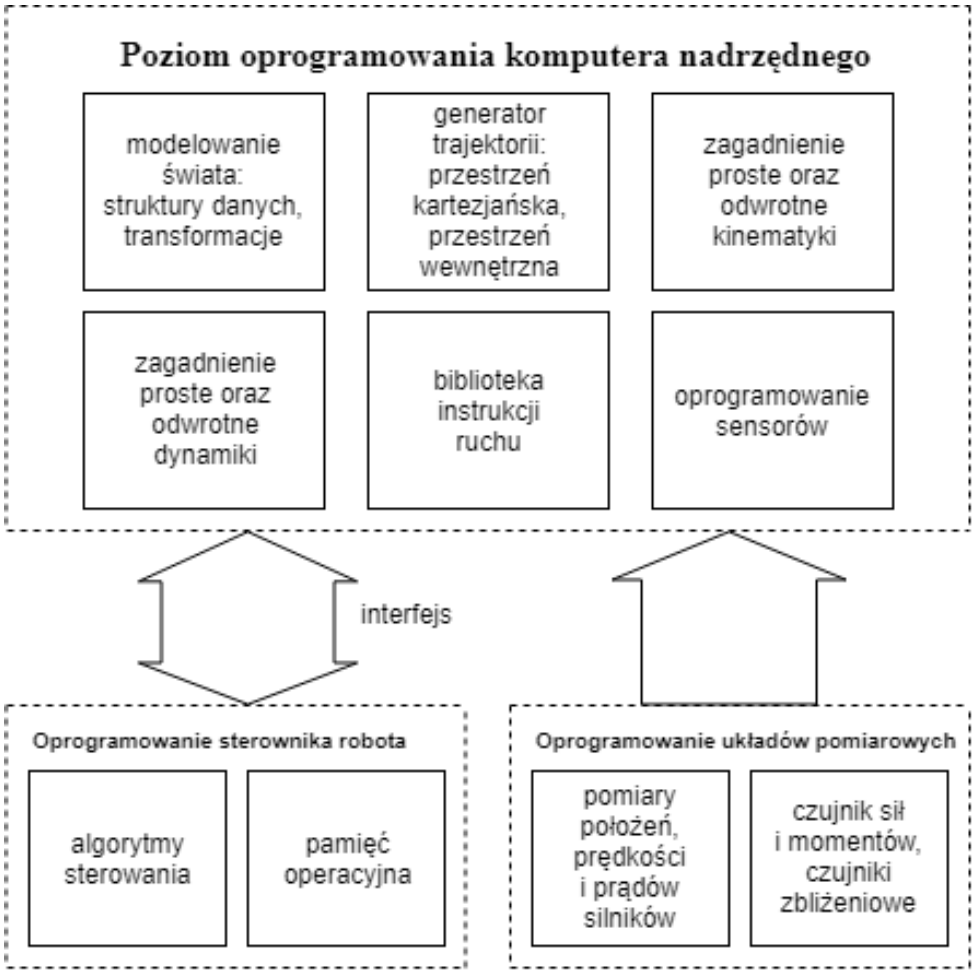
Przedstawione zostały oba rozwiązania. Autorzy stworzyli dwa stanowiska badawcze: jedno z nich oparte na manipulatorach przemysłowych Staübli RX60 z własnym interfejsem sieciowym, drugie na manipulatorze produkcji polskiej IRp-6 ze specjalnie do niego stworzonym interfejsem komunikacyjnym. W ten sposób udało się uzyskać uniwersalny system programowania bez ingerencji w oprogramowanie producenta składający się z trzech części (rysunek 1.1):

- sterownika robota
- urządzeń pomiarowych
- komputera nadrzędnego

Autorzy dodatkowo opracowali od podstaw oprogramowanie komputera nadrzędnego, które pozwoliło na rozszerzenie możliwości sterowania robotem poprzez napisanie własnych programów sterujących (w uniwersalnych językach wysokiego poziomu, takich jak C++). Taka architektura systemu daje większe możliwości względem specjalizowanych języków programowania wspomnianych wcześniej. Jednakże stworzenie programów sterujących od podstaw zajmuje dużo czasu. Z tego względu rozsądnym byłoby dołączenie dodatkowych, uniwersalnych bibliotek z szablonami potrzebnych funkcjonalności i dostosowaniu ich do badanego robota. Takim rozwiązaniem są programowe struktury ramowe (ang. frameworks).

W dalszej części artykułu [2] autor opisuje programowe struktury ramowe jako bibliotekę oraz wzorce jej wykorzystania. Użytkownik może ją dostosować do swojego urządzenia używając dobrze udokumentowane szablony oraz dostarczone z nią narzędzia, do których zaliczyć można symulatory czy debuggery. Dodatkowo autor wskazuje, że wykorzystanie programowych struktur ramowych w robotyce doprowadziło do zaniku podziału na oprogramowanie sterujące sprzętem, interpreter języka programowania robota i program użytkowy. Nie ma potrzeby użycia specjalnego interpretera z tego względu, że program użytkowy jest tworzony w tym samym języku co oprogramowanie sterujące. To rozwiązanie pozwala na znaczne przyspieszenie tworzenia oprogramowania, a także na łatwe dołączenie dodatkowych funkcjonalności systemu.

Jednym z przykładów takiej struktury jest ROS [4], który jest jednym z najpopularniejszych i najbardziej zaawansowanych narzędzi do tworzenia sterowników robotycznych. O jego popularności świadczyć może duża liczba literatury badawczej, w której pomocny był ten pakiet. Kilka pozycji zostało wymienione poniżej:



Rysunek 1.1: Schemat stworzonego systemu programowania *(Źródło: [11])*

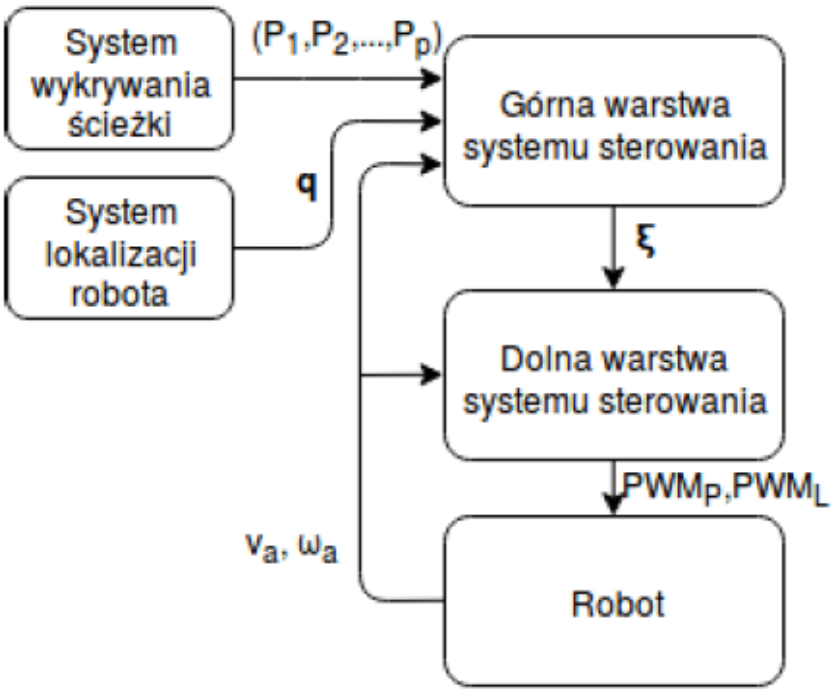
- *Sterowanie predykcyjne z wykorzystaniem wizji w zadaniu śledzenia ścieżki przez robota mobilnego[12]*:

  Praca przedstawia badanie wykorzystania algorytmów sterowania predykcyjnego do zadania śledzenia ścieżki. Zadanie wykonywane jest przez robota mobilnego klasy 2.0 na podstawie danych z kamery zamontowanej na robocie albo umieszczonej nad trasą, którą podążać ma robot. W pracy wykorzystano bibliotekę OpenCV do przetwarzania obrazu oraz środowisko MATLAB do implementacji algorytmów sterowania. System sterowania podzielono zgodnie z poniższym schematem:

  Ze względu na to, że system składa się z wielu równolegle działających procesów na trzech różnych platformach (komputer stacjonarny, komputer jednopłytkowy Raspberry Pi, mikrokontroler) autor wykorzystał pakiet ROS do komunikacji pomiędzy zadaniami oraz użył go do rejestracji i wizualizacji danych. ROS bardzo dobrze nadaje się do implementacji komunikujących się procesów. Poszczególne procesy są w nim odzwierciedlane jako węzły (ang. *nodes*), które komunikują się poprzez tematy (ang. *topics*).

- *System robotyczny chwytający obiekty [10]*:

  Celem pracy było stworzenie systemu robotycznego opartego o manipulator IRp-6, który umożliwiałby chwytanie obiektu. Zadanie chwytania wykonywane było stosując obraz z kamery 2D zamontowanej w kiści robota oraz znając model chwytanego przedmiotu. W tym przypadku system ROS został użyty w połączeniu z systemem IRPOS (IRp-6).



Rysunek 1.2: Struktura systemu sterowania robotem mobilnym *(Źródło: [12])*

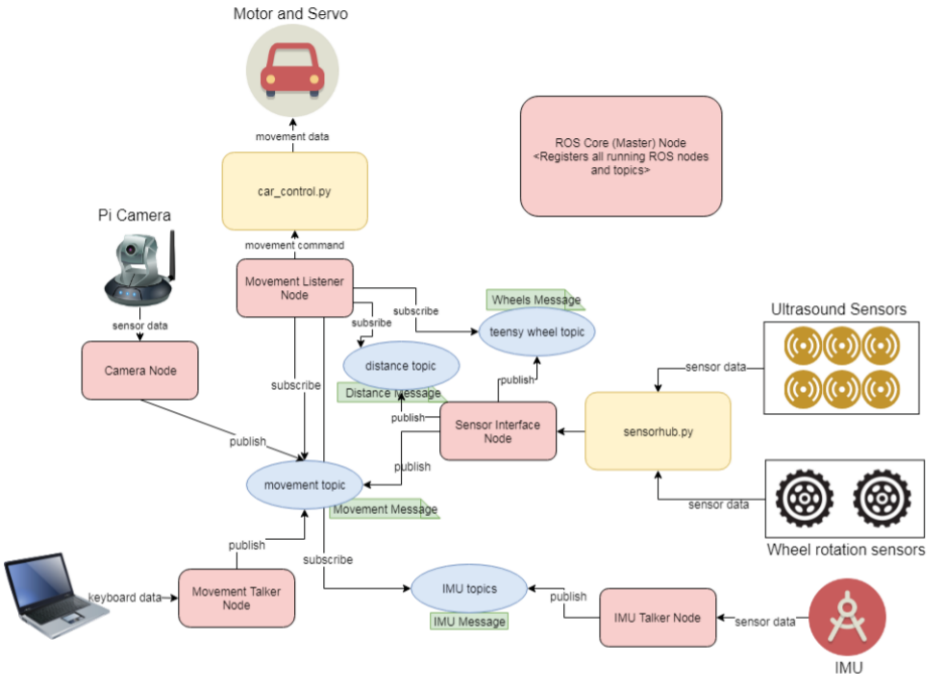
Służy on do sterowania dwoma robotami IRp-6 (o nazwach *Track* i *Postument*) i zawiera wiele gotowych rozwiązań ułatwiających wykonywanie skomplikowanych trajektorii ruchu ramienia robota. Dodatkowo do przetwarzania obrazu użyto biblioteki OpenCV oraz struktury ramowej DisCODe do przetwarzania danych uzyskanych z kamery. Wykorzystanie systemu ROS do komunikacji i procesowania wszystkich zadań daje możliwość dalszego rozwoju projektu poprzez dodanie kolejnych węzłów wykonujących nowe zadania. Ponadto różnorodność użytych komponentów i bibliotek wskazuje na bardzo elastyczną inkorporacje gotowych rozwiązań do systemu robotycznego opartego o ROS.

- *Development of autonomous driving using Robot Operating System [22])*:

  Praca skupia się na opisie wykorzystania ROS do stworzenia oprogramowania przeznaczonego do autonomicznej jazdy. W celu weryfikacji zaimplementowanych rozwiązań stworzony system zastosowany jest w zdalnie sterowanym pojeździe wyposażonym w jednopłytkowy komputer pojazdu Raspberry Pi 3, płytkę z mikrokontrolerem Arduino Uno do sterowania silnikami oraz czujniki takie jak: kamera 2D, ultradźwiękowe czujniki odległości i IMU (ang. *Intertial Measurement Unit*). W tym projekcie architektura systemu została całkowicie oparta o pakiet ROS zainstalowany na komputerze pojazdu z systemem operacyjnym Ubuntu 16.04.

  Zgodnie z grafiką 1.3 operacje każdego z elementów systemu (komunikacja zdalna z użytkownikiem, akwizycja oraz obróbka obrazu z kamery, czujników ultradźwiękowych i IMU, sterowanie silnikami) odbywa się w oddzielnym węźle ROS (oznaczone na czerwono). Przesyłanie danych odbywa się za pomocą zdefiniowanych tematów (oznaczone na niebiesko) i na podstawie tych danych sterowane są silniki pojazdu.

  Autor wskazuje na zalety systemu ROS, takie jak możliwość tworzenia modułowego oprogramowania oraz szeroką dostępność gotowych bibliotek czujników i narzędzi symu-



Rysunek 1.3: Struktura systemu sterowania autonomicznym pojazdem RC *(Źródło: [22])*

lacyjnych. W pracy użyto symulatora Gazebo przystosowanego do współpracy z ROS w celu przetestowania systemu sterowania pojazdem.

Oprócz tego zauważone zostają wady takie jak istnienie pojedynczego punktu awarii, jakim jest zbiór węzłów *roscore*. Są one wymagane w celu komunikacji pomiędzy uruchomionymi procesami (węzeł ROS Master). Ponadto autor wskazuje, że komunikacja w sieci ROS nie jest zabezpieczona, co może stanowić zagrożenie bezpieczeństwa systemu. Obie te niedogodności są rozwiązywane w ramach powstania nowej wersji oprogramowania ROS (ROS version 2).

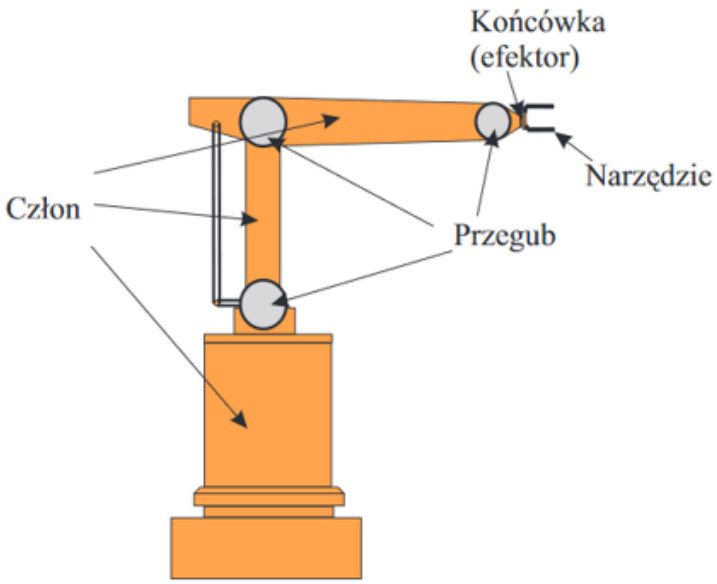
Szeroka gama gotowych modułów dodatkowych (takich jak symulator Gazebo), możliwość łatwego rozszerzania funkcjonalności systemu sterowania opartego o ROS oraz łatwość w dostosowaniu gotowych bibliotek do współpracy z pakietem zdecydowały o użyciu tego pakietu w niniejszej pracy.

2 Kinematyka mechanizmów wieloczłonowych

Niniejszy rozdział przedstawia podstawy teoretyczne sterowania manipulatorem o strukturze szeregowej. Manipulator jest definiowany jako mechanizm wieloczłonowy, czyli łancuch sztywnych członów połączonych ruchomymi przegubami, które pozwalają na ruch względny członów. Istnieje kilka róznych typów przegubów, jednakże najczęściej stosowanymi są tylko dwa:

- obrotowy - pozwalajacy na obrót członu wzgledem tylko jednej osi
- postępowy - pozwalający na ruch postępowy członu tylko w jednym kierunku

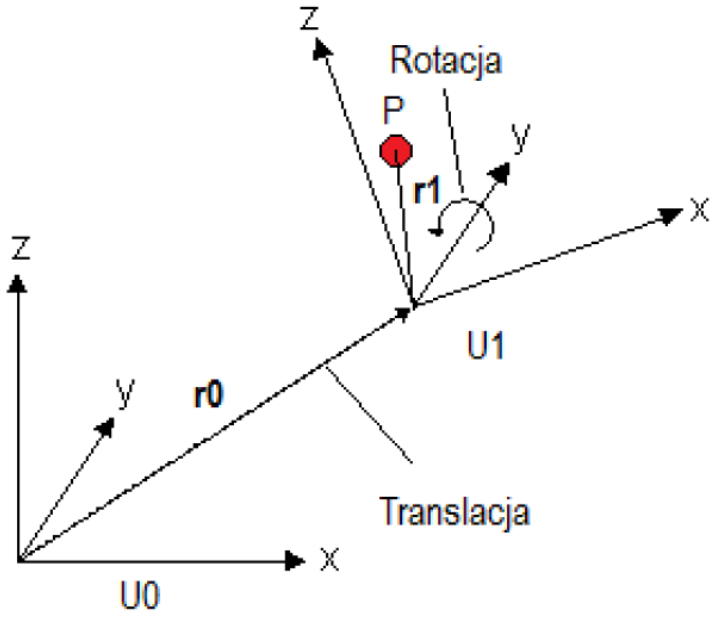
Podstawowym parametrem charakteryzującym manipulator jest liczba stopni swobody, która określa najmniejszą liczbę współrzędnych jednoznacznie opisujacą jego konfigurację (położenie i orientację w przestrzeni każdego członu). Ze względu na to, że najczęściej stosowane typy przegubów pozwalają na ruch postępowy albo obrotowy względem jednej osi, liczba stopni swobody często równa jest liczbie przegubów. Aby koncówka manipulatora (efektor) mogła uzyskać jednoznacznie zadane położenie i orientację w przestrzeni trójwymiarowej potrzebna jest znajomość 6 współrzędnych. Z tego wzgledu najczęściej stosowane są konstrukcje manipulatorów o 6 stopniach swobody.



Rysunek 2.1: Konstrukcja manipulatora szeregowego *(Źródło: [21])*

Do poprawnego sterowania manipulatorem robotycznym potrzebna jest znajomość położenia i prędkości poszczególnych jego członów względem bazowego, nieruchomego układu

współrzędnych, który nazywany jest układem globalnym. W tym celu dla każdego członu sztywnego definiuje sie związany z nim lokalny układ współrzędnych. Wtedy położenie kazdego członu można określić jako połozenie układu lokalnego wzgledem globalnego. Podstawowymi operacjami stosowanymi przy opisie kinematyki mechanizmów wieloczłonowych są rotacja i translacja o wektor. Pozwalają one na zdefiniowanie wektora przesunięcia oraz macierzy rotacji układu lokalnego względem układu globalnego (rysunek 2.2).



Rysunek 2.2: Położenie członu P i związanego z nim układu U_1 opisane względem globalnego układu U_0

Znając macierz rotacji $\boldsymbol{R}$, wektor translacji $\boldsymbol{r_0}$ układu lokalnego U_1 względem globalnego U_0 oraz wektor translacji członu P względem układu lokalnego U_1 możemy wyznaczyć połozenie członu P względem układu globalnego U_0:

$$\boldsymbol{r} = \boldsymbol{r_0} + \boldsymbol{R r_1} \tag{2.1}$$

Dodatkowo dla ułatwienia zapisu równań (2.1) wprowadzono pojęcie macierzy transformacji [11] pomiędzy układem $i-1$, a układem i. Jest to macierz 4x4, która tworzona jest na podstawie macierzy rotacji (3x3) oraz wektora translacji (3x1) i przedstawia się następująco:

$$\overset{i-1}{i}\boldsymbol{T} = \begin{bmatrix} \boldsymbol{R_{3x3}} & \boldsymbol{r_{i-1,3x1}} \\ 0_{1x3} & 1 \end{bmatrix} \tag{2.2}$$

Wtedy odnosząc się do rysunku 2.2 położenie członu P opisane względem globalnego układu U_0 można opisać jako:

$$\begin{bmatrix} \boldsymbol{r} \\ 1 \end{bmatrix} = \overset{0}{1}\boldsymbol{T} \begin{bmatrix} \boldsymbol{r_0} \\ 1 \end{bmatrix} \tag{2.3}$$

2.1 Notacja Denavita-Hartenberga

Powszechną metodą opisu położenia poszczególnych ogniw dla manipulatora posiadającego jedynie pary obrotowe i postępowe jest notacja Denavita-Hartenberga, w której każdemu członowi przyporządkowane są cztery wartości [11]:

- a_{i-1} - długość $i-1$ ogniwa, mierzona jako odległość między osiami przegubów $i-1$ oraz i,
- α_{i-1} - kąt skręcenia $i-1$ ogniwa prawoskrętnie wokół a_i, mierzony jako kąt między osiami przegubów i oraz $(i+1)$,
- d_i - odległość mierzoną wzdłuż osi i-tego przegubu między a_{i-1} i a_i,
- θ_i - kąt między a_{i-1} i a_i, określony prawoskrętnie wokół osi i-tego przegubu.

Metoda zakłada również, że osie ortogonalnego układu współrzędnych, związanego z i-tym ogniwem są skierowane następująco [11]:

- z_i pokrywa się z osią i-tego przegubu,
- x_i jest prostopadła do osi z_i oraz z_{i+1} i jest skierowana od przegubu i do $i+1$,
- y_i uzupełnia prawoskrętny układ współrzędnych.

Ze względu na przyjęte powyżej założenia notacja wymaga znajomości czterech parametrów do określenia wzajemnego położenia układów zamiast sześciu.

2.2 Zadanie proste i odwrotne kinematyki

W trakcie pracy robota jego układ sterowania musi być w stanie określić położenie końcówki na podstawie położenia każdego z jego napędów (a tym samym członów). Ponadto chcąc zmienić położenie końcówki należy wiedzieć w jaki sposób powinny być ułożone poszczególne człony, aby daną pozycję osiągnąć. Z tego względu wymagane jest rozwiązanie zadania prostego i odwrotnego kinematyki dla zadanego robota.

Zadanie proste kinematyki polega na tym, aby wyznaczyć współrzędne zewnętrzne manipulatora (wpółrzędne końcówki), kiedy dane są współrzędne wewnętrzne (każdego z członów). Sprowadza się więc ono zatem do znalezienia macierzy transformacji ${}^{0}_{n}\boldsymbol{T}$ (gdzie n to liczba stopni swobody) i obliczenia wektora współrzędnych $\boldsymbol{r}$, tak jak pokazano to w przykładzie z rysunku 2.2:

$$\begin{bmatrix} \boldsymbol{r} \\ 1 \end{bmatrix} = {}^{0}_{n}\boldsymbol{T} \begin{bmatrix} \boldsymbol{s} \\ 1 \end{bmatrix} \tag{2.4}$$

, przy czym $\boldsymbol{s}$ to współrzędne końcówki w układzie związanym z ostatnim stopniem swobody.

Zadanie odwrotne kinematyki służy obliczeniu współrzędnych członów znając położenie i orientację końcówki robota. Dla manipulatorów szeregowych wiążę się z rozwiązaniem nieliniowego układu równań i może nie mieć jednoznacznego rozwiązania (na przykład gdy manipulator ma więcej niż 6 stopni swobody). Ze względu na niekiedy trudne znalezienie rozwiązań analitycznych zadania odwrotnego dla skomplikowanych konstrukcji do jego rozwiązania stosuje się metody numeryczne. Przyjmując za szukany wektor współrzędnych wewnętrznych manipulatora $\boldsymbol{q}$, taki że:

$$\boldsymbol{q} = [q_1 q_2 \ldots q_n]^T \tag{2.5}$$

można przedstawić rozwiązywany układ równań jako:

$$\boldsymbol{\phi}(\boldsymbol{q}) = \begin{bmatrix} \phi^1(\boldsymbol{q}) \\ \phi^2(\boldsymbol{q}) \\ \dots \\ \phi^n(\boldsymbol{q}) \end{bmatrix} = 0_{nx1} \tag{2.6}$$

Mając postawiony problem w tej postaci można rozwiązać go stosując na przykład metodę Newtona-Rhapsona. Dodatkowo różniczkując powyższe równanie po czasie otrzymujemy zadanie odwrotne kinematyki o prędkościach, po rozwiązaniu którego wyznaczyć można prędkości każdego złącza niezbędne do uzyskania zadanej prędkości końcówki robota. Metody numeryczne są często wykorzystywane do rozwiązywania zadania odwrotnego w oprogramowaniu symulacyjnym (pakiet ROS także posiada biblioteki na to pozwalające).

3 Opis narzędzi

3.1 Sprzęt

3.1.1 Manipulator

System sterowania opracowywany jest do autorskiej konstrukcji manipulatora o 6 stopniach swobody. Głównym założeniem robota było jego wykorzystanie jako ramię operacyjne dla prototypu łazika marsjańskiego przeznaczonego do startu w zawodach URC 2018. Manipulator przedstawiony jest na zdjęciu poniżej 3.1:

Rysunek 3.1: Zaprojektowany manipulator wykonujący zadanie konkursowe

Konstrukcja stworzona była biorąc pod uwagę regulaminowe wymagania i ograniczenia konkursowe [17]:

- lekka konstrukcja o masie 13 kg
- udźwig do 5 kg
- zasięg maksymalny 1.2 m
- moźliwość wykonywania precyzyjnych zadań, takich jak odkręcanie zaworów, przełączanie przycisków, unoszenie przedmiotów o nieregularnym kształcie

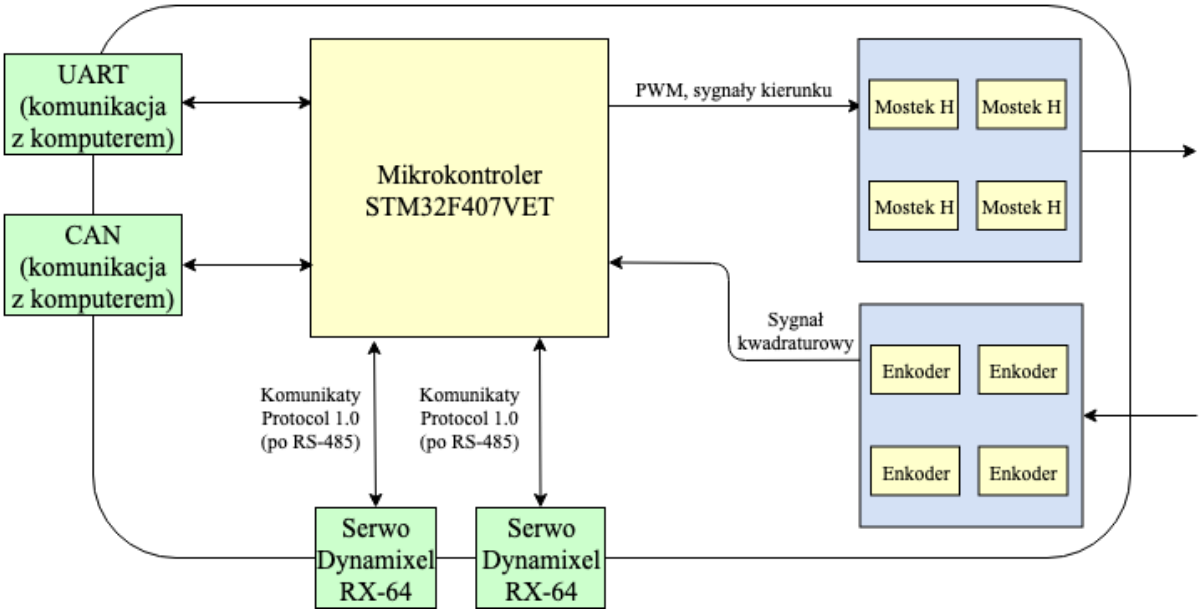
Skonstruowany manipulator wykorzystuje przeguby obrotowe do poruszania każdym stopniem swobody. Ze wzgledu na powyższe wymagania mechanizm został złożony wykorzystując elementy z giętej blachy aluminiowej oraz elementy wytworzone metodą druku 3D. Jako

napędów do poruszania przegubów użyto 3 silników DC, 1 siłownika i 2 serwomechanizmów Dynamixel RX-64 [14]. Manipulator zasilany jest napieciem 12 V. Sterowany jest za pomocą mikrokontrolera STM32F407-VET. Informacje o aktualnej prędkości i pozycji poszczególnych przegubów uzyskiwane są wykorzystując odczyty z enkoderów magnetycznych. Tam gdzie było to możliwe enkodery montowane były bezpośrednio na przegubie. W ten sposób mierzona jest prędkość członu, a nie wału silnika. W miejscach gdzie nie było to możliwe istnieje potrzeba uwzględnienia przełożeń napędów, co zostało pokazane w tabeli:

Tablica 3.1: Przełożenia uwzględniane przy odczycie prędkości związane z montażem enkoderów

Stopień swobody	Przełożenie
1	131 : 2
2	1 : 1
3	1 : 1
4	131 : 1
5	1 : 1
6	1 : 1

Robot wyposażony jest w 2 interfejsy komunikacyjne: UART oraz CAN. Ponadto do komunikacji z serwomechanizmami wykorzystywany jest interfejs RS-485. Uproszczony schemat elektroniczny manipulatora przedstawia poniższy diagram:



Rysunek 3.2: Schemat komunikacji mikrokontrolera sterującego robotem

Konfigurację poszczególnych członów manipulatora opisano wykorzystując następujące parametry Denavita-Hartenberga (zgodnie z ustalonymi układami współrzędnych każdego z członów):

Tablica 3.2: Parametry Denavita-Hartenberga manipulatora

i	$\alpha_{i-1}[rad]$	$a_{i-1}[m]$	$\delta_i[m]$	$\theta_i[rad]$
1	$\pi/2$	0	0.2005	θ_1
2	0	0.6	0	$\theta_2+\pi/2$
3	-$\pi/2$	0.408	0	$\theta_3-\pi/2$
4	$\pi/2$	0	0.129	θ_4
5	-$\pi/2$	0	0	θ_5
6	0	0	0.15	θ_6

3.1.2 Mysz 3D

Jako zadajnik ruchu wykorzystywana jest mysz 3D Magellan Space Mouse Plus [8] (rysunek 3.3). Pozwala ona na wygodne ręczne sterowanie manipulatorem z wykorzystaniem możliwości ruchu (translacji i obrotu) w 3 niezależnych osiach. Komunikacja z komputerem odbywa się poprzez interfejs szeregowy RS-232.

Rysunek 3.3: Mysz 3D Magellan Space Mouse Plus *(Źródło: [8])*

3.2 Oprogramowanie

3.2.1 SysML

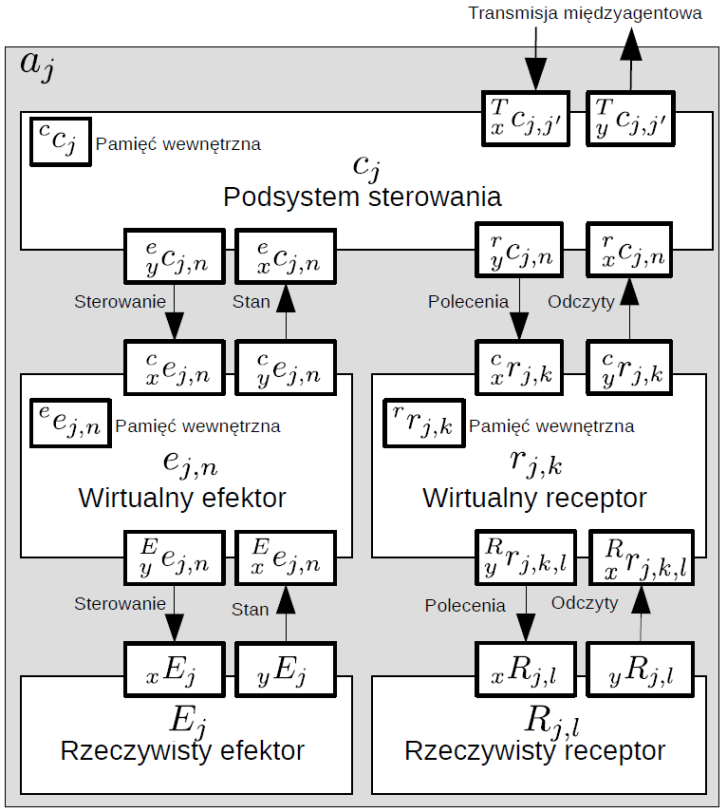
SysML (ang. System Modeling Language)[13] jest jedną z metod graficznego opisu specyfikacji oraz implementacji systemów. Jest to język modelowania, który powstał jako modyfikacja i rozbudowa standardu UML (ang. Unified Modeling Language). Zawiera on 9 typów diagramów podzielonych na 3 rodzaje:

- diagramy zachowań,
- diagramy wymagań systemowych,
- diagramy struktury.

W pracy wykorzystano diagram przypadków użycia (ang. Use Case Diagram), diagram maszyny stanowej (ang. State Machine Diagram), diagram klas.

3.2.2 Koncepcja agenta upostaciowionego

W pracy wykorzystano również koncepcje agenta upostaciowionego [1] w celu opisu systemu. Metodyka ta zakłada podział systemu na grupę agentów utrzymujących ze sobą komunikację. Ogólny schemat przedstawiający agenta upostaciowionego przedstawiony jest na rysunku 3.4.



Rysunek 3.4: Schemat agenta upostaciowionego *(Źródło: [20])*

Każdy agent (oznaczony na rysunku 3.4 jako a_j) posiada dokładnie jeden podsystem sterowania i może posiadać po kilka wirtualnych oraz rzeczywistych receptorów i efektorów, które opisane są poniżej:

- podsystem sterowania (c_j) - wykonuje zadanie zgodne z aktualnym stanem na podstawie danych z wirtualnych efektorów i receptorów poprzez wydawane im komendy; ponadto podsystem sterowania odpowiada za komunikację z innymi agentami w systemie;
- rzeczywisty efektor (E_j) - to fizyczne części, którymi agent może oddziaływać na środowisko zewnętrzne (w przypadku manipulatora to silniki do poruszania każdym z jego członów);
- rzeczywisty receptor ($R_{j,l}$) - są to czujniki dające wiedzę o stanie środowiska zewnętrznego
- wirtualny efektor ($e_{j,n}$) - jest to abstrakcyjna warstwa pośrednicząca między rzeczywistym efektorem, a podsystemem sterowania, jej zadaniem jest modyfikowanie komend sterujących i danych o stanie rzeczywistego efektora (przykładowo przelicza zadaną komendę prędkości z wartości wyrażonej w rad/s do wartości PWM na silnikach);
- wirtualny receptor ($r_{j,k}$) - tak jak wirtualny efektor pośredniczy w komunikacji pomiędzy rzeczywistym receptorem, a podsystemem sterowania (na przykład poprzez przeliczanie odczytów czujników).

3.2.3 ROS

ROS (Robot Operating System) [4] jest to ogólnodostępna programowa struktura ramowa służąca do tworzenia i rozwijania oprogramowania dla robotów. Zawiera biblioteki i narzędzia dostarczające gotowe sterowniki urządzeń i/lub rozwiązania przydatne do sterowania, testowania i symulowania pracy robota. ROS wspiera jezyki programowania C++ i Python. Program stworzony stosując pakiet ROS składa się z takich elementów jak:

- Węzły (ang. nodes) – pojedyncze procesy obliczeniowe, z których każdy odpowiada za pewną funkcjonalność. Węzły mogą komunikować się ze sobą za pomocą tematów (ang. topics);
- Zarządca (ang. master) - jest to proces odpowiedzialny za rejestrację i obserwacje węzłów w sieci. Dzięki niemu możliwa jest odszukanie się dwóch węzłów i komunikacja między nimi. Stąd do poprawnego działania wymagane jest uruchomienie węzła zarządcy. Ponadto zarządca udostępnia do użytku serwer parametrów;
- Wiadomości (ang. messages) – zdefiniowane przez użytkownika struktury danych, które mogą być przesyłane pomiędzy węzłami wykorzystując tematy;
- Tematy (ang. topics) – nazwane porty komunikacji pomiędzy węzłami, które mogą publikować lub subskrybować się na wiadomości nadchodzące do tematu;
- Usługi (ang. services) – typ komunikacji pytanie - odpowiedź. Pozwala na zdalne wywołanie procedury. Usługi mogą być blokujące lub nieblokujące
- Serwer parametrów (ang. parameter server) - służy do przechowywania danych, do których odnosić może się każdy węzeł za pomocą klucza.

W celu ujednolicenia sposobu definiowania kinematyki, dynamiki i wyglądu robotów oraz udostepnienia możliwości dzielenia się modelami przygotowanymi przez różnych użytkowników zdefiniowano standard opisu URDF (Unified Robotic Description Format) [7]. Pliki modeli URDF są to pliki XML, w których definiowane są parametry robota, takie jak:

- położenia i orientacje początkowe poszczególnych członów i związanych z nimi układów odniesienia,
- położenia i rodzaje przegubów łączących ze sobą zdefiniowane człony,
- rozmiary, masy, momenty bezwładności poszczególnych członów,
- ograniczenia ruchu w przegubach,
- parametry napedów sterujących ruchem przegubów.

3.2.4 Gazebo

Gazebo jest darmowym symulatorem dynamiki 3D stworzonym na potrzeby rozwoju robotyki. Umożliwia korzystanie z 4 silników fizycznych: ODE, Bullet, Simbody i DART (domyślnie używany jest ODE). Do renderowania otoczenia i modelu wykorzystuje silnik graficzny OGRE. Dodatkowo Gazebo posiada duży zbiór gotowych, popularnych modeli robotów, które można wykorzystać we własnych projektach, a także pozwala na uruchomienie w zdalnym środowisku lub w chmurze. Ponadto jest łatwo integrowany z pakietem ROS. W niniejszej pracy został użyty do symulowania i sprawdzania poprawności wykonania zadania przed uruchamianiem programu sterującego bezpośrednio na robocie.

3.2.5 ros_control

W celu połączenia oprogramowania symulacyjnego i sterownika robota użyto bibliotekę ros_control [6] [3]. Jest ona dedykowana do użytku z pakietem ROS i umożliwia szybką implementację oprogramowania sterującego. Biblioteka ta jest szablonem kontrolera robota, który można wykorzystać do własnego projektu. Ponadto posiada zestaw zaimplementowanych, szeroko używanych sterowników ruchu członów robota, które wykorzystują regulator PID. Są to między innymi:

- velocity_controllers - sterowanie prędkością/pozycją/mocą na podstawie zadanej prędkości,
- position_controllers - sterowanie prędkością/pozycją/mocą na podstawie zadanej pozycji,
- effort_controllers - sterowanie prędkością/pozycją/mocą na podstawie zadanej mocy.

Dodatkowo biblioteka pozwala na zarządzanie sterownikami w czasie działania za pomocą controller_managera, który odpowiedzialny jest za nadzorowanie pracy sterowników, inicjalizowanie ich oraz zajmowaniu się sytuacjami konfliktowymi między nimi.

Podstawą działania każdego sterownika pisanego przy użyciu ros_control jest sprzętowa warstwa abstrakcji (interfejs sprzętowy), która łączy rzeczywistego/symulowanego robota ze sterownikiem programowym. Ta warstwa abstrakcji dostarczona jest poprzez klasę *hardware_interface::RobotHW* (rysunek 3.5). Implementacja sterownika pod konkretnego robota musi dziedziczyć po tej klasie. W ten sposób możliwe jest pisanie oprogramowania, które może zostać w części lub w całości wykorzystane ponownie. Dodatkowo oznacza to, że sterownik napisany z pomocą ros_control jest niezależny od sprzętu użytego w konstrukcji robota, gdyż zastosowany jest interfejs sprzętowy.

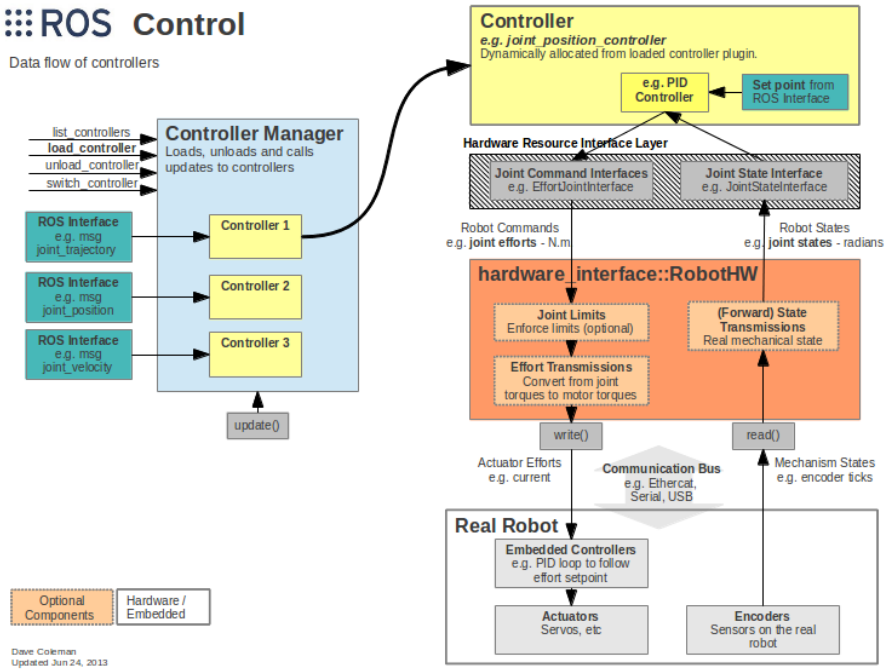
Biblioteka ros_control została w pracy użyta do zaimplementowania sterownika każdego z członów robota (zarówno w symulacji jak i rzeczywistego urządzenia), a także jako warstwa pośrednicząca w komunikacji pomiędzy mikrokontrolerem, a komputerem z uruchomionym oprogramowaniem sterującym i symulacyjnym.

3.2.6 Spacenav

Biblioteka spacenav [5] jest darmowym pakietem, który umożliwia użycie myszy 3D firmy 3Dconnexion w połączeniu z ROS. Pakiet oparty jest o darmowe sterowniki do tych urządzeń [19]. Zasada działania biblioteki opiera się o uruchomienie węzła w ROS o nazwie spacenav_node, który zbiera dane z podłączonego do komputera urządzenia i publikuje je w 4 tematach:

- spacenav/offset
- spacenav/rot_offset
- spacenav/twist
- spacenav/joy

W ten sposób możliwe jest odczytanie aktualnego wychylenia i obrotu względem osi urządzenia odzcytując dane z powyższych tematów.



Rysunek 3.5: Schemat działania sterownika robota opartego o dodatek ros_control *(Źródło: [6])*

3.2.7 STDPeriph

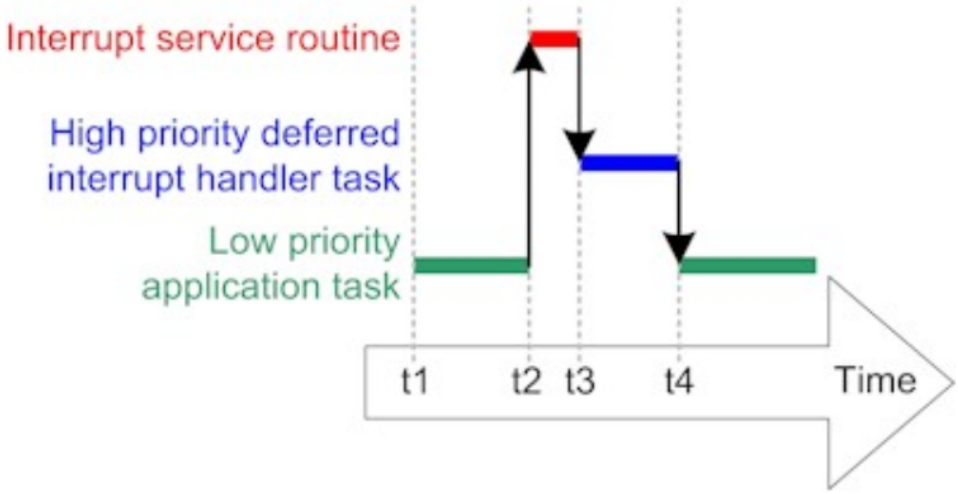
Do stworzenia oprogramowania dla mikrokontrolera STM32F407-VET zarządzającego peryferiami robota wykorzystano STDPeriph [18]. Jest to zbiór bibliotek, które uwalniają użytkownika od konieczności programowania procesorów STM32 stosując tylko zapis bitów do odpowiednich adresów rejestrów. STDPeriph posiada zdefiniowane struktury peryferiów, zmapowane adresy odpowiednich rejestrów dla każdego mikrokontrolera oraz udostępnia API, które ułatwiają inicjalizację i korzystanie z każdej funkcjonalności. Ponadto biblioteki stworzone zostały tak, aby napisany kod mógł zostać użyty bez zmian korzystając z innego mikrokontrolera z tej samej rodziny.

3.2.8 FreeRTOS

Aby zapewnić bezpieczne i pewne wykonywanie zadań przez mikrokontroler sterujący manipulatorem zdecydowano się na oparcie oprogramowania o system czasu rzeczywistego. Jednym z takich rozwiązań jest darmowy i szeroko wspierany system FreeRTOS [16], który bardzo dobrze nadaje się do takiego zastosowania ze względu na mały rozmiar i prostotę użytkowania. Jądro zawarte jest w 3 plikach C i jest kompilowane razem z kodem mikrokontrolera. System jest konfigurowalny za pomocą 1 pliku (*FreeRTOSConfig.h*). W ten sposób użytkownik decyduje, które komponenty chce wykorzystać.

Podstawą działania FreeRTOS jest scheduler działający priorytetowo. Każde ze zdefiniowanych zadań ma przypisany priorytet, który decyduje o tym, kiedy zostanie ono wykonane. W przypadku gdy w trakcie działania programu gotowość zgłosi zadanie o priorytecie wyższym nastąpi wywłaszczenie aktualnie wykonywanego zadania na rzecz nowego (zgodnie z 3.6). Jeżeli gotowe są dwa zadania o tym samym priorytecie to będą one kolejkowane na zasadzie

podziału czasu. Tak działający scheduler zapewnia terminowość krytycznych zadań. FreeRTOS używa cyklicznego, systemowego przerwania mikrokontrolera (w przypadku STM32 jest to przerwanie SysTick) do wywoływania kodu schedulera, w którym następuje zmiana kontekstu wykonywanego aktualnie zadania (ang. context switching).



Rysunek 3.6: Schemat kolejności wykonywania zadań we FreeRTOS *(Źródło: [16])*

Ponadto FreeRTOS oferuje dodatkowe funkcjonalności takie jak:

- system kolejek (ang. queues) stosowany do komunikacji między zadaniami;
- możliwość użycia programowego licznika czasu (ang. timer);
- mutexy, semafory i semafory binarne do zarządzania dostępem do zasobów współdzielonych i synchronizowania zadań
- mechanizm notyfikacji do synchronizacji wykonywania zadań
- obsługa błędów poprzez zdefiniowanie przerwań obsługujących przypadki, gdy: brak jest pamięci do stworzenia nowego zadania (przerwanie *vApplicationMallocFailedHook()*), odwołanie do niezdefiniowanego adresu (przerwanie *vApplicationStackOverflowHook()*);

4 Specyfikacja systemu

W poniższym rozdziale przedstawiono specyfikację systemu sterowania manipulatorem o 6 stopniach swobody. Opisano założenia, przypadki użycia i wyodrębniono agenty systemu.

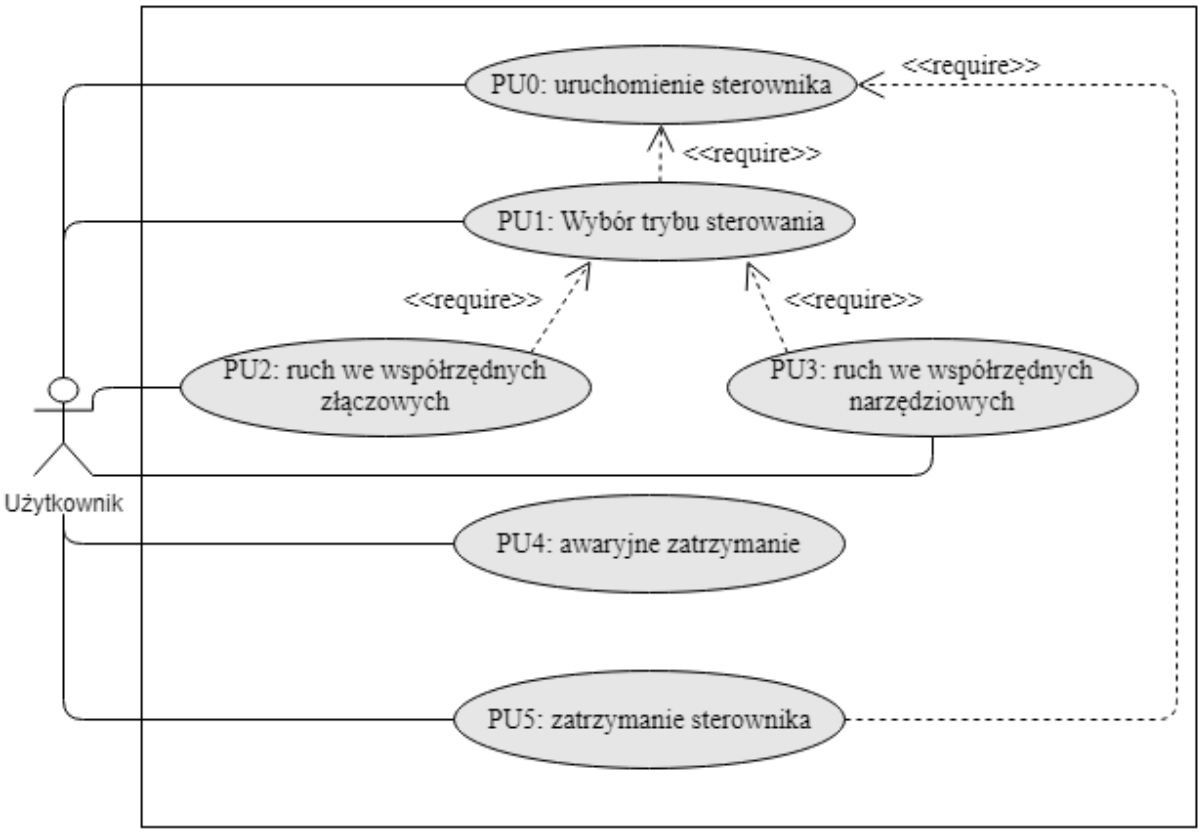
4.1 Założenia i wymagania

4.1.1 Założenia sprzętowe

- Sterowany manipulator ma 6 stopni swobody i wyposażony jest w chwytak dwupalczasty.
- Efektorami robota są 3 silniki prądu stałego, 1 siłownik i 2 serwa Dynamixel RX-64.
- Każdy napęd wyposażony jest w enkodery magnetyczne służące do pomiaru prędkości i położeń wałów napędów.
- Komunikacja z robotem odbywa się poprzez interfejs UART.

4.1.2 Założenia dotyczące systemu

- Aby system działał poprawnie wymagana jest obecność manipulatora oraz komputera PC.
- System pozyskuje dane o ruchu robota na podstawie danych z enkoderów magnetycznych.
- System pozwala na sterowanie pozycją manipulatora we współrzędnych złączowych lub współrzędnych narzędziowych.
- Sterowanie robotem może odbywać się poprzez napisanie programu definiującego ruch wykorzystując przygotowany interfejs programistyczny lub za pomocą podłączonego zadajnika.
- Komendy ruchu mogą być wykonywane przez rzeczywiste urządzenie lub oddzwierciedlone w symulacji dostarczonej ze sterownikiem.
- Do sterowania ręcznego ruchem manipulatora wymagana jest podłączona do komputera PC mysz 3D.
- System pozwala na sterowanie chwytakiem podłączonym do manipulatora poprzez jego proste zamknięcie lub otwarcie.



Rysunek 4.1: Diagram przypadków użycia systemu

4.2 Przypadki użycia systemu

Możliwe interakcje użytkownika z systemem ukazuje diagram przypadków użycia (rysunek 4.1):

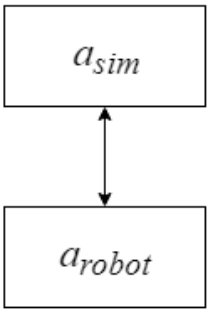
W założeniu jedynym aktorem jest użytkownik, który może korzystać ze sterownika za pomocą zadajnika lub używając interfejsu programistycznego. Wyodrębniono 6 przypadków użycia systemu:

- **PU0** uruchomienie sterownika - włączenie oprogramowania sterującego na komputerze PC oraz włączenie zasilania manipulatora.
- **PU1** wybór trybu sterowania - użytkownik może zdecydować, czy ruch ma być wykonywany w układzie współrzędnych związanym ze złączami, czy z narzędziem.
- **PU2** ruch we współrzędnych złączowych - użytkownik po wybraniu trybu pracy we współrzędnych złączowych może definiować ruch manipulatora zadając sterowanie każdemu z członów.
- **PU3** ruch we współrzędnych narzędziowych - użytkownik po wybraniu trybu pracy we współrzędnych narzędziowych może definiować ruch manipulatora zadając sterowanie końcówce robota.
- **PU4** awaryjne zatrzymanie - użytkownik może w sposób zamierzony lub nie zdecydować o zatrzymaniu pracy robota; manipulator zostanie zatrzymany w przypadku gdy: użytkownik zada taką komendę, nastąpi odłączenie zadajnika w ręcznym trybie pracy, nastąpi przerwanie komunikacji pomiędzy robotem, a komputerem.

- **PU5** zatrzymanie sterownika - wyłączenie oprogramowania sterującego na komputerze PC i/lub odłączenie zasilania manipulatora

4.3 Struktura systemu

System składa się z dwóch agentów: a_{robot} i a_{sim}. Agent a_{robot} jest odpowiedzialny za obsługę peryferiów manipulatora i wykonywanie zadanego ruchu poprzez sterowanie rzeczywistymi napędami oraz otwieranie/zamykanie chwytaka. Stąd agent a_{robot} oczekuje na komendy wydawane przez agenta a_{sim}, który decyduje o trybie pracy i sterowaniu ruchem na podstawie otrzymywanych danych.



Rysunek 4.2: Komunikacja pomiędzy agentami systemu

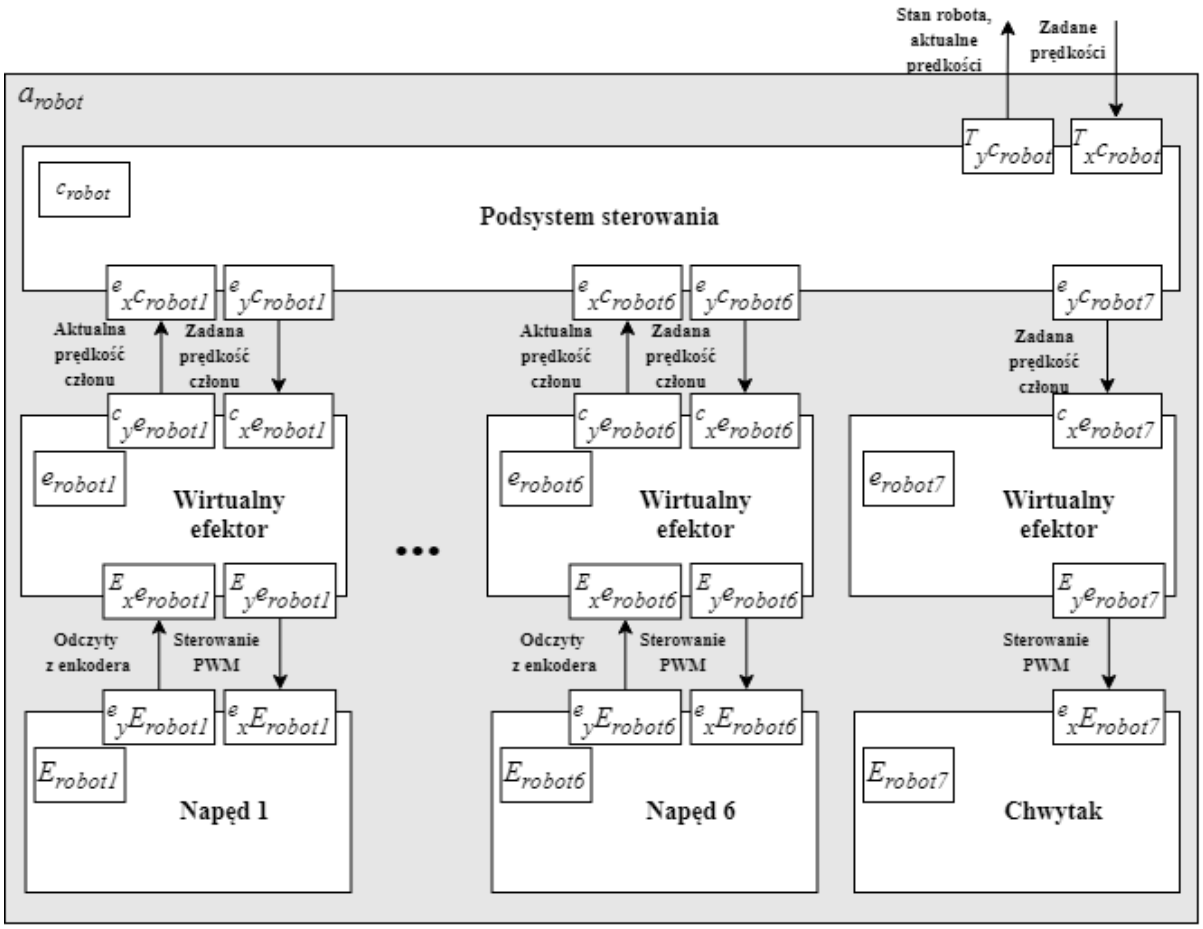
Agent a_{robot} posiada jedynie rzeczywiste efektory w postaci napędów sterujących pszczególnymi członami manipulatora ($E_{robot}1,...,E_{robot}6$), a także chwytakiem ($E_{robot}7$). Jego oprogramowanie składa się z jednego podsystemu sterowania oraz efektorów wirtualnych odpowiadających każdemu z efektorów rzeczywistych.

Agent a_{sim} ma wyspecyfikowany jeden rzeczywisty efektor jakim jest ekran monitora, który wizualizuje dane sterowania i symulacje robota. Ponadto agent zawiera dwa reczywiste receptory: klawiaturę, która służy do zmiany stanu manipulatora, a także zadajnik w postaci myszy 3D do ręcznego sterowania robotem. Oprogramowanie agenta składa się z wirtualnych efektorów i recepotrów do obsługi wspomnianych peryferiów, a także z podsystemu sterowania, który jest sterownikiem ruchu manipulatora, obsługuje symulacje oraz wydaje komendy ruchu do agenta a_{robot} na podstawie instrukcji zadanych przez użytkownika.

Agenty oraz ich podsystemy przesyłają dane za pośrednictwem buforów wejściowych i wyjściowych. Poniżej przedstawiono tabele 4.1, pokazującą komunikację międzyagentową oraz tabele dotyczące komunikacji wewnątrz agentów a_{robot} i a_{sim}.

Tablica 4.1: Komunikacja pomiędzy agentami a_{robot} i a_{sim}

Bufor wyjściowy	Bufor wejściowy	Dane
${}^{T}_{y}c_{sim}$	${}^{T}_{x}c_{robot}$	Komendy sterujące: tryb ruchu i prędkości poszczególnych członów
${}^{T}_{y}c_{robot}$	${}^{T}_{y}c_{sim}$	Stan robota i aktualne odczyty prędkości członów



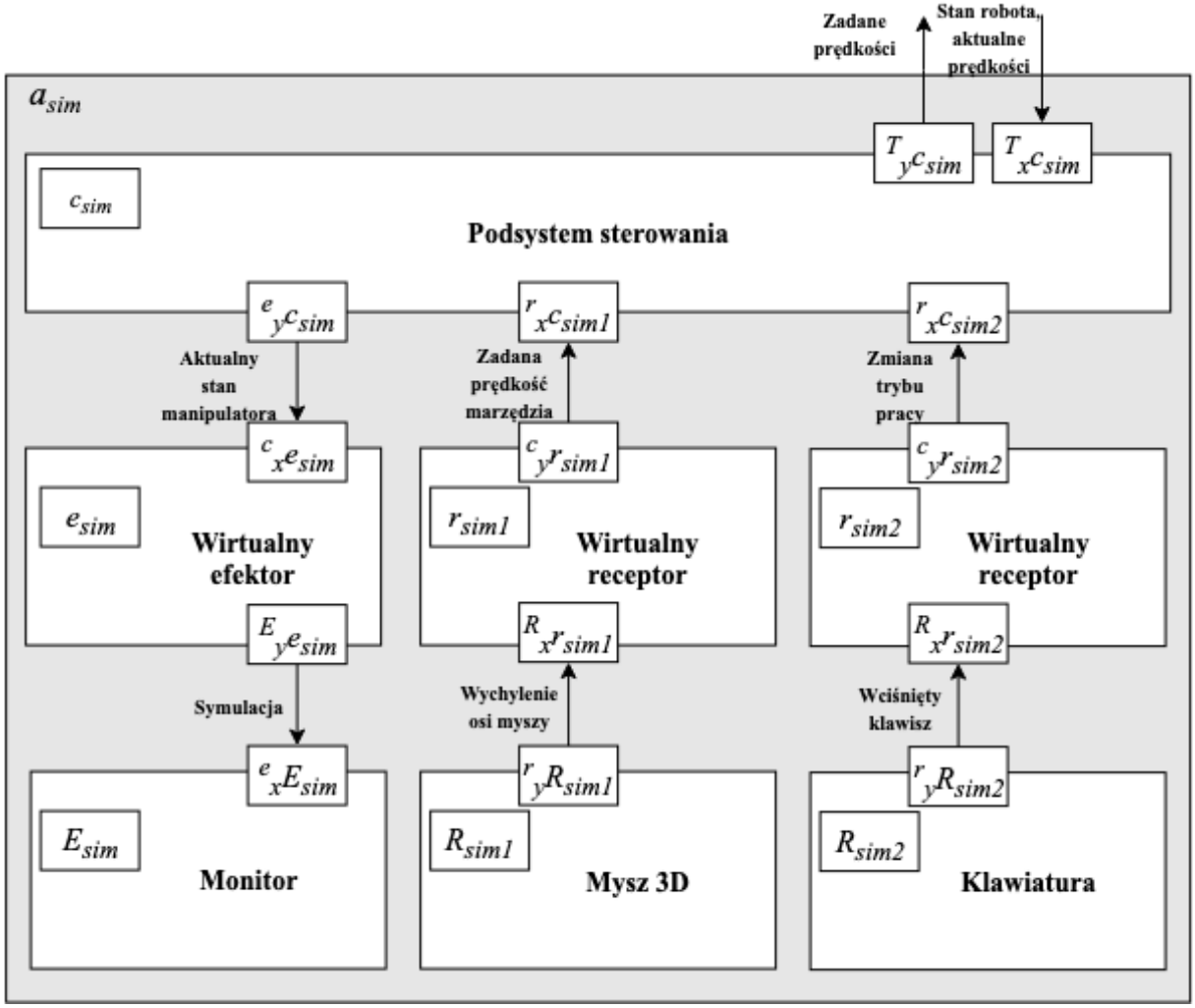
Rysunek 4.3: Schemat agenta a_{robot}

Tablica 4.2: Komunikacja podsystemów agenta a_{robot}

Bufor wyjściowy	Bufor wejściowy	Dane
$^{e}_{y}E_{robot1..6}$	$^{E}_{x}e_{robot1..6}$	Odczyty z enkoderów napędów członów 1..6
$^{E}_{y}e_{robot1..6}$	$^{e}_{x}E_{robot1..6}$	Przeliczony sygnał sterujący napędów członów 1..6
$^{c}_{y}e_{robot1..6}$	$^{e}_{x}c_{robot1..6}$	Przeliczona prędkość członów 1..6 (w [rad/s])
$^{e}_{y}c_{robot1..6}$	$^{c}_{x}e_{robot1..6}$	Zadana prędkość członów 1..6 (w [rad/s])
$^{E}_{y}e_{robot7}$	$^{e}_{x}E_{robot7}$	Sygnał sterujący zamykaniem/otwieraniem chwytaka
$^{e}_{y}c_{robot7}$	$^{c}_{x}e_{robot7}$	Polecenie zamknięcia/otwarcia chwytaka

4.3.1 Podsystem sterowania c_{robot}

W podsystemie sterowania wyodrębniono 6 stanów. Na początku w stanie *INIT* odbywa się inicjalizacja peryferiów robota, synchronizacja napędów i ich zatrzymanie. Jeżeli wszystkie



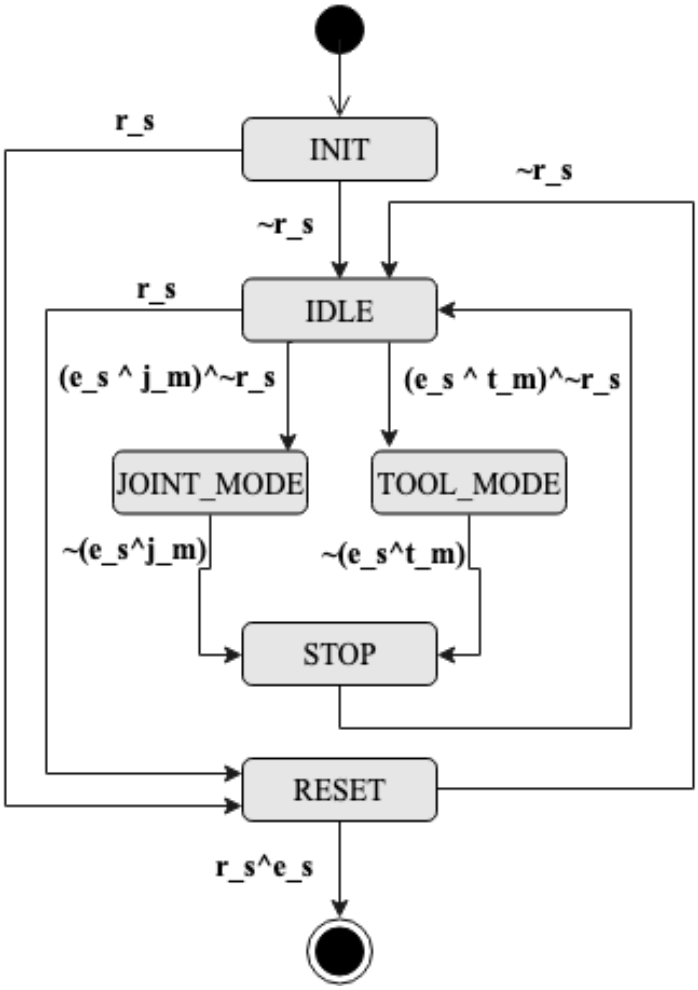
Rysunek 4.4: Schemat agenta a_{sim}

Tablica 4.3: Komunikacja podsystemów agenta a_{sim}

Bufor wyjściowy	Bufor wejściowy	Dane
${}^{E}_{y}e_{sim}$	${}^{e}_{x}E_{sim}$	Symulacja: stan manipulatora
${}^{e}_{y}c_{sim}$	${}^{c}_{x}e_{sim}$	Aktualny stan manipulatora: prędkości/położenia poszczególnych członów
${}^{r}_{y}R_{sim1}$	${}^{R}_{x}r_{sim1}$	Analogowe sygnały wychyleń i rotacji gałki myszy 3D względem każdej z osi
${}^{c}_{y}r_{sim1}$	${}^{r}_{x}c_{sim1}$	Przeliczone, procentowe wartości wychyleń i rotacji myszy 3D
${}^{r}_{y}R_{sim2}$	${}^{R}_{x}r_{sim2}$	Naciśnięty klawisz klawiatury
${}^{c}_{y}r_{sim2}$	${}^{r}_{x}c_{sim2}$	Odczytany klawisz i akcja z nim związana

peryferia zostaną zainicjalizowane poprawnie i nie ma potrzeby ponownego ich uruchamiania lub uśpienia kontrolera (~**r_s**) wtedy następuje przejście do stanu *IDLE*. W tym stanie system

oczekuje na komunikację z agentem a_{sim} jednocześnie utrzymując zadane prędkości zerowe na silnikach (zatrzymanie). W przypadku gdy następuje potrzeba ponownego ustawienia peryferiów (**r_s**) system przechodzi do stanu *RESET*, w którym następuje uśpienie, a w momencie gdy możliwa jest dalsza praca wykonywana jest reinicjalizacja i powrót do stanu *IDLE*. Jeżeli w stanie *IDLE* nawiązana zostanie komunikacja (**e_s**) to w zależności od trybu pracy (**j_m** lub **t_m**) wykonywane jest zadane sterowanie odpowiednio we współrzędnych złączowych (stan *JOINT_MODE*) lub narzędziowych (stan *TOOL_MODE*. Każda zmiana trybu pracy lub utrata komunikacji powoduje zatrzymanie napędów (stan *STOP*) i powrót do stanu *IDLE*.



Rysunek 4.5: Automat skończony podsystemu sterowania c_{robot}

Tablica 4.4: Zachowania podsystemu sterowania c_{robot} i odpowiadające im stany

Stan	Zachowanie	Opis
${}^{c}S_{robot,0}$	${}^{c}B_{robot,0}$	INIT: Inicjalizacja
${}^{c}S_{robot,1}$	${}^{c}B_{robot,1}$	IDLE: Bezczynność
${}^{c}S_{robot,2}$	${}^{c}B_{robot,2}$	JOINT_MODE: Sterowanie w trybie złączowym
${}^{c}S_{robot,3}$	${}^{c}B_{robot,3}$	TOOL_MODE: Sterowanie w trybie narzędziowym
${}^{c}S_{robot,4}$	${}^{c}B_{robot,4}$	STOP: Zatrzymanie
${}^{c}S_{robot,5}$	${}^{c}B_{robot,5}$	RESET: Reinicjalizacja

Poniżej przedstawiono opis zachowań podsystemu sterowania c_{robot}:

Inicjalizacja (${}^{c}B_{robot,0}$)

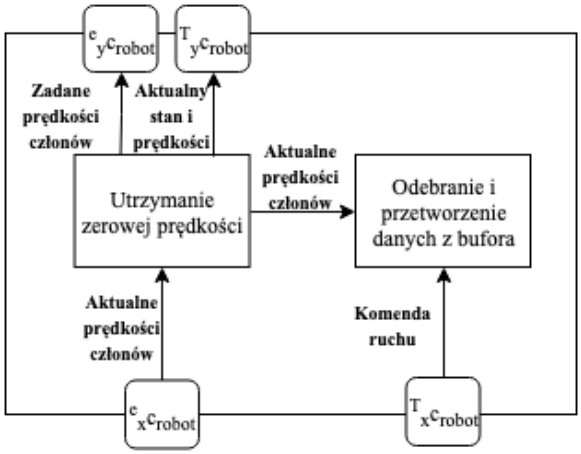
- warunek początkowy: włączenie zasilania
- warunek końcowy: inicjalizacja zakończona
- funkcja przejścia ${}^{c}f_{robot,0}$:

  Po włączeniu urządzenia następuje uruchomienie wymaganych do działania zegarów, peryferiów i oprogramowania sterującego.

Bezczynność (${}^{c}B_{robot,1}$)

- warunek początkowy: inicjalizacja przebiegła pomyślnie lub zatrzymano wszystkie napędy
- warunek końcowy: przesłano do bufora ${}^{T}_{x}c_{robot}$ polecenie ruchu lub wymagany reset
- funkcja przejścia ${}^{c}f_{robot,1}$:

  Zachowanie polega na oczekiwaniu na otrzymanie instrukcji sterowania przesyłanej przez bufor transmisyjny ${}^{T}_{x}c_{robot}$ przy jednoczesnym utrzymywaniu i informowaniu o bieżącej, stałej (zerowej) prędkości wszystkich członów.

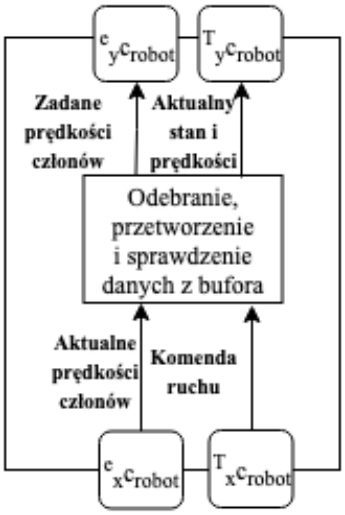


Rysunek 4.6: Funkcja przejścia ${}^{c}f_{robot,1}$

Sterowanie w trybie złączowym (${}^{c}B_{robot,2}$)

- warunek początkowy: odebranie komendy ruchu w trybie złączowym
- warunek końcowy: przerwanie komunikacji lub odebranie komendy ruchu w innym trybie
- funkcja przejścia ${}^{c}f_{robot,2}$:

  Zachowanie polega na przetwarzaniu otrzymywanych danych sterujących przesyłanych przez bufor transmisyjny ${}^{T}_{x}c_{robot}$ i wysyłaniu zadanych prędkości do efektorów oraz informowaniu o aktualnym stanie manipulatora. W trybie złączowym możliwe jest poruszanie jednocześnie tylko jednym członem, więc komenda ruchu sprawdzana jest pod tym kątem.



Rysunek 4.7: Funkcja przejścia ${}^{c}f_{robot,2}$

Sterowanie w trybie narzędziowym (${}^{c}B_{robot,3}$)

- warunek początkowy: odebranie komendy ruchu w trybie narzędziowym
- warunek końcowy: przerwanie komunikacji lub odebranie komendy ruchu w innym trybie
- funkcja przejścia ${}^{c}f_{robot,3}$:

  Zachowanie polega na przetwarzaniu otrzymywanych danych sterujących przesyłanych przez bufor transmisyjny ${}^{T}_{x}c_{robot}$ i wysyłaniu zadanych prędkości do efektorów oraz informowaniu o aktualnym stanie manipulatora.

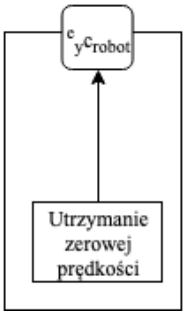
Rysunek 4.8: Funkcja przejścia ${}^{c}f_{robot,3}$

Zatrzymanie (${}^{c}B_{robot,4}$)

- warunek początkowy: przerwanie komunikacji lub odebranie komendy ruchu w innym trybie
- warunek końcowy: -

- funkcja przejścia ${}^{c}f_{robot,4}$:

  Zachowanie polega na bezwzględnym zatrzymaniu wszystkich napędów, czyli wysłaniu do efektorów prędkości zerowych.

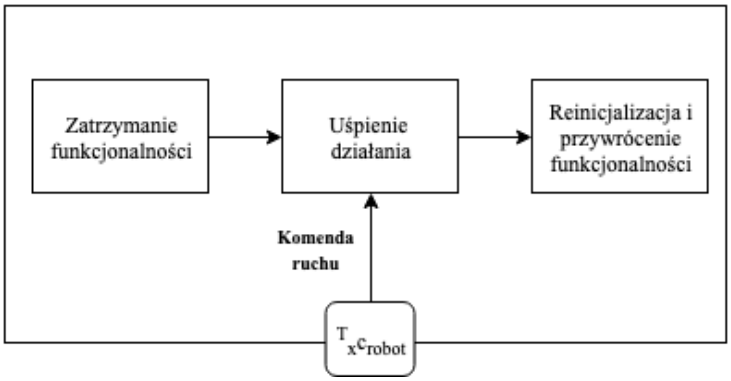


Rysunek 4.9: Funkcja przejścia ${}^{c}f_{robot,4}$

Reset (${}^{c}B_{robot,5}$)

- warunek początkowy: wykryto błąd lub wymuszono reset
- warunek końcowy: wybudzenie i poprawna inicjalizacja
- funkcja przejścia ${}^{c}f_{robot,5}$:

  Zachowanie polega na bezpiecznym wyłączeniu funkcji podsystemu i uśpieniu jego działania w przypadku gdy wykryto błąd lub operacja ta została wymuszona. Podsystem zostaje wybudzony w przypadku uzyskania sygnału do zresetowania funkcjonalności. Następuje wtedy reinicjalizacja i przejście do stanu bezczynności.



Rysunek 4.10: Funkcja przejścia ${}^{c}f_{robot,5}$

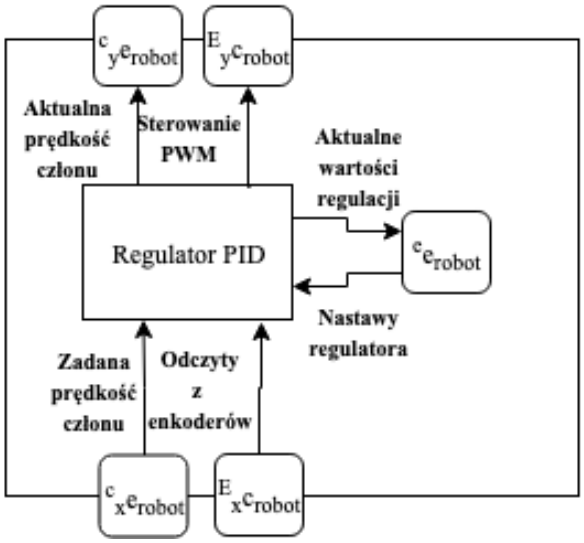
4.3.2 Rzeczywiste efektory $E_{robot1..7}$

Rzeczywistymi efektorami są napędy wyposażone w inkrementalne enkodery magnetyczne. Efektory pozwalają na sterowanie prędkością ruchu każdego z członów manipulatora oraz odczytywanie aktualnej prędkości. Sygnałem sterującym jest PWM (Pulse Width Modulation) dla silników prądu stałego oraz odpowiednia komenda ruchu dla serw Dynamixel RX-64. W przypadku efektora E_{robot7} (napędu sterującego otwieraniem i zamykaniem chwytaka) nie ma możliwości odczytu prędkości ze względu na brak enkodera, ani także sprawdzenia stanu (otwarty/zamknięty). Jest to bardzo duża niedogodność. Z tego względu kontrolowanie ruchu

chwytaka powinno odbywać się poprzez wizualne sprawdzenie stanu w trakcie ruchu i zatrzymanie go, gdy wykona operacje otwarcia/zamknięcia (przy sterowaniu ręcznym) lub poprzez czasowe nadanie odpowiedniego sygnału sterującego. Czas takiego ruchu powinien być zdefiniowany i znany na podstawie doświadczeń.

4.3.3 Wirtualne efektory $e_{robot1..6}$

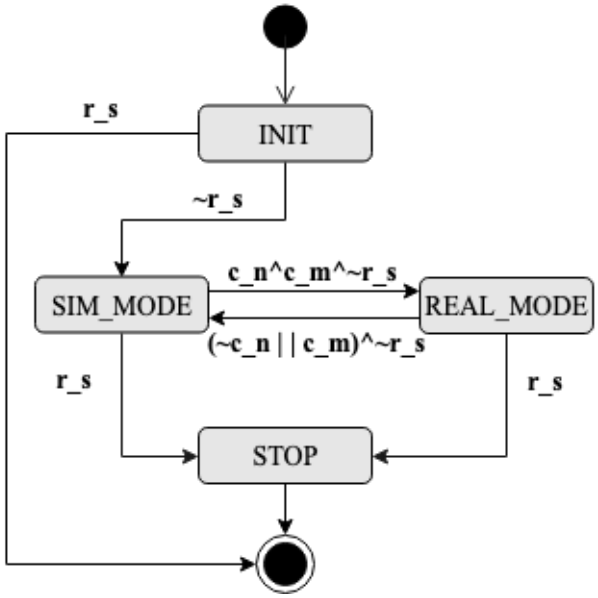
Wirtualne efektory są regulatorami PID prędkości każdego z napędów i działają na zasadzie sprzężenia zwrotnego od wyznaczonej prędkości na podstawie danych z enkoderów.



Rysunek 4.11: Funkcje przejścia ${}^{e}f_{robot,1..6}$

4.3.4 Podsystem sterowania c_{sim}

W podsystemie sterowania wyodrębniono 4 stany. Na początku w stanie *INIT* odbywa się inicjalizacja środowiska symulacyjnego, sterowników urządzeń, a także sterownika ruchu manipulatora. Jeżeli inicjalizacja przebiegła pomyślnie (~**r_s**) następuje przejście do stanu *SIM_MODE*. W przeciwnym przypadku (**r_s**) symulacja zostaje zatrzymana i wyłączona. W stanie *SIM_MODE* zadane sterowanie robota jest przekazywane tylko i wyłącznie do symulacji. Nie występuje wtedy komunikacja z agentem a_{robot} (rzeczywistym manipulatorem). W tym stanie możliwe jest podejrzenie w jaki sposób poruszać się będzie robot przy zadanym sterowaniu. W przypadku gdy wystąpi żądanie zmiany trybu pracy sterownika (**c_m**), możliwa jest komunikacja z robotem (**c_n**), a także nie wystąpił żaden błąd w trakcie działania (~**r_s**), następuje przejście do stanu *REAL_MODE*. Wtedy komendy sterujące wysyłane są do robota i odbierane są dane o jego stanie i prędkościach poszczególnych członów, na podstawie których operuje sterownik ruchu. Ponadto ruch manipulatora jest odzwierciedlany i wizualizowany w symulacji. Przejście spowrotem do stanu *SIM_MODE* odbywa się gdy utracone jest połączenie (~**c_n**) lub wystąpi żądanie zmiany trybu pracy sterownika (**c_m**). W przypadku gdy wystąpi błąd oprogramowania lub wymuszenie zakończenia działania (**r_s**) następuje przejście do stanu *STOP*, w którym wysyłana jest komenda zatrzymania robota, a następnie wyłączana jest symulacja i sterownik ruchu.



Rysunek 4.12: Automat skończony podsystemu sterowania c_{sim}

Tablica 4.5: Zachowania podsystemu sterowania c_{sim} i odpowiadające im stany

Stan	Zachowanie	Opis
${}^{c}S_{sim,0}$	${}^{c}B_{sim,0}$	INIT: Inicjalizacja
${}^{c}S_{sim,1}$	${}^{c}B_{sim,1}$	SIM_MODE: Praca w symulacji
${}^{c}S_{sim,2}$	${}^{c}B_{sim,2}$	REAL_MODE: Praca na rzeczywistym robocie
${}^{c}S_{sim,3}$	${}^{c}B_{sim,3}$	STOP_MODE: Zatrzymanie pracy

Inicjalizacja (${}^{c}B_{sim,0}$)

- warunek początkowy: Uruchomiono symulację
- warunek końcowy: inicjalizacja zakończona
- funkcja przejścia ${}^{c}f_{sim,0}$:

  Po włączeniu oprogramowania następuje inicjalizacja środowiska symulacyjnego, sterownika robota i sterowników urządzeń.

Praca w symulacji (${}^{c}B_{sim,1}$)

- warunek początkowy: poprawnie zainicjalizowano oprogramowanie lub zmieniono tryb pracy
- warunek końcowy: zmiana trybu pracy, wykrycie błędu lub wyłączenie oprogramowania
- funkcja przejścia ${}^{c}f_{sim,1}$:

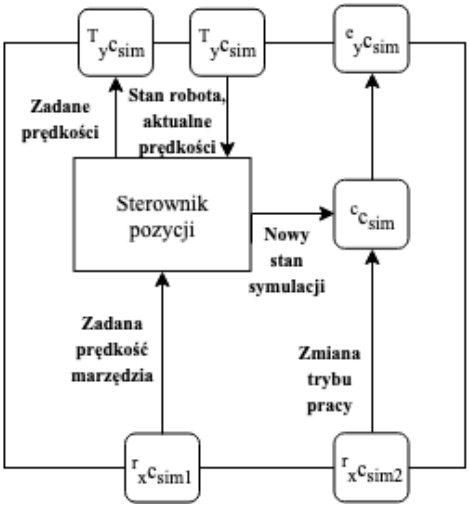
  Zachowanie polega na wykonywaniu sterowania na podstawie wirtualnego modelu robota. W tym trybie pracy nie występuje komunikacja pomiędzy agentami, a wszystkie komendy ruchu są przedstawiane w symulacji.

Rysunek 4.13: Funkcja przejścia ${}^{c}f_{sim,1}$

Praca na rzeczywistym robocie (${}^{c}B_{sim,2}$)

- warunek początkowy: zmieniono tryb pracy
- warunek końcowy: zmiana trybu pracy, wykrycie błędu lub wyłączenie oprogramowania
- funkcja przejścia ${}^{c}f_{sim,2}$:

  Zachowanie polega na wykonywaniu sterowania na podstawie danych o stanie robota otrzymywanych przez bufor komunikacyjny ${}^{T}_{x}c_{sim}$ wysyłanych przez rzeczywisty manipulator. W tym trybie pracy występuje komunikacja pomiędzy agentami, a wszystkie komendy ruchu są wysyłane do manipulatora poprzez bufor komunikacyjny ${}^{T}_{y}c_{sim}$. Symulacja staje się wtedy wizualizacją ruchów prawdziwego robota.

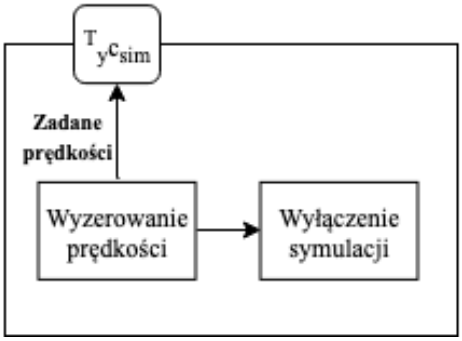


Rysunek 4.14: Funkcja przejścia ${}^{c}f_{sim,2}$

Zatrzymanie pracy (${}^{c}B_{sim,3}$)

- warunek początkowy: wykryto błąd lub wymuszono zatrzymanie pracy
- warunek końcowy: -
- funkcja przejścia ${}^{c}f_{sim,3}$:

  Zachowanie polega na bezpiecznym wyłączeniu funkcji podsystemu, zatrzymaniu manipulatora i przerwaniu z nim komunikacji oraz wyłączeniu symulacji i sterownika urządzenia.



Rysunek 4.15: Funkcja przejścia ${}^{c}f_{sim,3}$

4.3.5 Rzeczywisty receptor R_{sim1}

Rzeczywistym receptorem R_{sim1} jest zadajnik w postaci myszy 3D. Służy do ręcznego sterowania ruchem robota. Wychylenia względem każdej z osi myszy odpowiadają zadawaniu niezerowych prędkości we współrzędnych związanych z narzędziem.

4.3.6 Wirtualny receptor r_{sim1}

Wirtualnym receptorem r_{sim1} jest sterownik myszy 3D, dzięki któremu mechaniczne wychylenia gałki urządzenia są konwertowane na wartości liczbowe (od 0 do 1).

4.3.7 Rzeczywisty receptor R_{sim2}

Rzeczywistym receptorem R_{sim2} jest klawiatura, która w stworzonym systemie służy jedynie do zmiany trybu sterownika z trybu symulacji do trybu pracy na rzeczywistym robocie oraz z działania we współrzędnych złączowych do współrzędnych narzędziowych.

4.3.8 Wirtualny receptor r_{sim2}

Zadaniem wirtualnego receptora r_{sim2} jest wykrywanie naciśnięcia zdefiniowanych przycisków klawiatury służących do zmiany trybu i przesyłaniu informacji do podsystemu sterowania w celu rozpoczęcia pracy w innym trybie.

5 Implementacja systemu

Niniejszy rozdział przedstawia implementację systemu sterowania przedstawionego w rozdziale 4. Opisano w nim oparte na strukturze agentów a_{robot} i a_{sim} oprogramowanie odpowiednio kontrolera robota oraz sterownika ruchu i symulacji.

5.1 Oprogramowanie kontrolera

Implementacja oprogramowania mikrokontrolera STM32F407VET do sterowania robotem wykorzystuje system czasu rzeczywistego FreeRTOS oraz biblioteki STDPeriph opisane w rozdziale 3. Program napisany został w języku C++ i służy do komunikacji ze sterownikiem, kontrolowaniu ruchu napędów oraz monitorowaniu stanu manipulatora. Do stworzonych w ramach pracy komponentów zaliczają się:

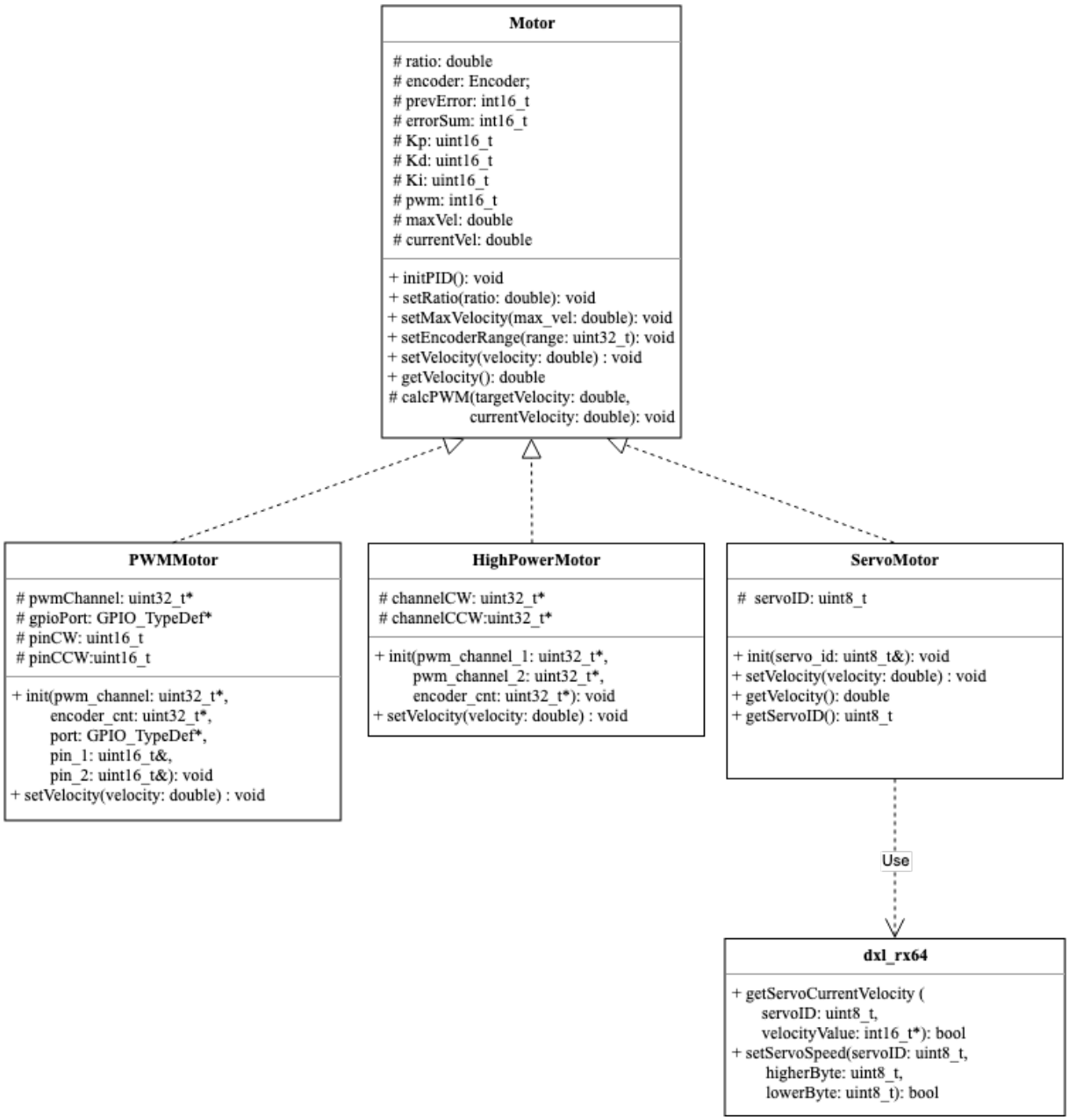
- podsystem sterowania c_{robot}
- wirtualne efektory e_{robot}

5.1.1 Wirtualne efektory $e_{robot1..6}$

Działanie efektorów polega na sterowaniu prędkością napędów na podstawie odczytów aktualnej prędkości z enkoderów oraz zadanej przez sterownik. W tym celu użyty został regulator typu PID. Ze względu na różnorodność napędów użytych w konstrukcji manipulatora występują różnice w sposobie ich sterowania. Silniki DC małej mocy (napędy 1, 3 i 4 stopnia swobody) sterowane są za pomocą jednego sygnału PWM i dwóch sygnałów decydujących o kierunku obrotu wału silnika. Silnik DC dużej mocy (napęd 2 stopnia swobody) jest sterowany za pomocą dwóch sygnałów PWM. Sterowanie serwami odpowiadającymi za 5 i 6 stopień swobody odbywa się poprzez wysyłanie i odbieranie komunikatów używając interfejs RS-485 zgodnie ze zdefiniowanym protokołem komunikacji Protocol 1.0 [15]. Stąd napisano trzy oddzielne klasy odpowiadające za sterowanie danym rodzajem napędów:

- *PWMMotor* - odpowiada za sterowanie silnikami DC małej mocy,
- *HighPowerMotor* - odpowiada za sterowanie silnikami DC dużej mocy,
- *ServoMotor* - odpowiada za sterowanie serwami Dynamixel RX-64.

Każda klasa dziedziczy po abstrakcyjnej klasie bazowej *Motor*, która stanowi wspólny interfejs do inicjalizacji, odczytywania i zadawania prędkości każdemu rodzajowi napędów.



Rysunek 5.1: Diagram klas sterujących napędami

Klasy *PWMMotor* i *HighPowerMotor* przechowują i wykorzystują dane o sterowaniu:

- zdefiniowane nastawy regulatora: *K_p*, *K_d*, *K_i*,
- dane o napędzie: maksymalna prędkość (*maxVel*) i przełożenie (*ratio*),
- odczyty enkoderów w postaci struktury zdefiniowanej jako *Encoder*:

```
typedef struct Encoder
{
        volatile uint32_t * currentCount; // aktualne wskazanie
        uint32_t previousCount; // poprzednie wskazanie
        uint32_t range; // zakres licznika enkodera
}Encoder;
```

- stan regulatora: poprzednia wartość błędu (*prevError*) i suma błędów sterowania (*errorSum*).

Na podstawie tak zgromadzonych danych zadana wartość prędkości przeliczana jest na wartość wypełnienia PWM napędu stosując funkcję *calcPWM()*, w której wykonywana jest regulacja PID według wzoru:

$$error = targetVelocity - currentVelocity \tag{5.1}$$

$$pwm+ = K\_p * error + K\_d * (error - prevError) + K\_i * errorSum \tag{5.2}$$

W przypadku serwa (klasa *ServoMotor*) odczytywanie i zadawanie prędkości odbywa się poprzez wysyłanie odpowiednich komend i odbieranie wiadomości zwrotnych z serwa wykorzystując protokół komunikacji Protocol 1.0 [15]. Regulacją prędkości zajmuje się oprogramowanie serwa dostarczone przez producenta. Stąd klasa *ServoMotor* nie implementuje regulatora PID, a jedynie interfejs do komunikacji szeregowej.

Odbierane i wysyłane komendy mają postać ramek danych o ściśle zdefiniowanej strukturze:

- pierwsze dwa bajty ramki muszą mieć wartość szesnastkową $0xFF$,
- kolejny bajt informuje o ID (od 0 do 253, 254 to ID rozgłoszeniowe) serwa, do którego wysyłana jest komenda,
- po ID wysyłana jest długość ramki, czyli ilość przesyłanych w ramce bajtów (ilość parametrów + 2),
- następnie występuje bajt idefiniujący rodzaj instrukcji (tutaj używane instrukcje WRITE ($0x03$) i READ ($0x02$)),
- kolejne bajty to parametry instrukcji,
- ostatecznie wysyłany jest bajt sumy kontrolnej, której wzór przedstawia się następująco (to operacja negacji bitowej):

$$suma\_kontrolna = (ID + dugo + parametr\_1 + ... + parametr\_N) \tag{5.3}$$

Tablica 5.1: Ramka danych wysyłana do serwa Dynamixel *(źródło: [15])*

$0xFF$	$0xFF$	ID	Długość	Instrukcja	Parametr_1	...	Parametr_N	Suma kontrolna

Po odebraniu przez serwo komendy odsyła ono ramkę powrotną, która różni się od wyżej wskazanej tylko tym, że bajt 4 wskazuje czy nie wystąpił żaden błąd zamiast długości:

Tablica 5.2: Ramka danych odsyłana od serwa Dynamixel *(źródło: [15])*

$0xFF$	$0xFF$	ID	Kod błędu	Instrukcja	Parametr_1	...	Parametr_N	Suma kontrolna

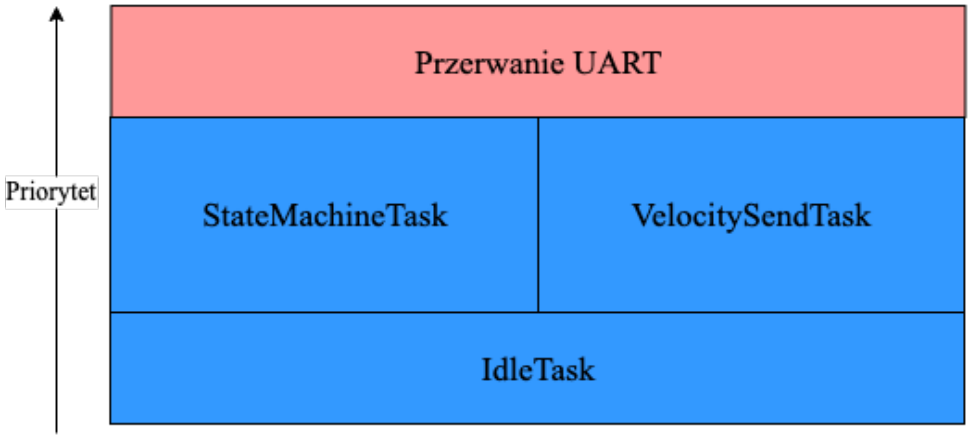
Klasa ServoMotor do implementacji funkcji *getVelocity()* i *setVelocity()* wykorzystuje autorską bibliotekę **dxl_rx64** napisaną w języku C, która stanowi wysokopoziomowy interfejs do wysyłania i odbioru ramek danych zdefiniowanych przez Protocol 1.0. Biblioteka wykorzystuje bufor cykliczny do odbierania bajtów i akumulowania ich do ramki danych.

5.1.2 Podsystem sterowania c_{robot}

Realizacja podsystemu sterowania c_{robot} wykorzystuje udogodnienia systemu czasu rzeczywistego FreeRTOS takie jak zadania (ang. tasks) i kolejki (ang. queues). Komunikacja ze sterownikiem ruchu (agentem a_{sim}) odbywa się poprzez interfejs UART. Działanie podzielono na 4 zadania:

- sterowanie napędami na podstawie odbieranych z bufora komunikacyjnego komend ruchu oraz odczytanych z napędów prędkości (zadanie *StateMachineTask*),
- wysyłanie ramek danych z prędkościami poszczególnych członów i aktualnym stanem poprzez bufor komunikacyjny (zadanie *VelocitySendTask*),
- odbieranie komend ruchu z bufora komunikacyjnego (zadanie to wykonywane jest w przerwaniu od interfejsu komunikacyjnego UART),
- zadanie bezczynności uruchamiane w momencie uśpienia zadania sterowania w przypadku wykrycia niepoprawnego działania i zatrzymania napędów (zadanie *IdleTask*).

Priorytety poszczególnych zadań zostały ustalone, aby sterowanie napędami zawsze wykonywane było na czas (zakładane maksymalne opóźnienie pomiędzy dwoma komendami ruchu wynosi 20 ms). Najniższy możliwy priorytet ma zadanie *IdleTask*, które służy głównie oszczędzaniu energii. Priorytety wyższe posiadają zadania sterowania napędami (*StateMachineTask*) oraz wysyłania danych o prędkościach (*VelocitySendTask*). Najwyższy priorytet ma przerwanie odbioru danych z interfejsu UART.

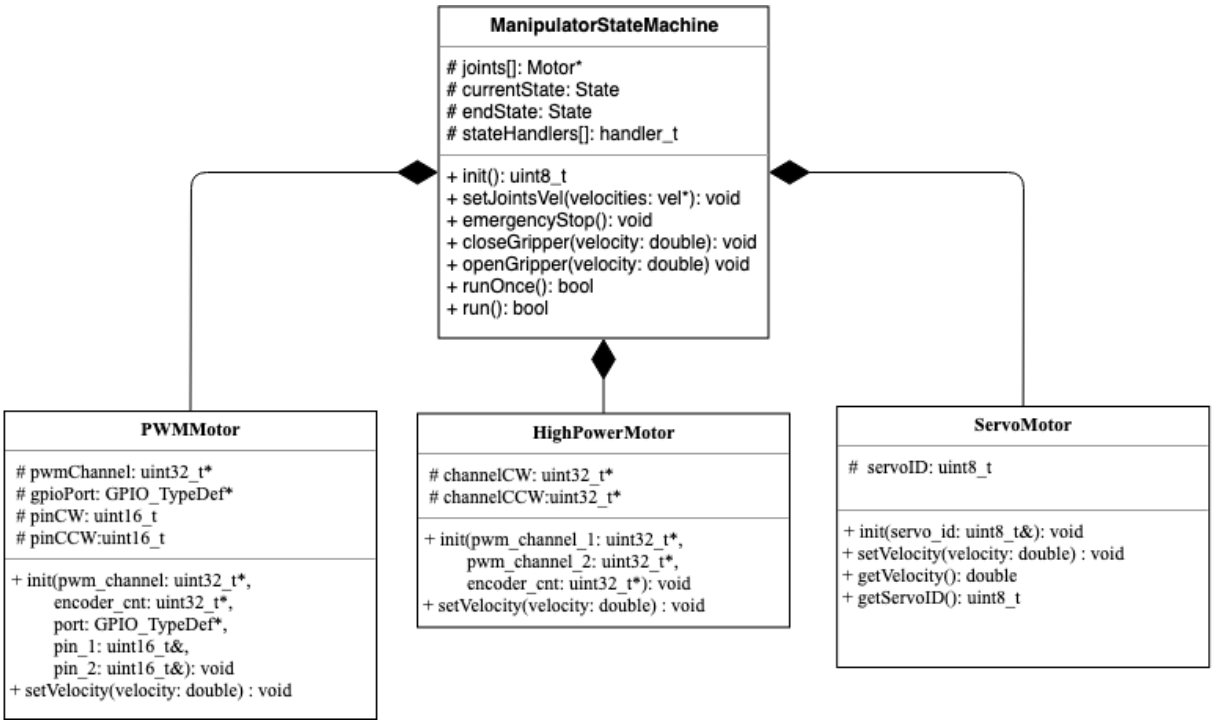


Rysunek 5.2: Priorytet wykonywania zadań podsystemu

Zadanie *StateMachineTask* oparte jest na automacie skończonym przedstawionym w rozdziale 4. Implementacją podsystemu jest klasa *ManipulatorStateMachine* napisana wykorzystując wzorzec projektowy maszyny stanu (ang. state machine).

Klasa *ManipulatorStateMachine* przechowuje tablicę wskaźników do obiektów typu *Motor* służących sterowaniu prędkością silników manipulatora oraz interfejs do zadawania prędkości każdemu z napędów (w postaci funkcji *setJointsVel()*), a także zatrzymywaniu ich w nieoczekiwanych sytuacjach (funkcja *emergencyStop()*).

Jak już wspomniano klasa realizuje automat skończony podsystemu sterowania poprzez przechowywanie i wykorzystywanie funkcji *handler_t*. Typ ten jest zdefiniowany jako:



Rysunek 5.3: Diagram klasy implementującej automat skończony podsystemu sterowania c_{robot}

```
typedef State (*handler_t)(void*);
```

, natomiast typ *State* jako:

```
enum State
{
        JOINT_STATE = 0,
        TOOL_STATE,
        IDLE_STATE,
        INIT_STATE,
        RESET_STATE,
};
```

Typ *handler_t* definiuje więc wskaźnik na funkcję przyjmującą wskaźnik na adres pamięci obiektu *ManipulatorStateMachine*, a zwraca jeden ze stanów *State*, do którego ma nastąpić przejście. Każdemu ze stanów przypisana jest jedna z procedur obsługi (*handler_t*) definiująca zachowanie w aktualnym stanie. Poniżej pokazano listę stanów, przypisane do nich procedury obsługi oraz opis ich działania (tablica 10).

Działanie klasy *ManipulatorStateMachine* obsługiwane jest na podstawie wiedzy o stanie aktualnym (*currentState*), stanie końcowym (*endState*) i przejściach pomiędzy stanami. Na początku następuje inicjalizacja klasy poprzez wypełnienie tablicy procedur (*stateHandlers[]*) oraz ustawienie stanu początkowego jako aktualnego i wybranie stanu końcowego. Następnie wywoływana jest funkcja *run()*, która uruchamia pierwszą i kolejne procedury obsługi działając w pętli. Następny stan wybierany jest na podstawie wartości zwracanej przez procedurę aktualnego. Wykonywane przez klasę zadanie zostaje zatrzymane w momencie wykonania stanu oznaczonego jako końcowy.

Tablica 5.3: Procedury obsługi stanów automatu skończonego manipulatora

Stan	**Procedura obsługi**	**Opis działania**
INIT_STATE	*runInit()*	Jest to stan początkowy, w którym inicjalizowane są wszystkie napędy oraz regulatory. Po udanej konfiguracji następuje przejście do stanu IDLE_STATE.
IDLE_STATE	*runIdle()*	Obsługa stanu polega na próbie odbioru danych z bufora komunikacyjnego odbierającego komendy ruchu z interfejsu UART. Bufor odbiorczy stanowi kolejka systemu FreeRTOS. Jeżeli odebrana komenda odpowiada za wykonywanie w jednym z trybów: złączowym (JOINT_STATE) lub narzędziowym (TOOL_STATE), to następuje przejście do odpowiadających im stanów. W przeciwnym wypadku oczekiwanie na komendę w stanie IDLE_STATE trwa.
RESET_STATE	*runReset()*	Przejście do tego stanu odbywa się w momencie gdy nie powiedzie się inicjalizacja w stanie INIT_STATE lub wystąpi problem z komunikacją (komendy nie będą przychodzić przez długi czas). W tym stanie następuje bezwzględne zatrzymanie napędów i zatrzymanie zadania systemu związanego z obsługą stanów na rzecz uśpienia mikrokontrolera. W momencie gdy mikrokontroler zostanie wybudzony (przy odebraniu jakiejkolwiek komendy) następuje ponowienie inicjalizacji napędów i regulatorów. W przypadku niepowodzenia następuje zakończenie zadania obsługującego poszczególne stany i reset mikrokontrolera. Jeżeli reinicjalizacja się uda następuje przejście do stanu otrzymanego w odebranej komendzie.
JOINT_STATE	*runJoint()*	W przypadku odebrania w stanie IDLE_STATE lub RESET_STATE komendy ruchu w trybie narzędziowym, wykonywana jest procedura obsługi stanu TOOL_STATE. Polega ona na cyklicznym odbieraniu oczekującym (z czasem oczekiwania 20 ms) komend ruchu w tym trybie i nadawaniu zadanej prędkości napędów z zastrzeżeniem, że prędkość niezerową może mieć w tym stanie tylko jeden człon (napęd). Jeżeli żaden komunikat nie dojdzie w okresie oczekiwania następuje przejście do stanu IDLE_STATE po uwczesnym zatrzymaniu napędów. W przypadku wystąpienia problemów w komunikacji z serwami Dynamixel następuje przejście do stanu RESET_STATE (czas oczekiwania także 20 ms).
TOOL_STATE	*runTool()*	Ten stan obsługiwany jest tak samo jak JOINT_STATE z tym wyjątkiem, że dozwala na sterowanie kilkoma napędami naraz.

Zadanie *VelocitySendTask* polega na wysyłaniu poprzez bufor komunikacyjny (interfejs UART) ramek danych zawierających dane o pracy manipulatora. Wykorzystując interfejs klasy *ManipulatorStateMachine* odczytywane są prędkości wszystkich członów oraz aktualny stan. Komunikacja ze sterownikiem jest wzorowana na protokole serw Dynamixel. Ramka składa się z dwóch pól nagłówka (o wartości $0xFF$), bajtu definiującego stan pracy, pola długości nagłówka. Po nich następuje następuje wysyłanie bajtów danych (wartości prędkości) i sumy kontrolnej (SK) obliczanej zgodnie ze wzorem (5.3).

Tablica 5.4: Ramka danych komunikacji manipulator- sterownik

$0xFF$	$0xFF$	Stan	Dł.	Pr_1 LSB	Pr_1 MSB	...	Pr_N LSB	Pr_N MSB	SK

Wartości prędkości są dzielone na 2 bajty i wysyłane w kolejności młodszy bajt (LSB), starszy bajt (MSB).

Zadanie odbierania komend ruchu odbywa się w przerwaniu, które występuje w momencie odebrania bajtu danych poprzez interfejs komunikacyjny UART. Bajty zapisywane są w buforze cyklicznym i w momencie otrzymania całej ramki danych (przedstawionej wyżej) sprawdzana jest jej poprawność (suma kontrolna). Jeżeli ramka jest poprawna, wybierane są z niej dane w postaci trybu pracy, ilości przesyłanych bajtów danych (długości) i wszystkich zadanych prędkości poszczególnych napędów. Następnie dane te są zapisywane w postaci struktury *ManipulatorMsg*:

```
typedef struct ManipulatorMsg
{
    State type;
    uint8_t length;
    uint8_t params[MANIPULATOR_MAX_PARAMS];
    uint8_t checksum;
} ManipulatorMsg;
```

W momencie uzyskania komendy w postaci powyższej struktury należy ją przesłać do zadania *StateMachineTask*. Komunikacja pomiędzy zadaniami odbywa się poprzez kolejki systemu FreeRTOS. W trakcie działania zadanie *StateMachineTask* odpytuje kolejke o nową komendę typu *ManipulatorMsg*. Oczekiwanie na nią trwa 20 ms i jeżeli w tym czasie nie przyszła nowa komenda oznacza to, że komunikacja została przerwana.

Zadanie *IdleTask* posiada najniższy priorytet i wykonywane jest tylko jeżeli pozostałe zadania zostaną przerwane. Jedynym celem zadania jest wprowadzenie mikrokontrolera w stan niższego poboru mocy (uśpienie). W tym celu wykorzystywane jest zdefiniowane w systemie FreeRTOS zadanie *vApplicationIdleTask()*, które wywołuje instrukcje *__WFI()*. Wybudzenie mikrokontrolera w tym stanie może nastąpić za pomocą przerwania (powrót do działania następuje w przypadku wystąpienia komunikacji poprzez interfejs UART).

5.2 Sterownik i symulacja

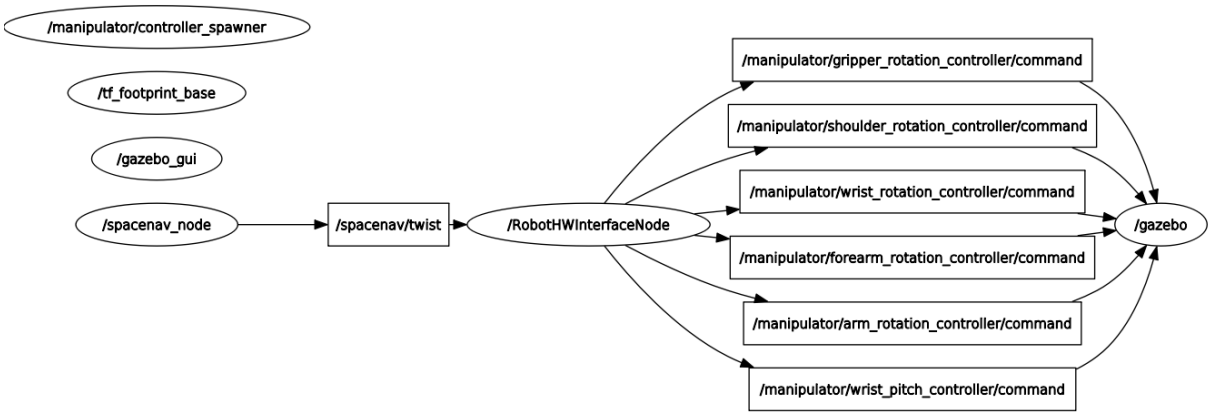
Implementacja oprogramowania sterownika ruchem i symulacji wykorzystuje pakiet ROS, biblioteki dodatkowe ros_control i spacenav oraz symulator Gazebo. Do stworzonych w ramach pracy komponentów zaliczają się:

- podsystem sterowania c_{sim}
- wirtualny receptor r_{sim1}
- wirtualny receptor r_{sim2}

Implementacja agenta a_{sim} wykorzystuje węzły ROS do komunikacji międzyprocesowej. Głównymi węzłami są:

- /gazebo - węzeł odpowiedzialny za proces symulacji,
- /RobotHWInterfaceNode - węzeł odpowiedzialny za sterowanie ruchem manipulatora,
- /spacenav_node - sterownik myszy 3D wysyłający dane o wychyleniu gałki myszy.

Poniższy diagram przedstawia jak przebiega komunikacja pomiędzy węzłami, na którym węzły oznaczone są jako elipsy, a tematy jako prostokąty:



Rysunek 5.4: Diagram komunikacji pomiędzy węzłami sterownika manipulatora

Węzeł sterujący publikuje komendy ruchu w tematach typu */command* dla każdego z 6 członów manipulatora. Dane te odczytywane są przez symulator i na podstawie modelu fizycznego wizualizowane w symulacji. Każdy z przegubów ma przypisany do siebie sterownik ruchu. W trybie ręcznym komendy ruchu w postaci prędkości narzędzia lub każdego z przegubów z osobna (w zależności od konfiguracji ruchu) są przekazywane do węzła sterującego za pomocą myszy 3D. Dane wysyłane są poprzez temat */spacenav/twist*, który zawiera aktualne wychylenia/skręcenia gałki urządzenia w 6 osiach. W przypadku sterowania w trybie automatycznym możliwe jest zadawanie prędkości lub pozycji i orientacji końcówki robota w

przestrzeni poprzez publikowanie wiadomości na odpowiednie tematy węzła */RobotHWInterfaceNode*. Ponadto w przypadku podłączenia interfejsu komunikacyjnego robota możliwe jest bezpośrednie sterowanie ruchem rzeczywistego manipulatora. W momencie gdy możliwa jest komunikacja sterownik- robot sterowanie ruchem odbywa się na podstawie danych przesyłanych z robota, a nie z symulacji. W ten sposób możliwe jest testowanie ruchu przed wykonywaniem go w rzeczywistości.

Sterownik robota (węzeł *RobotHWInterfaceNode*) zaimplementowano wykorzystując narzędzia dostarczone przez bibliotekę ros_control opisaną w rozdziale 3. Architektura rozwiązania bazuje na diagramie 3.5. Przy starcie programu uruchamiany jest *Controller Manager*, który odpowiada za załadowanie i przypisanie każdemu z przegubów sterownika ruchu (oznaczony na diagramie jako *Controller*). Rodzaje i parametry (np. nastawy PID) sterowników są definiowane w pliku typu *yaml* i ładowane przy starcie. Wtedy za pomocą węzła typu *spawner* (na diagramie 5.4 oznaczony jako */manipulator/controller_spawner*) następuje ich uruchomienie. Węzeł *RobotHWInterfaceNode* implementuje warstwy oprogramowania oznaczone na rysunku 3.5 jako *Hardware Resource Interface Layer*, a także *hardware_interface::RobotHW*. Pierwsza z nich stanowi interfejs do pobierania danych z kontrolerów o zadanych komendach ruchu (interfejs typu *command*), a także o stanie (prędkościach, pozycjach) poszczególnych przegubów (interfejs typu *state*). Druga jest pośredniczącym interfejsem sprzętowym. Klasa *hardware_interface::RobotHW* jest abstrakcyjna i stanowi szablon komunikacji pomiędzy sterownikiem, a robotem. W celu zaimplementowania interfejsu sprzętowego należy zdefiniować zadeklarowane funkcje *write()*, *read()* i *update()*, które zajmują się odpowiednio: wysyłaniem zadanego sterowania do robota, odczytywaniem danych z napędów (np. prędkości) oraz aktualizowaniem stanu sterowników.

Poniżej przedstawiono jak został zrealizowany węzeł *RobotHWInterfaceNode* wykorzystując bibliotekę *ros_control*:

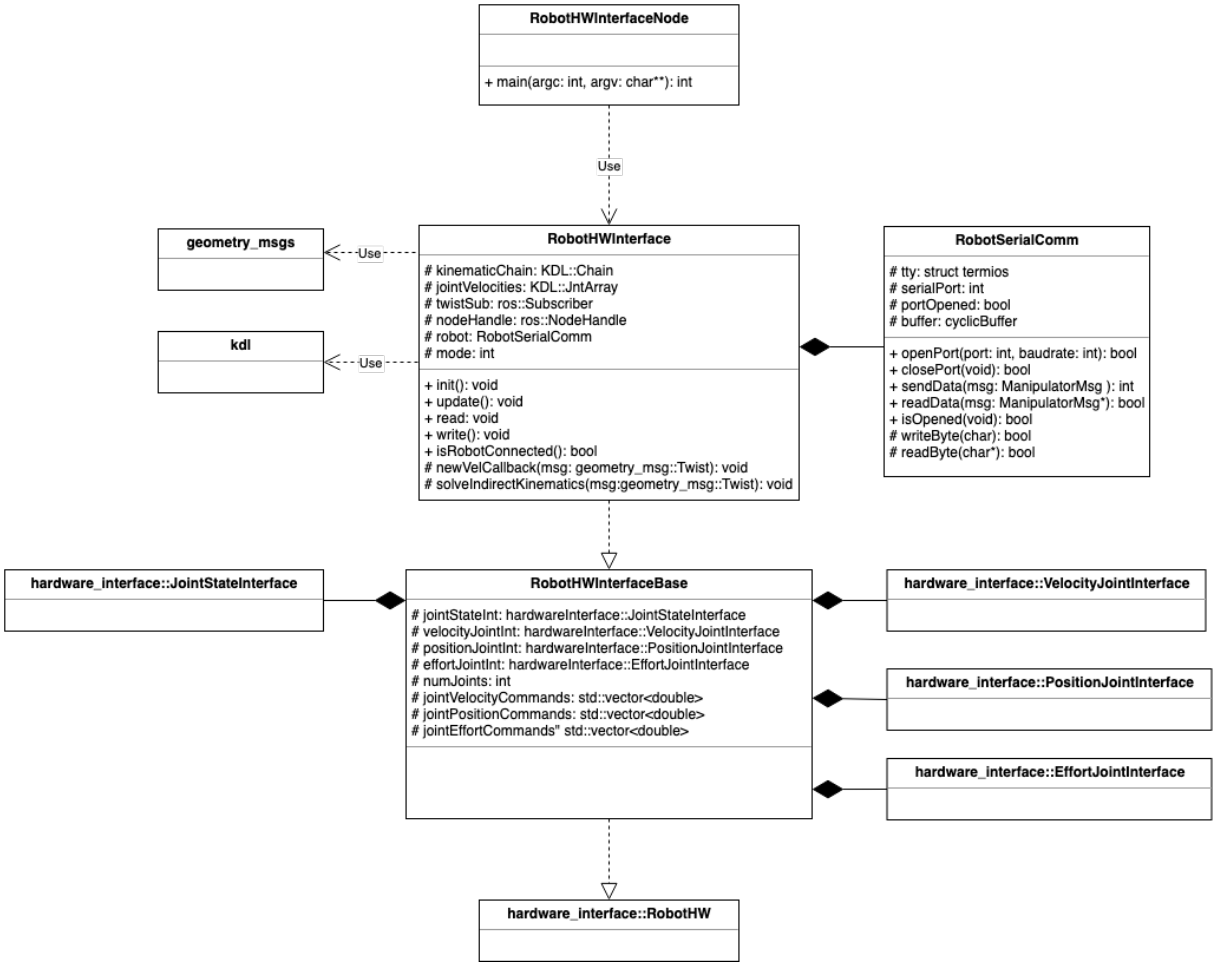
Klasa *RobotHWInterfaceBase* dziedziczy po interfejsie *RobotHW* oraz zawiera interfejsy stanów:

- *jointStateInt*,
- *velocityJointInt*,
- *positionJointInt*,
- *effortJointInt*.

Pozwalają one na odczytywanie aktualnych prędkości, pozycji, mocy z każdego sterownika w przypadku pracy w symulacji lub prędkości w przypadku pracy z rzeczywistym robotem. Ponadto klasa posiada interfejsy umożliwiające zadawaniu określonego sterowania względem prędkości, pozycji lub mocy:

- *jointVelocityCommands*,
- *jointPositionCommands*,
- *jointEffortCommands*.

Klasa ta stanowi bazę dla docelowego interfejsu sprzętowego *RobotHWInterface*, który implementuje 3 wymagane funkcje:



Rysunek 5.5: Diagram klas służących implementacji węzła sterującego ruchem manipulatora

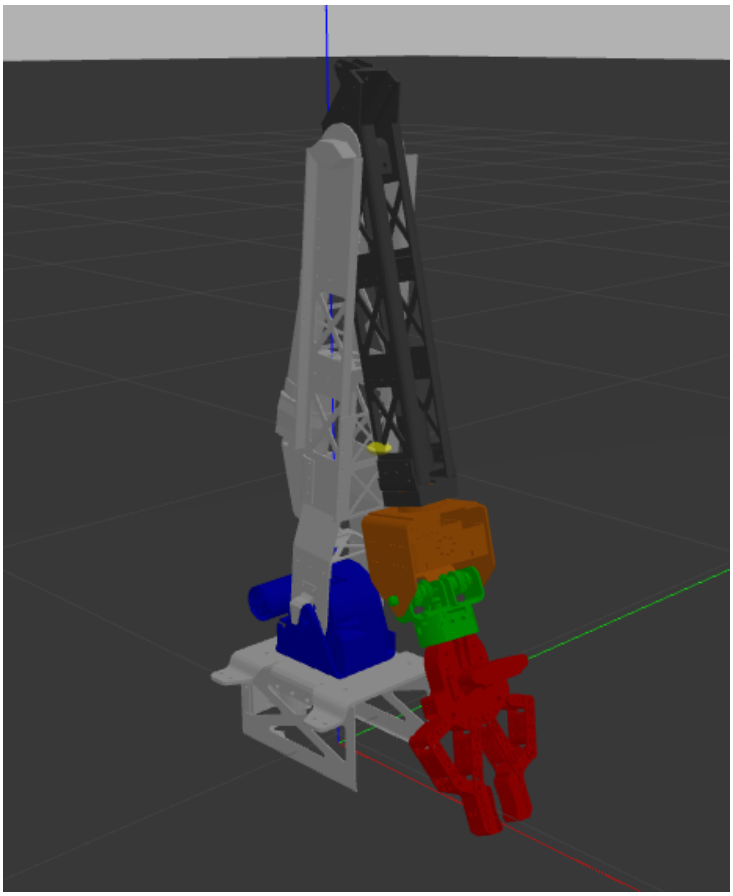
- *write()* - funkcja przesyła zadane prędkości do manipulatora w przypadku możliwości komunikacji z urządzeniem, w przeciwnym wypadku komendy ruchu wysyłane są do symulacji,
- *read()* - funkcja odczytuje prędkości i stan manipulatora w przypadku możliwości komunikacji z urządzeniem, w przeciwnym wypadku dane te odczytywane są z symulacji,
- *update()* - działanie funkcji polega na odczytaniu aktualnych prędkości (*read()*), aktualizacji stanu sterowników wykorzystując interfejs *controllerManager*, a na koniec przesłanie danych o sterowaniu (*write()*).

Interfejs sprzętowy wykorzystuje klasę *RobotSerialComm*, która pozwala na wysyłanie i odbieranie danych wykorzystując port szeregowy komputera. Jest ona wykorzystywana do ustawiania parametrów i komunikacji po interfejsie UART z manipulatorem. Przesyłane komunikaty mają taką samą postać jak przedstawiona w opisie komunikacji agenta a_{robot} (*ManipulatorMsg*). Do odbioru danych wykorzystywany jest także bufor cykliczny. W przypadku wystąpienia problemów (odczytanie danych nie powiodło się przez 20 ms) następuje przerwanie komunikacji.

Wirtualny receptor r_{sim1} wykorzystuje węzeł */spacenav* do konwertowania wartości wychyleń myszy 3D i wysyłania ich w temacie */spacenav/twist*. Do tego tematu zostaje podpięty subskrybent *twistSub*, który wywołuje funkcję *newVelCallback()* ilekroć na temacie spacenav

zostaje opublikowana nowa wiadomość. W zależności od trybu pracy dane pozyskane z tematu dotyczą zadania prędkości końcówki (narzędziu) lub pojedynczych przegubów. Jeżeli sterownik pracuje w trybie złączowym, wartość wychylenia przeliczane są do prędkości względem ich maksymalnych wartości. W innym przypadku dodatkowo wykorzystywana jest biblioteka *kdl*, która posiada solvery prostej i odwrotnej kinematyki. Zadana prędkość narzędzia przeliczana jest do wymaganych do jej uzyskania prędkości poszczególnych członów poprzez rozwiązania zadania odwrotnego kinematyki dla prędkości. Aby było to możliwe w trakcie inicjalizacji interfejsu definiowany jest łańcuch kinematyczny *kinematicChain* manipulatora, który jednoznacznie opisuje położenie członów względem siebie. Opis łańcucha kinematycznego wykonano na podstawie parametrów Denavita-Hartenberga manipulatora przedstawionych w rozdziale 3.

Symulacja została przygotowana wykorzystując program Gazebo. W celu zdefiniowania modelu fizycznego i wizualnego robota wykorzystano format URDF. Każdy z członów oprócz standardowych parametrów takich jak masa, punkt początkowy, materiał, geometria i rodzaj i ograniczenia przegubów ma zdefiniowany ponadto dodatkowy element *transmission*, który służy do połączenia sterowników biblioteki *ros_control* z symulatorem. Na podstawie pliku URDF tworzony jest model manipulatora pokazany poniżej:



Rysunek 5.6: Model manipulatora w środowisku symulacyjnym Gazebo

6 Weryfikacja

7 Podsumowanie

Celem niniejszej pracy było opracowanie systemu sterowania do manipulatora o 6 stopniach swobody, który pozwalałby na bezpieczne testowanie ruchu robota w środowisku symulacyjnym.

7.1 Wyniki działania systemu

7.2 Wnioski

7.3 Perspektywy rozwoju

DODATEK A. Zawartość płyty CD

Płyta CD, która została dołączona do pracy magisterskiej zawiera tekst w formacie pdf, programy symulacji i sterujący mikrokontrolera STM32F407-VET oraz pliki wynikowe przeprowadzonych testów. Układ danych w nośniku jest następujący:

- folder *tekst* zawiera plik Marcin_Baran_praca_mgr.pdf,
- folder *oprogramowanie* zawiera kod źródłowy mikrokontrolera i oprogramowania symulacyjnego odpowiednio w folderach *STM32F4* i *symulacja*,
- folder *testy* zawiera pliki z danymi uzyskanymi podczas weryfikacji systemu.

Bibliografia

[1] T. Winiarski i T. Kornuta C. Zielinski. „Agent-Based Structures of Robot Systems". W: *Trends in Advanced Intelligent Control, Optimization and Automation* 577 (2017), s. 493–502. DOI: doi:10.1007/978-3-319-60699-6_48.

[2] Tomasz Kornuta Cezary Zieliński. „Programowe struktury ramowe do tworzenia sterowników robotów". W: *Pomiary Automatyka Robotyka* 19.1 (2015), s. 5–14. DOI: 10.14313/PAR_215/5.

[3] Sachin Chitta i in. „ros control: A generic and simple control framework for ROS". W: *The Journal of Open Source Software* (2017). DOI: 10.21105/joss.00456.

[4] Open Source Robotics Foundation. *Opis pakietu Robot Operating System*. Ostatni dostęp 08.08.2019. URL: https://www.ros.org/about-ros/.

[5] Open Source Robotics Foundation. *Strona poświęcona bibliotece spacenav do obsługi urządzeń 3Dconnexion*. Ostatni dostęp 08.08.2019. URL: http://wiki.ros.org/spacenav.

[6] Open Source Robotics Foundation. *Strona poświęcona dodatkowi ros control*. Ostatni dostęp 08.08.2019. URL: http://ros.org/wiki/ros_control.

[7] Open Source Robotics Foundation. *Strona poświęcona standardowi opisu URDF*. Ostatni dostęp 08.08.2019. URL: http://wiki.ros.org/urdf.

[8] LogiCad3D GmbH. *Instrukcja myszy 3D Magellan Space Mouse*. Ostatni dostęp 08.08.2019. URL: https://www.microsoft.com/buxtoncollection/a/pdf/Magellan%20Manual.pdf.

[9] ISO. *Norma ISO 8373*. Ostatni dostęp 08.08.2019. URL: https://www.iso.org/obp/ui/#iso:std:iso:8373:ed-2:v1:en.

[10] Aleksandra Karbarczyk. „System robotyczny chwytający obiekty". Politechnika Warszawska, 2017.

[11] Waldemar Wróblewski Krzysztof Kozłowski Piotr Dutkiewicz. *Modelowanie i sterowanie robotów*. Warszawa: PWN, 2012, s. 344–391.

[12] Łukasz Maciej Meyer. „Sterowanie predykcyjne z wykorzystaniem wizji w zadaniu śledzenia ścieżki przez robota mobilnego". Politechnika Warszawska, 2019.

[13] SysML Partners. *Strona poświęcona SysML*. Ostatni dostęp 08.08.2019. URL: https://sysml.org/.

[14] Robotis. *Instrukcja serwomechanizmów Dynamixel RX-64*. Ostatni dostęp 08.08.2019. URL: http://emanual.robotis.com/docs/en/dxl/rx/rx-64/.

[15] Robotis. *Opis protokołu komunikacji z serwami Dynamixel*. Ostatni dostęp 29.08.2019. URL: http://emanual.robotis.com/docs/en/dxl/protocol1/.

[16] Amazon Web Services. *Strona poświęcona systemowi czasu rzeczywistego FreeRTOS*. Ostatni dostęp 08.08.2019. URL: https://www.freertos.org/.

[17] The Mars Society. *Requirements and guidelines for University Rover Challenge competition*. Ostatni dostęp 08.08.2019. URL: http://urc.marssociety.org/home/requirements-guidelines.

[18] ST. *Strona poświęcona bibliotekom Standard Peripheral Libraries*. Ostatni dostęp 08.08.2019. URL: https://www.st.com/en/embedded-software/stm32-standard-peripheral-libraries.html.

[19] John Tsiombikas. *Strona poświęcona sterownikom spacenav*. Ostatni dostęp 08.08.2019. URL: http://spacenav.sourceforge.net/.

[20] Maciej Piotr Węgierek. „Agentowa specyfikacja systemu sterującego robota manipulacyjnego IRp-6". Politechnika Warszawska, 2018.

[21] *Wykłady z przedmiotu Wstęp do Robotyki na wydz. EiTI Politechniki Warszawskiej*. Ostatni dostep 12.11.2018. URL: http://elka.pw/obieralne/wr/wyk/WR%20cz4.pdf.

[22] Aleksandar Zivkovic. „Development of autonomous driving using Robot Operating System". Universidad Politécnica de Madrid, 2018.

Wykaz symboli i skrótów

API Application Programming Interface

IMU Inertial Measurement Unit

ISO International Organisation for Standarization

KRL KUKA Robot Language

PID proportional–integral–derivative

PWM Pulse Width Modulation

ROS Robot Operating System

RTOS Real Time Operating System

STDPeriph Standard Peripheral Libraries

SysML System Modelling Language

UML Unified Modelling Language

URC University Rover Challenge

Spis rysunków

1.1 Schemat stworzonego systemu programowania *(Źródło: [11])* 12
1.2 Struktura systemu sterowania robotem mobilnym *(Źródło: [12])* 13
1.3 Struktura systemu sterowania autonomicznym pojazdem RC *(Źródło: [22])* . . 14
2.1 Konstrukcja manipulatora szeregowego *(Źródło: [21])* 15
2.2 Położenie członu P i związanego z nim układu U_1 opisane względem globalnego układu U_0 16
3.1 Zaprojektowany manipulator wykonujący zadanie konkursowe 19
3.2 Schemat komunikacji mikrokontrolera sterującego robotem 20
3.3 Mysz 3D Magellan Space Mouse Plus *(Źródło: [8])* 21
3.4 Schemat agenta upostaciowionego *(Źródło: [20])* 22
3.5 Schemat działania sterownika robota opartego o dodatek ros_control *(Źródło: [6])* 25
3.6 Schemat kolejności wykonywania zadań we FreeRTOS *(Źródło: [16])* 26
4.1 Diagram przypadków użycia systemu 28
4.2 Komunikacja pomiędzy agentami systemu 29
4.3 Schemat agenta a_{robot} 30
4.4 Schemat agenta a_{sim} 31
4.5 Automat skończony podsystemu sterowania c_{robot} 32
4.6 Funkcja przejścia ${}^{c}f_{robot,1}$ 33
4.7 Funkcja przejścia ${}^{c}f_{robot,2}$ 34
4.8 Funkcja przejścia ${}^{c}f_{robot,3}$ 34
4.9 Funkcja przejścia ${}^{c}f_{robot,4}$ 35
4.10 Funkcja przejścia ${}^{c}f_{robot,5}$ 35
4.11 Funkcje przejścia ${}^{e}f_{robot,1..6}$ 36
4.12 Automat skończony podsystemu sterowania c_{sim} 37
4.13 Funkcja przejścia ${}^{c}f_{sim,1}$ 38
4.14 Funkcja przejścia ${}^{c}f_{sim,2}$ 38
4.15 Funkcja przejścia ${}^{c}f_{sim,3}$ 39
5.1 Diagram klas sterujących napędami 41
5.2 Priorytet wykonywania zadań podsystemu 43
5.3 Diagram klasy implementującej automat skończony podsystemu sterowania c_{robot} 44
5.4 Diagram komunikacji pomiędzy węzłami sterownika manipulatora 47
5.5 Diagram klas służących implementacji węzła sterującego ruchem manipulatora 49
5.6 Model manipulatora w środowisku symulacyjnym Gazebo 50

Spis tablic

3.1 Przełożenia uwzględniane przy odczycie prędkości związane z montażem enkoderów 20
3.2 Parametry Denavita-Hartenberga manipulatora 21
4.1 Komunikacja pomiędzy agentami a_{robot} i a_{sim} 29
4.2 Komunikacja podsystemów agenta a_{robot} 30
4.3 Komunikacja podsystemów agenta a_{sim} 31
4.4 Zachowania podsystemu sterowania c_{robot} i odpowiadające im stany 32
4.5 Zachowania podsystemu sterowania c_{sim} i odpowiadające im stany 37
5.1 Ramka danych wysyłana do serwa Dynamixel *(źródło: [15])* 42
5.2 Ramka danych odsyłana od serwa Dynamixel *(źródło: [15])* 42
5.3 Procedury obsługi stanów automatu skończonego manipulatora 45
5.4 Ramka danych komunikacji manipulator- sterownik 46