Politechnika Warszawska

WYDZIAŁ MECHANICZNY
ENERGETYKI I LOTNICTWA

Instytut Techniki Lotniczej i Mechaniki Stosowanej

# Praca dyplomowa inżynierska

na kierunku Automatyka i Robotyka
w specjalności Robotyka

## Opracowanie aplikacji do zdalnej konfiguracji modułów elektronicznych robota

Julian Prolejko
Numer albumu 276392

promotor
dr inż. Andrzej Chmielniak

Warszawa, 2019

**Streszczenie**

Zadaniem postawionym w pracy jest stworzenie wieloplatformowej aplikacji do zdalnej konfiguracji modułów elektronicznych robota. Wynikające z współczesnej robotyki mobilnej złożone systemy sterowania wymagają doboru odpowiednich parametrów do prawidłowego działania. Dostosowywanie wartości i konfiguracja układów jest żmudną praktyką. Zazwyczaj konieczne jest zatrzymanie całego systemu i zajęcie się konkretną jego częścią. Ponieważ najczęściej to całościowa ocena pracy robota ma największe znaczenie, narzędzie opisane w pracy znacznie przyspiesza etap konfiguracji.

W pierwszym rozdziale opisane zostały podstawowe zagadnienia, które wykorzystano przy rozwoju aplikacji. Wstęp teoretyczny objął komunikacyjną magistralę CAN, jej historię powstania oraz zasadę działania. Zawarty został także opis procesu transmisji danych pomiędzy modułami elektronicznymi i sposób wykorzystania sieci tego typu do celów konfiguracyjnych. Rozdział pierwszy opisuje również przemysłowego robota mobilnego Kanboy, który posłużył za obiekt testowy aplikacji. Ponadto ta część pracy objęła także opis bibliotek języka Python, wykorzystanych przy opracowywaniu aplikacji: python-can i Flask. Uwzględniono również przegląd istniejących rozwiązań z zakresu zdalnej konfiguracji urządzeń.

Rozdział drugi zawiera ścisłe sformułowanie celu pracy, a także założenia, które zostały przyjęte dla rozwiązania. Opisano także zakres, jaki został objęty przez pracę.

W trzecim rozdziale opisano zrealizowaną aplikację. Umieszczony został opis ogólny oraz szczególny wszystkich części rozwiązania. Uwzględniono podział na część klienta i serwera, komunikację pomiędzy nimi i sposób, w jaki każda z funkcjonalności została zaimplementowana. Pokazano również proces transmisji danych przez magistralę CAN i sposób ich interpretacji z poziomu aplikacji.

Czwarty rozdział opisuje działające rozwiązanie i obsługę aplikacji. Zawarto również opis interfejsu użytkownika i właściwy proces konfiguracji urządzeń.

W rozdziale piątym opisany został szereg eksperymentów, potwierdzających prawidłowe działanie aplikacji. Przedstawiono stanowiska testowe i ich współpracę z narzędziem konfiguracyjnym. Zawarty został opis procedur, sprawdzających poprawne działanie wszystkich funkcjonalności.

Ostatnią częścią pracy jest podsumowanie, w którym przedstawiono wnioski końcowe i propozycje dalszego rozwoju narzędzia konfiguracyjnego.

*Słowa kluczowe: robotyka, zdalna konfiguracja, komunikacja, CAN, moduły elektroniczne, elektronika, technologie internetowe, Python.*

**Abstract**

Task posed in the study was to develop a multi-platform application for remote configuration of robot's electronic modules. Complex control systems resulted from the modern mobile robotics need an appropriate selection of parameters for correct functioning. Customization of every value and system configuration is a painstaking practice. Usually, it is needed to stop a whole setup to change only one particular part. Generally, overall evaluation of robot's operations is the most important and tool described in this study effectively accelerates the process of configuration.

First chapter describes basic issues, which are used in a development of application. Theoretical beginning covered description of communication via CAN bus, its history and principle of operation. Process of data transmission between electronic modules is also included. The way of usage of described network in configuration purposes is shown as well. Furthermore, the very first chapter describes an industrial mobile robot Kanboy, which was used as a test object of an application. In addition to this, this part of study covered description of Python language libraries, that were used in a process of development: python-can and Flask. The review of existing solutions in the field of remote configuration of electronic devices was also included.

Second chapter contains the exact formulation of purpose of the study, as well as the assumptions that were adopted for the solution. The range that was covered by the work was also described.

Third chapter of the study describes developed application. It contains general and specific description of all parts of the solution. The division into the client and server part, communication between them and the way in which each functionality was implemented is included. Also, it is shown how the process of data transmission via the CAN bus proceeds and the way how the data is interpreted in the application level.

Fourth chapter presents working solution and instructs how to use the application. Description of user interface and the proper process of devices configuration is also included.

A number of experiments were shown in the fifth chapter of study. They confirmed the proper operation of the application. The chapter consists of description of test environment and its cooperation with the configuration tool. A list of the procedures, which checked the correct working of all functionalities is also included.

The last part of study presents a summary, which consists of the final conclusions and proposals for further development of the configuration tool.

*Keywords: Robotics, remote configuration, communication, CAN, electronic modules, Electronics, web technologies, Python.*

## Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że przedstawiona praca dyplomowa:

- została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami,
- nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego lub stopnia naukowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

..... .....
data podpis autora (autorów) pracy

## Oświadczenie

Wyrażam zgodę / nie wyrażam zgody[*1] na udostępnianie osobom zainteresowanym mojej pracy dyplomowej. Praca może być udostępniana w pomieszczeniach biblioteki wydziałowej. Zgoda na udostępnienie pracy dyplomowej nie oznacza wyrażenia zgody na jej kopiowanie w całości lub w części.
Brak zgody nie oznacza ograniczenia dostępu do pracy dyplomowej osób:
- reprezentujących władze Politechniki Warszawskiej,
- członków Komisji Akredytacyjnych,
- funkcjonariuszy służb państwowych i innych osób uprawnionych, na mocy odpowiednich przepisów prawnych obowiązujących na terenie Rzeczypospolitej Polskiej, do swobodnego dostępu do materiałów chronionych międzynarodowymi przepisami o prawach autorskich.

Brak zgody nie wyklucza także kontroli tekstu pracy dyplomowej w systemie antyplagiatowym.

..... .....
data podpis autora (autorów) pracy

*1 - niepotrzebne skreślić

# Spis treści

# Rozdział 1

# Wstęp teoretyczny

## 1.1 Moduły elektroniczne w robotyce mobilnej

Postęp technologiczny jest możliwy do zaobserwowania w wielu technicznych dziedzinach. Robotyka mobilna, jako jedna z najmłodszych, ewoluowała na przestrzeni lat w bardzo znaczącym stopniu. Rozwój postępował od czasów II wojny światowej i pierwszych robotów mobilnych do celów militarnych, przechodząc przez łaziki wysyłane przez firmę *NASA* (ang. *National Aeronautics and Space Administration*) na Marsa i kończąc na czasach dzisiejszych. Roboty firmy *Boston Dynamics*, takie jak *ATLAS* - interaktywny humanoid o bardzo zaawansowanym układzie dynamicznym, czy *Spot Mini* - czworonóg, przypominający psa - szczególnie zwracają na siebie uwagę pod względem zaawansowania technicznego [6].

Robotyka mobilna z dzisiejszych czasów kojarzy się z wieloma zaawansowanymi systemami sterowania. Większość systemów zrealizowana jest za pomocą układów mikroprocesorowych, ze względu na ich łatwą dostępność i uniwersalność. Układy scalone tego typu, charakteryzują się wielozadaniowością. Implikuje to, że nie muszą spełniać tylko jednego zadania (dla przykładu, sterowanie procesem chłodzenia robota), ale również pełnić rolę komunikacji.

Daje to możliwość połączenia wszystkich urządzeń elektronicznych w jedną dużą sieć, w której każdy z węzłów ma kontakt z całą resztą. Dzięki takiemu rozwiązaniu, wszystkie pozornie niezależne od siebie procesy mogą przekazywać pomiędzy sobą informacje i dostosowywać się do warunków panujących w całym systemie robota. Przykładem realizacji sieci tego typu jest magistrala *CAN*, często wykorzystywana w robotyce mobilnej.

## 1.2 Opis magistrali komunikacyjnej CAN

### 1.2.1 Historia powstania

CAN (*ang. Controller Area Network*) jest szeregową magistralą komunikacyjną, powstałą w 1986 roku w firmie Robert Bosch GmbH na potrzeby przemysłu samochodowego. Pierwotnym zastosowaniem była wymiana wiadomości i rozkazów pomiędzy systemami w samochodzie a komputerem pokładowym. Dzięki swojej niezawodności zyskała na popularności i obszar jej zastosowań został powiększony. Konsekwencją wzrostu zainteresowania systemem komunikacji tego typu, powstało wiele standardów przesyłania (*CANopen, DeviceNet*) [7].

Ostatnia wersja standardu firmy Bosch została opublikowana w 1991 roku jako CAN 2.0 i składa się z dwóch części: CAN 2.0A oraz CAN 2.0B. Różnica pomiędzy A i B polega na różnicy w liczbie

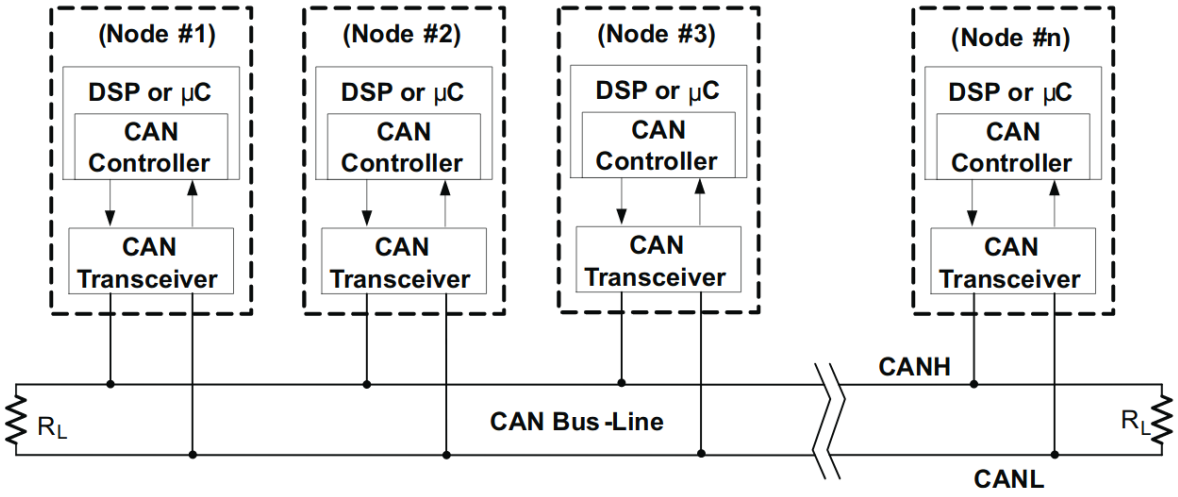
bitów w identyfikatorze wiadomości (CAN 2.0A - 11 bitów, CAN 2.0B - 29 bitów) [1].

W 1993 roku, organizacja ISO (International Organization for Standarization) opublikowała standard CAN ISO 11898 rozdzielony na część określającą warstwę łącza danych (ISO 11898-1) oraz warstwę fizyczną magistrali dużej przepustowości (ISO 11898-2). Oprócz głównego podziału na warstwy, w standardzie występuje również część ISO 11898-3 opisująca warstwę fizyczną magistrali CAN niskich prędkości z tolerancją błędów. Warstwowość standardu CAN opisana powyżej jest zgodna z modelem warstwowym OSI (pełna nazwa *ISO OSI RM - ISO Open Systems Interconnection Reference Model*) [3].

### 1.2.2 Zasada działania

CAN jest magistralą o charakterze szeregowym, wykorzystującą przewód dwużyłowy do celów transmisyjnych. Podstawową cechą omawianego połączenia jest brak jednostki nadrzędnej wśród urządzeń. Konsekwencją takiej struktury jest komunikacja o charakterze rozgłoszeniowym - każda wiadomość trafia do wszystkich urządzeń w sieci. Magistrala CAN pozwala na przesył o przepustowości do 1 Mb/s na odległości do 40 m. Wraz ze wzrostem odległości transmisyjnej, przepustowość kanału maleje. Sieci skonstruowane w takiej postaci spełniają wymagania czasu rzeczywistego.

Oba krańce magistrali CAN, w celu poprawnego funkcjonowania, muszą być zakończone terminacją w postaci rezystora o oporze $R_L = 120\Omega$. Wszystkie pozostałe urządzenia wpięte w magistralę są podłączone do obu linii układu - CAN High oraz CAN Low (na rysunku 1.1 oznaczane odpowiednio jako CANH oraz CANL). Obie linie są nośnikami sygnałów przebiegających przez każde z węzłów układu [2].



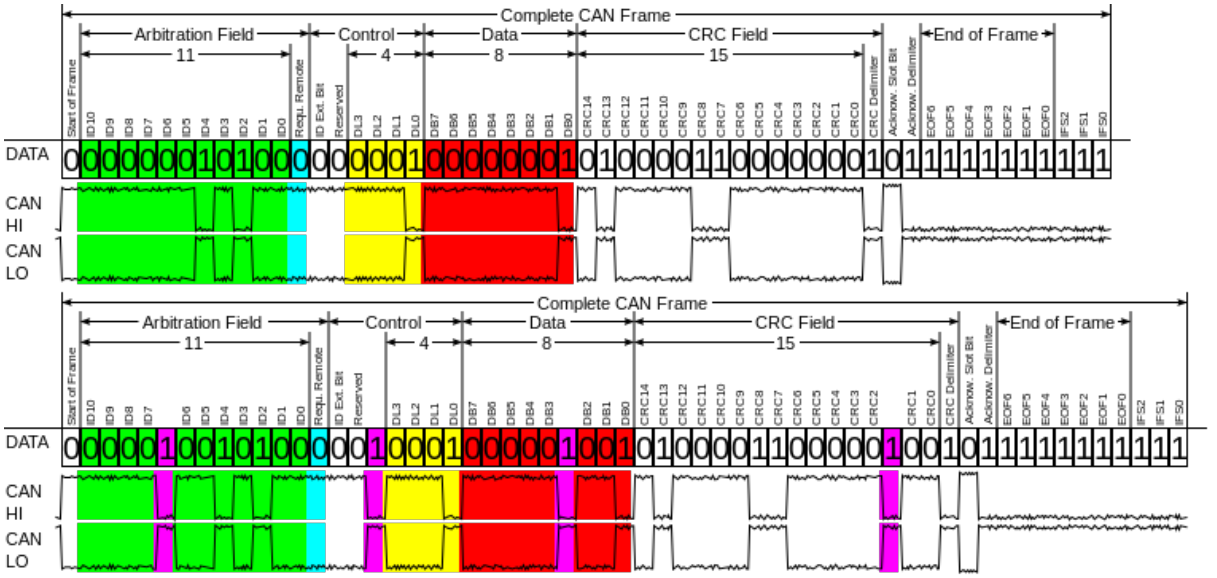

Rysunek 1.1: Struktura magistrali CAN [2]

Nośnikiem informacji w magistrali CAN jest różnicowy sygnał napięciowy. Stan sygnału jest określany na podstawie różnicy pomiędzy linią CAN High a CAN Low. Zastosowanie takiego rozwiązania jest związane z dużą odpornością komunikacji na zakłócenia i szumy zewnętrzne. Magistrala w trakcie działania przyjmuje jeden z dwóch stanów - recesywny lub dominujący (tabela 1.1). Stan recesywny występuje w chwili, gdy na obu liniach CAN pojawi się to samo napięcie (mieszczące się w tolerancji podanej w tabeli). Stan dominujący występuje w sytuacji przeciwnej - gdy na linii CAN High pojawi się sygnał o napięciu 3.5 V, a na linii CAN Low, sygnał o napięciu 1.5 V [3].

Tabela 1.1: Opis akceptowalnych napięć występujących w magistrali

| Linia | Stan recesywny | Stan dominujący |
|---|---|---|
| CAN High | 2-3 V | 2,75–4,5 V |
| CAN Low | 2-3 V | 0,5–2,25 V |

Kodowanie cyfrowe sygnału opiera się na przypisaniu logicznej jedynce - stanu recesywnego, a logicznemu zeru - stanu dominującego. W każdym węźle magistrali, sygnał jest próbkowany z określoną częstotliwością, a następnie interpretowany jako stan cyfrowy. W celu zwiększenia odporności na błędy konieczne jest zastosowanie mechanizmu *Bit stuffing*. W przypadku występowania tego samego stanu logicznego wielokrotnie pod rząd, co 5 bitów dodawany jest stan przeciwny, ułatwiający identyfikację kodu. Proces został przedstawiony na rysunku 1.2 [8].



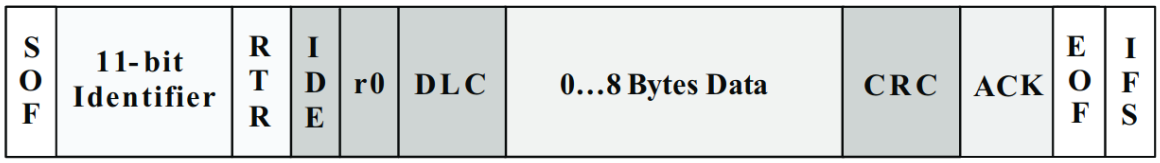

Rysunek 1.2: Mechanizm *Bit stuffing* z zaznaczonymi na fioletowo dodatkowymi bitami [8]

### 1.2.3 Warstwa łącza danych

Wiadomości przesyłane za pomocą magistrali CAN mają postać ramek. Można je rozdzielić na 4 podstawowe kategorie:

- *data frame* - przesyłająca dane pomiędzy węzłami,
- *remote frame* - wywołująca odpowiedź w odpowiednim węźle,
- *error frame* - sygnalizująca wystąpienie błędu,
- *overload frame* - sygnalizująca przepełnienie.

Ponadto każda sieć może obsługiwać równocześnie oba z opisanych standardów ramek danych: CAN 2.0A oraz CAN 2.0B. Różnica pomiędzy nimi występuje w długości identyfikatora ramki. Dla CAN 2.0A jest to 11 bitów, a odpowiednio dla CAN 2.0B - 29 bitów. Struktury wiadomości typu *data frame* w standardach CAN 2.0A i CAN 2.0B zostały przedstawione na rysunkach 1.3 i 1.4 [2].

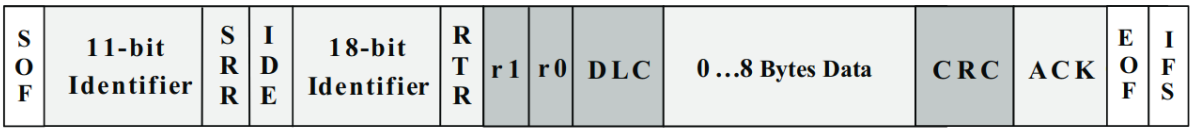



Rysunek 1.3: Format wiadomości *data frame* w standardzie CAN 2.0A [2]

Każda ramka danych jest rozpoczynana przez bit w stanie dominującym (logiczne zero) sygnalizujący początek ramki (*SOF*). Następnie transmitowany jest 11-sto bitowy identyfikator, który równocześnie pełni funkcje prioretyzacji (im niższa wartość, tym większy priorytet wiadomości). Następnikiem identyfikatora jest bit *RTR*, określający postać wiadomości (przesył danych - logiczne zero, czy prośba o odpowiedź - logiczne jeden). Kolejnym składnikiem ramki jest bit *IDE*, określający postać wiadomości (*standard* - CAN 2.0A lub *extended* - CAN 2.0B). Następny bit *r0* jest zarezerwowany dla prawdopodobnych prac rozwojowych nad nowszymi standardami warstwy łącza danych.

Kolejnymi jednostkami informacyjnymi w wiadomości są 4 bity *DLC* z zakodowaną liczbą bajtów danych. Następnie przesyłany jest ciąg złożony z maksymalnie 8 bajtów danych, zawierający informacje dostarczaną przez wiadomość. Po transmisji danych na magistrali pojawia się grupa 16 bitów reprezentująca cykliczny kod nadmiarowy *CRC* (ang. *Cyclic Redundancy Check*). Tego typu rozwiązanie polega na przesłaniu sumy kontrolnej informacji dostarczanej przez wiadomość, w celu detekcji i eliminacji błędów, które mogą pojawić się podczas transmisji.

Pozostała struktura w formacie ramki to 2 bitowa grupa *ACK* odpowiadająca za przechowywanie potwierdzenia o poprawnym odebraniu/transmisji ramki oraz bit *EOF* sygnalizujący zakończenie wiadomości. Ponadto pomiędzy każdą z kolejnych wiadomości występuje przerwa w postaci 7 bitów. Jest to konieczny zabieg, zapewniający odpowiedni czas na propagację ramki do odpowiednich obszarów w buforze.



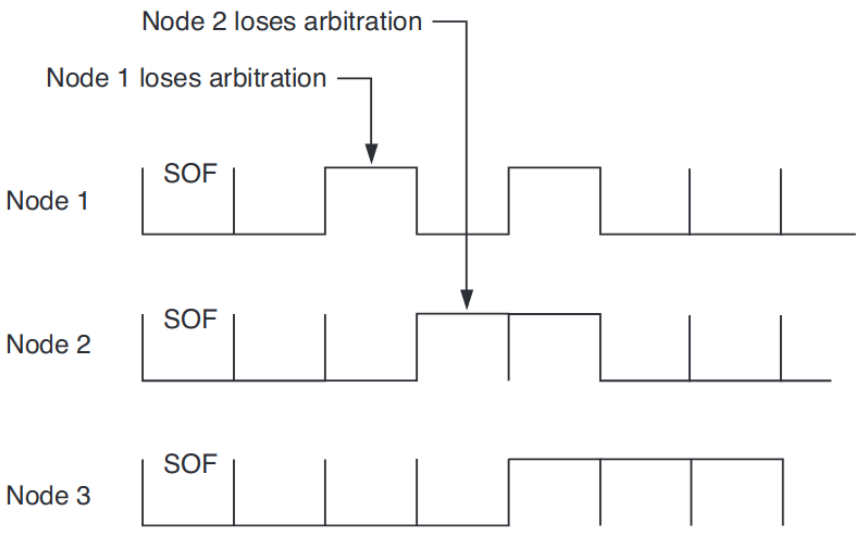

Rysunek 1.4: Format wiadomości *extended data frame* w standardzie CAN 2.0B [2]

Na rysunku 1.4 widoczna jest różnica pomiędzy standardową ramką danych a ramką *extended*. Dodatkowymi strukturami w formacie wydłużonym są bity:

- *SRR* - substytut bitu *RTR* ze standardu CAN 2.0A - pełni funkcję wypełniającą,
- *r1* - kolejny bit zarezerwowany do celów rozwojowych,
- 18-bitowe przedłużenie identyfikatora.

### 1.2.4 Arbitraż dostępu do magistrali

Polem wchodzącym w skład arbitrażu w magistrali CAN jest część ramki odpowiadająca za identyfikator (opisany w poprzedniej sekcji) wraz z bitem *RTR*. Schemat, wykorzystywany do ustalania dostępu do transmisji, jest nazwany *CSMA/CD* (ang. *Carrier Sense Multiple Access with Collision Detection*). W sytuacjach kolizyjnych mechanizm ten pozwala na propagację tylko jednej wiadomości - z ramką o najwyższym priorytecie. Priorytet wiadomości ustalany jest za pomocą pola arbitrażu. Im mniejsza jest wartość liczbowa tego pola, tym wyższy jest priorytet ramki.




Rysunek 1.5: Mechanizm detekcji kolizji wiadomości [4]

*CSMA/CD* wykorzystuje dwie właściwości magistrali CAN. Pierwszą z nich jest działanie stanów linii CAN na zasadzie iloczynu logicznego. Drugą, równie ważną własnością, jest równoczesne odczytywanie stanów magistrali przez transmitujący węzeł podczas wysyłania wiadomości.

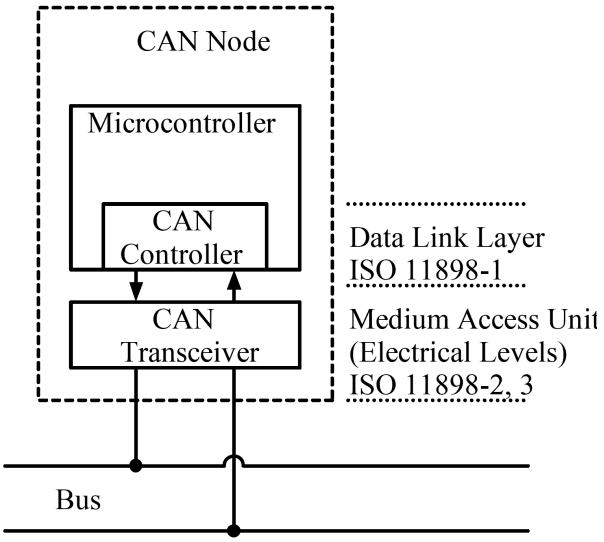
W przypadku jednoczesnego wysłania stanów przeciwnych przez różne węzły, na magistrali zawsze pojawi się stan dominujący (logiczne 0). Uczestnik transmisji, który odczyta bit niezgodny z wysłanym przez siebie, przerywa transmisję i pozwala na propagację wiadomości z niższym liczbowo polem arbitrażu. Mechanizm ten, został przedstawiony na rysunku 1.5 [4].

## 1.3 Proces komunikacji pomiędzy modułami elektronicznymi

### 1.3.1 Struktura układu

Każde urządzenie posiadające połączenie z magistralą CAN otrzymuje wszystkie pojawiające się w niej wiadomości. Każda z nich jest interpretowana i przetwarzana, jeżeli jest skierowana do konkretnego węzła. Rozpoznanie adresata wiadomości odbywa się na podstawie identyfikatora. Przykładowa struktura układu działającego w zadany sposób została przedstawiona na rysunku 1.6.

Podstawą działania poprawnego węzła magistrali są trzy elementy [8]:




Rysunek 1.6: Węzeł magistrali CAN [8]

- *Transceiver CAN* - nadajnik-odbiornik reprezentujący warstwę fizyczną standardu CAN. Podczas odbierania wiadomości odpowiada za przechwytywanie sygnałów występujących na liniach CAN High i CAN Low i przekazywanie ich do warstwy łącza danych. W trybie transmisji, odbiera dane od kontrolera CAN i zamienia ciągi bitów w odpowiednio nadawane sygnały.
- *Kontroler CAN* - reprezentacja warstwy łącza danych standardu CAN. W trybie odbioru, odbiera ciągi bitów od transceivera i magazynuje, aż do uzyskania kompletnej wiadomości. Następnie przekazuje odebrane dane do pamięci mikrokontrolera. Podczas nadawania wiadomości, otrzymuje dane od procesora i przekazuje je w postaci ciągu bitów do transceivera CAN.
- *Mikrokontroler* - zarządza logiką całego systemu. Decyduje o tym, które wiadomości są skierowane do danego węzła oraz o tym, jaką wiadomość wysłać przez magistralę CAN. Umożliwia podłączenie dodatkowych urządzeń poprzez interfejs wejścia/wyjścia.

Każdy z wyżej wymienionych elementów elektronicznych jest w postaci układu scalonego. Bardzo często w produkcji pojawiają się również moduły całościowo zapewniające wszystkie wymagane funkcjonalności (połączenie transceivera CAN, kontrolera CAN oraz mikrokontrolera w postaci jednego układu scalonego) lub składające się z dwóch układów scalonych (transceiver CAN oraz mikrokontroler połączony z kontrolerem CAN).

### 1.3.2 Transmisja wiadomości

Decyzja o wysłaniu wiadomości przez magistralę CAN jest podejmowana przez mikrokontroler. Dane, które mają być przesłane po liniach transmisyjnych, są czerpane z obliczeń lub pamięci nieulotnej (np. z wartości rejestrów w pamięci *flash*). Wraz z adresem docelowego węzła trafiają do kontrolera CAN. Układ ten, reprezentujący warstwę łącza danych w standardzie CAN, z wartości otrzymanych od mikrokontrolera, konstruuje ramkę (standardową lub rozszerzoną, w zależności od zapotrzebowania wiadomości) według schematów opisanych w podsekcji 1.2.3. Wiadomość w postaci łańcucha bitów jest następnie przekazywana do transceivera CAN. Warstwa fizyczna konwertuje każdy z bitów na stan dominujący (logiczne 0) lub stan recesywny (logiczne 1) i transmituje poprzez linie CAN High i CAN Low.

### 1.3.3 Odczytanie wiadomości

Zmiana potencjałów na liniach CAN High i CAN Low jest wychwytywana oraz interpretowana jako dane binarne przez transceiver CAN - reprezentację warstwy fizycznej standardu. Następnym modułem, który przejmuje odpowiedzialność komunikacji jest kontroler CAN, czyli warstwa łącza danych. Moduł ten interpretuje dane binarne, rozdzielając łańcuch bitów na odpowiednie podciągi według klasyfikacji pokazanej na rysunkach 1.3 i 1.4. Tak skategoryzowane informacje zostają przekazane do mikrokontrolera. W zależności od celów danej transmisji, dane znajdujące się w ramce CAN są przetwarzane, przesyłane dalej lub zapisywane do pamięci nieulotnej. W wypadku otrzymania wiadomości z prośbą o informacje zwrotną, mikrokontroler gromadzi dane i przygotowuje wiadomość dla kontrolera CAN.

## 1.4 Robot mobilny Kanboy

Wykorzystany w dalszej części pracy robot mobilny to przemysłowa platforma jezdna, wchodząca w skład systemów intralogistycznych. *Kanboy*, produkt firmy Versabox Sp. z o.o., jest urządzeniem o budowie z zachowaną modułowością. Jest zintegrowany z wielu złożonych podelementów.


Rysunek 1.7: Przemysłowy robot mobilny Kanboy

Możliwość ruchu robota została zrealizowana mechanicznie za pomocą napędu różnicowego. Koła napędzające platformę umieszczono z lewej i prawej strony w osi, przechodzącej przez środek nadwozia. Oprócz tego, w każdym z narożników urządzenia znajduje się koło samonastawne. Tak skonstruowane podwozie umożliwia ruch liniowy oraz ruch obrotowy w miejscu, co ułatwia sterowanie ruchem.

Moduły elektroniczne, znajdujące się na pokładzie robota, tworzą kompleksowy system. Jego główną częścią jest komputer *Intel NUC*, którego zadaniem jest przeprowadzanie wysokopoziomowych obliczeń, obejmujących SLAM (ang. *Simultaneous Localization and Mapping*), planowanie i realizację ścieżki oraz interakcję z użytkownikiem. Ze względu na podział zadań pod kątem poziomów abstrakcji, dodatkowym urządzeniem obliczeniowym jest mikrokomputer *BeagleBone Black*. Razem z komputerem *Intel NUC* tworzy sieć, zarządzającą wszystkimi modułami elektronicznymi.

Mikrokomputer pośredniczy w wymianie danych pomiędzy komputerem *Intel NUC* a resztą systemu. Przetwarza dane z wysokiego poziomu na niski i odwrotnie. Podłączenie do głównej magistrali CAN robota umożliwia mu odbiór danych z sensorów oraz transmisję rozkazów do innych modułów elektronicznych. Za pomocą odpowiednich wiadomości zarządza pracą między innymi sterowników napędów oraz modułów wejść-wyjść.

Robot mobilny umożliwia komunikację z użytkownikiem na trzy sposoby. Pierwszym z nich jest sieć bezprzewodowa, propagowana przez samego robota. Drugi sposób, najbardziej odpowiadający warunkom przemysłowym, to podłączenie do istniejącej już sieci bezprzewodowej i odwoływanie się do urządzenia za pomocą przypisanego mu adresu IP (ang. *Internet Protocol*). Ostatni ze sposobów polega na przewodowym połączeniu z komputerem robota, głównie stosowany w celach rozwojowych i serwisowych.

## 1.5 Biblioteki języka Python

### 1.5.1 Biblioteka Python-CAN

Python-CAN jest biblioteką wspomagającą obsługę magistrali CAN z poziomu aplikacji napisanych w języku Python. Została napisana przez zespół z firmy Dynamic Controls, zajmującej się produkcją zaawansowanych systemów sterowania do wózków inwalidzkich. W 2010 roku projekt przejął i udostępnił do ogólnego użytku Brian Thorne. Od tamtej pory biblioteka działa na licencji *LGPL* (*GNU Lesser General Public License*), co pozwala na wykorzystywanie jej w celach komercyjnych. Rozwój tego rozwiązania przebiega na zasadach wolnego oprogramowania, co skutkuje sporą społecznością skupioną wokół projektu [10].

Pakiet zapewnia funkcjonalności, potrzebne do podstawowej komunikacji. Może być wykorzystywany wszędzie tam, gdzie jest zainstalowany interpreter Python i istnieje podłączenie do magistrali CAN. W bibliotece zaimplementowane zostało wsparcie dla wielu rodzajów interfejsów. Przykład wykorzystania obsługi najpopularniejszego z nich - *SocketCAN*, zostało przedstawione na poniższym listingu.

```
import can

can_interface = "can0"
bus = can.interface.Bus(can_interface, bustype='socketcan')

message = can.Message(arbitration_id=0x0000aa,
                      data=[0, 0, 0, 0],
                      extended_id=False)
bus.send(message)
```

```
message = bus.recv()
```

Listing 1.1: Program wysyłający i odbierający przykładową wiadomość przez magistralę CAN

Wyżej opisany program wysyła czterobajtową paczkę danych, wykorzystując do tego interfejs o nazwie *can0*. Wiadomość składa się z identyfikatora o standardzie CAN 2.0A (patrz: podsekcja 1.2.3), zapisanego w formacie heksadecymalnym. Ostatnia linia kodu oczekuje na pojawienie się jakiejkolwiek wiadomości i jej zapis do obiektu *message*.

### 1.5.2 Biblioteka Flask

Flask jest platformą programistyczną (ang. *framework*), wspierającą tworzenie aplikacji internetowych w języku Python. Charakterystyczną cechą tego projektu jest prostota używania. Twórcą Flask jest Armin Ronacher ze zrzeszającej programistów społeczności Pocoo. Działa na zasadach wolnego oprogramowania i licencji *BSD*, co czyni go dostępnym dla projektów komercyjnych.

Biblioteka skupia w sobie dwa podstawowe moduły - *Werkzeug* oraz *Jinja*. Pierwszy z nich to moduł narzędzi WSGI (ang. Web Server Gateway Interface), obsługujący zapytania HTTP (ang. *Hypertext Transfer Protocol*) pomiędzy serwerem aplikacji a klientem. Druga część biblioteki to silnik szablonów dla języka Python, który pozwala na rozbudowanie warstwy prezentacyjnej aplikacji o część dynamiczną [5].

Funkcjonalności dostarczane razem z platformą Flask to między innymi:

- serwer rozwojowy i analizator błędów,
- wsparcie dla testów jednostkowych,
- obsługa żądań REST,
- wsparcie dla bezpiecznych plików cookie.

Budowa aplikacji opartych na bibliotece Flask sprowadza się do sprzęgania ze sobą metod z języka Python oraz zapytań HTTP. Przykładem kodu, który generuje prostą prezentację działania platformy jest listing 1.2.

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def index():
    return "Witaj swiecie!"
```

Listing 1.2: Aplikacja wyświetlająca napis "Witaj swiecie!" na ekranie przeglądarki internetowej.

## 1.6 Przegląd rozwiązań zadania zdalnej konfiguracji urządzeń

Konfiguracja jest nieodłącznym procesem produkcji i użytkowania każdego urządzenia. Do prawidłowej jego pracy konieczna jest przynajmniej w postaci podstawowej. Rozwiązania przedstawione

w literaturze i źródłach internetowych skupiają się głównie na aspektach zdalnej, wysokopoziomowej konfiguracji. Wynika to z braku dużego zapotrzebowania na zagłębianie się w wewnętrzną strukturę modułów elektronicznych przez użytkowników. Dostęp niskiego poziomu, wiąże się najczęściej z połączeniem przewodowym i wykorzystaniem odpowiedniego protokołu komunikacyjnego (np. *MODBUS* stworzony przez firmę Modicon do komunikacji z programowalnymi kontrolerami).

Zapotrzebowanie na zdalną konfigurację i diagnostykę urządzeń występuje po stronie producentów i integratorów. Z tego powodu powstały rozwiązania bardziej zbliżone do omawianego zagadnienia.

### 1.6.1 Narzędzie CANberry

Pierwszym z opisywanych rozwiązań jest narzędzie **CANberry**. Jest to prosta aplikacja, napisana w skryptowym języku Python, do zdalnej konfiguracji falowników *MOVIDRIVE*, firmy *SEW EURODRIVE*. Zrealizowana jako aplikacja internetowa osadzona na mikrokomputerze *Raspberry Pi*, pełniącym funkcje serwera. Komunikacja z urządzeniem jest możliwa dzięki magistrali CAN, obsługiwanej zarówno przez interfejs mikrokomputera i konfigurowany produkt.

Aplikacja zapewnia graficzny interfejs użytkownika w przeglądarce internetowej. Zdalna konfiguracja może być przeprowadzana po wcześniejszym bezprzewodowym połączeniu z siecią WiFi, w której znajduje się mikrokomputer Raspberry Pi. Narzędzie przechwytuje również odczyty z czujników urządzenia i umożliwia ich wyświetlenie w oknie przeglądarki.

Rozwiązanie korzysta z szeregu bibliotek pomocniczych. Jednym z nich jest między innymi przedstawione wcześniej narzędzie - *python-can* - do obsługi interfejsu CAN z poziomu interpretera języka. Oprócz tego, do zależności aplikacji wliczają się biblioteki ułatwiające rozwój technologii internetowych. Bardziej obszerny opis rozwiązania został przedstawiony na stronie domowej projektu [11].

### 1.6.2 Rozwiązania Przemysłu 4.0

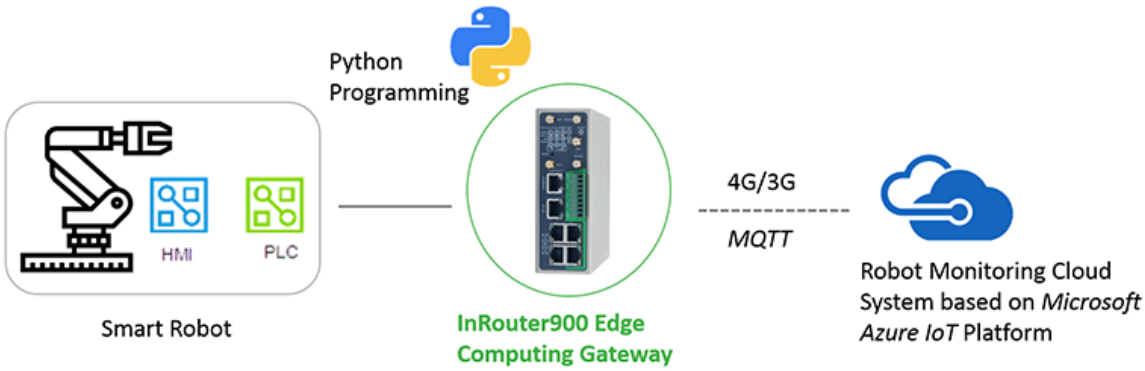
Współczesny przemysł coraz częściej jest w dużej części zautomatyzowany i zrobotyzowany. Zaawansowany technologicznie proces wymaga jednoczesnej diagnostyki i kontroli. Wprowadzanie idei Przemysłu 4.0 i obliczeń w chmurze pozwala na rozwiązanie problemu za pomocą akwizycji danych i wyświetlania ich w jednym, zunifikowanym systemie.

Firma *InHand Networks* opracowała aplikacje do zdalnego monitoringu i zarządzania systemem robotów przemysłowych. Rozwiązanie gromadzi dane z każdego urządzenia podłączonego do sieci i przedstawia je w postaci obrobionej. Dodatkowo zapewnia dostęp w czasie rzeczywistym do stanu robotów i pozwala na ich kontrolę. Aplikacja jest zrealizowana przy wykorzystaniu urządzeń *InRouter9000*, które odpowiadają za komunikację z maszynami przemysłowymi za pomocą wielu obsługiwanych protokołów i stanowią bramę do sieci Internet (chmury). Urządzenia te zbierają dane sensoryczne od programowalnych sterowników logicznych. Oprogramowanie napisane w języku Python, znajdujące się w systemie operacyjnym modułu *InRouter9000*, pozwala na odbiór danych z okresem 100 milisekund [13].

Tak zgromadzone dane przesyłane są do platformy *Microsoft Azure*. Z tego miejsca, zarówno odczyty jak i interfejs do kontroli, są dostępne dla użytkownika.

### 1.6.3 Konfiguracja sieciowa

Zadanie zdalnej konfiguracji pojawia się w wielu produktach, nie wymienionych w poprzednich podsekcjach. Przykładem mogą być moduły elektroniczne pełniące funkcje klientów sieci WiFi. Jed-




Rysunek 1.8: Architektura rozwiązania firmy InHand Networks [13]

nym z nich jest urządzenie ESP8266, które zostało wzbogacone o konfiguracyjną aplikację **WiFi Manager** [12]. Jest to rozwiązanie napisane w języku C++ z wykorzystaniem technologii internetowych. Zapewnia ingerencję w ustawienia modułów komunikacyjnych w postaci interfejsu użytkownika, wygenerowanego w przeglądarce internetowej. Konfiguracja odbywa się bezpośrednio przez sieć bezprzewodową.

# Rozdział 2

# Cel i założenia pracy

Celem pracy jest stworzenie wielopatformowej aplikacji do zdalnej konfiguracji robota i jego modułów elektronicznych w czasie rzeczywistym.

Podstawowym założeniem projektu jest występowanie komputera pokładowego, znajdującego się w robocie. System operacyjny, sterujący procesami obliczeniowymi, jest dystrybucją systemu operacyjnego *GNU/Linux* i umożliwia pracę z poziomu powłoki. To urządzenie powinno być przystosowane do obsługi serwera aplikacji, napisanego w języku Python. Z tego powodu zakłada się obecność interpretera tego języka, a także możliwość stabilnego połączenia sieciowego pomiędzy komputerem pokładowym oraz komputerem użytkownika.

Założenia, które obowiązują dla komputera użytkownika łączącego się z serwerem aplikacji, to między innymi obecność przeglądarki, obsługującej najnowsze standardy internetowe. Należy do nich asynchroniczny przesył danych pomiędzy częścią prezentacyjną a częścią obliczeniową oraz interpretacja języków JavaScript (*ECMAScript 2015*), HTML 5 i CSS 3. Dopuszcza to użycie zarówno komputera, jak i innych współczesnych urządzeń przenośnych.

Zakłada się, że system modułów elektronicznych robota jest połączony w sieć, zrealizowaną za pomocą magistrali CAN i protokołu CAN 2.0B. Każde urządzenie, w tym komputer pokładowy, posiada przypisany adres, umożliwiający identyfikację. Założeniem modułów elektronicznych jest budowa mikroprocesorowa z obsługą pamięci nieulotnej. Również wszystkie układy podłączone do magistrali CAN powinny być zaprogramowane tak, aby interpretować i odpowiednio przygotowywać wiadomości w formacie ustalonym w rozdziale 3. Jest to podstawą do odczytu danych zapisanych w rejestrach pamięci urządzeń, a także ich aktualizacji.

Dodatkowo aplikacja zakłada, że struktura rejestrów w pamięci konfigurowanych urządzeń jest znana i opisana w plikach o formacie XML. Każdy z nich, odpowiednio spreparowany, zawiera również wymagane parametry komunikacyjne i pozostały opis urządzenia (wielu urządzeń).

Zakres prac obejmuje zaprojektowanie struktury aplikacji oraz implementację jej dwóch warstw: klienta oraz serwera wraz ze skryptami instalacyjnymi i uruchomieniowymi. Wykonane zostaną również testy sprawdzające poprawność wszystkich funkcjonalności. Przebieg eksperymentów nastąpi dwuetapowo. Pierwszy etap odbędzie się przy użyciu specjalnie przygotowanego układu, symulującego system modułów elektronicznych robota. Druga część testów polegać będzie na sprawdzeniu funkcjonowania aplikacji na robocie mobilnym Kanboy.

# Rozdział 3

# Opis aplikacji

## 3.1 Opis ogólny

Aplikacja została stworzona przy wykorzystaniu technologii internetowych (ang. *web application*). Rozróżniany jest podział na część klienta (ang. *Front end*), odpowiedzialną za prezentację danych oraz za obsługę interakcji z użytkownikiem. Druga z części to warstwa serwera (ang. *Back end*), odpowiedzialna za komunikację przez magistralę CAN, główne obliczenia i przygotowanie danych w formie dopasowanej do klienta. Przekazywanie danych pomiędzy serwerem aplikacji i przeglądarką odbywa się przy pomocy formatu *JSON* (ang. *JavaScript Object Notation*) oraz akcji HTTP (ang. *Hypertext Transfer Protocol*) - *GET* i *POST*.

Konfiguracja i odczyt odbywa się za pomocą interfejsu graficznego. Aplikacja dostarcza możliwość wyboru typu urządzenia do konfiguracji, a także wielu urządzeń jednocześnie. Pomiar i zapis wszystkich wartości odbywa się automatycznie w tle lub na żądanie. Użytkownik ma możliwość dostosowania częstotliwości komunikacji w zależności od potrzeb. Istnieje możliwosć jednoczesnego zapisu wielu danych. Opisy struktury rejestrów w modułach elektronicznych i podstawowe parametry komunikacji są odczytywane z odpowiednio przygotowanych plików konfiguracyjnych w formacie *XML*. Użytkownik ma możliwość zmiany parametrów i ich zapisu do pliku o tym samym formacie. Ponadto, ustalone wartości rejestrów mogą być zapisane oraz odczytane z pliku o formacie *CSV*.

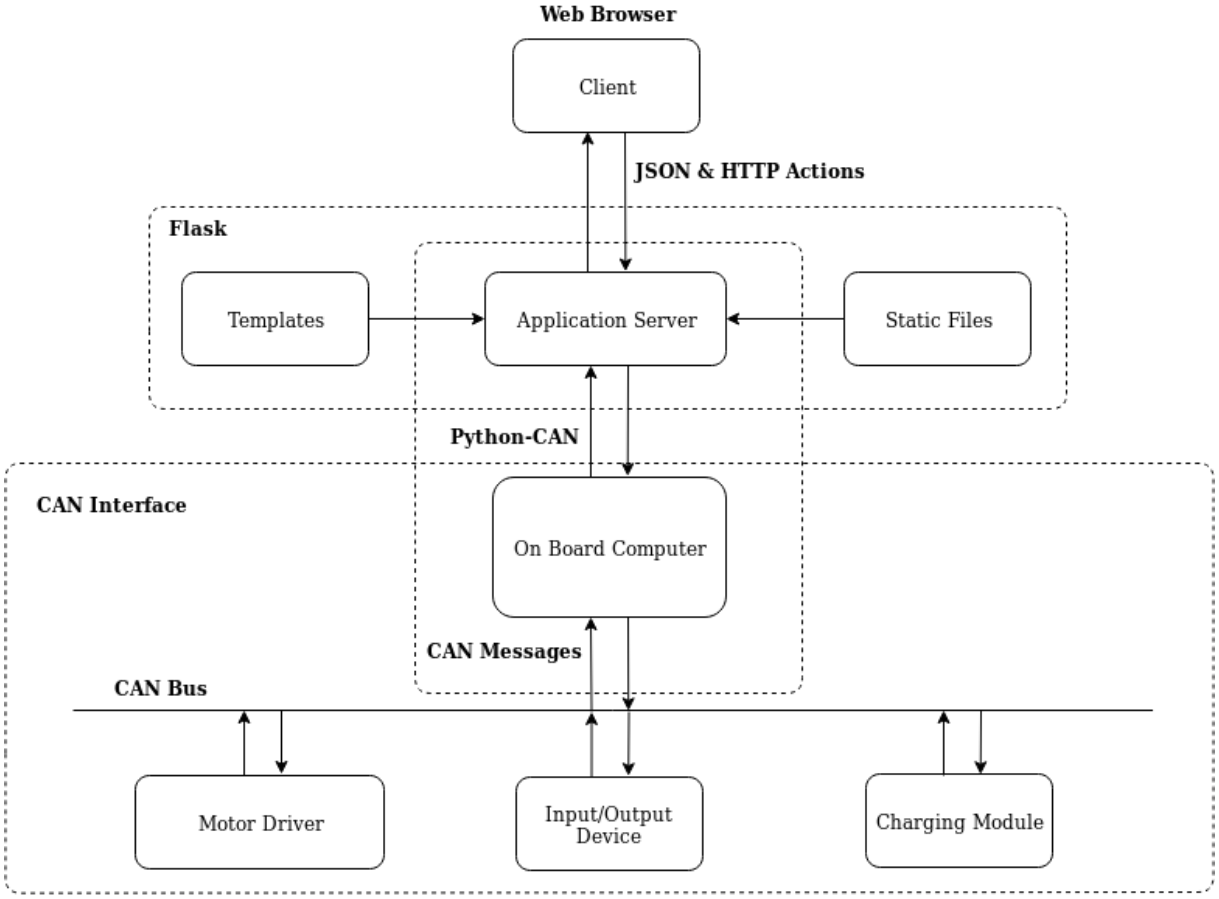
Tabela 3.1: Opis zależności projektu

| **Klient** | **Serwer** |
|---|---|
| HTML 5 + CSS 3 | Python 3.2 |
| JavaScript ECMAScript 2015 | Flask 1.0.2 |
| jQuery 3.1 | Python-CAN 2.1.0 |

Serwer aplikacji jest przystosowany do działania na systemach wbudowanych oraz zwykłych systemach operacyjnych zbudowanych na bazie jądra Linux. Kod źródłowy warstwy serwera jest napisany w języku interpretowanym *Python* w wersji 3.2, wraz z użyciem platformy programistycznej (ang. *Framework*) *Flask 1.0.2* [5] zapewniającej niezbędny interfejs do stworzenia aplikacji internetowej. Komunikacja pomiędzy serwerem a magistralą CAN jest zrealizowana z wykorzystaniem biblioteki *Python-CAN 2.1.0* [10]. Obie biblioteki działają na podstawie licencji umożliwiających darmowe wykorzystywanie w celach komercyjnych.

Interfejs graficzny aplikacji jest zaprojektowany jako automatycznie dostosowujący się do rozdziel-

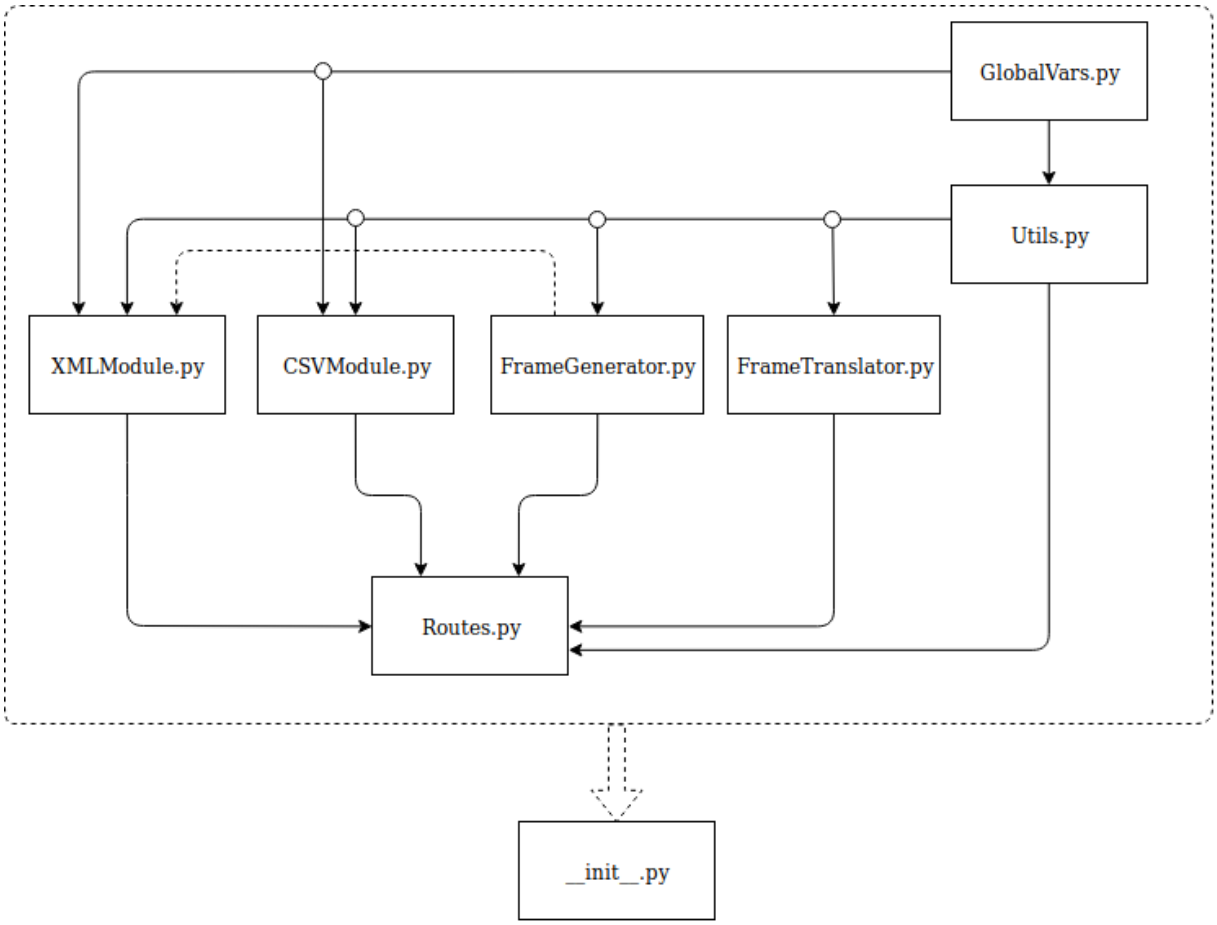
czości monitora użytkownika. Górne ograniczenie rozdzielczości nie istnieje. Jedynym ograniczeniem jest szerokość ekranu, która w pikselach musi wynosić minimalnie 1024 piksele. Warstwa wizualna w części klienta jest zrealizowana za pomocą języka zarządzania strukturą - *HTML 5* (ang. *Hypertext Markup Language*) - i oprawiona szatą graficzną za pomocą arkuszy stylów kaskadowych (*CSS 3* - ang. *Cascading Style Sheets*). Za interakcję z użytkownikiem i obsługę zdarzeń odpowiedzialna jest sekcja napisana w języku skryptowym *JavaScript* w wersji *ECMAScript 2015*, przy użyciu biblioteki *jQuery 3.1* [9]. Wspomniana biblioteka umożliwia jej powszechne użycie, pod warunkiem zamieszczenia odnośnika do głównej strony twórców w projektach.




Rysunek 3.1: Struktura aplikacji z przykładowymi urządzeniami elektronicznymi

## 3.2 Warstwa serwera

Wszystkie obliczenia znaczące dla komunikacji są przeprowadzane przez serwer aplikacji. Jest to element przetwarzający żądania użytkownika na wiadomości magistrali CAN (i odwrotnie). Ta część rozwiązania jest podzielona na moduły. Każdy z nich jest odpowiedzialny za inną część systemu, przez co kod źródłowy aplikacji jest bardziej czytelny. Struktura plików w warstwie serwera i zależności pomiędzy nimi zostały przedstawione na rysunku 3.2.




Rysunek 3.2: Podział warstwy serwera na moduły

### 3.2.1 Moduł inicjujący

Podstawową częścią aplikacji napisanej w języku Python jest plik **__init__.py**. Zawiera on wszystkie niezbędne dane o aplikacji i informuje interpreter, że dany katalog powinien być traktowany całościowo. Inicjuje również obiekt serwera oraz ustawienia aplikacji w postaci słownika. Parametry, które są przetrzymywane to: ścieżka względna do tymczasowego folderu z zapisywanymi plikami *XML* i *CSV* oraz lista obsługiwanych typów zmiennych.

```
app.config['FILES_FOLDER'] = 'files'
app.config['REGISTER_TYPES'] = {'int32_t': 32,
                                'int16_t': 16, 'uint16_t': 16}
```

Listing 3.1: Konfiguracja aplikacji w postaci par klucz - wartość

### 3.2.2 Klasa GlobalVars

Wszystkie wartości, które są wyświetlane w oknie przeglądarki, zostały zamknięte w jednej klasie - **GlobalVars**. Opisywana klasa została zaimplementowana jako kreacyjny wzorzec projektowy - *singleton*. Taki sposób reprezentacji danych wymusza istnienie tylko jednej instancji klasy i upraszcza dostęp do interesujących aplikację wartości. Podczas dowolnej próby odwołania się do obiektu *GlobalVars*, sprawdzane jest jego istnienie i w razie jego braku - tworzenie instancji.

Wartości przechowywane przez instancję klasy to między innymi opis wszystkich pól z mapy rejestrów konfigurowanych urządzeń oraz ich parametry. Obiekt przechowuje również ustawienia komunikacji po magistrali CAN oraz opcje aplikacji, wybrane przez użytkownika.

Metody składające się na definicję *GlobalVars* to:

- *get_instance()* - statyczna metoda, charakterystyczna dla wzorca singletonu, zwracająca jedyną instancję klasy,
- *__init__(self)* - wirtualnie prywatny konstruktor, używany przez metodę opisaną powyżej.

### 3.2.3 Moduł narzędzi pomocniczych

Pomoce programistyczne, ułatwiające rozwój oprogramowania zostały umieszczone w oddzielnym module - **Utils.py**. Plik ten jest źródłem, do którego odnoszą się wszystkie pozostałe części systemu. Złożony jest z funkcji pomocniczych, oferujących obsługę szeregu działań na liczbach binarnych, funkcji konwertujących oraz tych wykorzystywanych przez rejestrator komunikatów.

Obsługa konwersji liczb pomiędzy systemami: dwójkowym i dziesiętnym, została zrealizowana przy pomocy funkcji:

- *format_to_bin_from(value, type)* - funkcja konwertująca liczbę dziesiętną na liczbę binarną o odpowiedniej długości. Jako argumenty przyjmuje *value* - liczbę całkowitą oraz *type* - typ zmiennej. Na podstawie typu zmiennej i słownika konfiguracji opisanego w podsekcji 3.2.1 determinuje długość zwracanej liczby binarnej.
- *format_to_bin_with_len(value, length)* - funkcja analogiczna do poprzedniej z różnicą w postaci argumentu długośći - *length* zamiast typu zmiennej.
- *insert_bin_zeros(binary_value, length)* - funkcja uzupełniająca liczbę binarną (*binary_value*) tak, aby była długości podanej w argumencie (*length*). Brakujące bity są uzupełniane logicznymi zerami. Jest wykorzystywana przez wyżej wymienione funkcje.
- *not_binary(binary_value)* - wywołanie tej funkcji przeprowadza operację logicznego zaprzeczenia dla wszystkich bitów wartośći binarnej.

Oprócz funkcji matematycznych, moduł *Utils.py* zawiera funkcje niezbędne do konwersji wartości z rejestrów złożonych. Funkcje *value_to_values* oraz *values_to_value* przeprowadzają operacje rozbijania jednej wartości na wiele mniejszych (moment interpretacji otrzymanej wiadomości CAN) i składania wielu wartości w jedną dużą (moment przygotowania danych do transmisji). Przyjmują za argumenty odniesienie do rejestru w postaci jego numeru oraz indeksu urządzenia i indeksu zakładki, a także wartość do konwersji (lub wartości w przypadku przygotowywania danych do transmisji).

Moduł *Utils.py* składa się również z funkcji używanych przy komunikacji z użytkownikiem tj. *print_error*, *print_warning*, *print_info*. Funkcje te odpowiednio wyświetlają komunikat o błędzie, ostrzeżenie oraz informacje w przygotowanej dla rejestratora wiadomości postaci.

### 3.2.4 Obsługa plików XML

Aplikacja została przygotowana do konfiguracji wielu różnych typów urządzeń wraz z możliwością zwiększania ich liczby. W tym celu stworzony został moduł obsługujący odczyt i zapis plików *XML*.

Pliki te zawierają dokładny opis rejestrów w pamięci urządzenia konfigurowanego, a także parametry niezbędne przy komunikacji i domyślnie ładowane wartości. Serwer aplikacji interpretuje dane przesłane od użytkownika i odpowiednio preparuje środowisko konfiguracyjne do dalszego działania. Zmiana parametrów komunikacji jest możliwa do zapamiętania, poprzez zapis nowego pliku i jego pobranie.

```
<Configuration>
    <Slave address="64" description="I/O">
        <RegisterSet autoread="autoread" name="Tab"
            autoreadPeriod="100">
                <Register hidden="N" address="0"
                    description="Device ID" type="uint16_t"
                    widget="typed" access="R" />
                <Register hidden="N" address="1"
                    description="Device Version" type="uint16_t"
                    widget="typed" access="R" />
                <Register hidden="N" address="2"
                    description="Device Status" type="uint16_t"
                    widget="flag" access="R" />
                <Register hidden="N" address="3"
                    description="Digital Inputs" type="uint16_t"
                    widget="flag" access="R"/>
                <Register hidden="N" address="4"
                    description="Analog Input 1" type="uint16_t"
                    widget="cont" unit="mV" min="0" max="32767"
                    access="R"/>
                <Register hidden="N" address="5"
                    description="Analog Input 2" type="uint16_t"
                    widget="cont" unit="mV" min="0" max="32767"
                    access="R"/>
                <Register hidden="N" address="6"
                    description="Digital Outputs" type="uint16_t"
                    widget="flag" access="RW" f0="1" f1="2" f2="3"
                f3="4" f4="5" f5="6" f6="7" f7="8" f8="9" f9="10"
                f10="11" f11="12" f12="13" f13="14" f14="15"/>
        </RegisterSet>
    </Slave>
</Configuration>
```

Listing 3.2: Przykład pliku do konfiguracji jednego prostego modułu wejść/wyjść

Plik przedstawiony na listingu 3.2 odnosi się do konfiguracji z tylko jednym urządzeniem (ang. *Slave*) o adresie 64 w magistrali CAN. Komunikacja z nim odbywa się z poziomu jednej zakładki konfiguracyjnej (ang. *Register set*) z domyślnymi parametrami - automatycznym odczytem w tle co 100 milisekund. Wszystkie rejestry są opisane i przypisane do adresu. Dalszy opis możliwych pól został przedstawiony w rozdziale 4.

Cała funkcjonalność została zrealizowana w pliku *XMLModule.py*. Zapis i odczyt struktury re-

jestrów odbywa się za pomocą dwóch głównych funkcji: *load_file* i *save_file*. Argumentem pierwszej z nich jest obiekt przechowujący plik wczytany przez użytkownika, a drugiej - nazwa docelowego pliku do zapisu.

Funkcja wczytująca podczas działania czyści wszystkie obiekty (oprócz rejestratora komunikatów) i przygotowuje aplikację do działania z nową konfiguracją. Następnie, korzystając z domyślnych bibliotek języka Python, interpretuje plik *XML* i na jego podstawie tworzy nową strukturę wszystkich obiektów. Dodatkowo w tym celu napisane zostały funkcje pomocnicze *load_slaves_and_button* oraz *read_tab_configuration*. Kod pierwszej zajmuje się wczytaniem informacji o konfigurowanych urządzeniach, a drugiej - parametrów konfiguracyjnych dla zadanej zakładki.

Dostęp do wszystkich rejestrów odbywa się za pomocą klucza trójwartościowego (indeks urządzenia, indeks zakładki, adres rejestru). Każdy z nich przechowuje swoją aktualną wartość i jest opisany za pomocą:

- typu zmiennej (16-bitowa, 32-bitowa, ze znakiem lub bez),
- typu dostępu (odczyt, zapis, odczyt i zapis),
- typu pola (więcej w rozdziale 4),
- flagi, określającej czy rejestr jest ukryty.

Ponadto podczas wczytywania pliku aktualizowana jest maska komunikacyjna magistrali CAN. Wczytanie skonkretyzowanych adresów urządzeń umożliwia ustawienie odbieranych wiadomości o pożądanym identyfikatorze. Procedura odbywa się za pomocą modułu *FrameGenerator.py* opisanego w podsekcji 3.2.6 i biblioteki Python-CAN.

Częścią odpowiadającą za zapis zmodyfikowanego opisu komunikacji jest funkcja *save_file*. Generuje ona nowy plik *XML*, na podstawie danych znajdujących się w obiektach przechowywanych w serwerze aplikacji. Umożliwia to zapis nowej konfiguracji, np. dla urządzeń o zmienionych adresach lub z innymi parametrami komunikacyjnymi. Pomyślne wykonanie zwraca wygenerowany plik i przekazuje go do pobrania za pomocą przeglądarki.

### 3.2.5 Odczyt i zapis domyślnych wartości rejestrów

Aplikacja do zdalnej konfiguracji modułów elektronicznych robota została wyposażona w dodatkowy moduł obsługujący pliki z wartościami rejestrów. Umożliwiany jest zarówno odczyt, jak i zapis. Daje to możliwość przechowania domyślnej konfiguracji danego urządzenia i jej transfer do innych urządzeń, bez potrzeby ponownego ustawiania.

Funkcjonalność została zrealizowana za pomocą modułu *CSVModule.py*. Obsługiwane pliki zbudowane są według formatu CSV (ang. *Comma-Seperated Values*). Każdy z wierszy odpowiada jednemu z rejestrów urządzenia i składa się z czterech wartości: adresu urządzenia w magistrali CAN, indeksu zakładki konfiguracyjnej, adresu rejestru w pamięci urządzenia oraz docelowej wartości rejestru.

Moduł jest złożony z dwóch funkcji. Pierwsza z nich - *csv_save_file* - odpowiada zapisowi wartości do pliku *CSV*. Przyjmuje docelową nazwę pliku jako argument i preparuje odpowiedni format danych, korzystając z zawartości obiektu przechowującego wartości rejestrów. Za pomocą modułu funkcji pomocniczych, informuje użytkownika o powodzeniu procesu. W przypadku pomyślnego zapisu danych, zwraca plik z możliwością pobrania do przeglądarki. Druga funkcja - *csv_load_file* - zajmuje się odczytem wcześniej zapisanych plików i ich interpretacją. Sprawdza kompatybilność z wgraną strukturą

rejestrów i nadpisuje obiekt przechowujący zawartości rejestrów. Podobnie jak funkcja poprzednia, zwraca komunikat o statusie.

### 3.2.6 Generowanie i translacja ramek CAN

Moduł zajmujący się tworzeniem ramek CAN składa się z funkcji generującej odpowiedni nagłówek wiadomości, dla zadanych argumentów. Struktura nagłówka odpowiada standardowi CAN 2.0B. To oznacza, że jest on ciągiem 29 bitów. Urządzenia podłączone do magistrali CAN identyfikują wiadomości na jego podstawie, zatem konieczne jest zawarcie w nim odpowiednich informacji.

W tym celu identyfikator wiadomości złożony jest między innymi z informacji o typie funkcji wiadomości (zapis lub prośba o odczyt), adresie urządzenia docelowego i typie urządzenia. Wygenerowany nagłówek jest zwracany przez funkcję *gen* jako wartość całkowita w systemie dziesiętnym (w formie przystosowanej do biblioteki Python-CAN).

Tabela 3.2: Struktura wygenerowanego nagłówka w module *FrameGenerator.py*

| **Numer funkcji (6 bitów)** | **0** | **Typ urządzenia (4 bity)** | **00** | **Adres urządzenia (8 bitów)** | **Indeks wiadomości (7 bitów)** | **0** |
|---|---|---|---|---|---|---|

W zależności od pożądanej wiadomości, numer funkcji równy 50 odpowiada wysłaniu danych do zapisu, a 51 - prośbę o odczyt wartości. Zarówno typ urządzenia, jak i adres są wyznaczane na podstawie opisu znajdującego się w pliku XML. Tak przygotowany identyfikator wiadomości jest zwracany i wykorzystywany przez moduł główny aplikacji.

Treść wiadomości CAN, czyli część występująca po nagłówku, jest złożona z pojedynczych bajtów. Pierwszy z nich określa adres rejestru (do zapisu lub odczytu). Drugi oznacza długość rejestru (wielkość zmiennej przechowywanej przez rejestr - 2 lub 4 bajty). W tym miejscu dane wiadomości z prośbą o odczyt danych są zakończone. Jednakże, w przypadku zapisu do urządzenia, po pierwszych dwóch bajtach następuje również przesłanie bajtów z pożądanymi wartościami. Liczba bajtów jest związana z długością rejestru.

Plik *FrameTranslator.py* zawiera zaimplementowane funkcjonalności, używane przy odczycie otrzymanych wiadomości CAN. Składa się z dwóch funkcji: translacji nagłówka ramki, oraz translacji danych. Pierwsza z nich, na podstawie otrzymanej liczby reprezentującej nagłówek, zwraca odpowiednio przygotowany słownik. Składa się on z par klucz-wartość określających:

- typ funkcji otrzymanej wiadomości,
- typ urządzenia nadawczego,
- indeks wiadomości,
- adres urządzenia nadawczego.

Tak spreparowane dane są przekazywane do interpretacji przez inne moduły w aplikacji.

Druga funkcja, znajdująca się w omawianym pliku, zajmuje się składaniem danych otrzymanych w treści wiadomości CAN w całość. Część *translate_data* przyjmuje jako argument listę bajtów oraz informację o tym, czy dana wartość jest liczbą oznakowaną. Wyłuskuje dane o adresie i długości rejestru, a z pozostałych bajtów składa jedną liczbę, która reprezentuje wartość w pamięci urządzenia.

Przekazuje pozyskane informacje do innych części systemu za pomocą słownika par klucz-wartość (adres rejestru, długość rejestru oraz wartość).

### 3.2.7 Moduł główny

Modułem głównym serwera i trzonem całej aplikacji jest plik **Routes.py**. Zawiera on obsługę żądań użytkownika oraz łączy wszystkie części systemu w całość. Funkcje zaimplementowane w tym module można rozdzielić na wykonywane synchronicznie oraz asynchronicznie.

Głównymi częściami modułu, zwracającymi strony wyświetlane klientowi są funkcje *index* oraz *config_tabs*. Pierwsza z nich generuje szablon powitalny (wykonuje się po wpisaniu w przeglądarkę głównego adresu HTTP serwera), a druga wyświetla formularze konfiguracyjne rejestrów i jest wywoływana podczas poruszania się po zakładkach aplikacji.

Funkcja obsługująca wczytanie mapy i struktury rejestrów to *load_file_and_render*. Przechwytuje ona za pomocą zapytania HTTP plik XML przesłany przez użytkownika i przetwarza go za pomocą modułu opisanego w podsekcji 3.2.4. Dodatkowo, wykorzystując dane zawarte w mapie rejestrów, ustala podstawowe parametry komunikacji CAN, przy pomocy funkcji *get_networks_available*. Ta metoda sprawdza dostępne interfejsy komunikacyjne w systemie (w tym interfejsy magistrali CAN). Działa na zasadzie wywołania polecenia z systemu Linux `ifconfig -s` i odpowiedniej obróbki łańcucha znaków.

Moduł *Routes.py* zawiera również trzy dodatkowe funkcje obsługujące żądania operacji na plikach:

- *save_xml_file* - funkcja zapisująca plik XML, który jest aktualnie wykorzystywany. Pozwala na pobranie mapy rejestrów przez użytkownika, po ewentualnych zmianach w parametrach komunikacyjnych. Korzysta z modułu *XMLModule.py*.
- *csv_saver* - obsługa żądania zapisu wszystkich aktualnych wartości z rejestrów urządzenia. Zwraca gotowy plik z możliwością pobrania przez użytkownika, posługując się modułem *CSVModule.py*.
- *csv_loader* - funkcja przechwytująca plik CSV od użytkownika, zawierający wartości rejestrów. Wywołuje funkcję z modułu *CSVModule.py*.

Kolejną funkcją synchroniczną zawartą w tej części systemu jest *change_addresses*. Przechwytuje ona akcję *POST* od klienta, z prośbą o zmianę adresów komunikacyjnych urządzeń i potrzebnymi danymi. Wywołanie funkcji powoduje aktualizację danych i globalne wyłączenie komunikacji (w celach bezpieczeństwa na wypadek pomyłki).

Funkcja *submit* jest jedyną synchroniczną częścią kodu wywoływaną w celach komunikacyjnych po magistrali CAN. Odbiera dane o wszystkich ustawionych wartościach rejestrów w przeglądarce użytkownika za pomocą akcji *POST*. Następnie, interpretuje je i uaktualnia obiekty, które przechowują te wartości. Po wszystkich operacjach następuje przesłanie nowych wartości rejestrów poprzez magistralę CAN do układu elektronicznego. W tym celu, wykorzystywane są funkcje *background_sender* oraz adapter - *sender* - do iteracji po całej strukturze rejestrów, zmodyfikowanej przez użytkownika.

Druga część podziału funkcji w tym module, czyli część asynchroniczna to między innymi funkcja *background_submit*. Interpretuje ona i obsługuje ustawienie nowo wprowadzonej wartości rejestru przez użytkownika i zapamiętanie w obiekcie przechowującym. W wypadku zaznaczonej opcji automatycznego wysyłania, funkcja wywołuje wspomniany wcześniej *background_sender* w celu przesłania nowej wartości do urządzenia elektronicznego.

Transfer nowych informacji odbywa się przy wykorzystaniu funkcji pomocnicznych z modułu *Utils.py*, modułów do translacji i generowania ramek wiadomości oraz biblioteki *Python-CAN*. Na samym początku wywołania generowany jest nagłówek wiadomości oraz wartość do przesłania (konieczna jest konwersja, gdy dany rejestr jest złożony). W zależności od typu rejestru i typu przechowywanych w nim wartości, w funkcji generowany jest łańcuch zawierający wartość zapisaną w formacie binarnym. Następnie ta wartość jest dzielona na paczki bajtów i układana w odpowiedniej kolejności w ciągu danych. Tak przygotowana treść główna wiadomości ma strukturę [adres rejestru, liczba bajtów, pierwszy bajt wartości, drugi bajt wartości, ...]. Dodatkowo funkcja *button_sender* jest bliźniacza w stosunku do powyższej. Implementuje funkcjonalność konfigurowalnego przycisku i obsługuje przesył danych do aktywnego urządzenia.

Pozostałą funkcją komunikacyjną jest *background_reader*, czyli obsługa odczytu zawartości rejestrów w układzie elektronicznym. Wywołanie powoduje aktualizację widocznej części obiektu przechowującego wartości i zwrócenie go klientowi (przeglądarce). Odczyt zawartości odnosi się tylko do rejestrów, które aktualnie znajdują się w widocznym obszarze przeglądarki. Algorytm działania tej części aplikacji jest następujący i jest wykonywany dla każdego pożądanego rejestru:

- przygotowanie wiadomości z prośbą o udostępnienie informacji przez układ elektroniczny (przesyłając pożądany adres rejestru i liczbę spodziewanych bajtów),
- oczekiwanie na wiadomość zwrotną,
- w przypadku powodzenia - interpretacja danych i zapisanie w pamięci,
- w przypadku niepowodzenia - wyświetlenie komunikatu o błędzie w odczycie danego rejestru.

Proces odczytu wartości jest ograniczony dwiema wartościami czasowymi. Pierwsza z nich to maksymalny czas oczekiwania na wiadomość - 100 milisekund. W przypadku braku informacji zwrotnej od urządzenia (braku wiadomości o odpowiednim identyfikatorze) w tym czasie, funkcja informuje o błędnym odczycie danego rejestru i przechodzi do następnego. Druga wartość to maksymalny czas oczekiwania na wszystkie wiadomości podczas asynchronicznego odczytu w tle, co pewien okres czasu. Aktualizacja wszystkich danych nie może trwać dłużej niż ustawiony interwał. Z tego powodu funkcja przerywa swoje działanie w przypadku przekroczenia czasu oczekiwania i odpowiednio informuje użytkownika. Taka sytuacja nie występuje podczas synchronicznego wywołania funkcji (używane podczas odczytu bardzo obszernych map rejestrów na zatłoczonych magistralach CAN). W ogólności funkcja kończy swoje działanie na cztery sposoby: odczyt udany, odczyt częściowo udany, przekroczenie czasu oczekiwania na wszystkie wiadomości, odczyt nieudany.

Ponadto, moduł *Routes.py* składa się z szeregu funkcji do interakcji z użytkownikiem:

- *log_op* - ustawiająca globalne ustawienia rejestratora komunikatów,
- *auto_read_submit* - zmieniająca parametry automatycznego odczytu zawartości rejestrów (stan oraz interwał),
- *clear_log* - czyszcząca wszystkie komunikaty z rejestratora,
- *log_add* - dodająca nowy komunikat od użytkownika do rejestratora,
- *auto_send_fun* - zmieniająca parametr determinujący, czy zmiany wartości rejestrów są wysyłane automatycznie do urządzenia,

- *on_off_fun* - włączająca lub wyłączająca globalną komunikację z interfejsem CAN,
- *set_connection* - zmieniająca wybrany interfejs do komunikacji,
- *get_log* - zwracająca zawartość rejestratora komunikatów w celu wyświetlenia w oknie przeglądarki.

## 3.3 Warstwa klienta

Część aplikacji, która pełni rolę klienta to dynamiczna strona internetowa wyświetlająca się w oknie przeglądarki. Po wpisaniu odpowiedniego adresu, serwer odsyła odpowiednio wygenerowany widok z interfejsem, na podstawie wcześniej stworzonych szablonów. Za część statyczną strony odpowiadają pliki napisane w języku HTML. Wygląd został ostylowany za pomocą języka CSS. Dynamika i obsługa żądań użytkownika jest zapewniana przez JavaScript i pomocniczą bibliotekę jQuery.

Wymiana informacji pomiędzy klientem i serwerem występuje pod dwiema postaciami. Pierwszym sposobem jest wymiana synchroniczna, czyli zachodząca po ponownym przeładowaniu strony. Taka komunikacja zachodzi w przypadku prośby o globalny odczyt lub zapis danych, a także przy pobieraniu i ładowaniu plików.

Ze względu na uciążliwość i brak możliwości podglądu zaktualizowanych danych w czasie rzeczywistym, został zaimplementowany drugi sposób komunikacji pomiędzy serwerem i użytkownikiem. Następuje on asynchronicznie (w tle) i nie przeszkadza w obsługiwaniu aplikacji. Wykorzystywany jest do aktualizacji wartości znajdujących się w rejestrach urządzenia elektronicznego, a także do natychmiastowego transferu wartości nowo wprowadzonych.

Taki sposób komunikacji nosi nazwę *AJAX* (ang. *Asynchronous JavaScript and XML*) i polega na asynchronicznym przesyle tekstu i jego interpretacji. W aplikacji umieszczono rozwiązanie w postaci przesyłanego formatu JSON. Do tego celu została wykorzystana biblioteka jQuery i zawarta w niej funkcja *jQuery.getJSON()*.

```
{
    'slave0_address14_bit0_2_0': 2
    'slave0_address14_bit3_10_1': 34
    'slave0_address14_bit11_15_2': 10
}
```

Listing 3.3: Przykład struktury przesyłanej pomiędzy serwerem i klientem

Powyższy listing pokazuje obiekt w formacie JSON zawierający informacje o nowych wartościach w opisanych polach. Pokazana struktura odnosi się do pól, które są częścią złożonego rejestru o adresie 14 w urządzeniu o indeksie 0 (więcej w rozdziale 4).

## 3.4 Pozostałe części systemu

Oprócz warstwy serwera i klienta, aplikacja składa się dodatkowych części. Jedną z nich są skrypty instalacyjne i uruchomieniowe. Skrypt `install.sh` odpowiada za zgromadzenie wszystkich potrzebnych zależności do uruchomienia serwera na komputerze pokładowym. W tym celu wywołuje szereg poleceń z wykorzystaniem powłoki i narzędzia instalacyjnego *pip*.

Dodatkowo, aby ułatwić użytkowanie, stworzony został skrypt `setup.sh`, który automatycznie uruchamia serwer aplikacji. Eksportuje on wszystkie potrzebne zmienne środowiskowe (ang. *environment variables*), po czym wywołuje polecenie `flask run` z odpowiednimi parametrami.

Ponadto cały kod aplikacji został udokumentowany w celach rozwojowych i utrzymaniowych. Dokumentacja jest możliwa do wygenerowania za pomocą narzędzia *doxygen* i konfiguracji zawartej w paczce instalacyjnej. Zawiera ona niezbędny opis wszystkich metod i obiektów w kodzie. Pozwala na bardziej dogłębne zrozumienie struktury występującej w aplikacji. Wygenerowana dokumentacja została załączona do elektronicznej wersji pracy.

# Rozdział 4

# Działanie aplikacji

## 4.1 Pierwsze uruchomienie

Aplikacja została stworzona w postaci paczki umieszczonej w internetowym repozytorium. W celach uruchomieniowych, konieczne jest jej pobranie i umiejscowienie na komputerze pokładowym docelowego robota. Podczas pierwszego startu aplikacji wymagana jest instalacja wszystkich potrzebnych zależności. Umożliwia to uruchomienie instalacji w postaci skryptu `install.sh`, który gromadzi wszystkie potrzebne biblioteki opisane w tabeli 3.1 i umieszcza w odpowiednich miejscach systemu. Po zakończeniu procesu instalacji, aplikacja (część po stronie serwera) jest gotowa do uruchomienia za pomocą skryptu `start.sh`.

Prawidłowe zainstalowanie zależności i uruchomienie serwera powinno skutkować wyświetleniem komunikatów przedstawionych na listingu 4.1:

```
* Serving Flask app "VBarWeb/vbarweb.py"
* Environment: development
* Debug mode: off
* Running on http://0.0.0.0:3853/ (Press CTRL+C to quit)
```

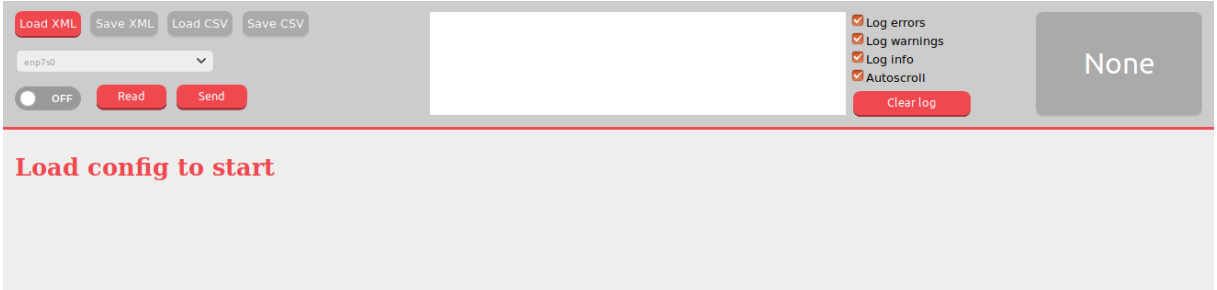
Listing 4.1: Komunikat o poprawnym starcie serwera aplikacji

Tekst komunikatów informuje użytkownika o załadowaniu ciała głównego aplikacji z pliku VBarWeb/vbarweb.py w trybie deweloperskim. Serwer wystartował z możliwością połączenia się z nim pod adresem IP (ang. *Internet Protocol*) równym adresowi IP komputera pokładowego.

Niezależnie od połączenia (wykorzystując interfejs bezprzewodowy *Wi-Fi* lub przewodowy *Ethernet*), od tego momentu aplikacja jest widoczna dla wszystkich użytkowników sieci, do której podłączony jest komputer pokładowy robota. Ze względów bezpieczeństwa zalecane są zabezpieczenia sieci, na przykład w postaci hasła dostępowego lub certyfikatów. Założeniem działania samej aplikacji jest fakt, że dostęp do robota jest umożliwiony tylko osobom do tego uprawnionym.

Włączona warstwa serwera na komputerze pokładowym pozwala użytkownikowi na uruchomienie klienta aplikacji na dowolnym komputerze spełniającym powyższe wymagania. W celu załadowania głównej strony aplikacji, należy uruchomić dowolną przeglądarkę obsługującą najnowsze, podstawowe standardy technologii internetowych i wpisać w pasku adresowym: `adres_robota:3853/` (podając jako adres_robota, jego adres IP, lub jeśli istnieje - domenę). Poprawne wczytanie strony powinno ukazać widok zbliżony do widoku na rysunku 4.1. Ewentualne rozbieżności mogą być spowodowane

różnicami w rozdzielczościach monitorów oraz wersjami systemów operacyjnych i przeglądarek. Zrzuty ekranu powstały na systemie operacyjnym *Ubuntu 18.04*, na monitorze o rozdzielczości 1366 pikseli na 768 pikseli i przy użyciu przeglądarki *Mozilla Firefox 63.0*.




Rysunek 4.1: Ekran główny aplikacji

Pierwszą informacją, którą uzyskuje użytkownik po uruchomieniu warstwy klienta, jest prośba o wczytanie pliku konfiguracyjnego opisującego strukturę rejestrów w pamięci modułu elektronicznego. Pożądany plik, o odpowiednim formacie *XML*, może być wczytany z komputera za pomocą przycisku **Load XML**, widocznego w lewym górnym rogu ekranu. Po sprawdzeniu poprawności danych wejściowych, aplikacja generuje interfejs pozwalający na dalszą pracę przy konfiguracji. Docelowe okno aplikacji widoczne jest na rysunku 4.2.


Rysunek 4.2: Okno widoczne po załadowaniu pliku konfiguracyjnego

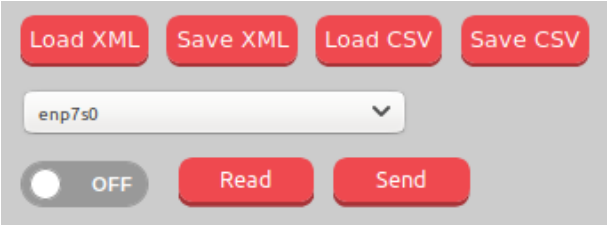
## 4.2 Opis interfejsu użytkownika

Okno, z którego składa się aplikacja do zdalnej konfiguracji modułów elektronicznych robota, jest podzielone na dwie części. Pierwsza z nich odpowiada za nadrzędny interfejs graficzny, zarządzający całą aplikacją. Druga część jest odpowiedzialna za wyświetlanie i obsługę docelowej konfiguracji sterowników. Większość elementów na stronie jest interaktywna i wymagająca szczegółowego opisu.

### 4.2.1 Nadrzędny interfejs użytkownika

Górna część ekranu jest zajęta przez warstwę aplikacji, dostępną od samego początku. Jeszcze przed załadowaniem pliku z opisem rejestrów modułu elektronicznego, niektóre z elementów są odblokowane i dostępne. Ten interfejs jest podzielony na trzy podstawowe elementy:

- część odpowiadająca za obsługę plików i komunikację z urządzeniami podłączonymi do magistrali CAN (rysunek 4.3),
- okno z bieżącymi komunikatami, dostarczające informacji na temat działania całej aplikacji (rysunek 4.5),
- konfigurowalny przycisk wszechstronnego użytku (rysunek 4.6).




Rysunek 4.3: Interfejs do obsługi plików i komunikacji

Na rysunku 4.3 w górnym wierszu widoczne są przyciski kolejno do odczytu i zapisu plików konfiguracyjnych w formacie *XML* oraz do odczytu i zapisu plików z wartościami rejestrów w formacie *CSV*. Kliknięcie w przyciski wczytujące pliki powoduje otwarcie standardowego dla przeglądarki okna do wyboru pliku z dysku komputera. Natomiast, chęć zapisu plików wywołuje pojawienie się okna modalnego, widocznego na rysunku 4.4.

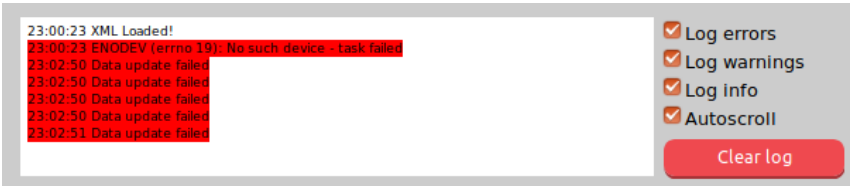

Rysunek 4.4: Okno pojawiające się po kliknięciu przycisków **Save XML** i **Save CSV** z prośbą o podanie nazwy pożądanego pliku

W środkowej części grupy interaktywnych elementów z rysunku 4.3 znajduje się rozsuwana lista z wykrytymi interfejsami komunikacyjnymi na komputerze pokładowym (komputerze z uruchomionym serwerem aplikacji). W celu rozpoczęcia pracy z programem, konieczne jest wybranie odpowiedniego interfejsu z połączeniem w postaci magistrali CAN. Na samym dole znajdują się elementy komunikacyjne: globalny przełącznik (zamykający lub otwierający transmisję) oraz przyciski do globalnego odczytu i zapisu wartości wszystkich aktualnie widocznych rejestrów, wybranego przez użytkownika odbiorcy.

Drugim z podstawowych elementów nadrzędnego interfejsu użytkownika jest rejestrator wiadomości od serwera aplikacji dla użytkownika (rysunek 4.5). Pojawiają się w nim wszystkie komunikaty o błędach, ostrzeżeniach oraz informacje o powodzeniach. Lista wszystkich możliwych wiadomości jest przedstawiona na poniższej liście:

- XML Loading passed! - pomyślne wczytanie mapy rejestrów,
- XML Saving passed! - informacja o prawidłowym zapisie pliku z mapą rejestrów i parametrami komunikacji,
- CSV Saving passed! - poprawny zapis aktualnych wartości rejestrów do pliku,
- CSV Loading passed! - odczyt wcześniej zapisanych wartości z pliku,
- Data approved - zapis nowej wartości rejestru/mapy rejestrów do serwera,
- Data updated - [Time] millis elapsed - pomyślny odczyt danych z magistrali CAN z informacją o czasie oczekiwania na wszystkie wiadomości,
- Interface change passed - informacja o pomyślnym wybraniu interfejsu komunikacyjnego (docelowy interfejs jest prawidłowo działającym interfejsem CAN),
- Data sent - poprawne wysłanie nowych wartości rejestrów przez magistralę,
- Reading partially failed (register [Number of register]) - timeout ocurred - ostrzeżenie o braku odpowiedzi ze strony urządzenia elektronicznego na zapytanie o konkretny rejestr z zaznaczeniem indeksu,
- XML Loading failed! - błąd wczytywania mapy rejestrów, związany z błędem parsowania pliku XML,
- XML Saving failed! - błąd zapisu mapy rejestrów i parametrów komunikacji,
- CSV Loading failed! - błąd odczytu wcześniej zapisanych wartości z pliku, wskazujący na zły format wczytywanego plilku,
- CSV Saving failed! - błąd zapisu aktualnych wartości rejestrów do pliku,
- Interface change failed - zmiana interfejsu komunikacyjnego zakończona niepowodzeniem (interfejs nieprawidłowo obsługuje magistralę CAN),
- [Error code] - task failed - błąd nieuwzględniony we wcześniejszych komunikatach z informacją o jego kodzie,
- Data sending failed - błąd podczas wysyłania wartości rejestrów,
- Refresh rate too high, data update failed - błąd informujący o zbyt dużej częstotliwości odświeżania odczytów.

Użytkownik ma również możliwość decyzji, które typy komunikatów są interesujące (błędy, ostrzeżenia lub informacje). Kolejną z dostępnych opcji jest automatyczne przewijanie do najnowszej wiadomości. Ta funkcjonalność jest bardzo pomocna przy dużej częstotliwości pojawiających się informacji. Ostatnim elementem tej części interfejsu jest przycisk **Clear log**, bezpowrotnie czyszczący cały rejestrator.



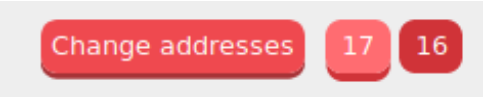

Rysunek 4.5: Rejestrator komunikatów zwracanych przez aplikację wraz z obsługą

Ostatnią z trzech części górnego interfejsu jest konfigurowalny przycisk. Wprowadzenie tej funkcjonalności zostało podyktowane częstą potrzebą szybkiej reakcji na zachowania elementów elektronicznych. W plikach konfiguracyjnych jest możliwość odpowiedniej definicji działania tego elementu. W przypadku braku takich określeń, przycisk pozostaje zablokowany. Wygląd elementu został przedstawiony na rysunku 4.6.


Rysunek 4.6: Konfigurowalny przycisk (w stanie blokady)

### 4.2.2 Część konfiguracyjna

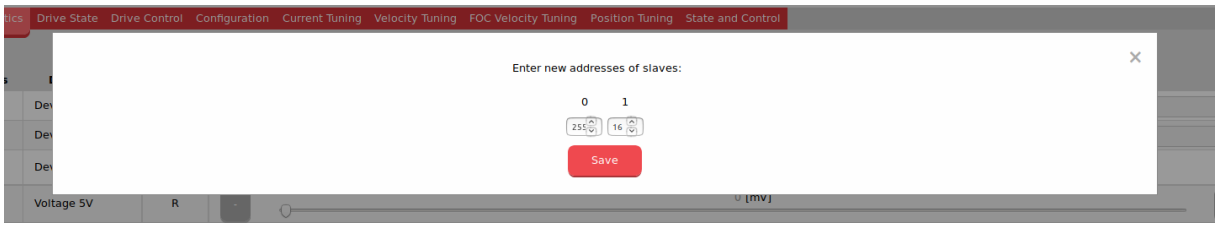
Druga z sekcji aplikacji pełni najważniejszą rolę. To tutaj wyświetlana jest cała mapa rejestrów konfigurowanego przez użytkownika urządzenia. Ponadto w nagłówku tej części znaleźć można informacje o nazwie bieżącego pliku opisującego tę strukturę. Z prawej strony znajduje się również menu nawigacyjne, pozwalające wybrać urządzenie do komunikacji. To rozwiązanie umożliwia szybkie przełączanie się pomiędzy wieloma urządzeniami podczas jednej konfiguracji.




Rysunek 4.7: Nawigacja pomiędzy konfigurowalnymi urządzeniami (wybór na podstawie ich adresu w magistrali)

Obok menu jest przycisk **Change addresses** otwierający moduł do zmiany adresów urządzeń. Nowo otwarte okno (rysunek 4.8) umożliwia zamianę istniejących adresów na inne, mieszczące się w zakresie od 0 do 255. W przypadku podania błędnych danych, użytkownik jest poinformowany stosownym komunikatem.

Poniżej znajduje się menu nawigacyjne pomiędzy zakładkami. Rozmieszczenie poszczególnych rejestrów w zakładkach jest wczytywane z pliku konfiguracyjnego *XML* i stanowi pełną dowolność. Jednakże, sposób podziału na strony jest istotny w momencie kliknięcia przycisku **Save**, ponieważ wysyłane są tylko wartości z zakładki aktywnej.




Rysunek 4.8: Moduł zmiany adresów urządzeń do konfiguracji

Każda ze stron rozpoczyna się od parametrów komunikacyjnych:

- *autoread* - determinuje odczyt samoczynny danej zakładki,
- *autoread period* - oznacza czas, co jaki następuje odczyt i odświeżenie wartości,
- *autosend* - determinuje, czy wysłanie wartości ma następować od razu po jej zmianie, czy tylko za sprawą przycisku **Save**.

Ustawienia opisane w liście poprzedzają właściwą część interfejsu do konfiguracji. Elementem spajającym wszystkie rejestry w jedną całość jest tabela, opisująca każdy z nich. Wartość każdej z komórek tabeli można podzielić na kolumny. Interpretowane od lewej oznaczają kolejno:

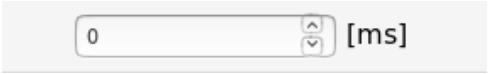
- *Address* - adres rejestru w pamięci urządzenia,
- *Description* - opis odpowiedzialności danego rejestru,
- *Access* - prawa do zapisu i odczytu rejestru (możliwe wartości to kombinacja liter **R** - odczyt i **W** - zapis),
- *Data* - pole wyświetlające zawartość rejestru i umożliwiające jego zmianę.

Podstawowym podziałem rejestrów na typy jest rozróżnienie pól prostych i złożonych. Rejestry proste składają się tylko z jednej wartości możliwej do odczytu i zmiany. Ich wartość jest wprost przekazywana pomiędzy aplikacją i urządzeniem. Rejestry złożone mogą składać się z wielu mniejszych pól, które podczas komunikacji obustronnej są złączane i rozdzielane w odpowiedni ciąg bitów.

Drugim sposobem rozróżnienia jest podział rejestrów na tylko do odczytu oraz te z możliwością zapisu. Jeden i drugi typ może występować w formie prostej lub złożonej, jednakże aplikacja sygnalizuje brak praw do zmiany wartości poprzez zablokowany interfejs i szary kolor przycisków.

Gatunki pól, które są możliwe do użycia, przy opisie rejestrów to:

- *Pole numeryczne* - najprostszy z gatunków. Wyświetla czystą wartość liczbową rejestru lub jego części. Zmiana odbywa się poprzez wpisanie cyfr za pomocą klawiatury i zatwierdzenie przyciskiem **ENTER**. Dodatkową informacją jest jednostka interpretowanej wartości liczbowej (rysunek 4.9).




Rysunek 4.9: Przykład pola numerycznego z wartością interpretowaną jako milisekundy

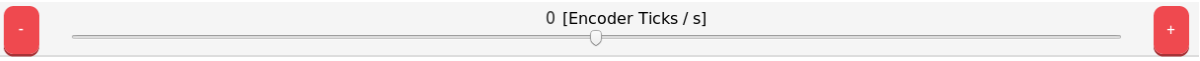
- *Flagi bitowe* - przedstawiające wartość rejestru w postaci zapalonych (1) lub zgaszonych (0) bitów. Za ich pomocą możliwa jest manipulacja liczbowa na poziomie dwójkowym. Zmiana statusu następuje w momencie zaznaczenia lub odznaczenia bitu przy pomocy myszki.


Rysunek 4.10: Przykład pola z flagami bitowymi dla rejestru 16-bitowego - przedstawiona wartość odpowiada liczbie 4404 w systemie dziesiętnym

- *Lista rozwijana* - umożliwiająca odczyt i wybór domyślnie przygotowanych opcji. Zawiera uwikłane wartości rejestrów wraz z opisem ich funkcjonalności. Zmiana odbywa się przez wybranie jednego z dostępnych elementów listy za pomocą myszki.


Rysunek 4.11: Przykład złożonego rejestru, składającego się z pięciu list rozwijanych

- *Suwak numeryczny* - najbardziej złożony gatunek pola. Podobnie do pola numerycznego jest czystą wartością liczbową, jednakże jej zmiana jest możliwa na więcej sposobów. Oprócz podmiany zawartości rejestru, poprzez wpisanie cyfr z klawiatury, zapis może nastąpić również za pomocą przesunięcia suwaka. W celach dokładniejszej zmiany wartości, istnieje też możliwość manipulacji przyciskami - i +.




Rysunek 4.12: Przykład prostego rejestru z możliwością edycji za pomocą suwaka numerycznego

Wszystkie pola, znajdujące się w aplikacji są zabezpieczone przed wpisaniem błędnej wartości. Użytkownik, po wpisaniu liczby nie znajdującej się w odpowiednim zakresie, zostaje poinformowany stosownym komunikatem z informacją o prawidłowych ograniczeniach. Zapobiega to błędom, które mogłyby wystąpić przy przesyle danych do urządzenia elektronicznego.

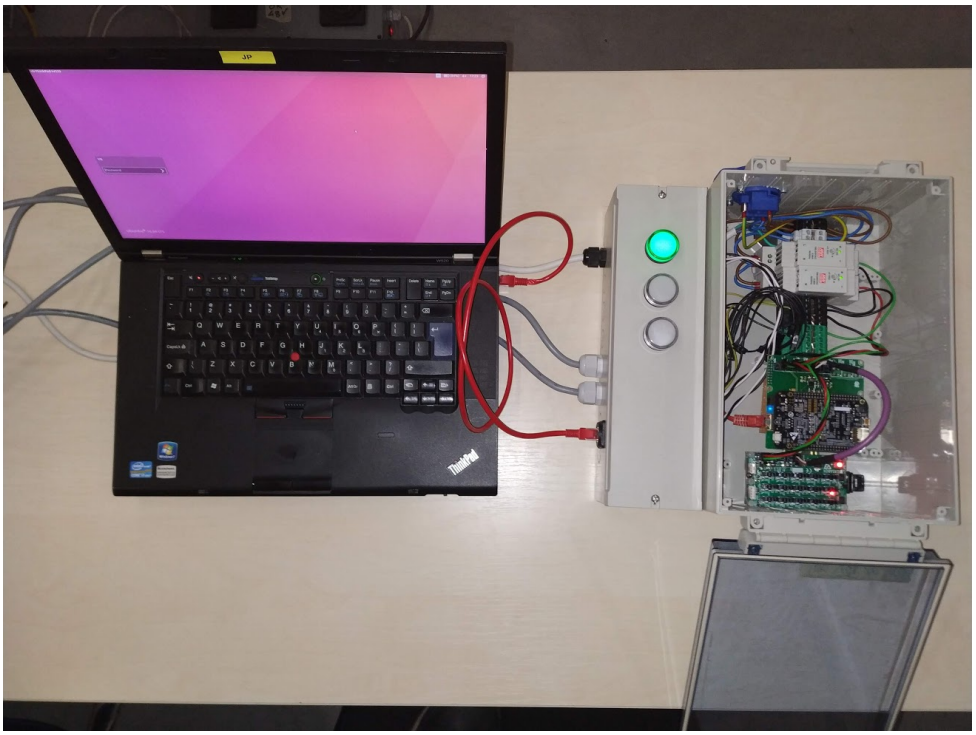
# Rozdział 5

# Testy

## 5.1 Pierwszy etap eksperymentów

### 5.1.1 Stanowisko testowe

Komunikacja w robotach mobilnych jest bardzo ważną częścią systemu, dlatego też jej zakłócenie przez nieodpowiednie działanie aplikacji może przynieść fatalne skutki. Ze względów bezpieczeństwa, testy aplikacji przebiegły dwuetapowo. Pierwszym etapem eksperymentów było sprawdzenie prawidłowości działania na odpowiednio przygotowanym układzie (rys. 5.1).



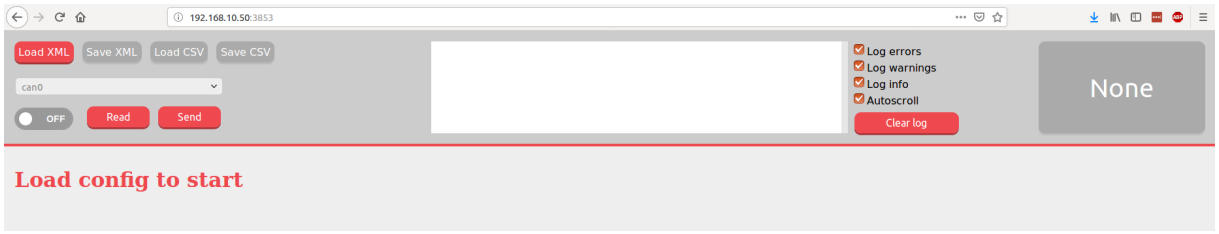
Rysunek 5.1: Stanowisko testowe w pierwszym etapie eksperymentów

Układ, do którego podłączony jest komputer użytkownika, jest złożony z elementów elektronicznych:

- zasilacza sieciowego, zapewniającego napięcie wyjściowe 24 V,

- komputera BeagleBone Black wraz z modułem pomocniczym,
- modułu wejść/wyjść - 1505_io, podłączonego do magistrali CAN,
- przewodów połączeniowych i gniazd stykowych.

Połączenie pomiędzy komputerem konfigurującym a komputerem symulującym komputer pokładowy robota zostało zrealizowane za pomocą przewodu Ethernet. Za pomocą programów *ssh* i *scp*, paczka zawierająca aplikację została przekopiowana, zainstalowana i uruchomiona na komputerze BeagleBone Black z dystrybucją systemu Linux - Debian. Po odpowiedniej konfiguracji połączenia Ethernet, możliwe było uruchomienie aplikacji w przeglądarce Mozilla Firefox, za pomocą adresu `192.168.10.50:3853`.




Rysunek 5.2: Uruchomienie aplikacji

### 5.1.2 Przebieg testów

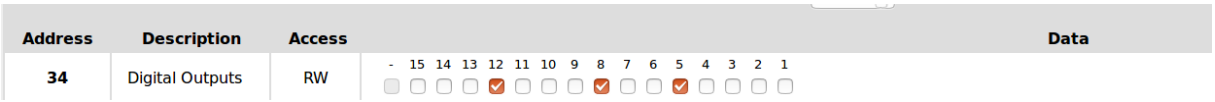
Czynności opisane w poniższej podsekcji, zostały uwiecznione na filmie, załączonym do elektronicznej wersji pracy. Przebieg rozpoczął się od przesłania przygotowanego wcześniej pliku konfiguracyjnego, określającego mapę rejestrów modułu wejść-wyjść, za pomocą przycisku *Load XML*. Aplikacja wyświetliła komunikat o poprawnym załadowaniu pliku i wygenerowała strukturę konfigurowalną. Proponowany przez aplikację interfejs *can0* obsługiwał magistralę CAN z podłączonym urządzeniem do konfiguracji.


Rysunek 5.3: Poprawny odczyt danych z urządzenia

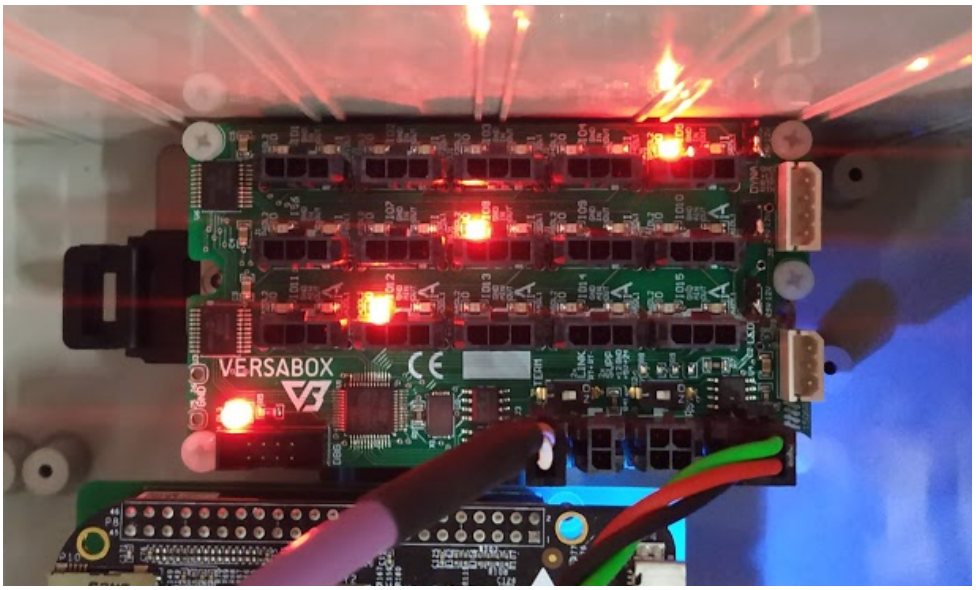
Włączenie komunikacji za pomocą przycisku *ON/OFF* skutkowało wyświetleniem informacji o poprawnym odczycie danych oraz aktualizacji wartości pól na stronie w czasie rzeczywistym. Oznaczało to, że ustawiona domyślnie w pliku konfiguracja, tj. urządzenie o adresie 64, jest prawidłowa.

Kolejnym etapem eksperymentu było sprawdzenie możliwości zmiany wartości pól i poprawnego ich przesłania. W tym celu otworzona została karta do kontroli urządzenia (ang. *Control*). Zawierała pola, których dostęp pozwalał na edycję zawartości (ang. *read and write*). Rejestr o adresie w pamięci numer 34 odpowiadał za sterowanie wyjściami cyfrowymi. Moduł 1505_io posiada ich 15 i każde

z nich jest obsługiwane osobno. Zaznaczanie flag w polach rejestru, skutkowało natychmiastowym przesłaniem żądania i zaświeceniem diod LED przy odpowiednich wyjściach urządzenia. Dodatkowo aplikacja przesłała również komunikat o przesłaniu danych.




Rysunek 5.4: Manipulacja stanem wyjść cyfrowych modułu - wyjścia z zaznaczonym haczykiem są w stanach wysokich

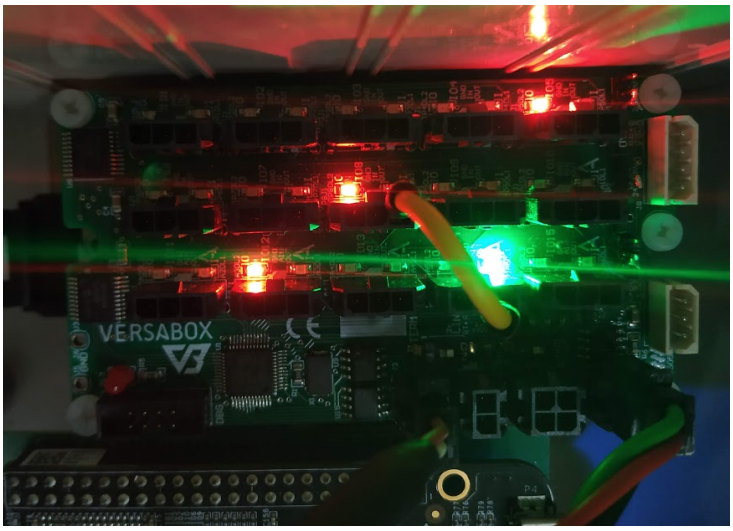



Rysunek 5.5: Stan modułu wejść/wyjść, odpowiadający rysunkowi 5.4

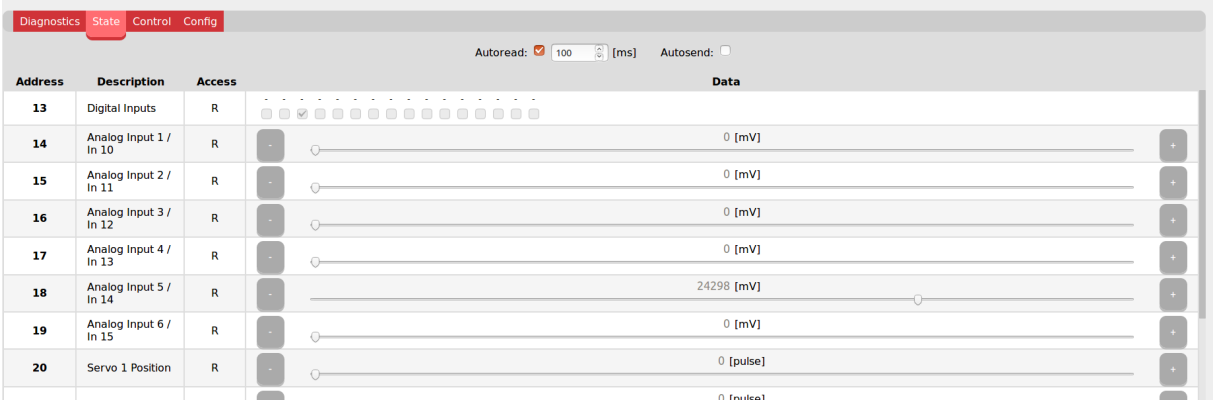
W celu sprawdzenia, czy odczyty z wejść analogowych i cyfrowych są prawidłowe, poprowadzono sygnał pochodzący od wyjścia cyfrowego numer 8 o stanie wysokim do wejścia analogowo-cyfrowego. Zostało to zrealizowane za pomocą krótkiego przewodu wtykowego. Stan modułu, po aktualizacji układu, został przedstawiony na rysunku 5.6.

Odczyt wejść modułu odbywa się w karcie stanu (ang. *State*). Rejestry występujące na tej podstronie służą tylko do odczytu. Wskazuje na to pole *Access*, które w każdym z rejestrów ma wartość R (ang. *read only*). Dodatkowym objawem są również zablokowane pola. Wcześniejsze odczyty pokazywały zerowe pomiary stanu urządzenia (z dokładnością do szumów pomiarowych). Automatyczny odczyt w tle zaktualizował wartości rejestrów, znajdujące się w tej części konfiguracyjnej.

Ze względu na podwójny charakter wejścia, do którego został doprowadzony sygnał, zmiana stanu nastąpiła w dwóch rejestrach. Pierwszy z nich (adres 13) przechowuje stany wszystkich wejść cyfrowych, a drugi (adres 18) przechowuje stan analogowy sygnału na wejściu analogowym 5 (cyfrowe wejście 14). Zgodnie z logiką, którą posługuje się ten moduł, cyfrowy stan wysoki wywołał pojawienie się sygnału analogowego o średniej wartości w okolicach 24 V (rysunek 5.7).

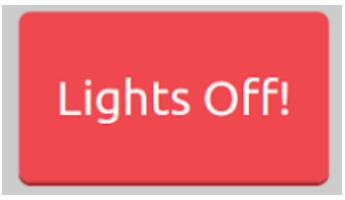



Rysunek 5.6: Świecąca dioda LED symbolizująca stan wysoki wejścia cyfrowego




Rysunek 5.7: Odczyty stanów wejść analogowych i cyfrowych w module wejść/wyjść

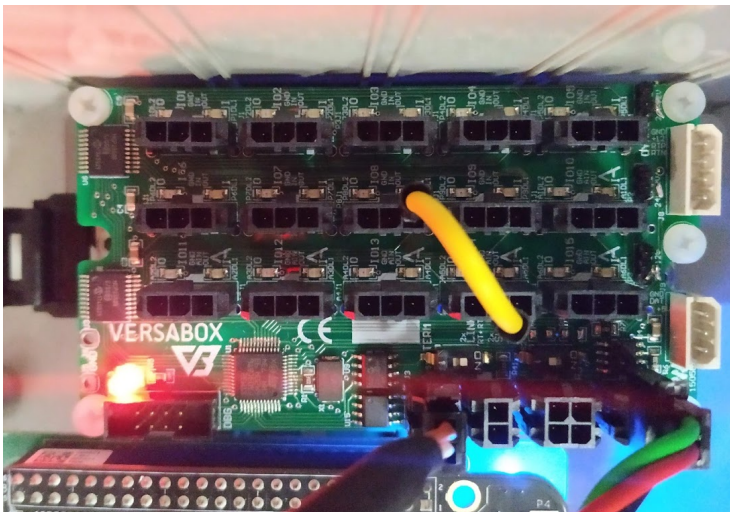
Ostatnią częścią testu było sprawdzenie działania konfigurowalnego przycisku. Plik XML, który został przesłany do serwera aplikacji, zawierał niezbędny dla aplikacji opis jego działania. Dla celów testowych, przycisk nazwany angielskim wyrażeniem "Lights off!" (rysunek 5.8), był przygotowany do nagłego wyłączenia wszystkich wyjść cyfrowych. Swoje działanie opierał na wysłaniu wiadomości do urządzenia, która nadpisywała zawartość rejestru, odpowiadającego za ich stan (adres 34) na 0.




Rysunek 5.8: Przycisk, znajdujący się w prawym górnym rogu aplikacji, wyłączający wszystkie wyjścia cyfrowe

Wciśnięcie przycisku spowodowało nagłe wyłączenie się wszystkich czerwonych diod LED, które były reprezentacjami stanów wyjść cyfrowych. Zostało to również potwierdzone odpowiednim ko-

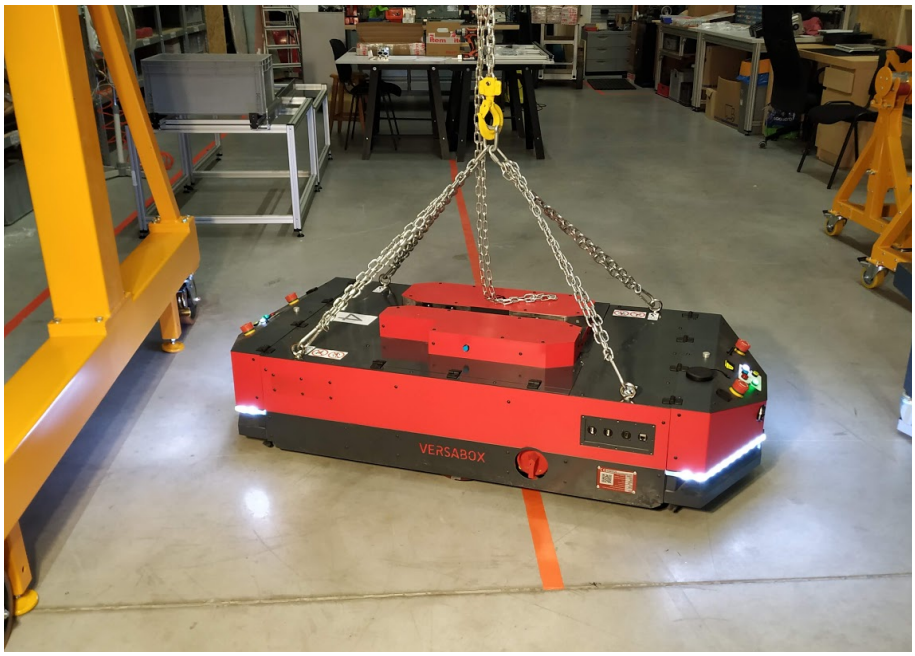
munikatem oraz brakiem zaznaczonych flag w rejestrze odpowiadającym za wyjścia. Stan końcowy urządzenia podłączonego do magistrali CAN został przedstawiony na rysunku 5.9.




Rysunek 5.9: Moduł wejść/wyjść po zakończonych testach

## 5.2 Działanie aplikacji na właściwym komputerze pokładowym

### 5.2.1 Stanowisko testowe




Rysunek 5.10: Robot mobilny Kanboy przed rozpoczęciem eksperymentów

Poprawne zachowanie aplikacji podczas komunikacji z odpowiednio przygotowanym układem pozwoliło na sprawdzenie działania na komputerze pokładowym robota. Urządzenie, które zostało wykorzystane w celach testowych, to przemysłowy robot mobilny **Kanboy** firmy Versabox Sp. z o.o.

Konfiguracja modułów elektronicznych, znajdujących się w jego wnętrzu, następowała po jego wcześniejszym umieszczeniu 3 centymetry nad podłożem.

Układ elektroniczny badanego podmiotu składa się z wielu urządzeń mikroprocesorowych. Wszystkie są podłączone do głównej magistrali CAN i istnieje możliwość ich konfiguracji za pomocą testowanej aplikacji. Głównym celem działań przeprowadzonych podczas testu było sprawdzenie poprawności działania omawianego systemu na obciążonej magistrali oraz zdalnego dostępu do modułów elektronicznych. Z tego powodu sprawdzona została możliwość jednoczesnej konfiguracji dwóch sterowników napędów - lewego oraz prawego koła. Połączenie pomiędzy komputerem użytkownika a robotem zostało zrealizowane za pomocą bezprzewodowej sieci WiFi.

Aby sprawdzić możliwości napędowe robota, koniecznym było włączenie stopni mocy. Doprowadzenie napięcia do sterowników zostało przeprowadzone za pomocą panelu sterowniczego na robocie. Świecąca na biało dioda kontrolna, oznaczała prawidłowo przeprowadzoną procedurę. Robot był gotowy do poruszania wszystkimi swoimi napędami.

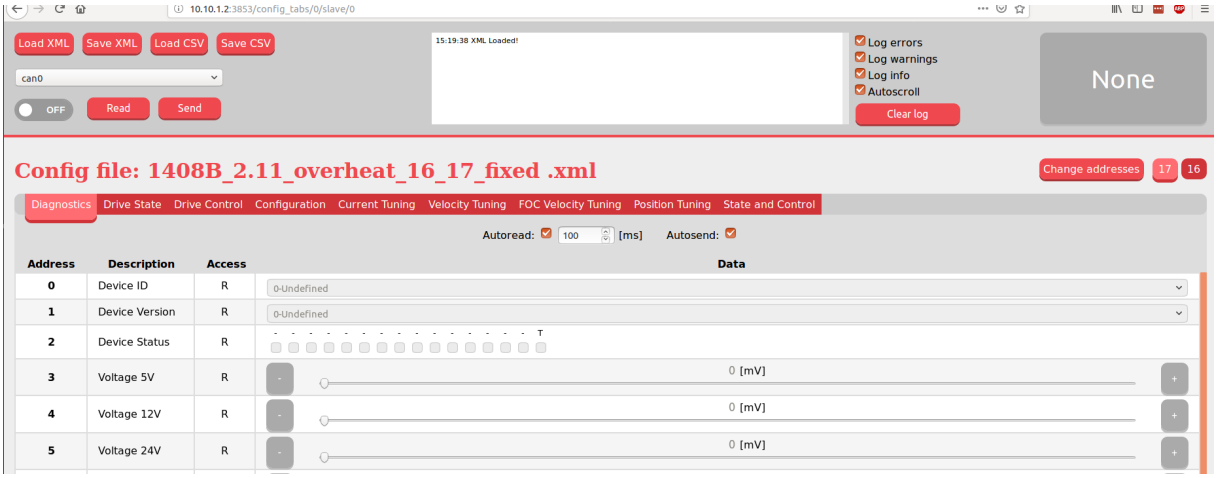
Rysunek 5.11: Panel sterowniczy, informujący o gotowości robota mobilnego

Dodatkowo, do zewnętrznej części obu kół, przyklejone zostały dwa kolorowe znaczniki. Miały na celu łatwe pokazanie ruchu, który mógłby być niewidoczny z dalszej odległości. Znaczniki zostały wykonane z zielonej taśmy samoprzylepnej.

### 5.2.2 Przebieg eksperymentów

Tak przygotowane stanowisko testowe umożliwiło rozpoczęcie drugiego, właściwego etapu eksperymentów. Ich przebieg został nagrany i załączony do elektronicznej wersji pracy.

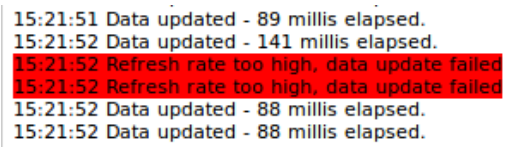
Po podłączeniu komputera użytkownika do sieci WiFi robota i skorzystaniu z programów *ssh* i *scp*, aplikacja została przekopiowana i zainstalowana na komputerze pokładowym. Pomyślne przygotowanie środowiska skutkowało prawidłowym uruchomieniem serwera. Możliwe było uruchomienie klienta aplikacji w przeglądarce Mozilla Firefox i wczytanie odpowiedniego pliku konfiguracyjnego XML.




Rysunek 5.12: Ekran aplikacji na komputerze użytkownika, po wczytaniu przestrzeni konfiguracyjnej dla sterowników napędów

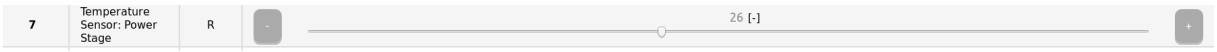
Aplikacja poinformowała o prawidłowym wygenerowaniu struktury kart i rejestrów. Po prawej stronie (widoczne na rysunku 5.12) pojawiło się menu do nawigacji pomiędzy urządzeniami. Sterowniki napędów, konfigurowane podczas testów, występowały pod adresami 16 i 17. Pierwsza faza eksperymentu odnosiła się do urządzenia sterującego ruchem lewego koła (adres 17). Główna magistrala CAN w robocie była obsługiwana przez interfejs *can1*, który został wykryty przez aplikację i pojawił się w liście rozwijanej.

Globalne włączenie komunikacji skutkowało aktualizacją danych na stronie, jednakże zdarzały się komunikaty o zbyt dużej częstotliwości odświeżania. Był to normalny objaw, ze względu na wysokie obciążenie magistrali CAN. Okres, pomiędzy dwoma odczytami danych na karcie diagnostycznej (ang. *Diagnostics*) został zwiększony do 200 milisekund za pomocą pola numerycznego.




Rysunek 5.13: Komunikat o błędzie, informujący o zbyt dużej częstotliwości odczytu

Wybrana karta diagnostyczna pozwalała na odczyt ważnych informacji, przechowywanych przez rejestry w pamięci modułu. Możliwy był odczyt wielu parametrów jednostki napędowej.




Rysunek 5.14: Temperatura odczytana z czujnika, umiejscowionego na sterowniku napędu lewego koła - rejestr z rysunku informuje o wartości 26 stopni Celsjusza

Pola znajdujące się w opisywanej karcie przechowywały również wartości napięć, występujących na odpowiednich przetwornicach w module. Za pomocą rejestrów o adresach 3, 4 i 5, możliwy był ich podgląd w czasie rzeczywistym.

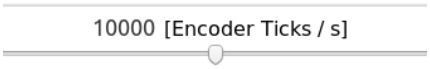

Rysunek 5.15: Pola reprezentujące wartości napięć wyjściowych z przetwornic, umieszczonych w układzie sterownika napędów

Druga z kolei karta z tej struktury przechowywała rejestry zawierające informacje o stanie napędu (ang. *Drive state*). Sprawdzona została poprawność odczytów - prędkość obrotowa napędu i natężenie prądu, przepływającego przez sterownik, pozostawały zerowe.

Kolejnym etapem testów było zadanie zdefiniowanej prędkości. W tym celu, użyta została karta sterownicza (ang. *Drive control*). Aby odblokować możliwość sterowania prędkością obrotową, konieczne było ustawienie modułu we właściwy tryb. Służyło do tego pole z listą wybieraną (rejestr o adresie 28).


Rysunek 5.16: Pole do odczytu i zapisu, przechowujące tryb napędu

Zmiana zawartości rejestru została potwierdzona opowiednim komunikatem. Następnym zadaniem było ustawienie stałej prędkości obrotowej. Jednostką używaną przy jej określaniu była liczba impulsów enkodera na sekundę. Nadpisanie zerowej wartości nową i zatwierdzenie przyciskiem *Enter* objawiło się poprawnym przesłaniem żądania i rozpoczęciem ruchu obrotowego koła.




Rysunek 5.17: Ustawiona prędkość obrotowa w odpowiednim polu konfiguracyjnym

Powrót do karty, zawierającej informacje o stanie napędu, pozwolił zweryfikować poprawność przesłanych danych. Rejestr o adresie 17 przechowywał 32 bitową wartość aktualnej prędkości obrotowej. Odczyty pojawiające się na tym polu oscylowały wokół zadanej prędkości (10 tys. impulsów enkodera na sekundę). Możliwe było również sprawdzenie aktualnego poboru prądu, którego wartość średnia wynosiła około 420 mA.

Następnym etapem testów na komputerze pokładowym robota, było przełączenie konfigurowanego urządzenia na prawy sterownik napędu (o adresie 16). Zmiana modułu za pomocą menu nawigacyjnego, umożliwiła podgląd zawartości jego rejestrów. Odczyty wyglądały podobnie do poprzednich, ze względu na symetrię robota.

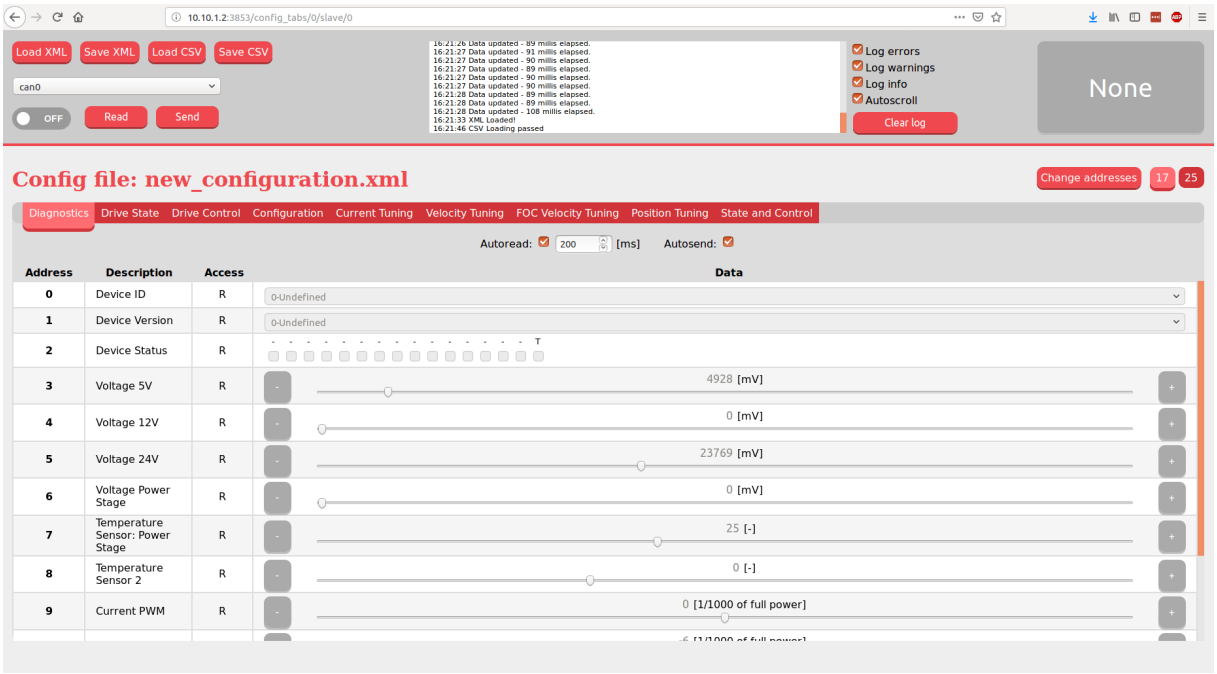
### 5.2.3 Działanie dodatkowych funkcjonalności

Podczas pracy serwera aplikacji na komputerze pokładowym robota sprawdzone zostały funkcjonalności, które nie zostały uwzględnione we wcześniejszych testach. Próba zmiany adresu konfigurowanego urządzenia za pomocą przycisku *Change addresses* przebiegła pomyślnie. Jednak wszystkie

próby komunikacji zostały zakończone niepowodzeniem i komunikatem, informującym o błędzie. Było to spowodowane brakiem urządzenia o adresie wpisanym przez użytkownika i z wczytaną wcześniej strukturą rejestrów.

Ostatnią częścią eksperymentów było sprawdzenie przetwarzania plików XML i CSV przez aplikację. Korzystając z poprzedniej konfiguracji dla dwóch sterowników napędów, aplikacja wykonała odczyt wszystkich zawartości z rejestrów. Następnie zmienione zostały parametry komunikacyjne, znajdujące się w kartach konfiguracyjnych. Tak przygotowana struktura została zapisana i pobrana w postaci pliku XML za pomocą przycisku *Save XML*. Również aktualne wartości rejestrów pobrano za pomocą przycisku *Save CSV*.

W celu sprawdzenia poprawności zapisywania i odczytywania plików przez aplikację, komunikacja została zatrzymana. Następnie wczytany został plik XML i wygenerowana zmodyfikowana struktura konfiguracyjna. Po weryfikacji prawidłowo zapisanego pliku, nastąpiło wprowadzenie danych z wcześniejszego działania aplikacji. Bez włączania globalnej komunikacji z modułami przesłany został wygenerowany plik CSV za pomocą przycisku *Load CSV*. Wszystkie wartości w ówczesnej strukturze zgadzały się z poprzednimi pomiarami.




Rysunek 5.18: Stan urządzenia przywrócony na podstawie pliku CSV

# Rozdział 6

# Podsumowanie

Prace wykonane oraz opisane w poprzednich rozdziałach przyczyniły się do opracowania poprawnej aplikacji do zdalnej konfiguracji modułów elektronicznych robota. Cel wyznaczony w rozdziale 2 został osiągnięty dla przyjętych założeń. Eksperymenty przeprowadzone przy użyciu powstałego systemu przebiegły pomyślnie i potwierdzają prawidłowe wykonanie zadania.

Niskopoziomowa konfiguracja modułów elektronicznych jest najczęściej związana z połączeniem przewodowym, pomiędzy komputerem a konfigurowanym urządzeniem. Zdalny dostęp do ich wnętrza znacząco przyspiesza proces zmiany parametrów. Umożliwia również łatwy podgląd urządzenia pod kątem diagnostycznym. Rozwiązanie zadania zdalnej konfiguracji urządzeń w postaci aplikacji internetowej wiąże ze sobą wiele korzyści. Tak skonstruowana architektura pozwala na wieloplatformowość (ułatwioną przenoszenie pomiędzy systemami). Dodatkowo użycie wyłącznie języków skryptowych do implementacji nie wymaga kompilacji, która dla różnych systemów operacyjnych może być kłopotliwa.

Testy opisane w rozdziale 5 przedstawiły możliwości, wynikające z zastosowania magistrali CAN w komunikacji wewnętrznej. Wykazano, że ten typ sieci jest odpowiedni do zastosowań konfiguracyjnych w czasie rzeczywistym. Mimo jej znacznego obciążenia podczas włączonego systemu robota, aplikacja była w stanie przeprowadzać dalsze działania. Możliwe było natychmiastowe sprawdzanie reakcji urządzeń na zastosowane zmiany i podgląd rzeczywistych wartości liczbowych.

Przedstawiona w pracy aplikacja jest otwarta na wiele dróg rozwojowych. Podstawowym sposobem na usprawnienie działania i dodanie nowych funkcjonalności jest zwiększenie liczby obsługiwanych typów rejestrów pamięci. Postawione ograniczenia uniemożliwiają użycie zmiennych liczbowych innych niż o wartościach całkowitych, mieszczących się w zakresie 32 bitów. Struktura rejestrów jest często bardzo różnorodna i dodatkowa obsługa rozszerzonej gamy zmiennych powiększyłaby grupę modułów nadających się do konfiguracji przez aplikację.

Sposobem na dalszy rozwój projektu jest również powiększenie interfejsu użytkownika o możliwość edycji struktury konfigurowalnej. Na aktualnym etapie prac, konieczne jest ręczne przygotowanie pliku z mapą pól do wygenerowania. Aplikacja umożliwia wprowadzanie zmian, jednakże dla małej ilości informacji. Dodanie trybu pełnej graficznej edycji usprawniłoby prace nad przygotowywaniem modułów o nowej strukturze pamięci.

# Bibliografia


[1] Bosch. *CAN Specification Version 2.0*. Robert Bosch GmbH, Postfach 50, D-7000 Stuttgart 1, Germany, 1991.

[2] Steve Corrigan. *Introduction to the Controller Area Network (CAN)*. Texas Instruments, 2002.

[3] Organización Internacional de Normalización. *ISO 11898 : Road Vehicles : Interchange of Digital Information : Controller Area Network (CAN) for High-speed Communication*. ISO, 1993.

[4] J. S. Freudenberg J. A. Cook. *Controller Area Network (CAN), EECS 461*. 2008.

[5] Źródło internetowe. *http://flask.pocoo.org - Strona domowa projektu Flask. Stan na dzień 16.12.2018 r.* 2018.

[6] Źródło internetowe. *https://bostondynamics.com - Strona firmy Boston Dynamics. Stan na dzień 15.11.2018 r.* 2018.

[7] Źródło internetowe. *https://can-cia.org/can-knowledge/can/can-history - Strona opisująca historię rozwoju magistrali CAN. Stan na dzień 21.10.2018 r.* 2018.

[8] Źródło internetowe. *https://en.wikipedia.org/wiki/CAN_bus - Anglojęzyczna strona serwisu Wikipedia.org. Stan na dzień 23.10.2018 r.* 2018.

[9] Źródło internetowe. *https://jquery.com - Strona domowa projektu jQuery. Stan na dzień 20.12.2018 r.* 2018.

[10] Źródło internetowe. *https://python-can.readthedocs.io/en/2.1.0 - Dokumentacja biblioteki Python-CAN. Stan na dzień 15.12.2018 r.* 2018.

[11] Źródło internetowe. *https://canberry.readthedocs.io - Strona domowa projektu CANberry. Stan na dzień 18.01.2019 r.* 2019.

[12] Źródło internetowe. *https://github.com/tzapu/WiFiManager - Repozytorium projektu WiFi Manager. Stan na dzień 16.01.2019 r.* 2019.

[13] Źródło internetowe. *https://www.inhandnetworks.com/solutions/industrial-robot-remote-monitoring.html - Opis przykładowego rozwiązania Przemysłu 4.0. Stan na dzień 15.01.2019 r.* 2019.

# Spis rysunków

# Spis tabel

# Załączniki do wersji elektronicznej

- *dokumentacja_kodu.zip* - Dokumentacja kodu aplikacji w wersji skompresowanej.
- *proces_konfiguracji_skrzynki.mp4* - Film przedstawiający eksperymenty, przeprowadzane na specjalnie przygotowanym układzie.
- *proces_konfiguracji_robota.mp4* - Film zawierający eksperymenty, przeprowadzana na robocie mobilnym Kanboy.