

Komunikacja z Robotem

Robot Astorino pozwala na komunikację z nim na 3 sposoby:

1. Poprzez USB
2. Poprzez Ethernet
3. Poprzez porty cyfrowego I/O

Robotowi można wgrać program, napisany w podzbiorze kompatybilnego z robotami Kawasaki języku AS Language, lecz programem tym należałoby jakoś sterować z zewnątrz, czy to wykorzystując sterownik PLC, komputer PC, czy jakiekolwiek urządzenie połączone z Internet of Things. Przeanalizujmy więc możliwe opcje implementacji takiej komunikacji z urządzeniami zewnętrznymi:

USB

Robot posiada port USB, lecz aktualnie jedyna znana nam metoda sterowania robotem z urządzenia połączonym przez ten port, to poprzez główną aplikację służącą do programowania robota. Oprócz samego środowiska programistycznego i konfiguracyjnego robota, aplikacja pozwala także na jego sterowanie. Problem w tym, że sterowanie te jest jedynie manualne. Nie jest znana nam żadna metoda na podpięcie zewnętrznego programu do aplikacji, aby wysłać do robota arbitralne komendy, inaczej niż wpisując je ręcznie w odpowiednim polu tekstowym, lub też klikając odpowiedni przycisk na interfejsie użytkownika. Nie została nam udostępniona żadna dokumentacja sterowników, czy też opis protokołu komunikacyjnego poprzez USB, która pozwalałaby na stworzenie własnej aplikacji potrafiącej to samo co ta główna. Chcąc wykorzystać połączenie USB do sterowania robotem zmuszeni byśmy byli więc do inżynierii wstępnej gotowej aplikacji, co wykracza poza nasze umiejętności, jak i chęci zaangażowania naszego czasu w projekt, dlatego sposób ten odpada.

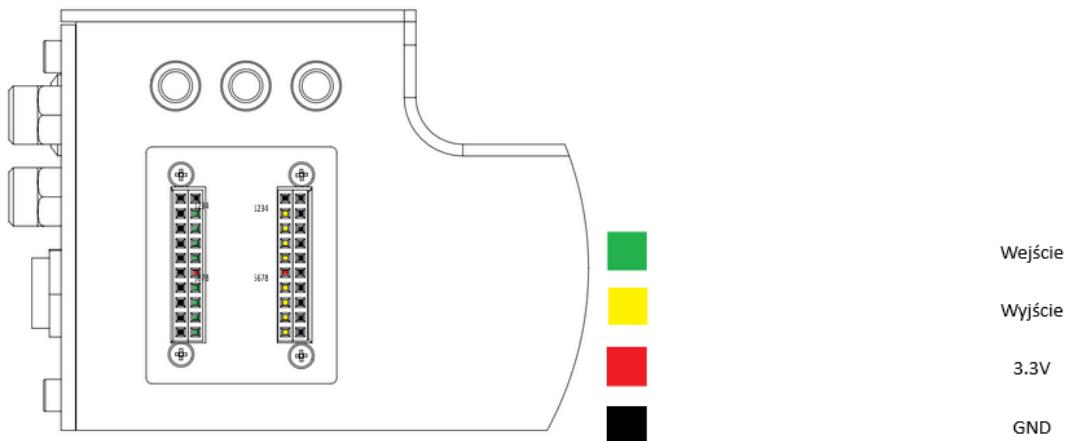
Ethernet

Bardziej odpowiadającym sposobem komunikacji wydawałby się ethernet. Robot posiada port RJ-45 na skrątkę internetową. Sama główna aplikacja sterująca robota posiada możliwość połączenia poprzez USB lub Ethernet, do wyboru. Prospekt napisania własnej aplikacji, imitującej tą główną, wykorzystującą Ethernet jako sposób komunikacji, byłby też o rzędy wielkości razy łatwiejszy niż w przypadku USB. Przy użyciu oprogramowania **Wireshark**, pozwalającego na podgląd zawartości pakietów przechodzących przez kartę sieciową, zauważono, że łącząc się poprzez Ethernet z robotem, aplikacja wykorzystuje protokół Telnet, czyli wysyła do robota komendy tekstowe. Oznaczałoby to, że w celu sterowania robotem, należałoby z poziomu dedykowanej aplikacji wykonać manualnie dane polecenie i w programie Wireshark odpalonym w tle podglądać format wysłanej komendy. Poznając w ten sposób protokół sterowania robotem, napisana przez nas aplikacja sterująca mogłaby wtedy wysłać komendę analogiczną do robota. Wszystko mogłoby się wydawać jakże akceptowalną opcją, jednak ten kanał komunikacyjny ma pewien mankament... nie działa. Przed wizytą konstruktora robota, aplikacja sterująca się zawieszała i wyłączała kilka sekund po inicjacji połączenia z robotem poprzez Ethernet. Mleliśmy nadzieję, że konstruktor robota znajdzie rozwiązanie problemu, lecz zamiast tego, w nowej wersji

oprogramowania jaką zainstalował podczas wizyty, komunikacja poprzez Ethernet została wyłączona. Tym samym zostaliśmy zmuszeni do wykorzystania trzeciego, ostatniego sposobu na komunikację z robotem...

Wejścia/Wyjścia Cyfrowe

Na boku podstawy robota znajdują się porty wejść i wyjść cyfrowych:

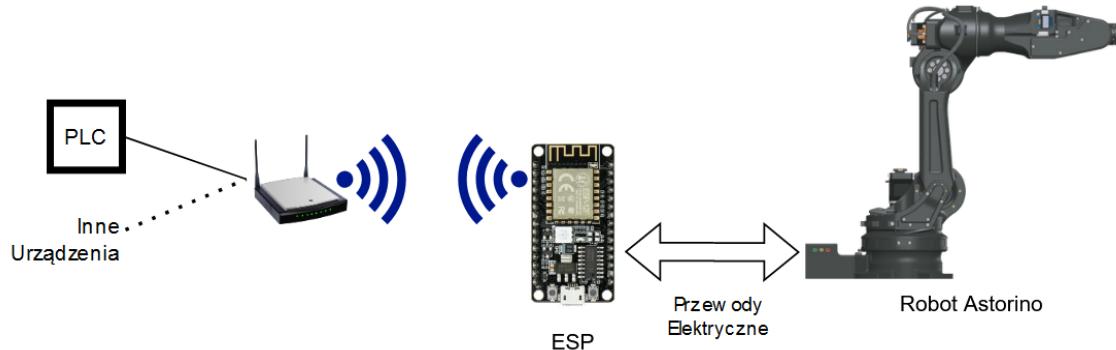


Dostępnych jest 8 wejść oraz 8 wyjść. Portu operują na napięciu 0-3,3V. Dostępny jest także osobny moduł do podłączenia do robota, pozwalający zwiększenie napięcia do 24V, lecz na nasze potrzeby nie będzie wykorzystywany. Stan każdego wejścia cyfrowego może być w dowolnym momencie odczytany w programie robota komendą **SIG**, jak i stan każdego wyjścia może być ustawiony, komendą **SIGNAL**. Warto też dodać, co początkowo wprowadziło nasz zespół w zakłopotanie, wyjścia ponumerowane są od 1 do 8, od dołu do góry, a wejścia od 1001 do 1008, od góry do dołu. Żeby sterować więc robotem poprzez te porty, musimy jedynie znaleźć urządzenie zdolne do podawania wybranych poziomów napięcia na odpowiednie wyjścia, jak i komunikacji z pozostałymi urządzeniami na platformie.

Takim urządzeniem mógłby być sterownik PLC. Jednak posiadany przez nas sterownik posiada jedynie 6 wyjść cyfrowych, do tego działających na 24V. Nie dość, że wymagałoby to dołączenie modułu 24-woltowego do portów robota, to nie wykorzystałoby pełni możliwości sterowania robotem, oraz dodatkowo wykorzystujemy wszystkie dostępne wyjścia ze sterownika, a przecież mogłyby się one potencjalnie przydać do sterowania pozostałymi urządzeniami platformy. Postanowiliśmy więc wykorzystać inne urządzenie, które będzie spełniać rolę pośrednika pomiędzy robotem i sterownikiem PLC (i potencjalnie jeszcze innymi urządzeniami), a jest nim mikrokontroler **ESP8266**.

Mikrokontroler ESP8266 jako pośrednik pomiędzy robotem Astorino i sterownikiem PLC

Wybrałyśmy mikrokontroler z rodziny ESP z dwóch powodów. Po pierwsze, działa on na napięciu 3,3V, tak samo jak robot Astorino. Pozwala to na bezpośrednie połączenie wyjść i wejść ogólnego przeznaczenia kontrolera z I/O robota, bez żadnego pośrednika zmieniającego poziomy napięć. Po drugie, mikrokontrolery ESP posiadają wbudowany moduł WiFi, co wielce ułatwia połączenie urządzenia z resztą urządzeń w sieci, ze względu na uniwersalność połączeń sieciowych.

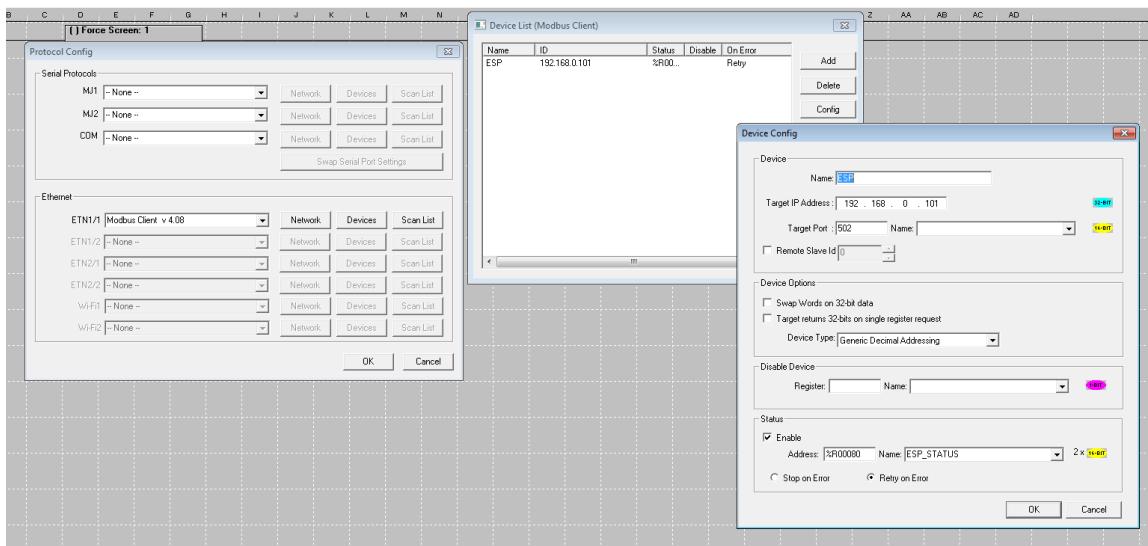


Używany sterownik PLC posiada port Ethernet. Został więc wpięty do switcha do tej samej sieci co ESP. Niestety PLC posiada ograniczone możliwości co do ilości sposobów na przesył danych poprzez Ethernet. Posiada zaimplementowany protokół ModbusTCP, działający w taki sam sposób jak opisany w poprzednich sekcjach protokół ModbusRTU, lecz zamiast wykorzystywania RS485, dane transmitowane są poprzez sieć Ethernet, poprzez tytułe TCP. W takiej sytuacji praktycznie znika pojęcie Master'a i Slave'a, a pojawia się Klient oraz Serwer, jako że urządzenia tworzą, za pomocą protokołu TCP, bezpośredni połączenia pomiędzy sobą. Pakiety nie są broadcastowe do wszystkich urządzeń, jak ma to miejsce z fizycznym połączeniem przy magistrali RS485, tylko wysyłane do konkretnego urządzenia o danym adresie IP. Eliminuje to możliwość wystąpienia kolizji, co sprawia że w sieci może jednocześnie działać więcej niż jeden Klient (Master) oraz urządzenia mogą być jednocześnie i Klientami i Serwerami.

W naszym przypadku PLC działa jako klient, a ESP jako serwer.

Konfiguracja PLC

Konfiguracja PLC polegała wyłącznie na zaznaczeniu odpowiedniej opcji w ustawieniach i wybraniu adresu IP serwera, do którego sterownik ma się łączyć. Cała reszta działa identycznie jak przy ModbusRTU, co opisane jest w poprzedniej sekcji.



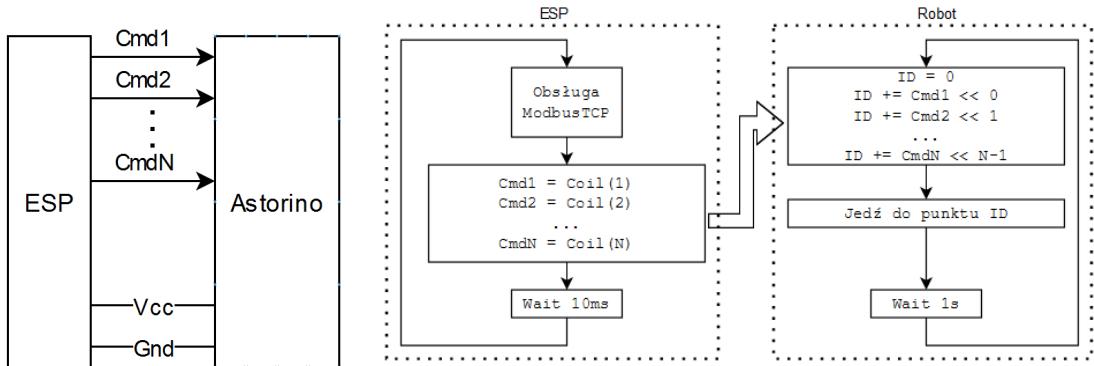
Konfiguracja ESP

Aby ESP działało jako serwer ModbusTCP, potrzebne było znalezienie odpowiedniej biblioteki, implementującej taką funkcjonalność. Mikrokontrolery ESP można programować w środowisku Arduino IDE, z dostępem do wszystkich funkcji i bibliotek kompatybilnych z popularnymi płytami Arduino. Jest to bardzo wygodny sposób na tworzenie oprogramowania na ESP, i poprzez swoją popularność, doczekał się wielu bibliotek napisanych z myślą właśnie o środowisku Arduino IDE. Wyszukiwarka bibliotek Arduino IDE znalazła bibliotekę dedykowaną do płyt z rodziny ESP o nazwie "**modbus-esp8266**" (<https://github.com/emelianov/modbus-esp8266>). Z niej też więc skorzystano. Pozwala ona na konfigurację Modbus-owych rejestrów, cewek itd. do których automatycznie każdy połączony klient może zapisać wybraną wartość, bądź też ją odczytać. Program napisany na ESP może wtedy w dowolnym momencie sprawdzić wartość dowolnego rejestru. Można także dodać callbacki, wywoływane w momencie zapisu/odczytu danych do wybranych rejestrów.

Pierwszy program testowy

W celach pierwszych testów postanowiono napisać program, który ustawia wartość wybranych 4 wyjść cyfrowych, podłączonych do wejść robota, na wartość skonfigurowanych w serwerze ModbusTCP 4 cewek. Na robota wgrano program, który co sekundę odczytywał stan wejść cyfrowych, a następnie ruszał się do odpowiedniego punktu, zależnie od wartości tych wejść. Na sterownik napisana została prosta aplikacja, synchronizująca wartość tych 4 cewek z wewnętrznym rejestrzem, modyfikowalnym z poziomu wyświetlacza HMI. Taka struktura pozwala na sterowanie podstawowe robotem poprzez sterownik PLC. Dodatkowo, w celach testów, na ESP dodano jeszcze jeden Holding Register, również synchronizowany z PLC, którego wartość jest zmieniana wewnętrzne poprzez ESP, co iterację głównej pętli. Dodano również wysyłanie logów poprzez port USB do podpiętego laptopa, co pozwalało na podgląd przychodzących do serwera ModbusTCP requestów. Na płytce prototypowej, na której znajduje się ESP, zamontowano także diody LED, pozwalające na podgląd aktualnego stanu wyjść powiązanych z cewkami.

ESP wysyła do Robotu ID komendy do wykonania, w postaci binarnej. Można użyć max. 8 takich połączeń (tyle wejść posiada Robot). Aktualnie jednak wykorzystywane są jedynie 4, co daje 16 (2^4) możliwych komend. Zasilanie ESP również dostarczane jest przez Robotą. Robot w pętli czeka na ID komendy i cokolwiek odczyta, tę komendę wykonuje



Schemat działania programów i komunikacji

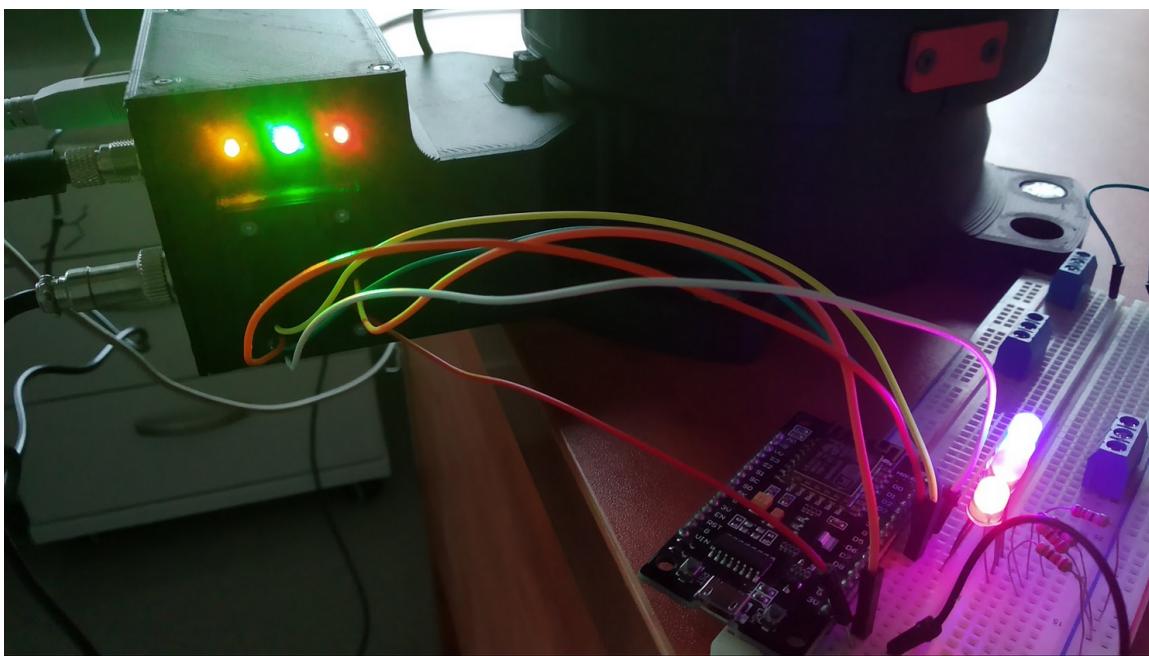
The screenshot shows the Arduino IDE interface with the following details:

- Arduino IDE:** Version 1.8.10, showing the file 'modbus_esp_test'.
- Serial Monitor:** Connected to 'COM7'. It displays the following log output:

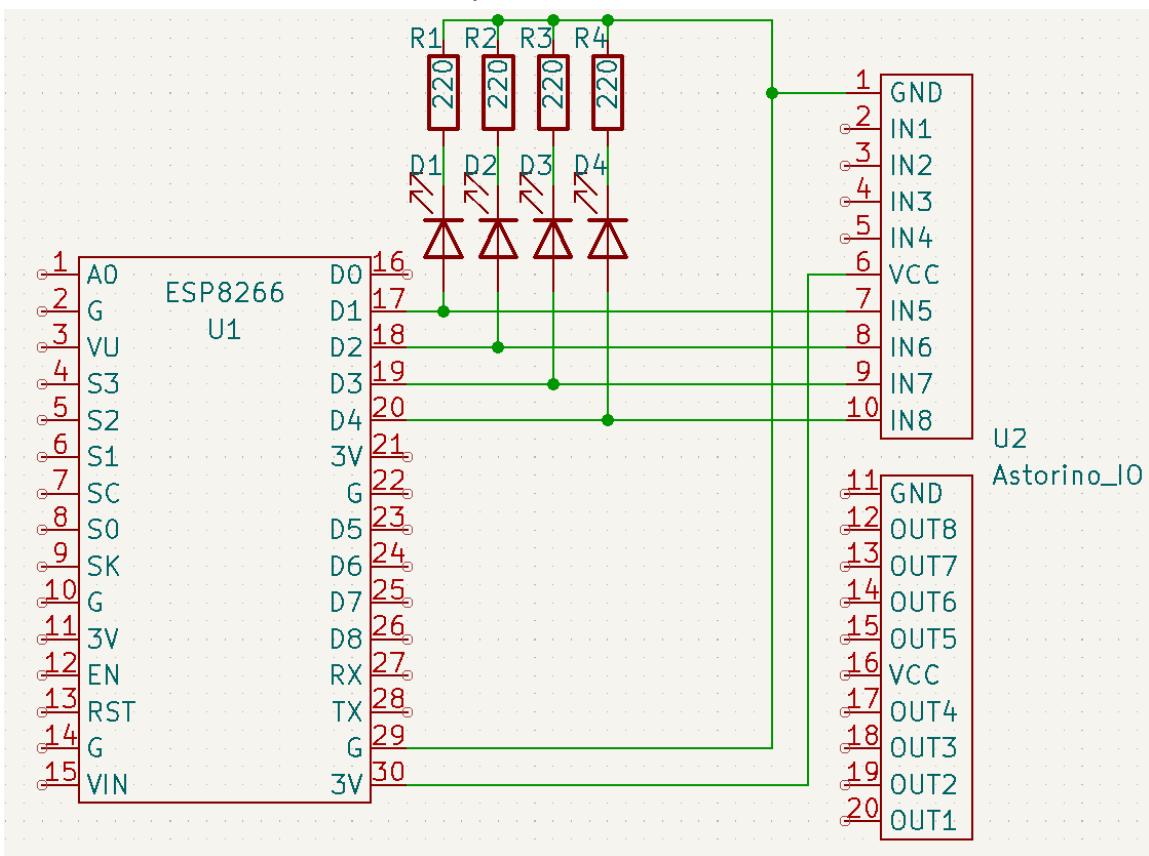

```

GC [101] (1)
GC [102] (1)
GC [103] (0)
GH [200] (9624)
GC [100] (1)
GC [101] (1)
GC [102] (1)
GC [103] (0)
GH [200] (9624)
SH [200] (9624) -> 9625
SC [104] (1) -> 1
GC [100] (1)
GC [101] (1)
GC [102] (1)
GC [103] (0)
GH [200] (962)
            
```
- Code:** The code for 'modbus_esp_test' is displayed in the editor, showing the logic for handling Modbus requests and writing to coils.

Program na ESP, wraz z logami



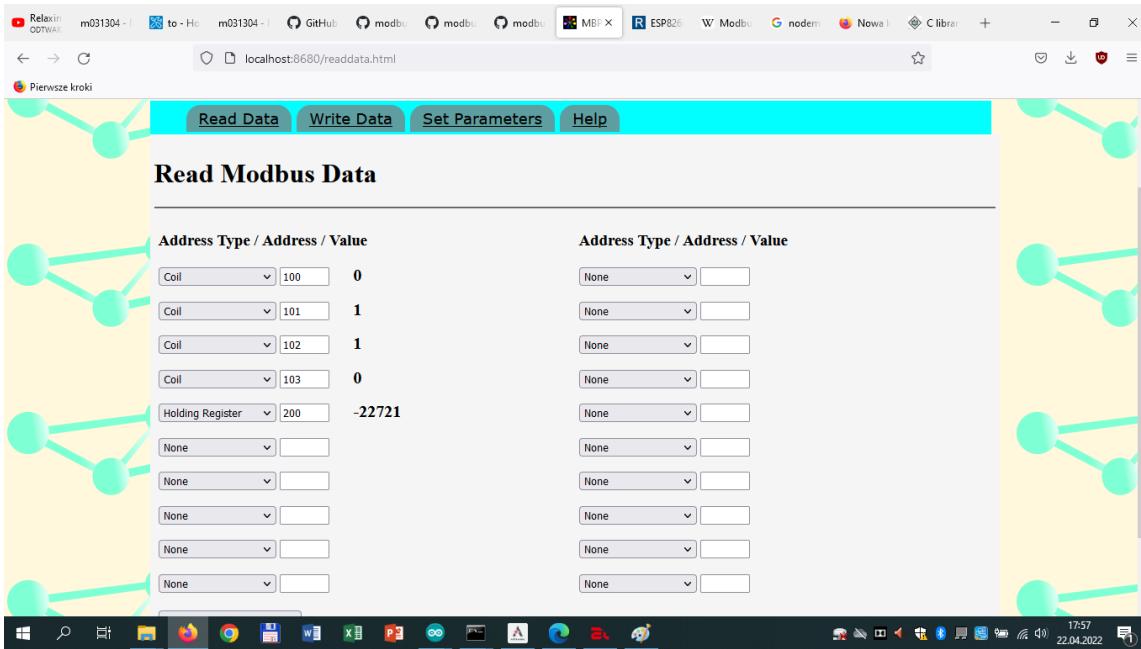
Połączenie ESP z Robotem



Schemat połączeń elektrycznych

Podczas testowania aplikacji, napotkaliśmy niespodziewany błąd. Sterownik PLC potrafił niespodziewanie się zawiesić w taki sposób, że przestał aktualizować wartości czytanych rejestrów/cewek z mikrokontrolera, a nadpisywał tylko te rejestrów/cewki, których stan był inny niż aktualnie myślał że jest. Po wielu godzinnych testach nie byliśmy w stanie

zdeterminować konkretnego powodu tego problemu. Sądzimy, że problem leżał po stronie sterownika PLC a nie ESP. Podczas testów, wykorzystany został darmowy program **MbProbe**, będącym jednym z narzędzi paczki **MbLogic**, zbioru przydatnych aplikacji powiązanych z symulacją i diagnozowaniem Modbus'a w różnej postaci, MbProbe to aplikacja napisana w Pythonie, uruchamiana na komputerze PC, udostępniająca interfejs do wysyłania arbitralnych komunikatów odczytu lub zapisu do urządzeń w sieci poprzez ModbusTCP. Zawsze bezbłędnie odczytywało i zapisywało wartości do ESP, nawet kiedy PLC przestawał działać poprawnie.



Interfejs MbProbe

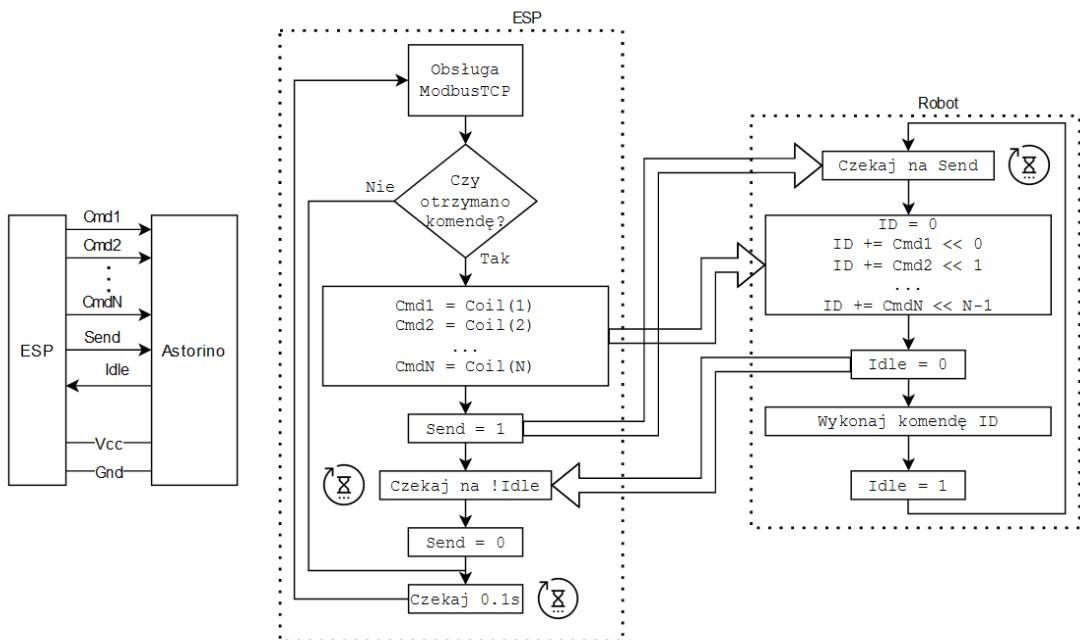
Podejrzewamy, że problem miał do czynienia ze zbyt częstym czytaniem wartości z mikrokontrolera (w celach synchronizacji) oraz tym, że wartość rejestru testowego mogła zostać nadpisana w momencie, kiedy PLC oczekiwał na odpowiedź, potwierdzającą, że zapisana wartość jest taka sama, jaką kazał tam zapisać. Problem ten można było na szczęście łatwo obejść, stosując zamiast automatycznej synchronizacji wartości rejestrów, wysyłanie i czytanie wartości programowo, w sposób kontrolowany, wywołując odpowiednią komendę na PLC. Podobne problemy nie wystąpiły jednak w przyszłych testach.

Rozbudowa komunikacji

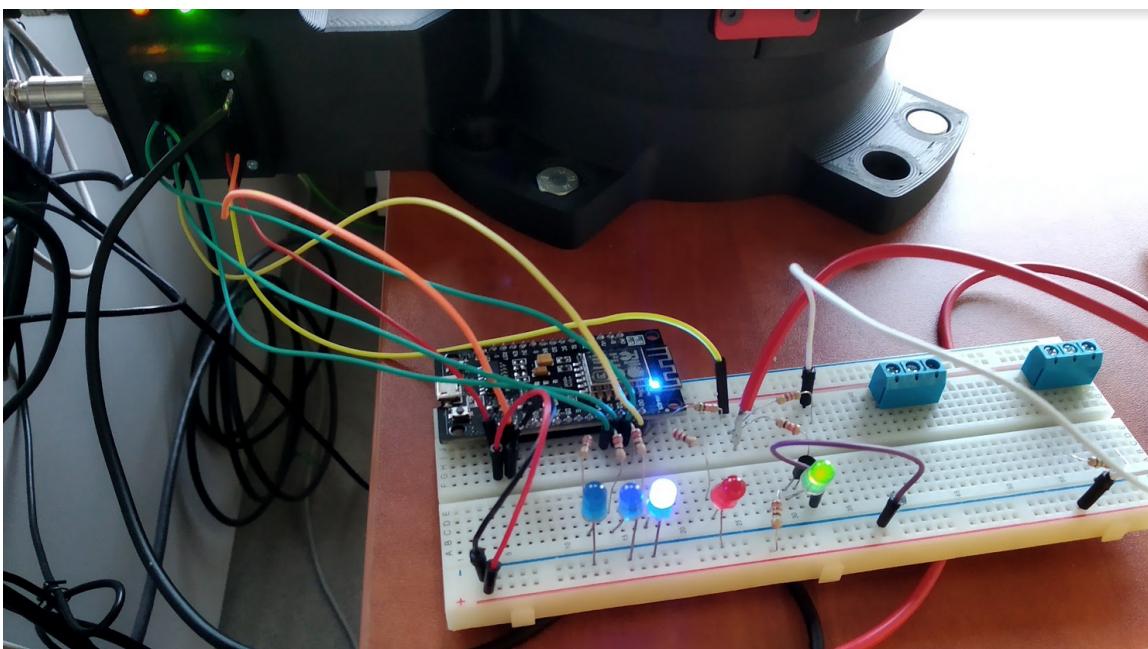
Aktualna aplikacja jest jednak zbyt prosta, aby można ją było zastosować w ostatecznej fazie projektu. Robot cały czas odczytuje wartości wszystkich wejść cyfrowych i wykonuje powiązane z nimi komendy. Oczekujemy jednak, by można było robotowi zadać jakąś konkretną komendę tylko raz. Aktualny stan rzeczy nie pozwala na dowiedzenie się, czy robot odebrał komendę, czy aktualnie jakąkolwiek komendę wykonuje, itd. Całkowity brak takiej synchronizacji mógłby łatwo doprowadzić do potencjalnego niewykonania wysłanej komendy, lub wykonania jakiejś komendy wielokrotnie. Projekt został więc lekko zmieniony.

Oprócz linii służących do przesyłania ID komendy w postaci binarnej, dodano jedno wejście i jedno wyjście, służące do synchronizacji. Jeśli ESP posiada zakolejkowaną komendę do wysłania do Robotu, wystawia sygnał "**Send**". Jeżeli robot aktualnie czeka na następną

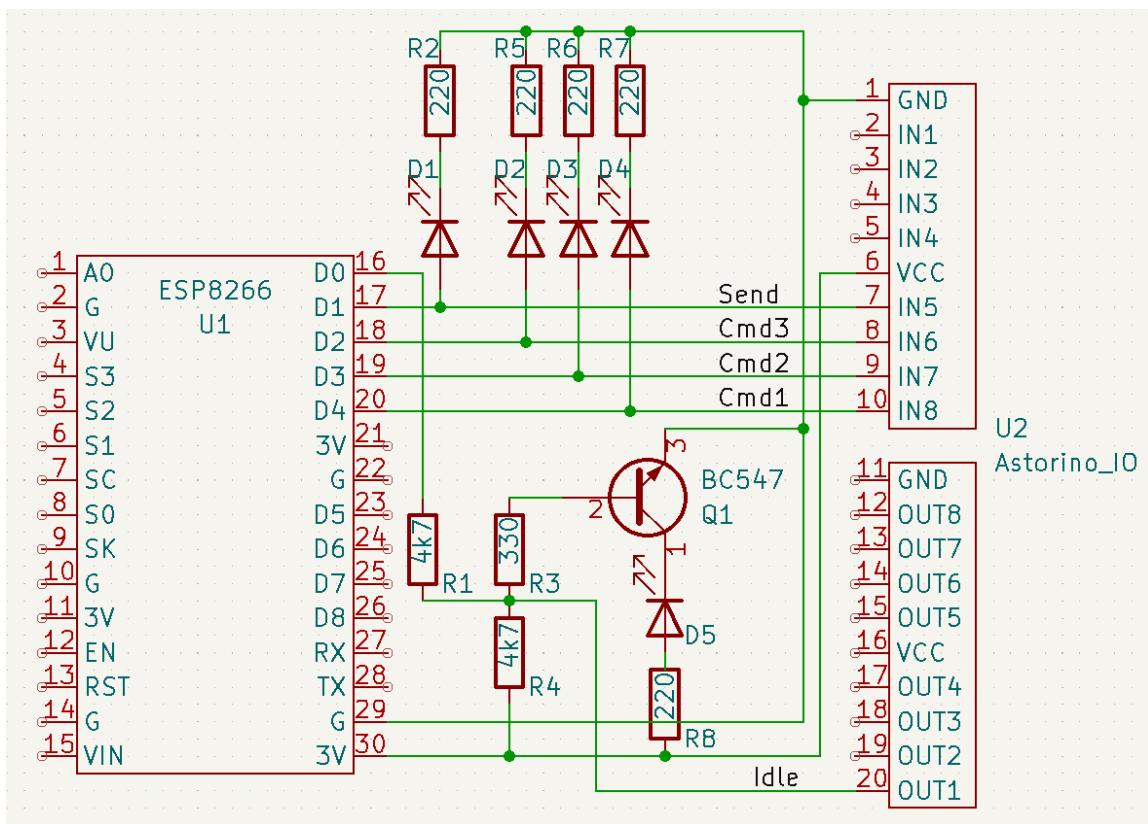
komendę, wystawia sygnał "Idle". Taki system zapewnia, że każda wystawiona komenda odczytana zostanie dokładnie 1 raz, nie mniej, nie więcej. Dodatkowo, stan pinu Idle odczytywany przez ESP przypisywany jest do odpowiedniego rejestru w serwerze ModbusTCP, co pozwala na odczyt przez PLC, czy jakiekolwiek inne urządzenie sterujące w sieci, czy robot aktualnie wykonuje jakąś komendę.



Schemat działania programów i komunikacji

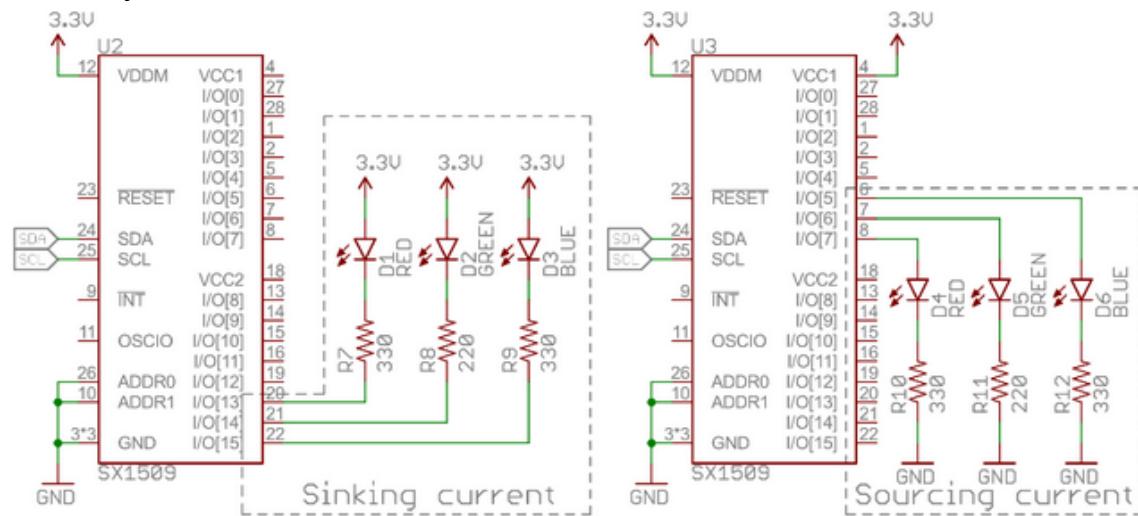


Połączenie ESP z Robotem



Schemat połączeń elektrycznych

Jak widać na zdjęciu, zastosowano więcej diod LED niż poprzednio. Niebieskie reprezentują stan bitów **Cmd**, czerwona stan **Send**, a zielona **Idle**. Zauważono, że zielona dioda, pomimo bycia podpięta przez tranzystor w celu wyeliminowania obciążenia na wyjściu robota jakie by powodowała, nie włącza się. Okazało się, że wyjścia robota wymagają dodania rezystora Pull-Up, dlatego tak też zrobiono. Jest to jednak niezgodne ze schematem zamieszczonym w instrukcji robota:



Schematy połączeń I/O do diod LED z instrukcją Robotu Astorino

Jak widać, ewidentnie wyjścia powinny być w stanie bezpośrednio zasilić diodę. Robot może pracować w dwóch trybach: NPN (po lewej) oraz PNP (po prawej). Niezależnie jednak w

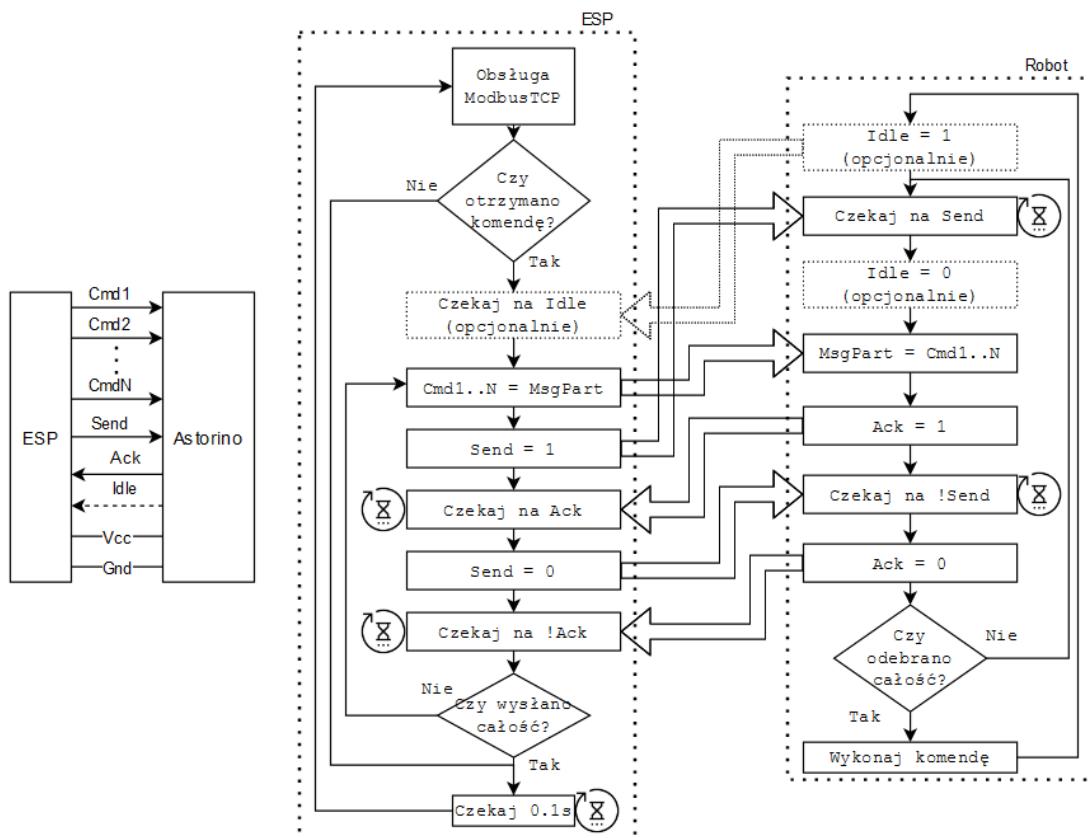
jakim się znajdował, żaden sposób podłączenia diody nie powodował jej zaświecenia. Robot nie był w stanie wyprodukować odpowiedniego prądu, dodano więc rezystor wspomniany rezystor Pull-Up. Warto jeszcze dodać, że zauważaliśmy, że robot pracując w trybie NPN neguje wejścia przy odczytywaniu ich w programie, a w trybie PNP neguje wyjścia. Nie wiemy dlaczego postanowiono, by działał w taki sposób. Aktualnie pracuje w trybie NPN. Podczas testów tego systemu nie zaobserwowano żadnych desynchronizacji ze sterownikiem PLC.

Dalsza rozbudowa komunikacji - zwiększenie ilości komend

W aktualnej formie, możliwe jest zaprogramowanie maksymalnie 8 komend na robocie. Wynika to z tego, że przesyłane komendy są 3-bitowe ($2^3 = 8$). Potencjalnie możemy jednak chcieć móc wykonać więcej poleceń, zależnie od ostatecznego zastosowania.

Pierwszym oczywistym sposobem na zwiększenie liczby komend jest zwiększenie podłączonych wejść robota do ESP. Robot posiada 8 wejść, z czego jedno musi być wykorzystane jako sygnał "Send". Dlatego takie rozwiązanie może pozwolić na maksymalnie $2^7 = 128$ możliwych komend. Jest to najprawdopodobniej wystarczająca liczba do większości możliwych zastosowań robota. Są jednak z tym rozwiązaniem 2 problemy. Po pierwsze - ograniczona ilość wyjść cyfrowych na ESP. Można ten problem jednak obejść poprzez dodanie adresonalnego portu rozszerzeń, np. na magistrali SPI. Pozwoliłoby to całkowicie wyeliminować ten problem. Innym mankamentem jest jednak to, że wykorzystujemy wszystkie wejścia Robota. Ogranicza to jego możliwości, uniemożliwia np. sprawdzenie przez robota czy aktualnie chwytak jest otwarty lub zamknięty, nie można utworzyć sygnału awaryjnego wymuszającego zatrzymanie robota programowo, itd. Każdy nowy sygnał zmniejszyłby maksymalną ilość komend.

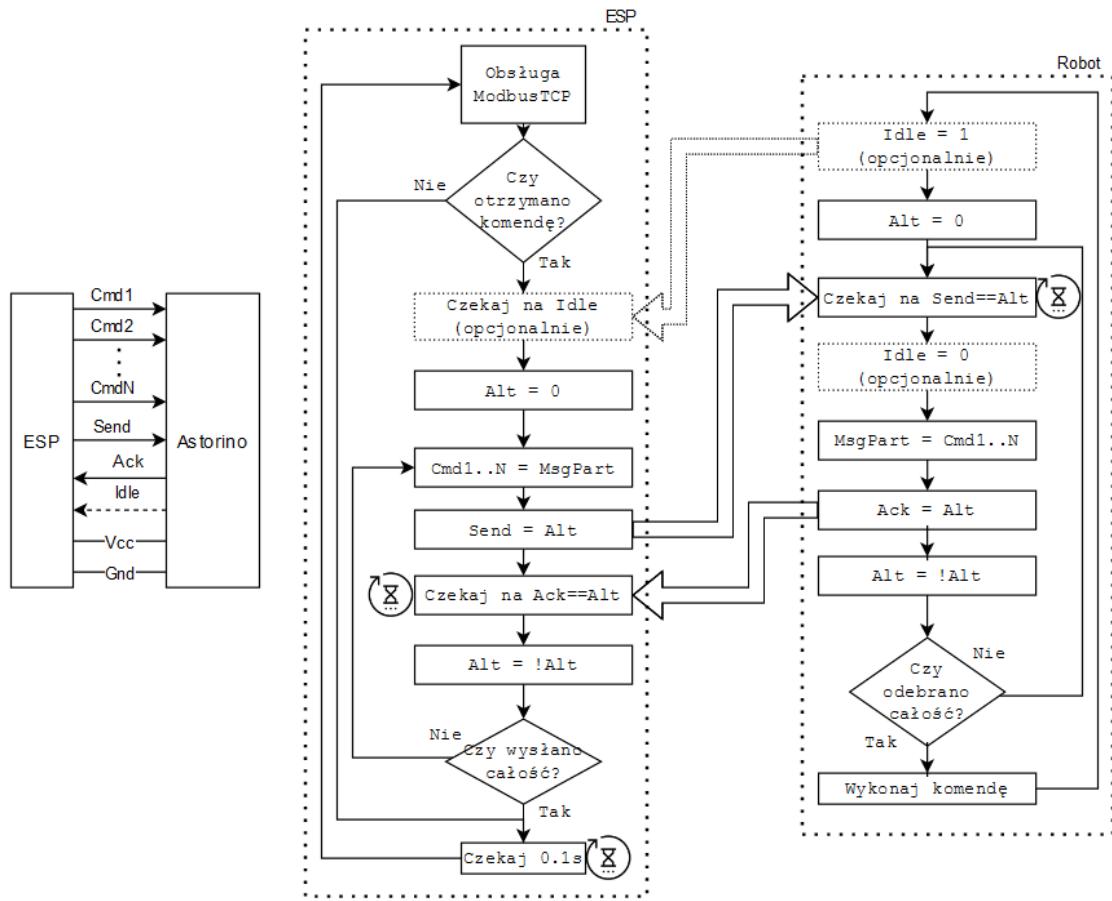
Rozwiązaniem pozwalającym na posiadanie dużej ilości wejść innego przeznaczenia, niż do przesyłania komend, jest lekkie zmodyfikowanie algorytmu przesyłania komend. Ponieważ przesyłanie te jest synchronizowane, zarówno ESP jak i Robot wiedzą w jakim stanie aktualnie znajdują się nawzajem. Przesłanie ID komendy mogłoby zostać podzielone na arbitralnie dużą ilość etapów. ESP wysłałby następujące po sobie kilka komend, a Robot odczytał je wszystkie po kolei, a następnie skonkatenował ze sobą, otrzymując ostateczne ID komendy do wykonania. Takie rozwiązanie pozwoliłoby na praktycznie **arbitralnie dużą ilość komend do zaprogramowania, i to przy wykorzystaniu wyłącznie 3 połączeń pomiędzy ESP i Robotem**: Send, Idle oraz Cmd1. ID komend wysyłane były wtedy bit po bicie. Zwiększenie połączeń Cmd do przesyłania ID komendy spowodowałoby jedynie zwiększenie prędkości przesyłu. Nie można jednak naiwnie zakolejkować po prostu kilka komend (będących tak naprawdę częściami całej komendy) i przesłać je po kolejnych z aktualnym algorytmem, ponieważ aktualnie synchronizacja urządzeń opiera się na tym, że robot przez jakiś niekrótki czas wykonuje komendę. Należało więc dodać jeszcze jedno wyjście z robota, nazwane "Ack", i przy jego pomocy zaimplementować pełnoprawny handshaking. Sygnał Idle stałby się wtedy opcjonalny, lecz może na chwilę obecną pozostać, ponieważ pozwala na podgląd aktualnego stanu w jakim znajduje się robot, co może być użyteczne przy kolejkowaniu nowych komend i debugowaniu.



Schemat działania programów i komunikacji

Polepszenie handshacking'u

Warto zauważyć, że głównym powodem dla którego trzeba w tym rozwiążaniu wprowadzić handshaking, a nie np. zwykłą linię pulsów zegara, jest nieznana prędkość pracy Robota. Zaprezentowany wyżej algorytm wykorzystuje 4-stronny handshaking, co po dłuższej analizie okazało się być nadmiarowym rozwiązaniem. ESP musi jedynie wiedzieć, kiedy Robot odczytał wszystkie przesłane bity komendy. Nie musi jednak wiedzieć, że, potocznie mówiąc, robot wie, że ESP wie. Etapu "Send = 0" oraz "Czekaj na !Ack" w ESP oraz "Czekaj na !Send" i "Ack = 0" na Robocie są więc zbędne. Zwykłe usunięcie tych kroków z algorytmu wprowadza jednak inny błąd - Robot działa zbyt wolno by móc niezawodnie wykrywać zbocza zbyt szybko zmieniających się sygnałów. Urządzenia potrafią wykryć jedynie aktualny stan sygnału, dlatego nie można zastąpić np. polecenia "Send = 1" krótkim pulsem, po którym sygnał Send wraca do 0. Jeśli by zastąpić to polecenie długim pulsem, to przeciwodzieliby to optymalizacji polegającej na przyśpieszeniu algorytmu przesyłania. Rozwiązaniem jest więc nie przebudowywać algorytmu w taki sposób, by oczekiwano na narastające zbocza, ale by oczekiwano na jakiekolwiek zbocza. Z każdą iteracją pętli przesyłu komendy sygnały Send i Ack ustawiany by były na zmianę na 1 i na 0, zaś urządzenie czekałyby na zmianę danego w stosunku do wartości z poprzedniej iteracji.



Schemat działania programów i komunikacji

Powyższy diagram przedstawia ostatecznie zastosowany algorytm przesyłu komend do Robotu.

Badanie prędkości przesyłu

Dokonano pomiaru czasu przesyłu kodu komendy do robota w zależności od zastosowanego algorytmu. Brane były pod uwagę długość słowa komendy, ilość bitów sygnałów Cmd, oraz to, czy zastosowana została optymalizacja opisana w poprzednim akapicie.

Długość słowa	Ilość sygnałów Cmd	Oczekiwanie na dowolne zbocze	Czas przesyłu jednej komendy
16 bity	1	nie	0,869 s
16 bity	2	tak	0,255 s
16 bity	4	nie	0,260 s
16 bity	4	tak	0,150 s
32 bity	2	tak	0,461 s

32 bity	4	nie	0,470 s
32 bity	4	tak	0,250 s

Jak widać, każda forma optymalizacji (zwiększenie dwukrotnie ilości sygnałów Cmd, oczekiwanie na dowolne zbocze) powoduje około 2x przyspieszenie przesyłania komend, zaś zwiększenie dwukrotnie długości komendy, powoduje 2x spowolnienie. W ostatecznej wersji komunikacji zastosowano 16-bitowe słowo, 2 sygnały Cmd, oraz oczywiście oczekiwanie na dowolne zbocze.

Zestawienie zaprogramowanych komend i nauczonych punktów robota

Poniżej zamieszczono tabelę reprezentującą wszystkie zaprogramowane komendy robota:

Kod	Opis
0,1,2	Wzięcie części dolnej elementu z odpowiedniego magazynu wejściowego oraz ustawienie do czujnika odległości, aby sprawdzić czy została wzięta
5,6,7	Wzięcie części górnej elementu z odpowiedniego magazynu wejściowego oraz ustawienie do czujnika odległości, aby sprawdzić czy została wzięta
20	Ustaw wziętą dolną część na obrotnej montażowej
21	Zamontuj wziętą część górną na części dolnej, ustawionej na obrotnej.
10	Przenieś złożony element z obrotnej montażowej do magazynu wyjściowego
15	Przenieś część dolną z obrotnej montażowej do magazynu zbytowego
16	Zdemontuj z elementu i przenieś część górną z obrotnej montażowej do magazynu zbytowego
17	Przenieś złożony element z obrotnej montażowej do magazynu zbytowego

Oprócz tych komend, wykonywanych podczas automatycznej pracy stacji, stworzono także wiele prostszych komend, wykorzystywanych w fazie testów.

Kod	Opis
50	Zamknij chwytak
51	Otwórz chwytak
100, 101, 102	Wzięcie części dolnej elementu z odpowiedniego magazynu wejściowego
105, 106, 107	Wzięcie części górnej elementu z odpowiedniego magazynu wejściowego
110	Weź część dolną z obrotnej montażowej

111	Zdemontuj z elementu i weź część górną z obrotnicy montażowej
112	Weź zmontowany element z obrotnicy montażowej
120	Odstaw część dolną na obrotnicy montażowej
121	Zamontuj część górną na części dolnej na obrotnicy montażowej
122	Odstaw zmontowany element na obrotnicy montażowej
120	Wyrzuć trzymaną część dolną
121	Wyrzuć trzymaną część górną
122	Wyrzuć trzymany złożony element
140	Odstaw trzymany złożony element do magazynu wyjściowego

Lista punktów nauczonych na robocie, wykorzystywanych w implementacji komend:

Numer	Opis
P5	Magazyn wejściowy części dolnych 1
P6	Magazyn wejściowy części dolnych 2
P7	Magazyn wejściowy części dolnych 3
P8	Magazyn wejściowy części górnych 1
P9	Magazyn wejściowy części górnych 2
P10	Magazyn wejściowy części górnych 3
P19	Sprawdzanie obecności wziętej części czujnikiem odległości
P20	Części dolna na obrotnicy montażowej
P21	Nieprzykręcona część górną na obrotnicy montażowej
P22	Przykręcana część górną na obrotnicy montażowej
P25	Punkt pośredni pomiędzy obrotnicą a magazynem wyjściowym/zbytowym
P26	Punkt pośredni pomiędzy czujnikiem odległości a obrotnicą montażową
P30	Magazyn wyjściowy
P31	Magazyn zbytowy

Problem z wejściami cyfrowymi robota

Początkowo np. oczekивание на sygnały Ack oraz Send w programie robota prezentowały się za pomocą zwykłej pętli while:

```
WHILE ((SIG(1008)) == awaitsend) DO
    TWAIT 0.2
END
/// odczyt i obsługa bitów Cmd...
```

Powyższy blok kodu sprawdza, czy sygnał wejściowy o numerze **8** (w naszym przypadku Send) będzie miał wartość zmiennej "**awaitsend**". Jeżeli ma, to czeka dalej, jeżeli nie ma, to wychodzi z pętli i wykonuje dalszą część programu. Pewnego jednak razu postanowiono zmniejszyć opóźnienie wewnątrz pętli z 0.2s na możliwie mały - **0,01s**. Kiedy zostawiliśmy na chwilę bezczynnie robota po tej zmianie, zauważaliśmy, że zaczął samoczynnie wykonywać komendę. Po kilku testach okazało się, że jeżeli na wejście robota podane jest zasilanie (logiczna 1) to średnio raz na około 1000 wywołań komendy "**SIG**" odczytywane jest niepoprawnie 0. Wcześniej tego błędu nie zauważono, ponieważ z opóźnieniem **0,2s** nie zdążyło się wykonać odpowiednio dużo iteracji, lecz pozostawiając robota na kilkanaście minut i tak zaobserwowano problem. Jest to oczywiście bardzo niepożądane zjawisko. Błąd został zgłoszony konstruktorowi robota, lecz nie spieszyl się z utworzeniem aktualizacji oprogramowania, dlatego byliśmy zmuszeni obejść problem na własną rękę.

Jednym z pomysłów obejścia problemu było zastosowanie komendy **SWAIT**, która stopuje wykonanie programu do momentu napotkania sygnału wejściowego o odpowiedniej wartości. Pozwoliłaby ona na zdecydowanie krótszy kod i większą wygodę jego pisania, lecz, jak zostało wspomniane, blokowałaby wykonanie programu, co przy dalszej jego rozbudowie mogłoby powodować problemy, jeśli np. chcielibyśmy w połowie odczytywania komendy przerwać procedurę. Niestety, okazało się że ta polecenie to cierpi na to samo co **SIG**.

Prowadząc dalej testy, poprzez stworzenie wielu pętli bez żadnego opóźnienia, w których sprawdzane zostały stany wejść, zauważaliśmy, że błędny odczyt nigdy nie występuje 2 razy z rzędu (a przynajmniej na tyle rzadko, by ani razu tego nie uświadomić). Postanowiono więc sprawdzać wartość stanu wejścia dwukrotnie:

```
ret = awaitsend
WHILE ret == awaitsend DO
    WHILE ((SIG(1008)) == awaitsend) DO
        TWAIT 0.2
    END
    ret = (SIG(1008))
END
/// odczyt i obsługa bitów Cmd...
```

Jak widać, powyższa pętla wewnętrzna oczekuje na stan wejścia, tak jak miało to miejsce wcześniej. Jeśli pętla zostanie przerwana, to wejście jest odczytywane ponownie, i pętla zewnętrzna sprawdza, czy odczytane wyjście ma faktycznie oczekiwana wartość. Pozostawiając robota wykonującego tą pętlę przez kilkadesiąt minut nie zaobserwowano ani jednego podwójnie błędnie odczytanego bitu. Co jeszcze warto dodać, w tym przypadku

nie trzeba zmieniać opóźnienia w poleceniu **TWAIT** na mniejsze, ponieważ mikrokontroler z którym robot się komunikuje działa znacznie szybciej niż robot, przez co, jeśli transmisja została rozpoczęta, to sygnał **SEND** zdąży się zmienić na odpowiedni pomiędzy pierwszymi wywołaniami pętli zewnętrznej, nawet ani razu (nie licząc pierwszego bitu) nie wywołując polecenia **TWAIT**.

Kiedy już otrzymaliśmy aktualizację oprogramowania, która rzekomo miała poprawić błąd z błędym odczytem, to doznaliśmy w zespole niemałego zaskoczenia. Otóż w celu naprawy błędu, nowy firmware robota przy każdym wywołaniu komendy **SIG** dodaje opóźnienie **100ms**, co spowodowało, że jedna 16-bitowa komenda wysyłała się do robota w zamiast **0,2s**, to w **7s!**. Jest to o **3400%** wolniej niż przed aktualizacją. A co najgorsze... aktualizacja nie naprawiła błędu - spowodowała jedynie że występuje rzadziej !!!

Zostaliśmy więc zmuszeni pozostać przy starszej wersji oprogramowania. Na szczęście, po kilku tygodniach i osobistej wizycie konstruktora robota w naszym laboratorium, oprogramowanie zostało poprawnie i błąd faktycznie wyeliminowany, a opóźnienie komend **SIG** zmniejszone do **20ms**. Niestety sprawiało to, że dalej komendy przesyłają się wolniej niż wcześniej (w około **2s**). Ponieważ jednak błąd został wyeliminowany, to mogliśmy pozbyć się redundantnych wywołań poleceń **SIG**, zmniejszając ostatecznie czas przesyłania komendy do około **1,3s**.

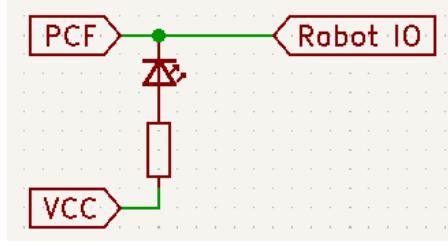
Porty rozszerzeń

Zastosowane ESP ma dostępne tylko 8 wejść/wyjść cyfrowych ogólnego przeznaczenia (GPIO). Każda linia sygnałowa występująca w powyższym opisie algorytmu przesyłu komend do robota wymaga własnego GPIO. Dodając do tego wszystkie inne sygnały kontrolne, sygnały z czujników na stanowisku, bardzo szybko zaczyna tych GPIO brakować. Rozwiązaniem jest zastosowanie portów rozszerzeń, czyli układów, które posiadają własne GPIO i pozwalają na komunikację z mikrokontrolerem poprzez jakąś konkretną magistralę szeregową.

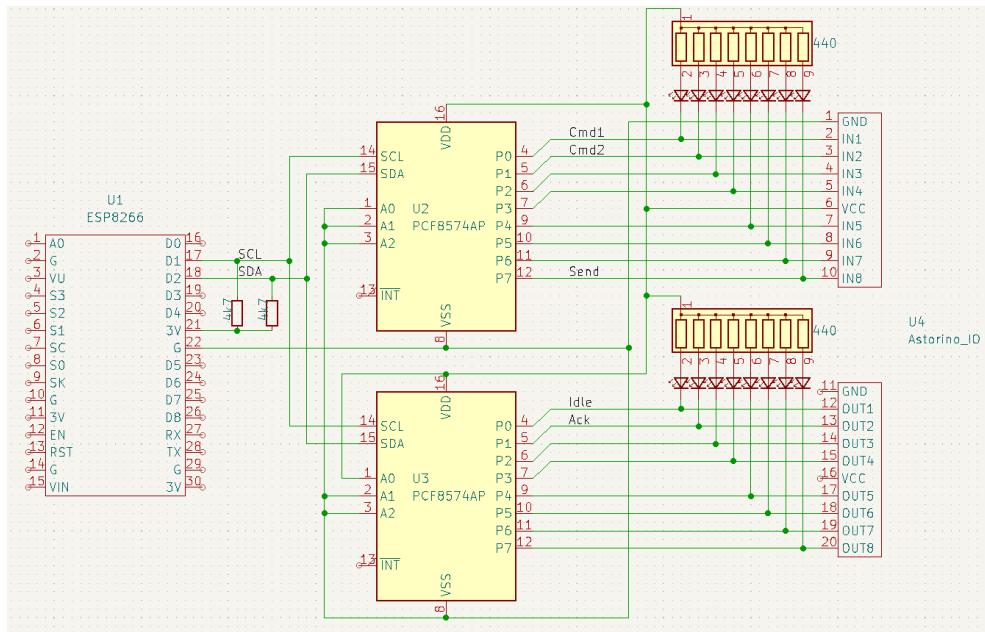
Do naszego projektu wybraliśmy układy **PCF8574AP**. Wybór akurat tych układów wynikał wyłącznie z tego, że mieliśmy kilka takich układów na stanie, oraz spełniały one wszystkie wymagania potrzebne do poprawnego działania systemu.

Zgodnie z notą katalogową, układy PCF to dwukierunkowe, 8-bitowe porty rozszerzeń, pracujące na napięciu od 2,5V do 6V, działające na magistrali **I²C**. Magistrala ta wymaga połączenia mikrokontrolera dwoma liniami sygnałowymi : danych (SDA) oraz zegara (SCL). Obsługa magistrali posiada domyślną implementację w bibliotece środowiska Arduino IDE, z której korzysta ESP, dlatego nie stanowi to żadnego problemu. Trzeba jedynie pamiętać, że linie te muszą mieć podpięty rezystor pull-up.

Każdy bit modułu PCF można osobno ustawić na logiczną 1 lub 0, oznaczające odpowiednio ustawienie napięcia na pinie na napięcie zasilania (z dość ograniczającym limiterem wyjściowego natężenia) lub zwarcie pinu do masy (z maksymalnym prądem wpływającym do pinu rzędu 25 mA). Urządzenie te jest więc typu open-collector. Dodatkowo, jeżeli na bieżąco ustalona jest aktualnie 1 (czyli nie jest zwarty do masy), to można odczytać stan pinu, jakby był pinem wejściowym. Połączenie pomiędzy pojedynczym pinem PCF a jednym pinem IO robota można zaprezentować następująco:



Obecność diody LED jest opcjonalna i służy wyłącznie do określenia aktualnego stanu sygnału. Ważny jest jednak rezystor, pełniący rolę pull-up'a, który (jeśli ani PCF ani Robot nie zwróci linii do masy) daje domyślnie wysoki sygnał. Warto dodać, że dioda jest invertująca. Powyższa konfigurację zastosowano przy wszystkich 16 złączach IO robota, co wymagało dwóch 8-bitowych modułów PCF.



Schemat połączenia ESP i Robotu poprzez porty rozszerzeń PCF

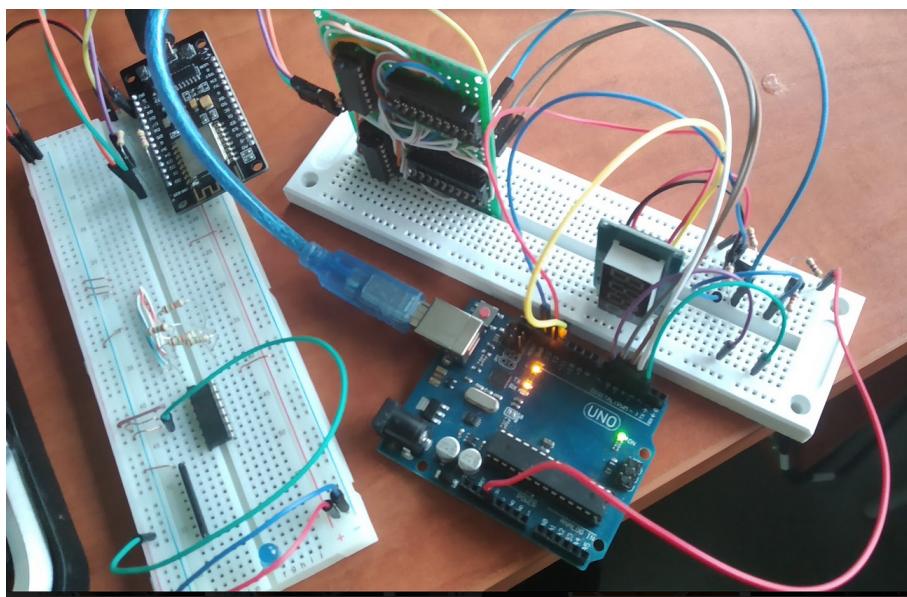


Podłączone mikrokontroler, płytka z modułami PCF oraz robot

Symulator Robota

W celu ułatwienia procesu testowania oprogramowania mikrokontrolera niezbędny jest dostęp do robota, który będzie przetwarzał przesypane komendy oraz zwracał odpowiednie sygnały wyjściowe. Niestety, dostęp do robota jest wyłącznie w laboratorium, co uniemożliwia testowanie oprogramowania z domu. Dodatkowo, patrząc na częstotliwość psucia się robota, często nawet w laboratorium jest on niedysponowany. Stworzono więc swego rodzaju moduł symulujący działanie robota.

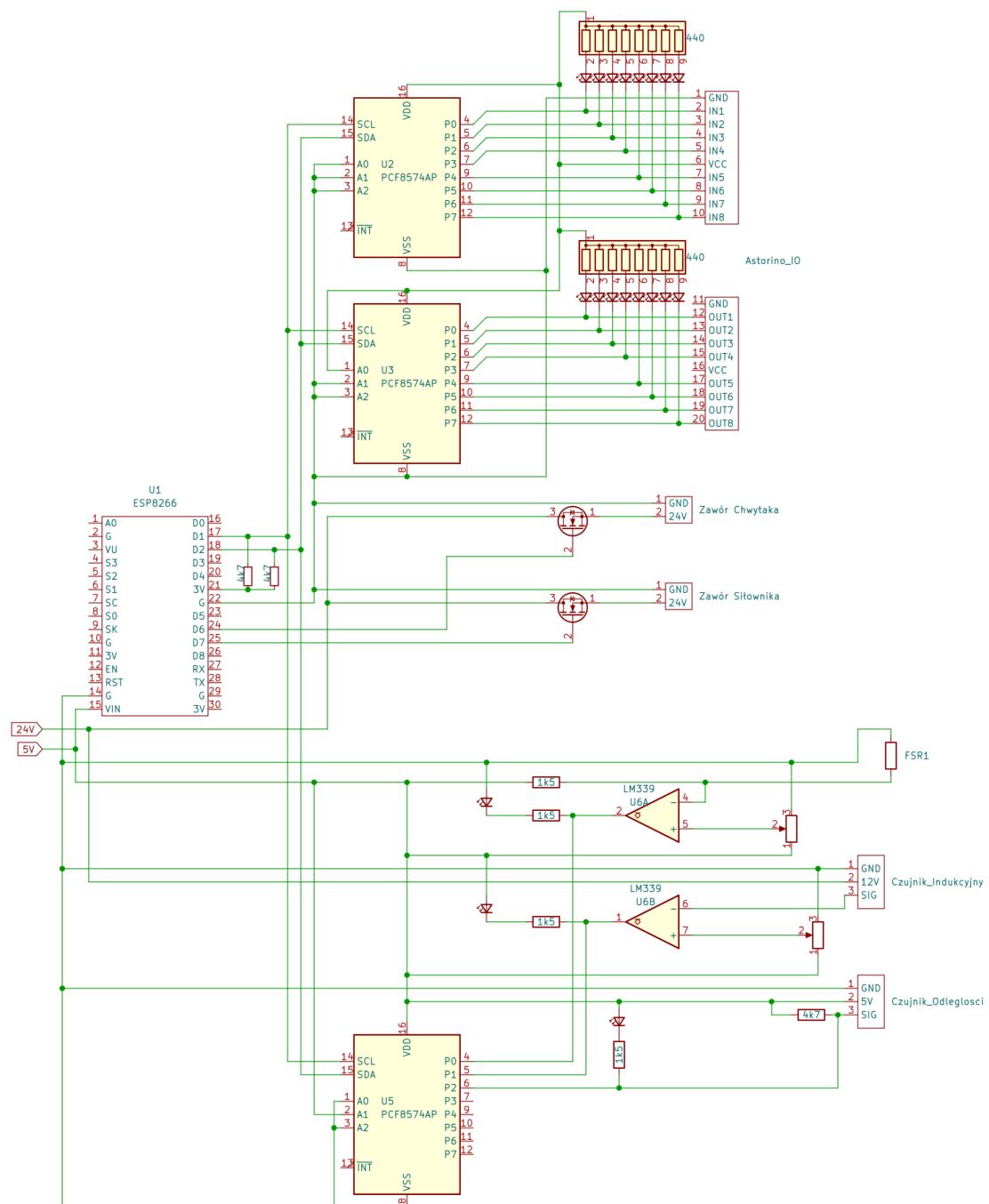
Modułem tym jest płytka Arduino Uno. Wgrane na nią zostało oprogramowanie udające robota. Fragment programu odpowiadający za transmisję komend został praktycznie 1-1 przekopiowany z kodu robota, przerobiony jedynie w taki sposób, by był zgodny ze standardem języka C++, wykorzystywanego na arduino. Samo wykonywanie przesłanych komend zostało zastąpione kilkusekundowym opóźnieniem. Arduino na szynie I2C posiada podpięty wyświetlacz 8-segmentowy, na którym wyświetlany jest kod aktualnie wykonywanej komendy. Wystarczy więc podpiąć moduły PCF zamiast do robota, to do odpowiednich pinów Arduino Uno, co daje nam możliwość testowania oprogramowania mikrokontrolera bez potrzeby podpięcia do robota, oraz bez jakiejkolwiek ingerencji w logikę mikrokontrolera - z jego punktu widzenia wykonywany jest dokładnie ten sam program, co jakby był faktycznie podpięty do robota.

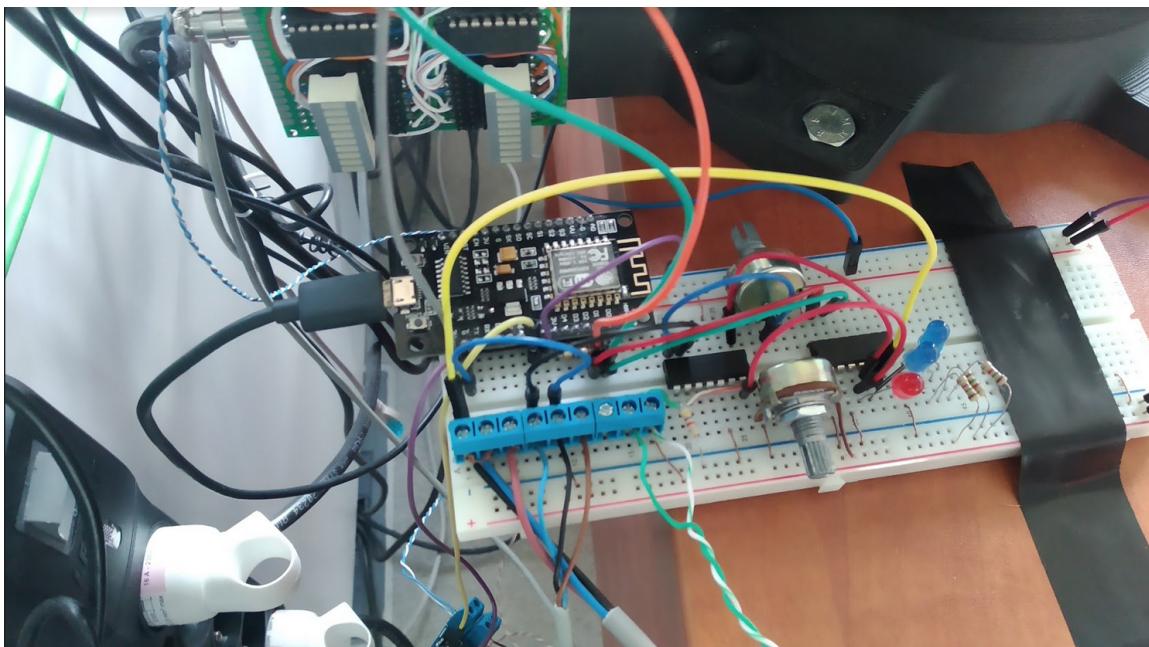


Mikrokontroler ESPO podpięty do symulatora robota na Arduino Uno

Kompletny schemat połączeń elektrycznych

Poniżej zamieszczono kompletny schemat połączeń elektrycznych wszystkich urządzeń. W górnym fragmencie widać opisane wcześniej podłączone do magistrali I²C porty rozszerzeń PCF pośredniczące z IO robota. Potem widać sterowanie zaworami elektrycznymi. Ponieważ wymagają one zasilania 24V, wykorzystaliśmy tranzystory MOSFET, sterowane bezpośrednio sygnałami wyjściowymi z mikrokontrolera. W dolnej sekcji widać wszystkie czujniki zamontowane na stanowisku. Rezystor siło-czuły FSR daje sygnał analogowy, dlatego należy go przepuścić przez komparator. Poziom napięcia, zależny od siły nacisku na FSR, wykrywany przez komparator może być ustawiany potencjometrem. Czujnik indukcyjny zwraca napięcie o bardzo małej wartości (zaledwie 0,5V), dlatego również postanowiliśmy go przepuścić przez komparator, gdyż inaczej nie bylibyśmy go w stanie wykryć. Czujnik odległości zwraca już sygnał cyfrowy 0 lub 5V, dlatego został podłączony bezpośrednio do modułu PCF.





MQTT

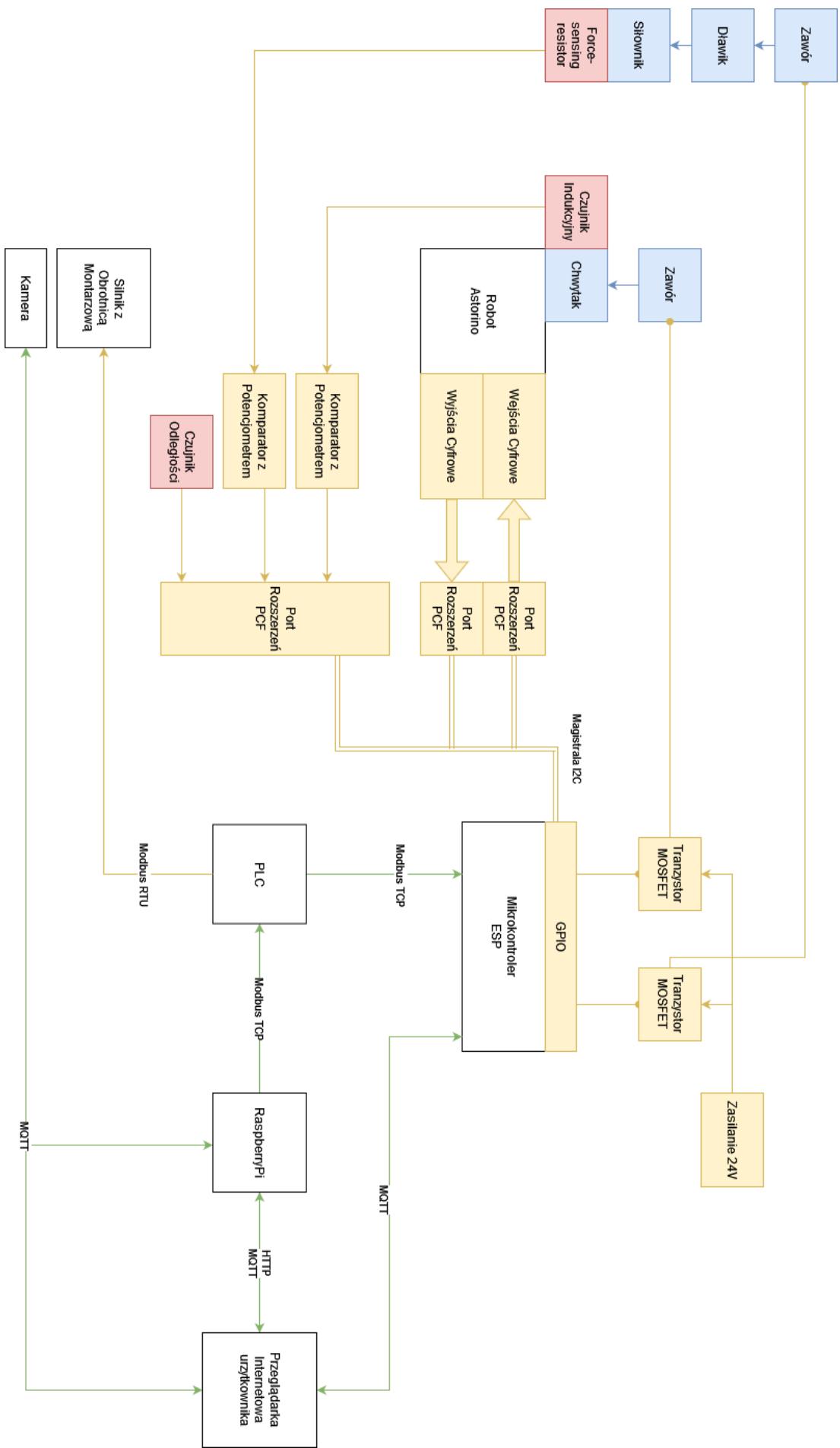
Poniżej przedstawiono zestawienie wszystkich zaimplementowanych rodzajów wiadomości przesyłanych poprzez protokół **MQTT**, jak i ich format oraz które urządzenia je przetwarzają. Urządzenie o nazwie “**UI**” odnosi się do przeglądarki użytkownika, połączonej z serwerem **HTTP** hostowanym na **Raspberry Pi**, gdyż przeglądarka staje się wtedy osobnym klientem **MQTT**.

Nazwa	Format	Urządzenia	Opis
img/jpeg	Binarny	Nadające: ESP32 Cam Nasłuchujące: Raspberry Pi UI	Obraz zarejestrowany przez kamerę w formacie JPEG. Raspberry Pi po odebraniu obrazu przeprowadza analizę koloru widzianego elementu, a na UI ten obraz się wyświetla.
analysis	JSON	Nadające: Raspberry Pi Nasłuchujące: UI	Wyniki analizy koloru obrazu. Zawiera uśredniony kolor analizowanego wycinka w formatach RGB oraz HSL, odchylenie standardowe kanałów RGB w analizowanym wycinku obrazu, oraz rozmiar wykorzystanego wycinka
serverstate	JSON	Nadające: Raspberry Pi Nasłuchujące: UI	Stan serwera, konkretnie to, czy sterownik PLC jest aktualnie połączony poprzez Modbus TCP, oraz aktualny stan sterownika PLC.
robotstate	Binarny	Nadające: Kontroler Robota Nasłuchujące: UI	Aktualny stan wszystkich wejść i wyjść cyfrowych robota oraz ESP, dodatkowe informacje takie jak to, czy aktualnie robot jest w stanie bezczynności lub jaką komendę wykonuje.

assemblyRequest	JSON	Nadające: UI Nasłuchujące: Raspberry Pi	Nowe zlecenie montażu elementu, o zadanych kolorach części dolnej oraz górnej.
customCommand	Binarny	Nadające: UI Nasłuchujące: Kontroler Robota	Wymuszenie, aby kontroler robota wysłał do robota komendę o danym kodzie.

Schemat połączeń pomiędzy elementami stanowiska

// TODO opis



Niebieski - elementy Pneumatyczne

Żółty - elementy elektryczne

Zielony - połączenia bezprzewodowe / ethernetowe

Czerwony - czujniki