AKADEMIA GÓRNICZO – HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INŻYNIERII MECHANICZNEJ I ROBOTYKI

PRACA DYPLOMOWA

Inżynierska

Techniki wymiany danych między układem sterowania robota przemysłowego i urządzeniami peryferyjnymi.

Techniques of data exchange between the industrial robot control system and peripheral devices.

Autor: **Norbert Krzysztof Acedański**

Kierunek studiów: Automatyka i Robotyka

Opiekun pracy: **dr hab. inż.,**

Wojciech Lisowski **……**

podpis

Kraków, rok 2020

Spis treści

1. Wstęp……...……3

2. Cel i założenia pracy……5

3. Metody komunikacji pomiędzy środowiskami programowania…….7

4. Metody komunikacji z robotami firmy ABB na podstawie aplikacji sterującej robotem 6-osiowym……13

4.1 Opis aplikacji napisanej w programie RobotStudio…….18

4.2 Opis aplikacji napisanej w języku Python……...26

5. Porównanie wydajności i płynności metod komunikacji……..….30

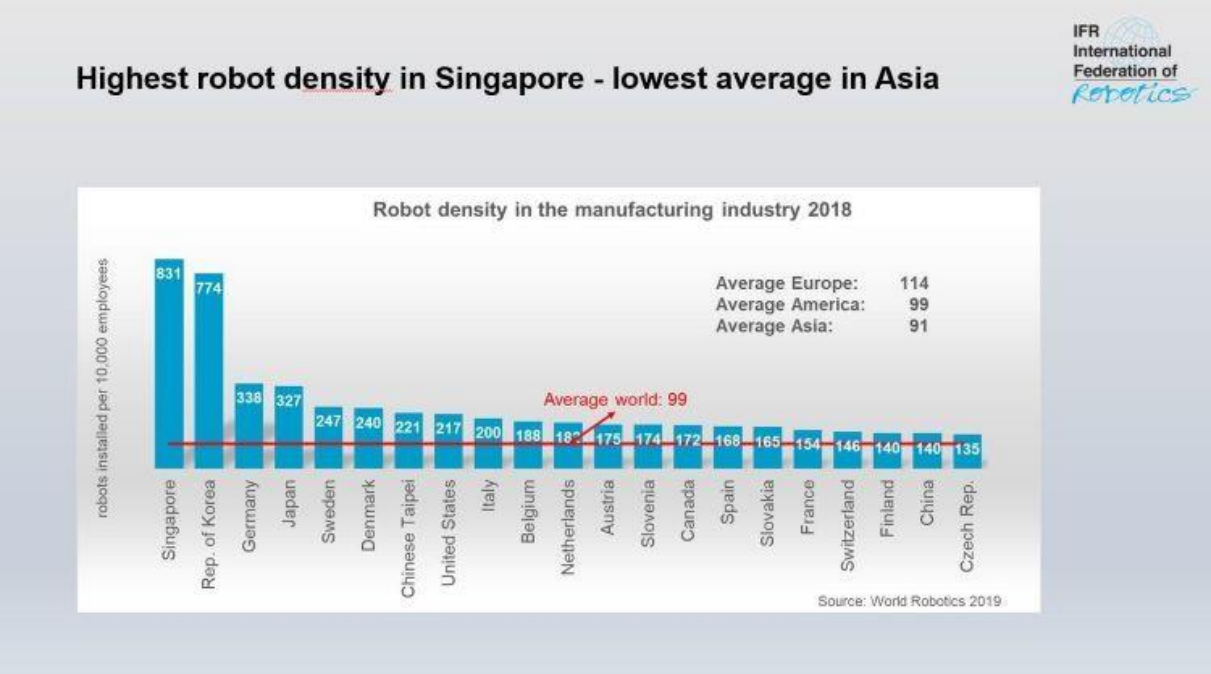
6. Zastosowanie i dalszy rozwój……...36

7. Wnioski oraz podsumowanie……38

Literatura, źródła……...……...39

1. Wstęp

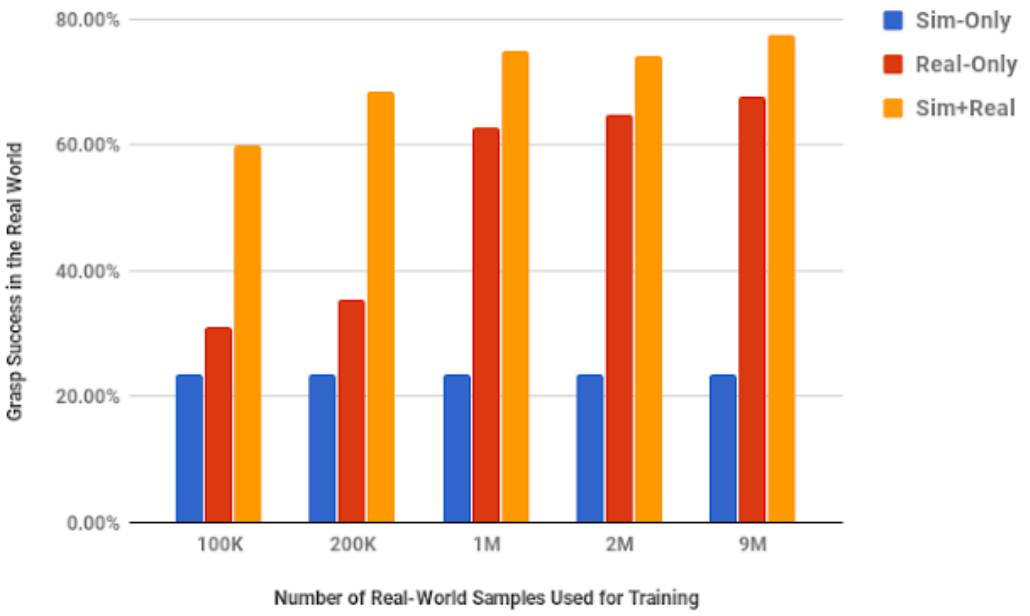
Proces postępującej automatyzacji i robotyzacji w przemyśle zachodzi na naszych oczach i z roku na rok staje się coraz szybszy a zdaniem specjalistów trend ten będzie się utrzymywał przez najbliższe lata. Ilość kupowanych robotów przemysłowych w każdym roku rośnie co zwiększa produkcję, poprawia jakość i usuwa błędy ludzkie z procesu tworzenia dóbr. Biorąc pod uwagę gęstość robotyzacji w danym kraju na czele znajdują się Singapur, Korea Południowa oraz Niemcy, co sprawia, że wzrasta zapotrzebowanie na specjalistów zajmujących się programowaniem robotów przemysłowych, czy to w symulacjach typu offline, czy online[1].



Rysunek 1 Kraje o największej gęstości robotyzacji

https://ifr.org/ifr-press-releases/news/robot-investment-reaches-record-16.5-billion-usd

Firma Google[2] porównała sposoby przygotowywania stanowisk zrobotyzowanych. W badaniu pod uwagę zostały wzięte stanowiska zbudowane wyłącznie jako symulacje, stanowiska uruchomione bez wcześniejszej symulacji, oraz stanowiska które przed uruchomieniem zostały przetestowane w środowisku symulacyjnym.



The performance effect of using 8 million simulated samples of procedural objects with no randomization and various amounts of real data.

Rysunek 2 Porównanie typów programowania robotów

https://ai.googleblog.com/2017/10/closing-simulation-to-reality-gap-for.html?m=1&fbclid=IwAR1UIQ78hsvW7KjPbZ7ORnFiGAxBQYNO1KCm4euODW7zdDMaM9ailYMxZm0

Badania pokazały jak ważne jest testowanie stanowiska wirtualnego, przed jego rzeczywistym stworzeniem.

2. Cel i założenia projektu.

W niniejszej pracy podjęto się stworzenia aplikacji służącej do komunikacji z robotem i sterowaniu nim przy pomocy urządzeń peryferyjnych w celu łatwiejszego przyswajania wiedzy i umiejętności w programowaniu robotów przemysłowych. Zadaniem aplikacji będzie odczyt danych z wybranych urządzeń peryferyjnych, przetworzenie ich w sposób zrozumiały dla robota i za pomocą wybranych rodzajów komunikacji wysłanie ich do robota.

W pierwszym rozdziale skupiono się na celu i założeniach pracy, następnie przedstawiono metody komunikacji środowisk programowania, po czym następuje opis metod komunikacji oraz analiza programu sterowania z robotami firmy ABB. W kolejnym rozdziale przedstawiono wyniki testów wraz z prezentacją możliwości dalszego rozwoju.

Założeniami pracy jest realizacja dwóch zadań. Pierwszym jest zaprogramowanie i implementacja aplikacji mającej na celu dekodowanie informacji przychodzących z wybranych urządzeń peryferyjnych (myszka komputerowa, pad), następnie odpowiednia interpretacja tych danych, aby móc wysłać przetworzone informacje do robota 6-osiowego. Zadanie to będzie realizowane za pomocą trzech rodzajów komunikacji: TCP/IP, UDP oraz EGM (jest to technika komunikacji mogąca służyć do sterowania robotami firmy ABB). Aplikacja przetwarzająca dane wysyłane z urządzeń peryferyjnych została stworzona w Pythonie, natomiast odbiorcą przetworzonych danych jest program napisany w języku RAPID w programie RobotStudio, które jest programem dedykowanym dla urządzeń firmy ABB.

Robotem sterowanym w pracy będzie IRB 140. Jest to ważna informacja, ponieważ każdy robot posiada inne ograniczenia poruszania się i inne zakresy ruchów. Program napisany w języku Python będzie miał możliwość przełączenia się pomiędzy metodami komunikacji z robotem w symulacji RobotStudio za pomocą przycisków na klawiaturze, które będą odpowiednio zaprogramowane. Sama aplikacja będzie wykrywać dostępne urządzenia peryferyjne oraz, przełączając się pomiędzy nimi, będzie interpretować nadchodzące z nich dane (przesunięcie myszą, poruszenie gałką w padzie). Aplikacja będzie wykrywać nadchodzące sygnały błędów zbyt dużych wychyleń robota jak również zbyt dużych wartości obrotu poszczególnych złączy. W aplikacji będzie również możliwość przełączania się pomiędzy trybami pracy robota. Jednym z trybów będzie sterowania każdym złączem osobno, w drugim zaś będzie możliwość sterowania robotem liniowo na płaszczyźnie.

Drugim zadaniem będzie porównanie łatwości implementacji danej metody komunikacji, ich wydajności, obsługi błędów oraz płynności ruchów robota. Ogólnym celem pracy jest natomiast przybliżenie problemu sterowania robotem przy pomocy urządzeń dobrze rozpoznawalnych. Urządzeniami tymi będą myszka oraz pad.

3. Metody komunikacji pomiędzy środowiskami programowania

Metody komunikacji można inaczej nazwać protokołami komunikacji. Zarówno one, jak i aplikacje są niejako „napędami" świata biznesu i efektywnego Industry 4.0, ponieważ umożliwiają efektywną oraz szybką wymianę danych pomiędzy urządzeniami, użytkownikami, czy kombinacją tych dwóch. Protokoły są, w skrócie, zbiorem software'owych i hardware'owych reguł, których muszą przestrzegać końcowe punkty komunikacji (ang. „communication end-points"), aby móc przesłać, lub odebrać jakiś rodzaj informacji. Istnieje tysiące rodzajów protokołów komunikacji używanych zarówno w analogowych jak i cyfrowych urządzeniach, a sieci komputerowe nie mogłyby istnieć bez protokołów komunikacji. Jednymi z najczęściej używanych i najlepiej rozpoznawalnych są: FTP (File Transfer Protocol), TCP/IP (Transmission Control Protocol/Internet Protocol), UDP (User Datagram Protocol), Bluetooth, HTTP (Hypertext Transfer Protocol)[3]. Dodatkowo w tym rozdziale będzie rozważana metoda komunikacji dedykowana dla robotów firmy ABB – EGM (Externally Guided Motion).

Pierwszym typem komunikacji opisywanym w tym rozdziale jest FTP. Jest to standardowy protokół używany w celu przesyłania plików pomiędzy klientem a serwerem w sieci komputerowej. FTP jest zbudowany na standardowym modelu klient-serwer przy użyciu oddzielnych połączeń sterowania i danych pomiędzy klientem a serwerem. Do komunikacji używane są 2 połączenia TCP, z których jedno służy do transmisji danych a drugie do przesyłania poleceń. FTP może działać w 2 trybach: aktywnym i pasywnym. Jeśli FTP działa w trybie aktywnym, używa portu 21 w celu przesyłania poleceń (jest to zestawiane przez klienta), natomiast portu 20 do przesyłania danych (co z kolei jest zestawiane przez serwer). W trybie pasywnym zaś, tak samo jak w przypadku aktywnego trybu, FTP używa portu 21 do przesyłania poleceń, natomiast portu o numerze powyżej 1024 w celu transmisji danych oraz klient zestawia oba połączenia[4].

Kolejnym trybem komunikacji jest połączenie TCP/IP. Teoretycznie protokół transmisji może być bardzo prosty, jednak TCP nie może zostać tak nazwanym. Skoro nie jest to prosta metoda komunikacji, dlaczego jej się używa? Najważniejszym powodem jest zawodność IP, chociaż tak na prawdę wszystkie poziomy (layers) poniżej TCP są zawodne. TCP zapewnia

dodatkowe usługi do poziomu IP oraz do wyższych poziomów, jednak co najważniejsze zapewnia całkowicie kompletny przesył danych oraz informacji. Ta metoda komunikacji jest tym samym protokołem sprawdzania poprawności wiadomości. Jeśli informacja, lub jej część zostanie uszkodzona, wybrakowana, bądź zaginie w trakcie przesyłania, to TCP zajmuje się retransmisją, bez ingerencji instrukcji wyższego poziomu. Dodatkowo, gdy wybieramy połączenie TCP z aplikacjami, najpierw konieczne jest zawiązanie połączenia pomiędzy serwerem a klientem. W tej pracy inżynierskiej są to szczególnie ważne informacje, ponieważ jest to jedna z wybranych metod komunikacji z robotem [5].

Struktura nagłówka TCP:

TCP

Offset	Oktet	0								1								2								3							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Port nadawcy																Port odbiorcy															
4	32	Numer sekwencyjny																															
8	64	Numer potwierdzenia (jeżeli flaga ACK jest ustawiona)																															
12	96	Długość nagłówka				Zarezerwowane			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Szerokość okna															
16	128	Suma kontrolna																Wskaźnik priorytetu (jeżeli flaga URG jest ustawiona)															
20 ...	160 ...	Opcje (jeżeli długość nagłówka > 5, to pole jest uzupełniane "0") ...																															

Rysunek 3 Struktura nagłówka TCP

https://pl.wikipedia.org/wiki/Protok%C3%B3%C5%82_sterowania_transmisj%C4%85

Następnym omawianym protokołem komunikacyjnym, który również będzie wykorzystany w pracy inżynierskiej jest protokół UDP (User Datagram Protocol). W przeciwieństwie do TCP, UDP nie wymaga nawiązania połączenia między klientem a serwerem. Ten protokół komunikacji po prostu wysyła dane, dlatego jest nazywany protokołem bezpołączeniowym „connectionless protocol”. Ta metoda komunikacji jest znacznie szybsza od TCP, jednak może tracić dane podczas wysyłania. Dane wysłane za pomocą tego protokołu mogą dochodzić w innych kolejnościach, niż zostały wysłane, pakiety danych mogą ginąć po drodze z aplikacji do aplikacji. Samo UDP nie zawiera także zabezpieczeń przed tego typu zdarzeniami, dlatego nie używa się go do przesyłania ważnych informacji, danych z baz danych itp. Idealnie nadaje się za to do przesyłania obrazów, filmów, streamingu audio, ponieważ jest szybkie [6].

+	Bity 0 – 15	16 – 31
0	Port nadawcy	Port odbiorcy
32	Długość	Suma kontrolna
64	Dane	

Rysunek 4 Struktura nagłówka UDP

https://pl.wikipedia.org/wiki/User_Datagram_Protocol

Jak widać po nagłówkach protokołów UDP i TCP znacząco różnią się one długością i skomplikowaniem. Udowadnia to wady i zalety każdej z metod komunikacji.

UDP v/s TCP		
Characteristics/ Description	**UDP**	**TCP**
General Description	Simple High speed low functionality "wrapper" that interface applications to the network layer and does little else	Full-featured protocol that allows applications to send data reliably without worrying about network layer issues.
Protocol connection Setup	Connection less data is sent without setup	Connection-oriented; Connection must be Established prior to transmission.
Data interface to application	Message base-based is sent in discrete packages by the application.	Stream-based; data is sent by the application with no particular structure
Reliability and Acknowledgements	Unreliable best-effort delivery without acknowledgements	Reliable delivery of message all data is acknowledged.
Retransmissions	Not performed. Application must detect lost data and retransmit if needed.	Delivery of all data is managed, and lost data is retransmitted automatically.
Features Provided to Manage flow of Data	None	Flow control using sliding windows; window size adjustment heuristics; congestion avoidance algorithms
Overhead	Very Low	Low, but higher than UDP
Transmission speed	Very High	High but not as high as UDP
Data Quantity Suitability	Small to moderate amounts of data.	Small to very large amounts of data.

Rysunek 5 Porównanie UDP – TCP

https://en.wikibooks.org/wiki/Communication_Networks/TCP_and_UDP_Protocols

Na rysunku 4 zostało przedstawione porównanie protokołów UDP oraz TCP pod względem niezawodności, zabezpieczeń, prędkości i innych ważnych dla przesyłu danych informacji. Najważniejszymi są dane dotyczące niezawodności – UDP jest zawodnym protokołem w przeciwieństwie do TCP, UDP jest szybsze niż TCP, UDP daje możliwość wysyłania małych oraz średnich paczek danych, natomiast TCP nawet dużych ilości informacji oraz TCP „dba" o to, aby każdy bit informacji został dostarczony w odpowiednie miejsce, w przeciwieństwie do UDP [7].

Zarówno metoda TCP/IP jak i UDP jest prosta w implementacji w języku Python jak również w języku RAPID. Nie wymaga dodatkowych ustawień w samym środowisku

RobotStudio, gdzie będzie implementowany moduł komunikacji, ani także w Pythonie, wymaga jedynie dodatkowej biblioteki (socket).

Bluetooth jest krótko zasięgowym protokołem komunikacyjnym pozwalającym na bezprzewodową wymianę danych pomiędzy telefonami, komputerami i innymi urządzeniami peryferyjnymi. Celem tego protokołu jest usunięcie okablowania z drogi między urządzeniami, ale pozostawiając bezpieczną komunikację między nimi. Zasięg Bluetooth wynosi około 10m, a w tym promieniu może połączyć się od dwóch do ośmiu urządzeń. Ta metoda komunikacji zużywa zarówno mniej energii niż Wi-Fi, jak również jest mniej kosztowna w implementacji. Dzięki zużywaniu mniejszej ilości energii Bluetooth cierpi również mniej na interferencję z innymi urządzeniami używającymi tego samego zakresu fal w komunikacji (2.4GHz) [8].

Ostatnim z omawianych w tym rozdziale metod komunikacji, będzie metoda dedykowana dla oprogramowania RobotStudio w języku RAPID – EGM (Externally Guided Motion), która bez pośrednio łączy się z robotem (nie nawiązując połączenia standardowo poprzez język RAPID) i wysyła instrukcje ruchu. EGM ma 2 rodzaje – Position Guidance oraz Path Correction. W tej pracy będzie wykorzystywany pierwszy rodzaj EGM w celu bezpośredniego sterowania robotem, ponieważ w tym trybie robot nie wykonuje ścieżki zadanej w programie RobotStudio, tylko ścieżkę generowaną z urządzenia peryferyjnego. W trybie Path Correction natomiast, robot podąża wytyczoną w programie ścieżką z uwzględnieniem poprawek przychodzących z urządzenia zewnętrznego[9][10].

Implementacja tego modułu komunikacji jest o tyle problematyczna, że wymaga zbudowania odpowiedniego pliku oraz odpowiednich bibliotek w języku, którego chcemy użyć, jednak Application Manual[11] do RobotStudio zapewnia nam odpowiednie instrukcje aby to zrobić.

How to build an EGM sensor communication endpoint using .Net

This guide assumes that you build and compile using Visual Studio and are familiar with its operation.

Here is a short description on how to install and create a simple test application using *protobuf-csharp-port*.

	Action
1	Download protobuf-csharp binaries from: https://code.google.com/p/protobuf-csharp-port/.
2	Unpack the zip-file.
3	Copy the *egm.proto* file to a sub catalogue where protobuf-csharp was un-zipped, e.g. *~\protobuf-csharp\tools\egm*.
4	Start a Windows console in the tools directory, e.g. *~\protobuf-csharp\tools*.
5	Generate an EGM C# file (*egm.cs*) from the *egm.proto* file by typing in the Windows console: `protogen .\egm\egm.proto --proto_path=.\egm`
6	Create a C# console application in Visual Studio. Create a C# Windows console application in Visual Studio, e.g. *EgmSensorApp*.
7	Install NuGet, in Visual Studio, click **Tools** and then **Extension Manager**. Go to **Online**, find the *NuGet Package Manager extension* and click **Download**.
8	Install protobuf-csharp in the solution for the C# Windows Console application using NuGet. The solution has to be open in Visual Studio.
9	In Visual Studio select, **Tools**, **Nuget Package Manager**, and **Package Manager Console**. Type *PM>Install-Package Google.ProtocolBuffers*
10	Add the generated file *egm.cs* to the Visual Studio project (add existing item).
11	Copy the example code into the Visual Studio Windows Console application file (*EgmSensorApp.cpp*) and then compile, link and run.

Rysunek 4 Budowanie plików .Net

ABB Robotics: Application Manual, Västerås, Sweden (2007) – str.341

How to build an EGM sensor communication endpoint using C++

When building using C++ there are no other third party libraries needed.

C++ is supported by Google. It can be a bit tricky to build the Google tools in Windows but here is a guide on how to build protobuf for Windows.

Use the following procedure when you have built *libprotobuf.lib* and *protoc.exe*:

	Action
1	Run Google protoc to generate access classes, `protoc --cpp_out=. egm.proto`
2	Create a win32 console application
3	Add Protobuf source as include directory.
4	Add the generated *egm.pb.cc* file to the project, exclude the file from precompile headers.
5	Copy the code from the *egm-sensor.cpp* file, see *UdpUc code examples on page 355*.
6	Compile and run.

Rysunek 5 Budowanie plików w C++

ABB Robotics: Application Manual, Västerås, Sweden (2007) – str.341-342

Po poprawnym zbudowaniu odpowiednich plików potrzebne są jedynie 2 pliki w celu implementacji odpowiednich funkcji wysyłania danych. Pliki o nazwach egm_pb2 oraz rpi_abb_irc5. W nich znajdują się definicje odpowiednich instrukcji oraz klas wymaganych do odpowiedniej komunikacji poprzez EGM. Odpowiednie instrukcje w programie RobotStudio zostały bliżej przedstawione w książce Technical Reference Manual[12] oraz zostaną opisane w następnym rozdziale przy okazji analizy programu w języku RAPID.

4. Metody komunikacji z robotami firmy ABB na podstawie aplikacji sterującej robotem 6-osiowym

W tym rozdziale zostały utworzone aplikacje mające na celu umożliwić komunikację urządzenie peryferyjne – robot wraz z interpretacją danych pochodzących z tych urządzeń.

Programowanie offline pozwala na przygotowanie oprogramowania dla robotów bez przerywania procesu technologicznego, zatrzymywania produkcji, czy wchodzenia w strefę roboczą manipulatora lub pedipulatora a także na bezpieczną manipulację kodu bez możliwości zepsucia, bądź uszkodzenia robota lub stanowiska i sprzętu znajdującego się w strefie roboczej, co jest najlepszą drogą do maksymalizacji zwrotu kosztów w systemy zrobotyzowane.

Taki rodzaj programowania pozwala na pracę na komputerze na przykład z biura lub nawet domu. Nie wymaga tez znajomości ograniczeń fizycznych robota, ani narzędzi. Wszystko ukazuje symulacja w odpowiednim środowisku.

Jednym z takich środowisk jest RobotStudio.

Program RobotStudio jest produktem firmy ABB powstałym w 1998 roku. Było to pierwsze narzędzie symulacyjne oparte na wirtualnym kontrolerze, w którym oprogramowanie jest dokładnie odwzorowane ze swojego rzeczywistego wzorca, co zrewolucjonizowało programowanie off-line.

Od tego czasu program przechodził wiele zmian i upgrade-ów przez wprowadzenie paczek do spawania, możliwości dodawania własnych modeli i endefektorów, Smart Components czy choćby dodanie fizyki.

Oprogramowanie RobotStudio pozwala na import wybranego modelu robota 6-osiowego do środowiska symulacji, symulację z użyciem wirtualnego kontrolera, tworzenie własnych Inteligentnych Komponentów, a także na symulację kodu, który w przyszłości może być wgrany do robota, a język programowania RAPID pozwala na edycję kodu, który jest generowany, gdy tworzymy instrukcje dla robota. Dodatkowo RobotStudio zapewnia bardzo przydatną funkcję „Pack&Go”, która powoduje zapisanie obecnej symulacji do pliku rspag, który można w dowolnym miejscu rozpakować aby otworzyć całą stację. Jest to o tyle przydatne, ponieważ gdy przypadkowo zmieni się nazwę folderu, w którym jest symulacja,

projekt czasami nie jest w stanie załadować kontroler do symulacji. Funkcja „Unpack&Work" powoduje rozpakowanie projektu z pliku rspag.

RobotStudio pozwala na zaprogramowanie komunikacji za pomocą trzech protokołów komunikacyjnych: TCP/IP, UDP oraz EGM. Opis tych metod komunikacji został bliżej zarysowany oraz opisany w poprzednim rozdziale. Programowanie w języku RAPID w celu umożliwienia komunikacji po TCP/IP oraz UDP nie różni się niczym oprócz nazw instrukcji od innych języków programowania, co sprawia, że napisanie komunikacji jest rzeczą bardzo prostą. Jedynie komunikacja za pomocą modułu EGM jest problematyczna i wymaga dodatkowych plików i ustawień zarówno w języku Python jak i RAPID, czego problem zostanie bliżej przybliżony poniżej.

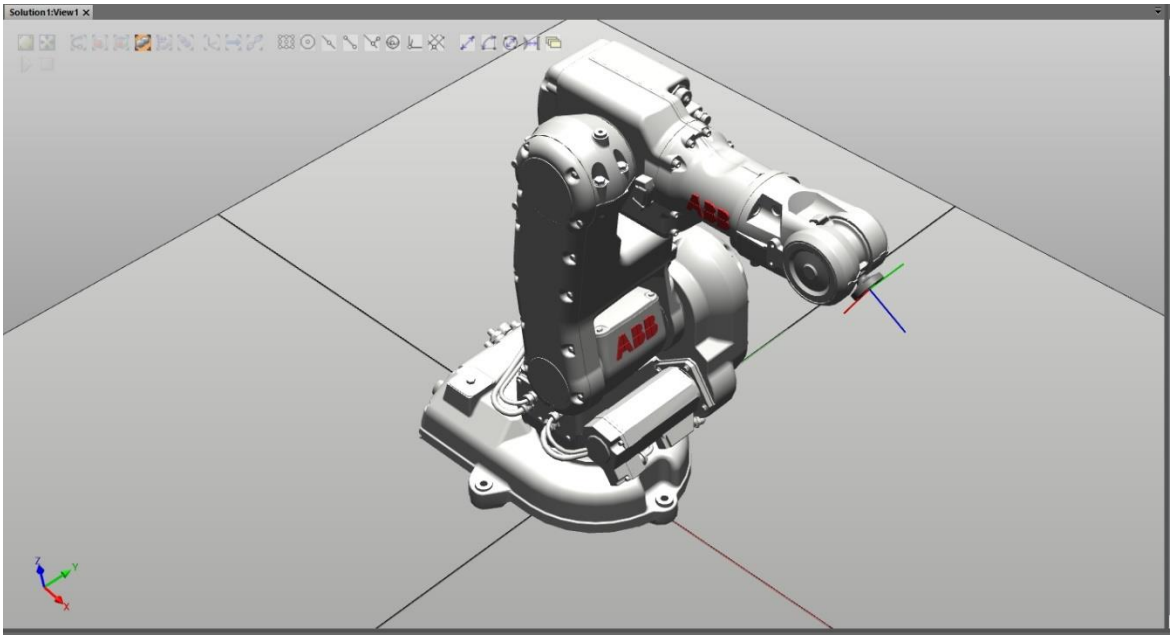
Programy pisane w tej pracy w Pythonie będą zawierały następujące elementy:

1. Komunikacja nawiąże połączenie odpowiednio TCP, UDP, lub EGM z programem napisanym w programie RobotStudio.
2. Program następnie będzie nasłuchiwał zdarzenia przychodzącego z myszki, pada, lub klawiatury, jeśli takowe nastąpi program przestanie słuchać z drugiego urządzenia peryferyjnego (nie będzie konfliktu w wysyłanych danych), będzie nasłuchiwać tylko zdarzenia z klawiatury.
3. Gdy nastąpi zdarzenie, program będzie interpretował przychodzące dane i wysyłał je za pomocą odpowiedniej komunikacji. Zarówno w przypadku pada jak i myszki będzie możliwość przełączania się pomiędzy trybami obsługi (poruszanie każdego złącza niezależnie, liniowe poruszanie TCP robota) poprzez wciśnięcie odpowiedniego przycisku na danym urządzeniu peryferyjnym.
4. Gdy nastąpi przerwanie za pomocą przycisku na klawiaturze program przejdzie do nasłuchiwania zdarzenia z urządzeń peryferyjnych, a robot powróci do pozycji domowej.

Programy pisane w języku RAPID natomiast będą zawierały następujące założenia:

1. Nawiązanie odpowiedniej komunikacji (TCP, UDP) z aplikacją kontrolującą w osobnym tasku. Komunikacja za pomocą EGM nie wymaga osobnego tasku do komunikacji.
2. Powrót robota do pozycji domowej.

3. W przypadku nadejścia danych, będą one zapisywane w odpowiedni sposób w instrukcji ruchu, następnie po ustawieniu odpowiedniej flagi robot dostanie informację, że może się poruszyć. W przypadku, gdy ruch będzie wykraczał poza zakres robota, bądź poza zakres kątowy poszczególnych złączy, program wyśle informacje do programu w Pythonie i nie pozwoli na dalszy ruch.
4. W przypadku zmiany trybu ruchu, task obsługujący komunikację odpowiednio zmodyfikuje instrukcje ruchu.
5. W przypadku przerwania od strony aplikacji kontrolującej robot powróci do pozycji domowej.

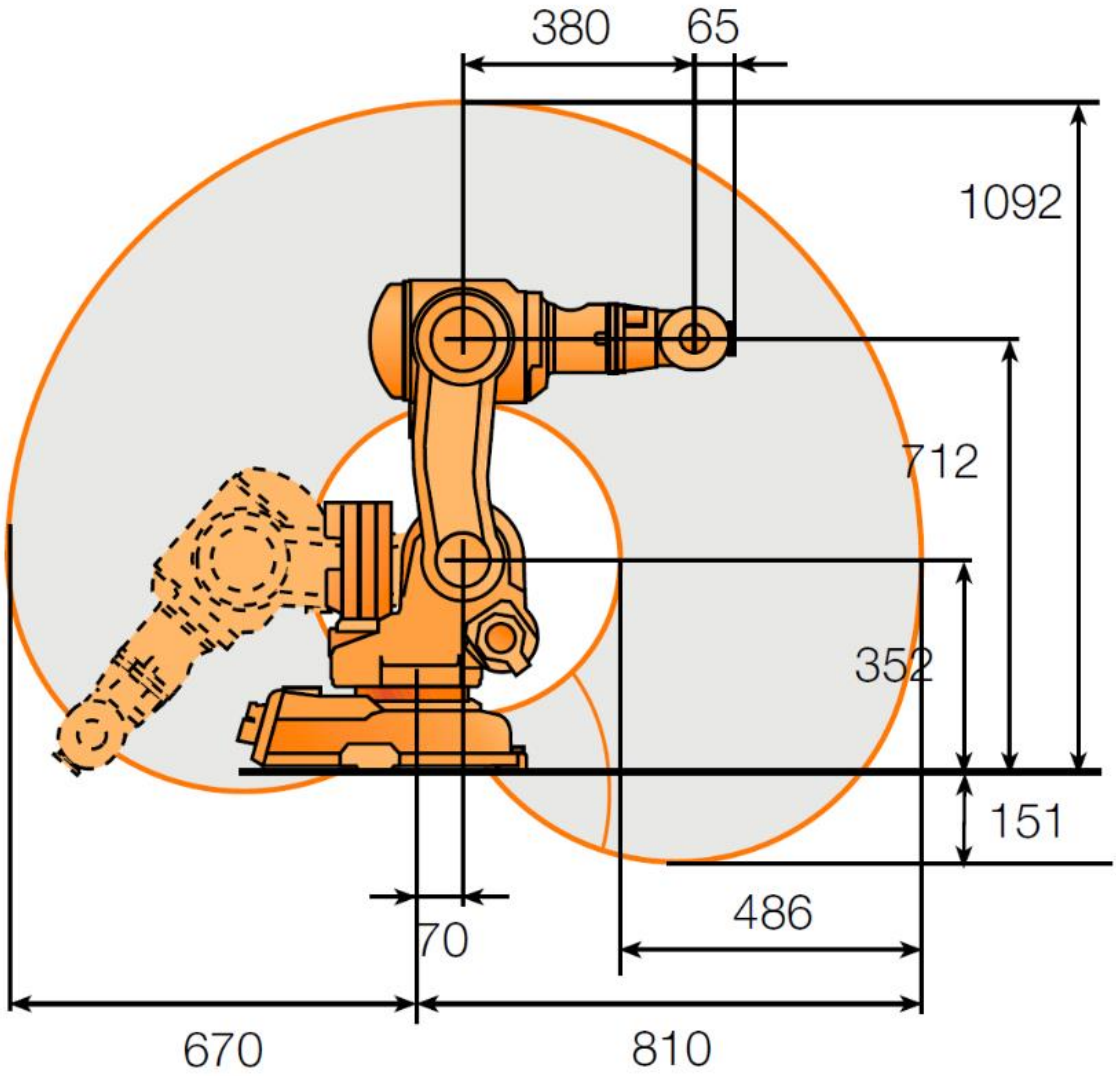


Rysunek 6 Robot IRB140 w symulacji RobotStudio

Robotem sterowanym w niniejszej pracy jest IRB140 wybrany z biblioteki robotów ABB.

IRB 140 jest średniej wielkości robotem 6-osiowym produkowanym przez firmę ABB od 1999 roku. Jest robotem mało zawodnym ze względu na niską wartość współczynnika MTBF (średni czas międzyawaryjny – ang. Mean Time Between Failures) oraz posiada niewielkie zapotrzebowanie na konserwację i charakteryzuje się krótkim czasem naprawy. Osiąga powtarzalność pozycjonowania do 0.03mm i wysoką dokładność odtwarzania trajektorii. Obciążenie maksymalne robota wynosi około 6 kg a maksymalny zasięg – 810mm. Ponadto IRB 140 może być mocowane pod różnymi kątami do podłoża, na ścianie a także na suficie, co ułatwia projektowanie linii produkcyjnych a także rozmieszczenia innych urządzeń.

Dodatkowym atutem jest możliwość obrotu osi 1 o pełne 360 stopni. Głównymi zastosowaniami opisywanego robota są spawanie, montaż, czyszczenie, obsługa maszyn, przenoszenie, pakowanie i gratowanie. Robot waży 98kg[13].



Rysunek 9 Zasięg robota IRB 140

https://new.abb.com/products/robotics/pl/roboty-przemyslowe/irb-140/irb-140-dane-techniczne

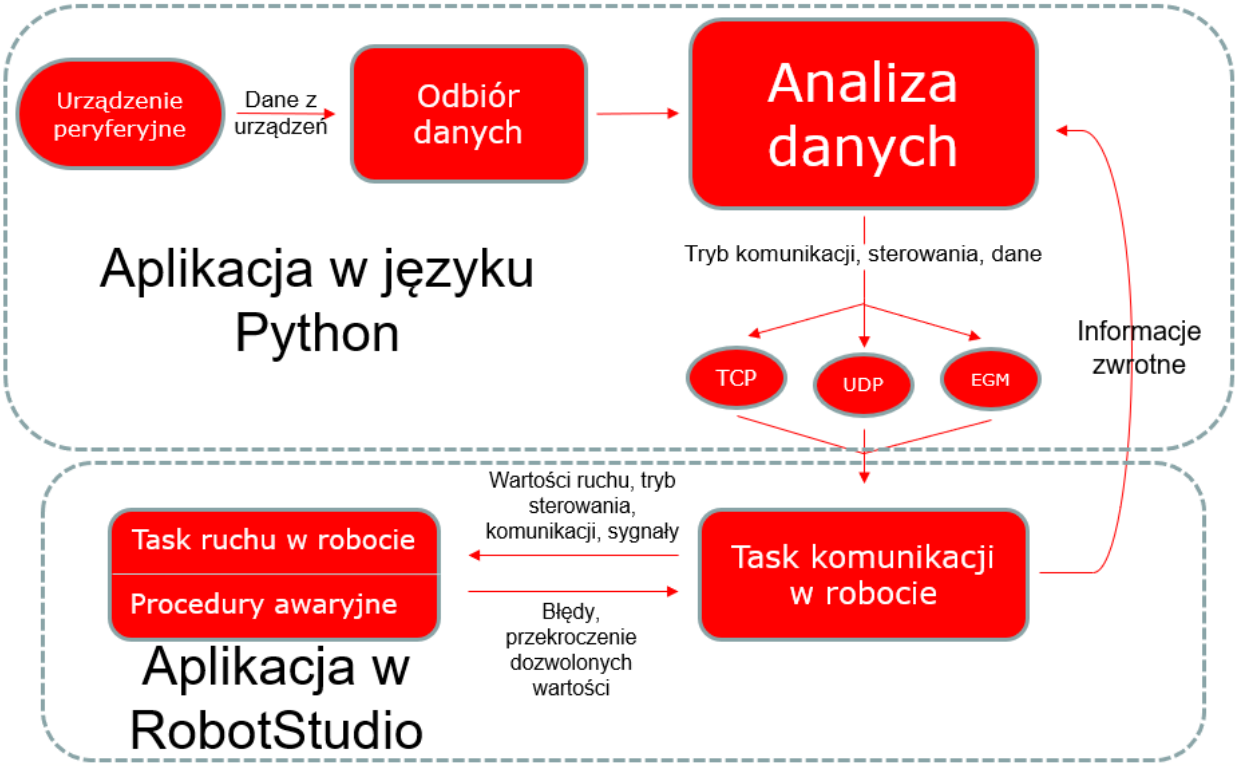
Type of motion	Range of movement
Axis 1: Rotation motion	+ 180° to - 180°
Axis 2: Arm motion	+ 110° to - 90°
Axis 3: Arm motion	+ 50° to - 230°
Axis 4: Wrist motion	+ 200° to - 200° Default + 165 revolutions to - 165 revolutions Max. i
Axis 5: Bend motion	+ 115° to - 115°
Axis 6: Turn motion	+ 400° to - 400° Default + 163 revolutions to -163 revolutions Max. i

Rysunek 10 Zasięg ruchu w stopniach dla poszczególnych osi robota IRB 140

Product specification IRB 140 [13]

Informacja o zakresie kątowym robota będzie szczególnie ważna w celu zaprogramowania ograniczeń ruchowych robota, aby nie wywoływać błędu związanego ze zbyt dużym obrotem danej osi.

Schemat działania aplikacji w języku Python oraz w języku RAPID jest następujący:



Rysunek 11 Działanie i połączenie aplikacji

Opracowanie własne

4.1 Opis aplikacji napisanej w programie RobotStudio.

Najważniejszym zadaniem aplikacji w RobotStudio jest wykonywanie zadań przychodzących z aplikacji napisanej w języku Python. W języku RAPID napisane są 2 taski (analogicznie do języka Python są to odpowiedniki wątków, każdy task działa niezależnie, jednak mogą być synchronizowane poprzez sygnały lub instrukcje synchronizujące). Pierwszy task obsługuje wszystkie ruchy robota oraz zarządza bezpośrednio trybami ruchu. Drugi task natomiast obsługuje komunikację pomiędzy RobotStudio a aplikacją w języku Python, dekoduje informacje oraz ustawia odpowiednie sygnały i zmienne do odczytania przez pierwszy task.

Na początku pierwszego tasku (odtąd nazywany taskiem ruchu) zadeklarowane są wszystkie niezbędne zmienne zarówno do komunikacji między taskami, jak również do modułów awaryjnych oraz procedur zabezpieczających i procedur ruchów.

```
PROC Settings()
    IF settingsSignal THEN
        CommunitationType := "";
        JointCoordinateSignal := "";
        MoveSignal := FALSE;
        jointAngles := [0,0,0,0,0,0];
        coordinates := [0,0,0];
        settingsSignal := FALSE;
        coordinatesOutOfRange := FALSE;
        jointsOutOfRange := FALSE;
        MoveAbsJ HOME,v1000,fine,tool0\WObj:=wobj0;
    ENDIF
ENDPROC
```

Rysunek 12 Procedura Settings w języku RAPID

W główniej części programu a więc w procedurze main() pierwszą procedurą jest procedura resetująca wszystkie zmienne, które mają zasięg międzytaskowy (niektóre zmienne w programie RobotStudio po zadeklarowaniu w wielu taskach mogą być w tych taskach zmieniane i modyfikowane, dodatkowo ich stan jest zapamiętany nawet po wyłączeniu symulacji, stąd konieczność resetowania wszystkich takich zmiennych na początku symulacji). Dodatkowo dla bezpieczeństwa resetowane są również pozostałe zmienne. Task ruchu jak i task komunikacji działają w trybie ciągłym, to znaczy, że zachowują się, jakby były napisane w nieskończonej pętli while.

```
PROC main()
    Settings;
    WaitUntil CommunitationType <> "";
    WHILE CommunitationType = "TCP/UDP" DO
        tcpUdpJoinMove;
        tcpUdpCoordinateMove;
    ENDWHILE
    WHILE  CommunitationType = "EGM" DO
        EGMReset egmID1;
        checkingEGMState;
        egmJoinMove;
        EGMReset egmID1;
        egmCoordinateMove;
    ENDWHILE
ENDPROC
```

Rysunek 13 Procedura main w języku RAPID

Po zresetowaniu wszystkich zmiennych task ruchu czeka na informacje z tasku komunikacji. Tutaj rozpoczyna się wybór odpowiedniego rodzaju komunikacji oraz trybu sterowania. W przypadku wybrania opcji komunikacji za pomocą TCP lub UDP znacznik programu przechodzi do wyboru metody sterowania. Wybór trybu „JOINT" aktywuje odczytanie wartości z tablicy jointAngles zawierającej przychodzące absolutne wartości kątów złączowych robota. Następnie procedura specjalna sprawdza czy dane wartości zawierają się w odpowiednich granicach przytoczonych na stronie 19.

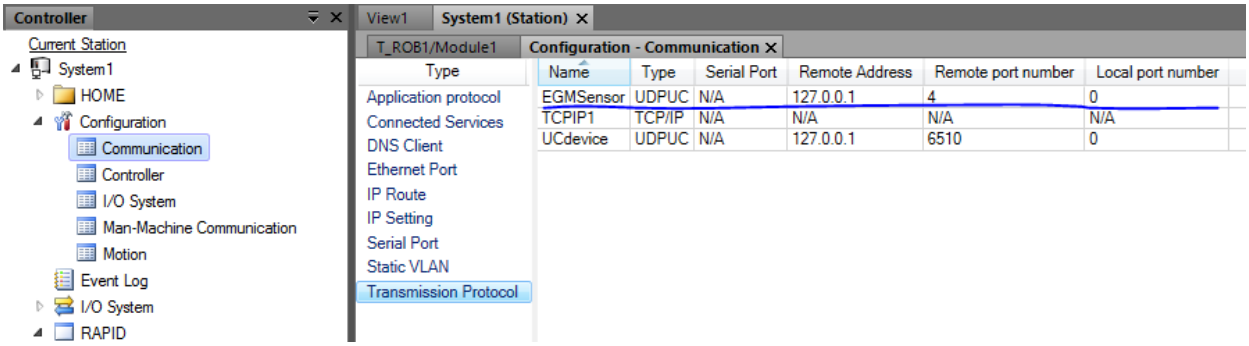
```
FUNC num clamp(num value, num minValue, num maxValue)
    IF value > maxValue THEN
        RETURN maxValue;
    ELSEIF value < minValue THEN
        RETURN minValue;
    ELSE
        RETURN value;
    ENDIF
ENDFUNC

PROC restrictionsToJointMove()
    jointAngles{1} := clamp(jointAngles{1}, -179, 179);
    jointAngles{2} := clamp(jointAngles{2}, -89, 109);
    jointAngles{3} := clamp(jointAngles{3}, -229, 49);
    jointAngles{4} := clamp(jointAngles{4}, -199, 199);
    jointAngles{5} := clamp(jointAngles{5}, -114, 114);
    jointAngles{6} := clamp(jointAngles{6}, -399, 399);
ENDPROC
```

Rysunek 14 Funkcja clamp oraz procedura ograniczająca

Jeśli wartości kątów złączowych zawierają się w odpowiednich granicach program przechodzi dalej i aktywowana jest instrukcja przypisania wartości z tablicy do targetu. W przeciwnym wypadku procedura ogranicza wartości kątów i zwracana jest informacja o nieprawidłowym położeniu robota do tasku komunikacji, co z kolei aktywuje funkcję

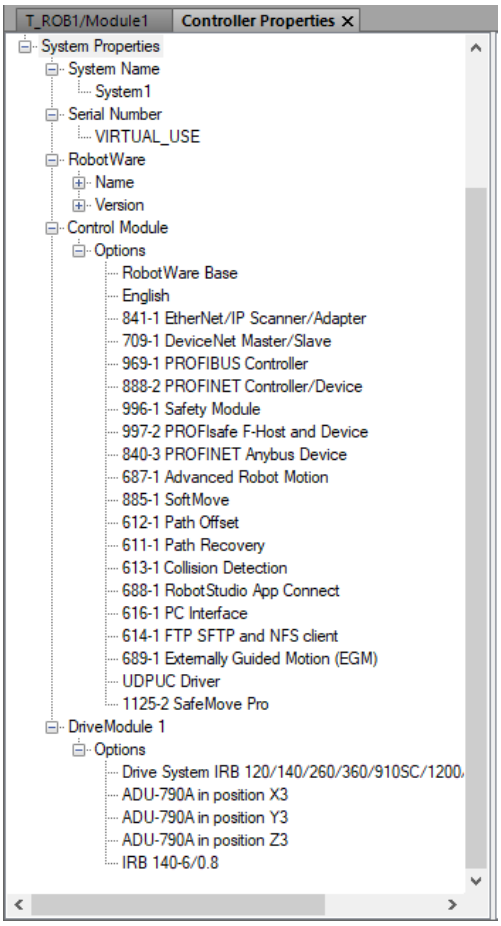
wysyłającą informację o błędzie do aplikacji w języku Python. Po udanym przypisaniu instrukcja ruchu powoduje osiągnięcie przez robota odpowiedniej pozycji. Program tkwi w tej procedurze (ruch złączowy) dopóki task komunikacji nie dostanie informacji o zmianie trybu pracy, bądź zmianie trybu komunikacji. Jeśli zostanie wybrany drugi tryb sterowania instrukcje wyglądają podobnie. Robot ustawiany jest w pozycji do poruszania się na płaszczyźnie, następnie sprawdzane są wartości przysyłanych współrzędnych xyz. W przypadku poprawnych wartości (znajdujących się w odpowiednim zakresie – zakres został wyznaczony obliczeniowo oraz doprecyzowany doświadczalnie biorąc pod uwagę wymiary robota oraz minimalne i maksymalne odchyły wartości kątów poszczególnych złączy jako $\sqrt{x^2 + y^2 + (z - 352)^2} < 800$ oraz $\sqrt{x^2 + y^2 + (z - 352)^2} > 320$, jeśli warunek nie jest spełniony robot otrzymuje instrukcję ruchu do pozycji domyślnej) robot otrzymuje polecenie ruchu do danej współrzędnej. Podobnie jak w przypadku metody ruchu złączowego, ruch xyz odbywa się dopóki task komunikacji nie otrzyma informacji o zmianie trybu, lub sposobu komunikacji.

Gdy wybrana jest metoda komunikacji EGM na początku resetowane są odpowiednie zmienne i sprawdzany jest stan komunikacji (EGM_STATE_CONNECTED, EGM_STATE_DISCONNECTED lub EGM_STATE_RUNNING). Aby było to możliwe najpierw należy zadeklarować w zmiennych komunikacyjnych odpowiedni protokół transmisji – EGMSensor – tak jak jest napisane w Technicznym Manualu dla RobotStudio dla komunikacji EGM.



Rysunek 15 Deklaracja protokołu transmisji

Nazwa protokołu musi być taka sama jak nazwa zmiennej w programie, aby komunikacja mogła działać. W przypadku Remote Address host jest lokalny, a Remote port numer może być dowolny.Dodatkowym warunkiem działania komunikacji EGM jest wybranie odpowiedniej opcji przy tworzeniu programu.



Rysunek 16 Moduły potrzebne do działania programu z EGM

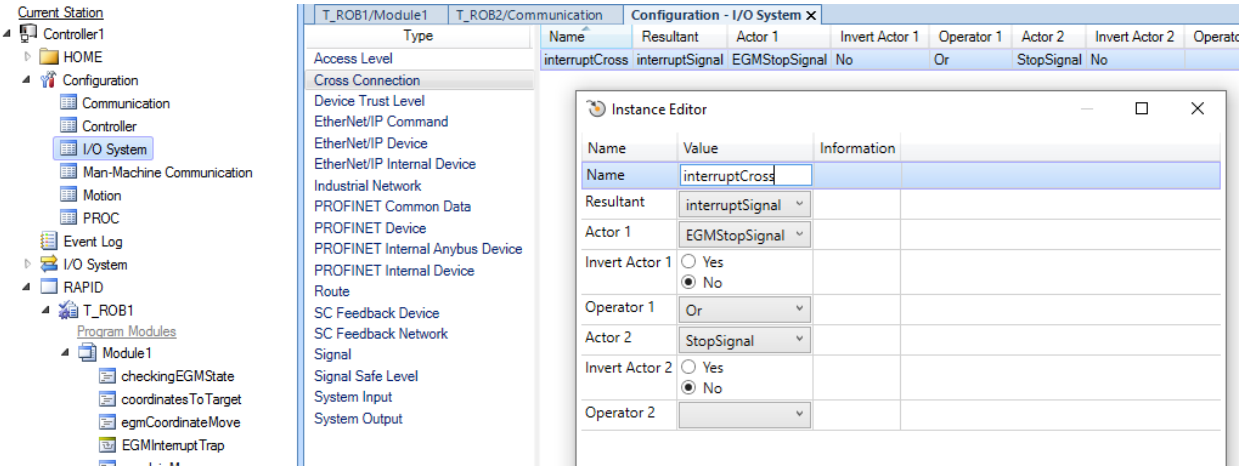
Jest to moduł o numerze 689-1 Externally Guided Motion (EGM).

Po sprawdzeniu i zresetowaniu odpowiednich zmiennych program decyduje, którą metodę sterowania wybrał użytkownik aplikacji i odpowiednio wybiera sprawdzając warunki pętli. W przypadku wybrania metody JOINT a więc współrzędnych złączowych aktywowane jest połączenie. Warto nadmienić, że EGM działa podobnie do protokołu komunikacyjnego UDP, gdzie klient jest napisany w programie RobotStudio, natomiast serwer jest napisany w wybranym języku (w przypadku tej pracy – Python). Po aktywowaniu połączenia rozpoczyna się nakładanie ograniczeń oraz wymagań odnośnie dokładności poruszania się robota. Po odpowiednich ustawieniach instrukcja EGMRunJoint nasłuchuje informacji przychodzących z aplikacji (w tym wypadku tablicy z kątami obrotu złączy robota). W przypadku nieotrzymania wiadomości z aplikacji dopiero po wyznaczonym czasie COMM_TIMEOUT następuje zerwanie połączenia i przejście do następnej instrukcji, dlatego konieczne jest zapisane również procedury Trap, która przerwie połączenie po żądaniu zmiany trybu sterowania, bądź zmiany komunikacji.

```
TRAP EGMInterruptTrap
    IF DInput(EGMStopSignal) = 1 THEN
        RETURN;
    ELSEIF DInput(StopSignal) = 1 THEN
        Stop;
    ENDIF
ENDTRAP
```

Rysunek 17 Przerwania

Trap działa jak przerwanie, jeśli nadejdzie odpowiedni sygnał wejściowy – w tym przypadku sygnał EGMStopSignal lub StopSignal – załącza się ta procedura i wykonywana jest każda instrukcja a następnie program wraca do przerwanej instrukcji (dotyczy to również instrukcji ruchu (jeśli trap aktywowany jest w trakcie ruchu robota, robot nie zatrzyma się, chyba że w samym trapie tego zażądamy. Ruch można wstrzymać, lub zatrzymać za pomocą odpowiednich instrukcji). Trap jest aktywowany jednym sygnałem, więc naturalną konsekwencją jest umożliwienie, aby jeden z tych sygnałów mógł aktywować tę procedurę. Jest to osiągnięte za pomocą Instrukcji Cross Connection.



Rysunek 18 Połączenie krzyżowe

Aby to osiągnąć sygnał interruptSignal dziedziczy wartości po dwóch sygnałach wymienionych powyżej. W przypadku aktywowania jednego z nich, aktywuje się sygnał przerywający i aktywowany zostaje trap.

Drugim omawianym taskiem jest task komunikacji. Obsługuje on wszystkie wiadomości przychodzące z aplikacji napisanej w języku Python i wysyła odpowiednie wiadomości zwrotne (zazwyczaj są to tylko wiadomości potwierdzające odbiór, jednak w przypadku zbyt dużego wychylenia robota sygnał będzie odpowiednio inny i sprawi, że na ekranie pojawi się informacja o niebezpieczeństwie lub w przypadku pada będzie to informacja połączona z wibracją pada).

W przypadku tasku komunikacji nie ma potrzeby resetowania dodatkowych zmiennych, ponieważ wszystkie potrzebne zmienne zostały już zmienione w tasku ruchu. Procedura main składa się z 3 funkcji – ListenTCP, ListenUDP oraz ListenEGM. Każda z tych procedur nasłuchuje wiadomości przychodzących w odpowiednim wariancie komunikacji.

```
PROC ListenTCP()
    IF CommType = "TCP" THEN
        createServerTCP;
        connectWithClientTCP;
        WHILE CommType = "TCP" DO
            SocketReceive clientSocketTCP\Str:=Message;
            analyzingMessageCommType;
            IF CommType = "UDP" OR CommType = "EGM" THEN
                IF CommType = "UDP" THEN
                    SocketClose clientSocketTCP;
                ENDIF
                RETURN;
            ELSEIF CommType = "EGMSTOP" THEN
                EGMStopSignal := 1;
                RETURN;
            ELSEIF CommType = "STOP" THEN
                StopSignal := 1;
                Stop;
            ELSE
                CommunitationType := "TCP/UDP";
                CommType := "TCP";
            ENDIF
            analyzingMode;
            IF JointCoordinateSignal = "JOINT" THEN
                analyzingJointMessage;
            ELSEIF JointCoordinateSignal = "COORDINATES" THEN
                analyzingCoordinateMessage;
            ENDIF
            IF coordinatesOutOfRange THEN
                SocketSend clientSocketTCP\Str:="COORDINATES_OUT_OF_RANGE";
            ELSEIF jointsOutOfRange THEN
                SocketSend clientSocketTCP\Str:="JOINTS_OUT_OF_RANGE";
            ENDIF
        ENDWHILE
    ENDIF

    ERROR
        IF ERRNO = ERR_SOCK_CLOSED THEN
            TPWrite "Connection closed";
            CommType := "TCP";
            CommunitationType := "";
            MoveSignal := FALSE;
        ENDIF
ENDPROC
```

Rysunek 19 Procedura ListenTCP

Domyślnym protokołem komunikacji jest protokół TCP, na początku tworzony jest serwer w aplikacji RobotStudio i aktywowane jest połączenie z klientem. Następnie program wchodzi do pętli, którą opuszcza dopiero wtedy, gdy z aplikacji w języku Python nadejdzie informacja, że został zmieniony tryb komunikacji. Od razu po wejściu do pętli while program czeka na odbiór wiadomości. Tutaj warto zaznaczyć jaką strukturę ma przychodząca wiadomość:

Protokół komunikacji/ Zatrzymanie programu	;	Tryb sterowania	;	Wartości dla poszczególnych ruchów
TCP/UDP/EGM/ EGMSTOP/STOP		JOINT/COORDINATES		jAngle1;jAngle2;jAngle3;jAngle4;jAngle5;jAngle6;/ xCoordinate;yCoordinate;zCoordinate;

Rysunek 20 Struktura przychodzącej wiadomości

Do pierwszego średnika odczytywana jest metoda komunikacji, lub awaryjne wyłączanie EGM, lub sygnał wyłączenia aplikacji. Do drugiego średnika odczytywana jest metoda sterowania, następnie w przypadku trybu Joint – sześć kolejnych średników determinuje wartości kątów złączowych robota, a w przypadku trybu sterowania Coordinates – trzy średniki wyznaczają trzy współrzędne ruchu.

Po odszyfrowaniu sposobu komunikacji aplikacja decyduje czy w tym miejscu należy zakończyć wykonywanie danej procedury, czy też przejść dalej. W przypadku braku zmiany sposobu komunikacji aplikacja czyta tryb sterowania i odpowiednio później wywołuje instrukcje w celu przypisania wartości do tablic. Poniżej przykład funkcji odczytującej wartości kątów złączowych robota.

```
PROC analyzingJointMessage()
    semicolonsPositions{3} := StrFind(Message, semicolonsPositions{2} + 1, ";");
    semicolonsPositions{4} := StrFind(Message, semicolonsPositions{3} + 1, ";");
    semicolonsPositions{5} := StrFind(Message, semicolonsPositions{4} + 1, ";");
    semicolonsPositions{6} := StrFind(Message, semicolonsPositions{5} + 1, ";");
    semicolonsPositions{7} := StrFind(Message, semicolonsPositions{6} + 1, ";");
    semicolonsPositions{8} := StrFind(Message, semicolonsPositions{7} + 1, ";");
    analyzingStatus := StrToVal(StrPart(Message,semicolonsPositions{2}+1,semicolonsPositions{3}-semicolonsPositions{2}-1),jointAngles{1});
    analyzingStatus := StrToVal(StrPart(Message,semicolonsPositions{3}+1,semicolonsPositions{4}-semicolonsPositions{3}-1),jointAngles{2});
    analyzingStatus := StrToVal(StrPart(Message,semicolonsPositions{4}+1,semicolonsPositions{5}-semicolonsPositions{4}-1),jointAngles{3});
    analyzingStatus := StrToVal(StrPart(Message,semicolonsPositions{5}+1,semicolonsPositions{6}-semicolonsPositions{5}-1),jointAngles{4});
    analyzingStatus := StrToVal(StrPart(Message,semicolonsPositions{6}+1,semicolonsPositions{7}-semicolonsPositions{6}-1),jointAngles{5});
    analyzingStatus := StrToVal(StrPart(Message,semicolonsPositions{7}+1,semicolonsPositions{8}-semicolonsPositions{7}-1),jointAngles{6});
    Message := "";
    MoveSignal := TRUE;
    WaitUntil MoveSignal = FALSE;
ENDPROC
```

Rysunek 21 Analiza wiadomości metody Joints

```
PROC analyzingCoordinateMessage()
    semicolonsPositions{3} := StrFind(Message, semicolonsPositions{2} + 1, ";");
    semicolonsPositions{4} := StrFind(Message, semicolonsPositions{3} + 1, ";");
    semicolonsPositions{5} := StrFind(Message, semicolonsPositions{4} + 1, ";");
    analyzingStatus := StrToVal(StrPart(Message,semicolonsPositions{2}+1,semicolonsPositions{3}-semicolonsPositions{2}-1),coordinates{1});
    analyzingStatus := StrToVal(StrPart(Message,semicolonsPositions{3}+1,semicolonsPositions{4}-semicolonsPositions{3}-1),coordinates{2});
    analyzingStatus := StrToVal(StrPart(Message,semicolonsPositions{4}+1,semicolonsPositions{5}-semicolonsPositions{4}-1),coordinates{3});
    Message := "";
    MoveSignal := TRUE;
    WaitUntil MoveSignal = FALSE;
ENDPROC
```

Rysunek 22 Analiza wiadomości metody Coordinates

Powyżej znajduje się kod procedury odczytującej i zapisującej dane odnośnie ruchu xyz.

Warto zaznaczyć, że jest to łatwy sposób przesyłania wiadomości, ponieważ wystarczy znaleźć średnik i aplikacja wie, że wartości przesyłane znajdują się przed danym średnikiem. Dodatkowo współrzędne średników w stringu wiadomości zapisywane są w tablicy, co dodatkowo poprawia przejrzystość kodu.

Procedura ListenUDP nie różni się bardzo od procedury ListenTCP. Nie istnieje tam tylko procedura połączenia z klientem, ponieważ tak działa protokół komunikacji UDP, że serwer przyjmuje każdą wiadomość, niezależnie od nadawcy i nie wymaga zawiązania połączenia. Dodatkowo na końcu wyżej wymienionych procedur znajduje się obsługa błędu zerwania połączenia, co resetuje wskaźnik programu do metody komunikacji TCP.

W przypadku ListenEGM jest nieco inaczej, ponieważ metoda EGM pozwala jedynie na przesłanie koordynatów lub kątów złączowych robota. Nie pozwala natomiast na przesłanie żądania wyłączenia przesyłania danych ani żadnego innego sygnału, który mógłby posłużyć za sygnał końca transmisji, dlatego w przypadku wybrania metody komunikacji EGM dodatkową koniecznością jest zawiązanie połączenia dodatkowego, które taką informację będzie w stanie przesłać. Tym trybem komunikacji jest TCP.

```
PROC ListenEGM()
    IF CommType = "EGM" THEN
        analyzingMode;
        CommunitationType := "EGM";
        CommType := "TCP";
    ENDIF
ENDPROC
```

Rysunek 23 Procedura ListenEGM

W przypadku EGM aplikacja odczytuje tryb sterowania i ustawia odpowiednie zmienne i ustawia zmienną CommType na TCP, co w konsekwencji sprawi, że program znów stworzy serwer TCP oraz zacznie nasłuchiwać wiadomości z aplikacji o zakończeniu trybu EGM, bądź zmianie trybu sterowania.

4.2 Opis aplikacji napisanej w języku Python.

Najważniejszym zadaniem aplikacji napisanej w języku Python jest przechwytywanie i odczyt danych z urządzeń peryferyjnych (klawiatura, myszka, pad) oraz wysyłanie przetworzonych danych do programu w RobotStudio. Aplikacja napisana jest w taki sposób, aby można ją było łatwo rozbudowywać o kolejne urządzenia peryferyjne oraz interpretację przychodzących z nich sygnałów.

Założeniem programu jest brak interferencji ze sobą pomiędzy wykorzystanymi urządzeniami peryferyjnymi. Gdy zdecydujemy się na sterowanie myszką, program nie obsługuje wtedy zdarzeń przychodzących z pada i na odwrót, natomiast zawsze słucha przychodzących danych z klawiatury, gdyż jest ona urządzeniem nadrzędnym.

W programie zdefiniowana jest funkcja EventListener, która nasłuchuje każdych nadchodzących eventów. Zdefiniowane są klasy takie jak MouseListener, KeyboardListener, PadListener, które dziedziczą po klasie EventListener. Każda z osobna klasa uaktywnia odpowiednie procedury, gdy zostanie wywołany odpowiedni event nadchodzący z urządzenia peryferyjnego.

Odpowiednie funkcje zarządzają obróbką nadchodzących danych z pada, myszki oraz klawiatury. Jest to o tyle istotne, że dane nie są oznaczone w sposób jednolity. Zarówno pad, jak i myszka, czy klawiatura przysyłają sygnały o różnych nazwach i różnych zakresach wartości, więc koniecznością jest zadeklarowanie osobnych funkcji dla każdej metody dekodowania oraz zamiany na odpowiednie wartości dla robota.

Rysunek 24 Nazwy przychodzących zdarzeń z pada

Na rysunku powyżej zostało oznaczone, w jaki sposób program w języku Python odbiera zdarzenia przychodzące z pada. Podpisy opisują jakie mają oznaczenia poszczególne przyciski i gałki oraz jaki jest zakres ich danych z nich przychodzących – jeśli dane rozdziela średnik, znaczy to że tylko te dwie wartości są możliwe w ramach danych przychodzących, jeśli natomiast oddziela je przecinek, znaczy to, że wartości przychodzące znajdują się w danym przedziale.

Rysunek 25 Interpretacja danych z pada

Na rysunku na poprzedniej stronie oznaczony został sposób interpretacji danych przychodzących z pada. Główne sterowanie będzie się odbywać za pomocą dwóch gałek oraz przycisków po lewej stronie, dodatkowe zaś, czyli reset kątów i współrzędnych (aby przywrócić robota do pozycji domyślnej) oraz zmiana trybu sterowania.

Kolejnym urządzeniem peryferyjnym wykorzystanym w tej pracy jest myszka. Wartości zmiennych przychodzących w zależności od ruchu myszki są następujące:

W aplikacji zdefiniowane są trzy funkcje odczytujące zmiany stanów przychodzących z myszki. Są to on_move(x, y), on_click(x, y, button, pressed) oraz on_scroll(x, y, dx, dy).

Funkcja on_move(x, y) odczytuje aktualną pozycję myszki na ekranie komputera w formie koordynatów x oraz y. Nie jest to różnica pomiędzy poprzednim a aktualnym stanem tylko nierelatywna pozycja myszki. Pozwala to w łatwy sposób interpretować dane do wysyłania w trybie komunikacji TCP oraz UDP.

Funkcja on_click(x, y, button, pressed) zwraca aktualną pozycję myszki jeśli został naciśnięty któryś przycisk, zwraca nazwę wciśniętego przycisku oraz jego stan(0 lub 1). Pozwala to na przypisanie przyciskom odpowiednich funkcji, które są opisane w jednym z kolejnych akapitów.

Funkcja on_scroll(x, y, dx, dy) zwraca natomiast pozycję myszki gdzie został użyty środkowy przycisk myszki do przewijania oraz różnicę między poprzednim a aktualnym stanem przewijania. Warto zaznaczyć, że w zwykłej myszce komputerowej tylko zmienna dy podlega odczytowi, ponieważ scroll działa tylko w jednej płaszczyźnie. Zmienna dx zawsze jest równa 0.

Interpretacja nadchodzących sygnałów jest następująca: ruch myszki na płaszczyźnie porusza robotem w trybie sterowania xyz na wybranej płaszczyźnie (domyślnie jest to płaszczyzna xy, natomiast używając przycisku typu scroll ruchami góra, dół zmieniamy położenie robota w osi z), po naciśnięciu lewego przycisku myszy płaszczyzna zmienia się na yz, natomiast scroll obsługuje ruch na osi x, a po kolejnym na xz a scroll obsługuje ruch na osi y w nieskończonym cyklu. Naciśnięcie prawego przycisku myszy resetuje współrzędne wysyłane do robota, a więc ustawia go w pozycji domowej. W przypadku naciśnięcia środkowego przycisku myszy następuje zmiana trybu sterowania na złączowy. W tym przypadku poruszanie myszką na płaszczyźnie powoduje zmianę w pierwszej i drugiej współrzędnej złączowej, po naciśnięciu przycisku lewego przycisku myszy następuje zmiana

sterowania na trzecią i czwartą współrzędną złączową, a po kolejnym naciśnięciu – na piątą i szóstą. Prawym przyciskiem myszy resetuje się ustawienie wszystkich złączy, tak jak w trybie sterowania xyz, natomiast środkowym przyciskiem myszy resetuje się tylko wartości kątów, które aktualnie są wybrane (w przypadku sterowania trzecią i czwartą, tylko te zostaną wyzerowane).

Ostatnim, jednak najważniejszym urządzeniem peryferyjnym jest klawiatura. Obsługuje ona zmianę trybu wysyłania wiadomości (protokół komunikacyjny), zmianę wybranego urządzenia peryferyjnego oraz całkowite wyjście z aplikacji. Zmiana protokołu odbywa się za pomocą przycisku „P" natomiast tryb sterowania za pomocą przycisku „T". Zakończenie programu następuje po wciśnięciu przycisku „Esc".

5. Porównanie wydajności i płynności metod komunikacji na przykładach użycia aplikacji.

W celu sprawdzenia funkcjonalności aplikacji i porównania implementacji metod komunikacji zostaną przeprowadzone testy. Testy będą zakładały sprawdzenie następujących funkcji: przetestowanie każdego protokołu, każdego trybu sterowania oraz każdej funkcjonalności urządzeń peryferyjnych.

W celu łatwiejszego pokazania jakie testy zostaną przeprowadzone, zostaną użyte następujące skróty: TCP – połączenie za pomocą protokołu TCP, UDP – połączenie za pomocą protokołu UDP, EGM – połączenie za pomocą protokołu EGM, JOINTS – tryb sterowania kątami złączowymi robota, COORDINATES – tryb sterowania w przestrzeni za pomocą współrzędnych xyz.

Zarówno przypadku Pada jak i w przypadku myszki testy wyglądały następująco:

TCP-JOINTS	UDP-JOINTS	EGM-JOINTS
TCP-COORDINATES	UDP-COORDINATES	EGM-COORDINATES

Na tej podstawie zostanie porównane sterowanie za pomocą poszczególnych protokołów. Dodatkowo przełączenie pomiędzy każdym rodzajem testu zostanie wykonane poprzez zaimplementowane funkcje za pośrednictwem urządzeń peryferyjnych oraz za pomocą klawiatury. Każdy test zostanie przeprowadzony na tyle długo aby możliwe było uzyskanie odpowiedzi na interesujące pytania i problemy. Z każdego testu zostanie utworzony plik .exe, który będzie stanowił materiał poddany analizie w celu wysnucia wniosków na temat wydajności i płynności działania.

Jako pierwsza zostanie porównana implementacja poszczególnych metod komunikacji poprzez sterowanie padem. W przypadku metod TCP, UDP oraz EGM jest widoczna znaczna różnica jeśli chodzi o szybkość wykonywania, obsługę odbioru dużej ilości współrzędnych oraz płynność przechodzenia pomiędzy poszczególnymi ruchami.

Porównując obsługę nadchodzącej dużej ilości współrzędnych zdecydowanie daje się zauważyć, że w przypadku metody komunikacji UDP nie jest ona w stanie nadążyć za nadchodzącymi wiadomościami i, jeśli odpowiednio dużo informacji zostanie wysłane, program w języku RAPID nie jest w stanie nadążyć, przez co wydaje się, że robot porusza się

sam. W przypadku dużej ilości nadchodzących współrzędnych opóźnienie pomiędzy wysłaniem a wykonaniem jest znaczne (potrafi przekroczyć parę sekund). W przypadku TCP oraz EGM różnica nie jest tak bardzo zauważalna.

Biorąc pod uwagę płynność ruchu na pierwszym miejscu zdecydowanie plasuje się metoda EGM. Przeskok braku informacji pomiędzy poszczególnymi wysyłanymi wiadomościami jest praktycznie niezauważalny. Nie ma wrażenia „schodkowego" i urywanego poruszania się robota, co sprawia, że siły złączowe są mniejsze i robot nie zużywa dużych ilości energii oraz nie zużywają się elementy. W przypadku TCP oraz UDP przerywany ruch robota jest wyraźnie widoczny. Jest to konsekwencja działania programu w języku RAPID. Instrukcje wykonują się po kolei, co również dotyczy instrukcji ruchu robota. Robot kończy ruch a następnie Program Pointer przechodzi do następnej instrukcji, co trwa niezerową ilość czasu.

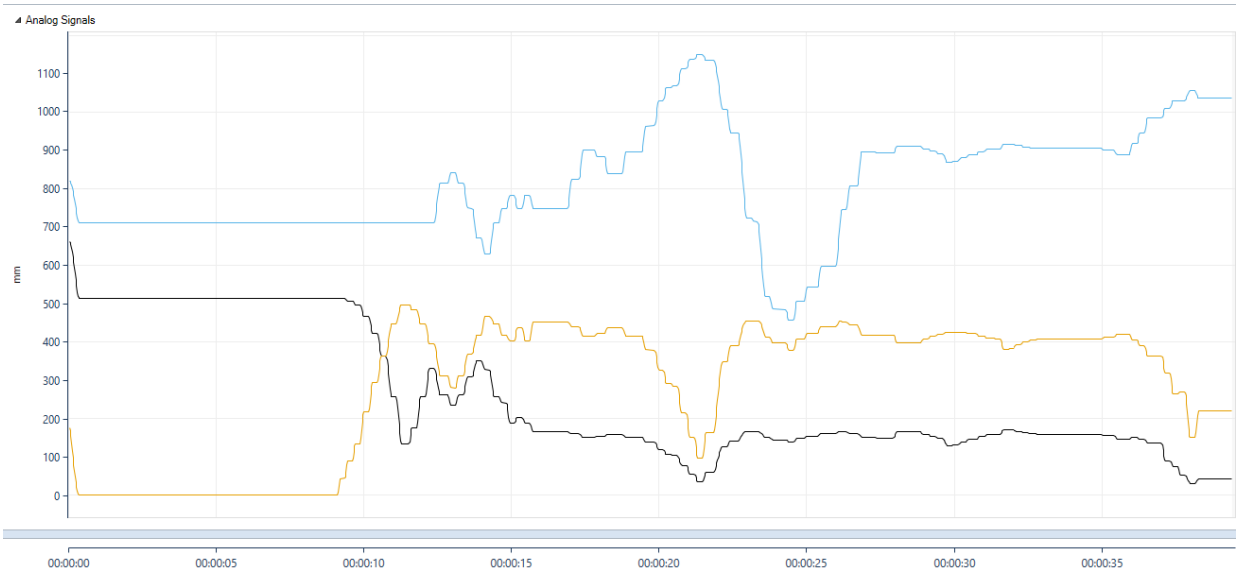
Ruch robota w przypadku sterowania padem jest dość dokładny. Program jest w stanie w odpowiedni sposób zinterpretować nadchodzące wiadomości oraz odpowiednio je przetworzyć przed wysłaniem do robota. Niestety pad nie jest idealny, ponieważ dane przychodzące z niego nie są ciągłe. Po wciśnięciu i przytrzymaniu przycisku wiadomość wysyłana jest tylko raz. Nie jest wysyłana w sposób ciągły o określonym czasie próbkowania, podobnie jak w przypadku operowania gałkami. Gdyby była wysyłana w sposób ciągły nie trzeba było by zmieniać położenia gałki ani co jakiś krótki czas naciskać przyciski.

Jako drugie urządzenie peryferyjne sterujące ruchami robota została wykorzystana myszka komputerowa. Zarówno w przypadku pada jak i w przypadku myszki różnice przy użyciu poszczególnych form komunikacji są bezprecedensowo widoczne na pierwszy rzut oka.

Porównując obsługę dużych ilości informacji na raz na pierwszym miejscu zdecydowanie plasuje się komunikacja przez EGM. Ruch robota nie ma zauważalnych ani uciążliwych opóźnień, co zdecydowanie wpływa na duży plus dla tej formy komunikacji. W przypadku TCP różnica przy dużej ilości napływających paczkach informacji jest już zauważalna, jednak nie jest tak bardzo uciążliwa i można w sposób bezproblemowy pracować przy tym protokole komunikacji. Na ostatnim miejscu zdecydowanie znajduje się połączenie po UDP. Ruch przy dużej ilości napływających informacji jest bardzo opóźniony, niekiedy nawet wielosekundowo (przy testach czas ten przekraczał nawet 10 sekund w niektórych przypadkach) . Ruch jest opóźniony nawet przy średniej ilości wysyłanych danych w ciągu określonego czasu.

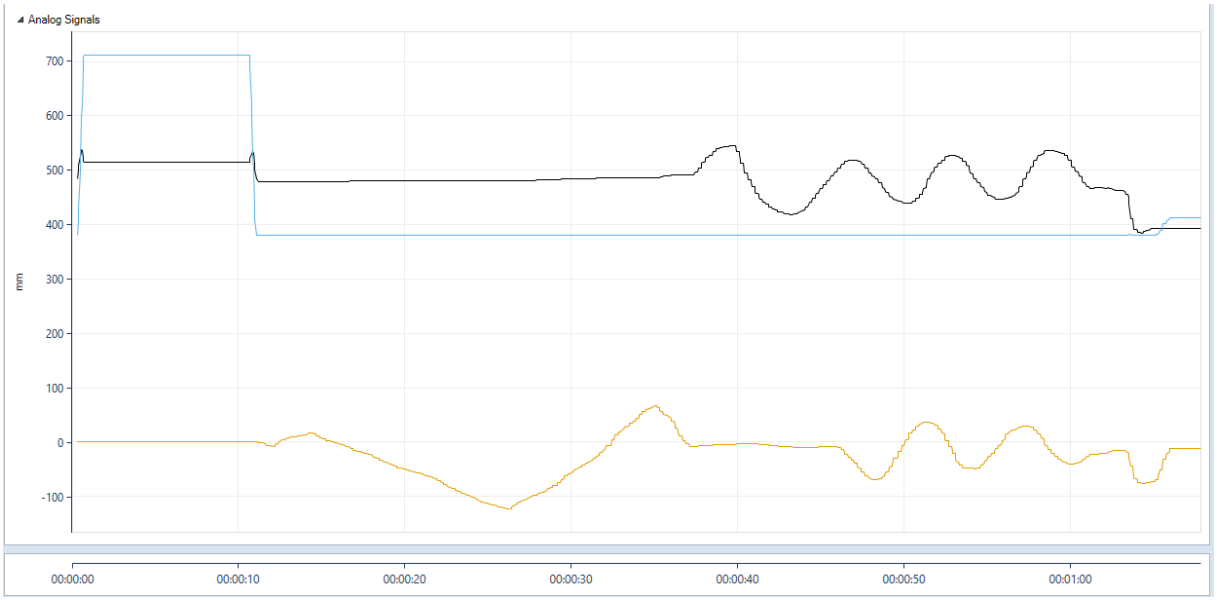
Płynność ruchu również zdecydowanie jest atrybutem EGM, podobnie jak w przypadku sterowania padem. Robot porusza się tak płynnie jakby pozycje lub kąty złączowe zostały wysłane nie za pośrednictwem aplikacji a za pośrednictwem instrukcji ruchu w języku RAPID, co jest ogromną zaletą. W przypadku sterowania za pomocą TCP oraz UDP ruch jest przerywany, zdarzają się postoje kilkuset milisekundowe, co zdecydowanie wpływa na odbiór wizualny jego ruchu. W języku RAPID niemożliwe jest uzyskanie lepszego wykonania nadchodzących instrukcji, co zostało powiedziane na stronie wcześniej przy porównaniu metod komunikacji w zależności od płynności ruchu przy zastosowaniu pada.

Teraz zostaną omówione trzy testy – pierwszy zakłada poruszanie padem, protokół TCP, metoda sterowania JOINTS, następnie drugi test poruszanie myszką, protokół UDP, metoda sterowania COORDINATES. Ostatni test zakłada poruszanie padem, protokół EGM oraz metoda sterowania JOINTS.



Rysunek 26 Test komunikacji 1

W tym wypadku pierwszego testu z użyciem protokołu TCP można zauważyć niedługie poziome linie, które są czasem opóźnienia między wysłaniem wiadomości, odczytaniem oraz wywołaniem instrukcji ruchu. Średni czas po przeliczeniu danych zapisanych w Excelu wynosi około 136 ms. Po zaprzestaniu wysyłania wiadomości (ostatnim ruchu gałką na padzie) program zatrzymał się po około 100ms, to znaczy, że ostatni pakiet wiadomości dotarł praktycznie natychmiast.



Rysunek 27 Test komunikacji 2

W przypadku UDP średnia wartość opóźnienia między kolejnymi wiadomościami jest krótsza niż w przypadku TCP, jednak jest to niewielka wartość (około 125ms). Jest to spowodowane prawdopodobnie tym, że samo opóźnienie między wywołaniem instrukcji ruchu a rzeczywistym poruszeniem się robota jest znacząca. Testy te zostały wykonane w miejscu pracy, więc nie mogą zostać tutaj pokazane. Biorąc pod uwagę opóźnienie maksymalne to przekroczyło ono 10s. Gdy na wykresie w obu przypadkach zaczynają pojawiać się sinusoidy, wartości te zostały zadanie po pierwszej górce na żółtym wykresie.

Rysunek 28 Test komunikacji 3

Jak łatwo zauważyć w przypadku EGM, nie ma żadnych zauważalnych poziomych linii w trakcie zadawania pozycji dla danego kąta złączowego. Sygnał zawsze jest ciągły a poziome

linie na wykresie wynikają jedynie z faktu, iż w tym czasie nie były wysyłane sygnały do danego złącza. Po zadaniu ostatniego kąta program również zatrzymuje się z niezauważalnym opóźnieniem. Niestety w tym przypadku nie jest możliwe śledzenie TCP w Signal Analyzer. Może to być zrobione jedynie dla innych metod komunikacji.

Sterowanie za pomocą klawiatury działa bardzo płynnie i niezawodnie. Klawiatura po wciśnięciu przycisku „P" program przechodzi do trybu wysyłania za pomocą kolejnego protokołu komunikacji. Zostaje zerwane połączenie dla przykładu TCP i nawiązywane jest kolejne połączenie UDP. Przycisk „T" działa dokładnie tak jak zakładano oraz tak jak przycisk „Y" a więc zmienia tryb sterowania robotem. Klawisz „R" powoduje powrót do wyboru urządzenia. Natomiast gdy zostanie wciśnięty przycisk „Esc" program wysyła wiadomość kończącą program w RobotStudio, robot przechodzi do pozycji HOME a program w Pythonie kończy się.

Porównywany problem	Protokół komunikacji		
	TCP	UDP	EGM
Opóźnienie pomiędzy wysyłanymi danymi a ruchem robota	Małe (opóźnienie nie przekracza 0.5s)	Duże (opóźnienie w przypadku wysyłania dużych ilości danych przekracza nawet 10s)	Zerowe/Niezauważalne
Płynność ruchu robota	Widoczne przerywanie ruchu robota (średnio 136ms różnicy)	Widoczne przerywanie ruchu robota (około 125ms)	Idealna (brak wrażenia przerywanego ruchu)
Wykonanie nagłego dużego ruchu	Brak problemów z wykonaniem	Brak problemów z wykonaniem	Błąd zbyt dużego obciążenia
Zgubione wartości przemieszczenia	Brak	Brak/Niewielka ilość	Brak
Implementacja	Łatwa	Łatwa	Trudniejsza niż TCP i UDP

Tabela 1 Porównanie protokołów komunikacji

Jak pokazują dane w tabeli powyżej najlepiej biorąc pod uwagę opóźnienia, płynność ruchu oraz zgubione przychodzące wiadomości, EGM jest bez wątpienia najlepszym

protokołem komunikacyjnym. Problemem jest jedynie skomplikowane przygotowanie w celu implementacji tego protokołu w programach oraz okazjonalne błędy przy zbyt dużych przesunięciach. Protokół TCP bardzo dobrze radzi sobie jeśli chodzi o opóźnienia oraz gubienie wiadomości a także jest łatwą w implementacji metodą komunikacji.

6. Zastosowanie i dalszy rozwój.

Aplikacja może zostać wykorzystana w celach nauki oraz rozwoju początkujących programistów robotów. Przybliża ona zagadnienie sterowania robotem poprzez znane urządzenia peryferyjne i prowadzi do poznania odpowiednich instrukcji zarówno na poziomie stricte kontroli robota (programowanie w języku RAPID w programie RobotStudio) oraz wysyłanie odpowiednich komend za pośrednictwem aplikacji (program w języku Python).

Połączenie tych dwóch metod pozwala na pełną kontrolę robota za pomocą dowolnego urządzenia po uprzednim rozszerzeniu aplikacji o odczyt i dekodowanie przychodzących danych, co pozwoli na szybką naukę i szybkie zrozumienie zagadnień związanych z bezpośrednią kontrolą robotów.

Dodatkowo przy przejściu na mniejsze roboty (np. Dobot Magician) będzie można zainteresować najmłodszych robotyką poprzez zabawę. Zajęcia zaznajamiające dzieci i młodzież z robotami oraz z możliwością ich kontroli poprzez znane im proste urządzenia kontrolujące np. myszka, pad, klawiatura, czy joystick jak również te bardziej zaawansowane (CadMouse, kamera odczytująca pozycję urządzeń sterujących w przestrzeni), stanowią kierunek dalszego rozwoju, który pozwoli na edukację zabawą, co z kolei zachęci w przyszłości dzieci i młodzież do aktywnego brania udziału w festiwalach nauki, czy w dalszej przyszłości do wyboru studiów związanych z robotyką.

Kolejną przydatną implementacją będzie przesyłanie danych odnośnie bezpośredniego załączania i wyłączania sygnałów mających na celu ruch dodatkowych urządzeń zamocowanych przy robocie (chwytaków, przyssawek, czy przenośników) oraz dodatkowo obsługa danych przychodzących z dodatkowych urządzeń do robota (aktualne parametry taśmociągu, lub też stanu przyssawek i kamer).

Dodatkowo dalszym rozwojem może być dodanie całej bazy robotów i odpowiednich instrukcji i wymagań odnośnie poszczególnych modeli, tj. ich wymiarów, ograniczeń ruchowych (jak w przypadku robota IRB360), większej ilości osi (IRB14000 „YuMi") czy też robota typu SCARA (IRB910SC). Zostanie to zaimplementowane w programie w języku Python. Na początku włączania programu będzie istniała możliwość wyboru poszczególnych klas robotów i podklas modeli w zakresie jednego numeru (np. IRB140 oraz IRB140T), jednak każdy robot będzie musiał mieć zdefiniowaną klasę w języku Python, ponieważ każdy posiada

inny zakres ruchów, inne wartości odchyleń kątów złączowych a także prędkości i przyspieszenia.

Proste w użyciu biblioteki oraz łatwość implementacji kolejnych urządzeń jak na przykład CadMouse (myszka 3D) oraz Joystick a także w przyszłości kamera odczytująca pozycję i orientację urządzenia, lub nawet człowieka stanowi docelowy rozwój niniejszej pracy. Rozszerzenie programu Python o nowe modele robotów i ich parametry, dodatkowe zuniwersalizowanie programów poprzez wysyłanie odpowiednich parametrów do uniwersalnego programu w RobotStudio.

7. Podsumowanie i wnioski.

Sformułowany cel pracy został osiągnięty. Zostały utworzone programy zapewniające komunikację między robotem i aplikacją sterującą wykorzystującą przychodzące informacje z różnych urządzeń peryferyjnych na podstawie kilku protokołów komunikacji i metod sterowania robotem. Dodatkowo zostały przeprowadzone testy pokazujące wady i zalety poszczególnych protokołów.

Sterowanie robotem za pomocą różnych protokołów komunikacji pozwala na przeanalizowanie zalet oraz wad każdej metody komunikacji oraz współdziałania zależnych od siebie programów w języku Python oraz w RobotStudio. Z przeprowadzonych testów i badań wynika, że najbardziej dokładnym sposobem komunikacji i sposobem, który pozostawia najmniejszą wartość opóźnienia w wysyłaniu i odbieraniu informacji o położeniu robota poprzez sterowanie rzeczywiste urządzeniami peryferyjnymi jest protokół EGM stworzony na potrzeby robotów ABB. Jest on jednak trudniejszy w zaimplementowaniu niż zwykła metoda komunikacji jak TCP, czy UDP oraz trudniejszy w kontroli, ponieważ aby ograniczyć ruch robota w przestrzeni należy dodać kolejny task kontrolujący aktualną pozycję robota w trybie komunikacji EGM, czy będzie to sterowanie kątami złączowymi robota, czy też ruch na płaszczyźnie.

W przypadku chęci sterowania robotem poprzez urządzenie peryferyjne protokół EGM jest najlepszym wyborem dla natychmiastowej i bezproblemowej komunikacji. Należy jednak uważać na ograniczenia i niebezpieczeństwa z tym związanie (dodatkowe ograniczenia w osobnym tasku oraz brak możliwości wysyłania dodatkowych sygnałów tylko za pośrednictwem EGM).

Gdy zadanie wymaga sterowanie robotem tylko poprzez wysyłanie danych współrzędnych do robota – TCP jest modułem komunikacyjnym, którego należy użyć. Gwarantuje szybki ruch, względny brak opóźnień oraz pewność w przesyłaniu wiadomości. Dodatkowo umożliwia przesyłanie wszystkich informacji tą metodą bez konieczności zawiązywania dodatkowych połączeń. W przypadku płynnego ruchu, gdzie informacje o pozycji są wysyłane w sposób ciągły najlepszym wyborem jest protokół komunikacyjny EGM.

Literatura, źródła

[1] https://ifr.org/ifr-press-releases/news/robot-investment-reaches-record-16.5-billion-usd

[2] https://ai.googleblog.com/2017/10/closing-simulation-to-reality-gap-for.html?m=1&fbclid=IwAR1UIQ78hsvW7KjPbZ7ORnFiGAxBQYNO1KCm4euODW7zdDMaM9ailYMxZm0

[3] https://www.techopedia.com/definition/25705/communication-protocol

[4] Jon Postel, Joyce K. Reynolds, File Transfer Protocol, STD 9, RFC 959, IETF, październik 1985

[5] Forouzan, B.A. (2000). *TCP/IP: Protocol Suite* (1st ed.). New Delhi, India: Tata McGraw-Hill Publishing Company Limited.

[6] https://pl.wikipedia.org/wiki/User_Datagram_Protocol

[7] https://en.wikibooks.org/wiki/Communication_Networks/TCP_and_UDP_Protocols

[8] https://www.lifewire.com/what-is-bluetooth-2377412

[9]ABB Robotics: RAPID Reference Manual, Västerås, Sweden (2007)

[10] ABB Robotics: Operating Manual RobotStudio, Västerås, Sweden (2007)

[11] ABB Robotics: Application Manual, Västerås, Sweden (2007)

[12] ABB Robotics: Technical Reference Manual, Västerås, Sweden (2007)

[13] https://new.abb.com/products/robotics/pl/roboty-przemyslowe/irb-140/irb-140-dane-techniczne

Dodatek do pracy

Kod programu napisany w języku Python:

```
# TODO: import devices modules

import enum
import socket
import time
import sys
import winput
from IPy import IP
from inputs import devices
from inputs import get_gamepad
from inputs import get_mouse
import inputs
import rpi_abb_irc5
from pynput.mouse import Listener, Button

class CurrentDevice(enum.Enum):
    NONE = 0
    PAD = 1
    MOUSE = 2


class EventLisner:
    def __init__(self, steering_type="JOINTS"):
        self.state = None
        self.prev_state = None
        self.currentTime = time.time()
        self.steering_type = steering_type
        self.jointAnglesToSend = [0,0,0,0,0,0]
        self.cartesianPositionToSend = [0,0,0]

    def get_state(self):
        return self.state[:] if self.state != self.prev_state else None


class MouseLisner(EventLisner):

    def __init__(self):
        super.__init__()
        self.zeroX = 960
        self.zeroY = 540
        self.jointsToMove = 0
        self.cartesianToMove = 0
```

```
    def on_move(self,  x, y):
        if self.steering_type == "JOINTS":
            if self.jointsToMove == 0:
                difference_x = round((x - self.zeroX) * 3 / 16, 1)
                difference_y = round((y - self.zeroY) * 5 / 27 + 10, 1)
            elif self.jointsToMove == 2:
                # difference_x = round((x - zeroX) * 7 / 48 - 90, 1)
                difference_x = round((x - self.zeroX) * 7 / 48, 1)
                difference_y = round((self.zeroY - y) * 10 / 27, 1)
            else:
                difference_x = round((x - self.zeroX) * 19 / 64, 1)
                difference_y = round((self.zeroY - y) * 20 / 27, 1)
            self.jointAnglesToSend[self.jointsToMove] = difference_x
            self.jointAnglesToSend[self.jointsToMove + 1] = difference_y
            self.state = "JOINTS;" + \
            str(self.jointAnglesToSend[0]) + ";" + \
            str(self.jointAnglesToSend[1]) + ";" + \
            str(self.jointAnglesToSend[2]) + ";" + \
            str(self.jointAnglesToSend[3]) + ";" + \
            str(self.jointAnglesToSend[4]) + ";" + \
            str(self.jointAnglesToSend[5])
        elif self.steering_type == "COORDINATES":
            if self.jointsToMove == 0:
                difference_x = round((x - self.zeroX) * 3 / 16, 1)
                difference_y = round((y - self.zeroY) * 5 / 27, 1)
            elif self.jointsToMove == 2:
                difference_x = round((x - self.zeroX) * 7 / 48, 1)
                difference_y = round((self.zeroY - y) * 10 / 27, 1)
            else:
                difference_x = round((x - self.zeroX) * 19 / 64, 1)
                difference_y = round((self.zeroY - y) * 20 / 27, 1)
            self.zeroX = x
            self.zeroY = y
            self.cartesianPositionToSend[self.cartesianToMove] += difference_y
            self.cartesianPositionToSend[(self.cartesianToMove + 1) % 3] += di
fference_x
            self.state = "COORDINATES" + \
            str(self.cartesianPositionToSend[0]) + ";" + \
            str(self.cartesianPositionToSend[1]) + ";" + \
            str(self.cartesianPositionToSend[2])

    def on_click(self, x, y, button, pressed):
        if pressed:
            if button == button.right:
                if self.steering_type == "JOINTS":
                    self.jointAnglesToSend = [0,0,0,0,0,0]
                    self.state = "JOINTS;0;0;0;0;0;0;"
                elif self.steering_type == "COORDINATES":
                    self.cartesianPositionToSend = [0,0,0]
```

```
                    self.state = "COORDINATES;0;0;0;"
            elif button == button.left:
                if self.steering_type == "JOINTS":
                    self.jointsToMove = (self.jointsToMove + 2) % 6
                elif self.steering_type == "COORDINATES":
                    self.cartesianToMove = (self.cartesianToMove + 1) % 3
            elif button == button.middle:
                if self.steering_type == "JOINT":
                    self.steering_type = "COORDINATES"
                    print("Coordinates move")
                else:
                    self.steering_type = "JOINT"
                    print("Joints move")

    def on_scroll(self,  x, y, dx, dy):
        if self.steering_type == "JOINTS":
            pass
        elif self.steering_type == "COORDINATES":
            self.cartesianPositionToSend[(self.cartesianToMove + 2) % 3] += dy
*10
            self.state = "COORDINATES" + \
            str(self.cartesianPositionToSend[0]) + ";" + \
            str(self.cartesianPositionToSend[1]) + ";" + \
            str(self.cartesianPositionToSend[2])

    def listen(self):
        with Listener(on_move=self.on_move, on_click=self.on_click, on_scroll=
self.on_scroll) as listener:
            listener.join()

    def get_state(self, steering_type):
        return self.state


class KeyboardLisner(EventLisner):

    def __init__(self):
        super.__init__()

    def listen(self, event, comtype):
        if event.vkCode == winput.VK_ESCAPE:
            self.steering_type = "EXIT"
        elif event.vkCode == winput.VK_P:
            if comtype == "TCP":
                comtype = "UDP"
            elif comtype == "UDP":
                comtype = "EGM"
            elif comtype == "EGM":
                comtype = "TCP"
```

```
        elif event.vkCode == winput.VK_T:
            if self.steering_type == "JOINT":
                self.steering_type = "COORDINATES"
                print("Coordinates move")
            else:
                self.steering_type = "JOINT"
                print("Joints move")
        elif event.vkCode == winput.VK_R:
            self.steering_type = "RESET"


class PadLisner(EventLisner):

    def __init__(self):
        super.__init__()

    def listen(self):
        events = get_gamepad()
        for event in events:
            if event.ev_type != 'Sync' and ( abs(event.state) == 1 and
                event.code != 'ABS_X' and event.code != 'ABS_Y' and
                event.code != 'ABS_RX' and event.code != 'ABS_RY' or
                event.code == 'ABS_RZ' or event.code == 'ABS_Z' or abs(event.s
tate) > 2500):
                    if self.steering_type == "COORDINATES":
                        if event.code == 'ABS_HAT0Y':
                            # cartesianPositionToSend[0] = - 200 * event.state
                            self.cartesianPositionToSend[0] += 10 * event.stat
e
                            # cartesianPositionToSend[1] = 0
                            self.cartesianPositionToSend[0] = round(self.carte
sianPositionToSend[0], 1)
                        elif event.code == 'ABS_HAT0X':
                            # cartesianPositionToSend[1] = 200 * event.state
                            # cartesianPositionToSend[0] = 0
                            self.cartesianPositionToSend[1] += 10 * event.stat
e
                            self.cartesianPositionToSend[1] = round(self.carte
sianPositionToSend[1], 1)
                        elif event.code == 'ABS_X':
                            # cartesianPositionToSend[1] = event.state / 100
                            # cartesianPositionToSend[2] = 0
                            self.cartesianPositionToSend[1] += event.state / 2
000
                            self.cartesianPositionToSend[1] = round(self.carte
sianPositionToSend[1], 1)
                        elif event.code == 'ABS_Y':
                            # cartesianPositionToSend[2] = - event.state / 100
                            # cartesianPositionToSend[1] = 0
```

```
                            self.cartesianPositionToSend[2] += event.state / 2
000
                            self.cartesianPositionToSend[2] = round(self.carte
sianPositionToSend[2], 1)
                        elif event.code == 'ABS_RY':
                            # cartesianPositionToSend[2] = - event.state / 100
                            # cartesianPositionToSend[0] = 0
                            self.cartesianPositionToSend[2] += event.state / 2
000
                            self.cartesianPositionToSend[2] = round(self.carte
sianPositionToSend[2], 1)
                        elif event.code == 'ABS_RX':
                            # cartesianPositionToSend[0] = - event.state / 100
                            # cartesianPositionToSend[2] = 0
                            self.cartesianPositionToSend[0] += event.state / 2
000
                            self.cartesianPositionToSend[0] = round(self.carte
sianPositionToSend[0], 1)
                        elif event.code == 'BTN_SOUTH' and event.state == 1:
                            # cartesianPositionToSend = [0,0,0]
                            self.cartesianPositionToSend = [484.224139982, 0,
381.030051802]
                        elif event.code == 'BTN_NORTH' and event.state == 1:
                            if self.steering_type == "JOINT":
                                self.steering_type = "COORDINATES"
                                print("Coordinates move")
                            else:
                                self.steering_type = "JOINT"
                                print("Joints move")
                        self.state = "COORDINATES" + \
                        str(self.cartesianPositionToSend[0]) + ";" + \
                        str(self.cartesianPositionToSend[1]) + ";" + \
                        str(self.cartesianPositionToSend[2])

                    elif self.steering_type == "JOINT":
                        if event.code == 'ABS_HAT0X':
                            self.jointAnglesToSend[0] += 5 * event.state
                        elif event.code == 'ABS_HAT0Y':
                            self.jointAnglesToSend[1] += 5 * event.state
                        elif event.code == 'ABS_Y':
                            self.jointAnglesToSend[2] += - event.state / 3277
                            self.jointAnglesToSend[2] = round(self.jointAngles
ToSend[2], 1)
                        elif event.code == 'ABS_X':
                            self.jointAnglesToSend[3] += - event.state / 3277
                            self.jointAnglesToSend[3] = round(self.jointAngles
ToSend[3], 1)
                        elif event.code == 'ABS_RY':
                            self.jointAnglesToSend[4] += - event.state / 3277
```

```
                            self.jointAnglesToSend[4] = round(self.jointAngles
ToSend[4], 1)
                        elif event.code == 'ABS_RX':
                            self.jointAnglesToSend[5] += - event.state / 3277
                            self.jointAnglesToSend[5] = round(self.jointAngles
ToSend[5], 1)
                        elif event.code == 'BTN_SOUTH' and event.state == 1:
                            self.jointAnglesToSend = [0,0,0,0,0,0]
                        elif event.code == 'BTN_NORTH' and event.state == 1:
                            if self.steering_type == "JOINT":
                                self.steering_type = "COORDINATES"
                                print("Coordinates move")
                            else:
                                self.steering_type = "JOINT"
                                print("Joints move")
                        self.state = "JOINTS;" + \
                        str(self.jointAnglesToSend[0]) + ";" + \
                        str(self.jointAnglesToSend[1]) + ";" + \
                        str(self.jointAnglesToSend[2]) + ";" + \
                        str(self.jointAnglesToSend[3]) + ";" + \
                        str(self.jointAnglesToSend[4]) + ";" + \
                        str(self.jointAnglesToSend[5])


class EventsManager:

    def __init__(self, current_device=CurrentDevice.NONE):
        self.devices = {}
        self.init_devices()
        self.exit = False
        self.current_device = CurrentDevice.NONE

    def init_devices(self):

        if EventsManager.get_keyboard():
            self.devices["keyboard"] == KeyboardLisner()
        else:
            raise Exception("Please make sure, that the keyboard is connected.
")

        if EventsManager.get_mouse():
            self.devices["mouse"] == MouseLisner()

        if EventsManager.get_pad():
            self.devices["pad"] == PadLisner()

        if len(self.devices) <=1:
            raise Exception("cos oprocz klawiatury")
```

```
    @staticmethod
    def get_mouse():
        return False #Check if exists

    @staticmethod
    def get_pad():
        return False

    @staticmethod
    def get_keyboard():
        return False

    def get_steering_msg(self):

        keyboard_state =self.listen_keyboard()
        if keyboard_state == "EXIT":
            return "EXIT;"

        elif keyboard_state == "RESET":
            self.current_device = CurrentDevice.NONE

        steering_msg = None

        if (self.current_device is CurrentDevice.MOUSE or self.current_device
is CurrentDevice.NONE) and "mouse" in self.devices.keys():
            steering_msg = self.listen_mouse()
            self.current_device = CurrentDevice.MOUSE

        elif (self.current_device is CurrentDevice.PAD or self.current_device
is CurrentDevice.NONE) and "pad" in self.devices.keys():
            steering_msg = self.listen_pad()
            self.current_device = CurrentDevice.PAD

        return steering_msg

    def listen_keyboard(self):
        return self.devices["keyboard"].listen() # args

    def listen_mouse(self):
        return self.devices["mouse"].listen() # args

    def listen_pad(self):
        return self.devices["pad"].listen() #args

class RobotSteering:
    def __init__(self, ip, port, comtype, steering_type):
        self.ip = ip
        self.port = port
        self.comtype = comtype
```

```
        self.event_menager = EventsManager()
        self.s = None
        self.exit = False
        self.set_socket(self.comtype)
        self.steering_type = steering_type
        self.egm=rpi_abb_irc5.EGM()

    def __del__(self):
        self.close_connection(self.comtype)

    def send(self, steering_msg):

        if self.comtype == "TCP":
            self.s.send("TCP;".encode() + steering_msg.encode())
        elif self.comtype == "UDP":
            self.s.sendto("TCP;".encode() + steering_msg.encode(), (self.ip, s
elf.port))
        elif self.comtype == "EGM":
            steering = [float(steer) for steer in steering_msg.split(";")[1:-
1]]
            if self.steering_type == "JOINTS":
                self.egm.send_to_robot(steering, None)
            elif self.steering_type == "COORDINATES":
                self.egm.send_to_robot(None, steering)

    def connect(self, comtype):
        self.close_connection(comtype)
        self.set_socket(comtype)

    def set_socket(self, comtype):
        if comtype == "TCP":
            self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.s.connect((self.ip, self.port))
        elif comtype == "UDP":
            self.s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        elif comtype == "EGM":
            res, state=self.egm.receive_from_robot(.01)
            self.set_socket("TCP")
            self.s.send("EGM;JOINTS;".encode())
        else:
            raise Exception ("Undefined type of communication.")

    def close_connection(self, comtype):
        if self.s:
            if self.comtype == "TCP":
                if comtype == "UDP":
                    self.s.send("UDP;JOINTS".encode())
                elif comtype == "EGM":
                    self.s.send("EGM;JOINTS".encode())
```

```
                self.s.close()
            elif self.comtype == "UDP":
                if comtype == "TCP":
                    self.s.sendto("TCP;JOINTS".encode(), (self.ip, self.port))
                elif comtype == "EGM":
                    self.s.sendto("EGM;JOINTS".encode(), (self.ip, self.port))
                self.s.close()

    def run(self):
        steering_msg = self.event_menager.get_steering_msg()
        if steering_msg:
            self.send(steering_msg)
        self.exit = steering_msg == "EXIT;"

    @staticmethod
    def validate_adress(ip_address, port):
        if port < 1024 or port > 65535:
            raise ValueError("Wrong port number.")
        IP(ip_address)

if __name__ == "__main__":

    try:
        steering = RobotSteering("127.0.0.1", 8888, "TCP", "JOINTS")
        while not steering.exit:
            steering.run()

    except ValueError as ve:
        print(ve)

    except Exception as e:
        print(e)
```

Kod programu napisany w języku RAPID:

Task 1:

```
MODULE Module1
    PERS string CommunitationType;
    PERS string JointCoordinateSignal;
    PERS bool MoveSignal;
    PERS num jointAngles{6};
    PERS num coordinates{3};
    PERS bool coordinatesOutOfRange;
    PERS bool jointsOutOfRange;
    LOCAL CONST jointtarget
HOME:=[[0,0,0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
    LOCAL VAR jointtarget
PosUDPTCPJoint:=[[0,0,0,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
    LOCAL VAR robtarget
PosUDPTCPCoordinate:=[[484.224139982,0,381.030051802],[0,0,1,0],[0,0,0,0],[9E+09,9E+
09,9E+09,9E+09,9E+09,9E+09]];
    LOCAL VAR bool settingsSignal := TRUE;
    LOCAL VAR egmident egmID1;
    LOCAL VAR egmstate egmSt1;
    LOCAL CONST egm_minmax egm_minmax_lin1:=[-1,1];
    LOCAL CONST egm_minmax egm_minmax_rot1:=[-0.5,0.5];
    LOCAL CONST egm_minmax egm_minmax_joint1:=[-0.5,0.5];
    LOCAL VAR pose corr_frame_offs:=[[0,0,0],[1,0,0,0]];
    LOCAL CONST string egmSensor:="EGMSensor";
    LOCAL CONST num COMM_TIMEOUT:=5;
    LOCAL VAR intnum EGMInterrupt;

    PROC main()
        Settings;
        WaitUntil CommunitationType <> "";
        WHILE CommunitationType = "TCP/UDP" DO
            tcpUdpJoinMove;
            tcpUdpCoordinateMove;
        ENDWHILE
        WHILE  CommunitationType = "EGM" DO
            EGMReset egmID1;
            checkingEGMState;
            egmJoinMove;
            egmCoordinateMove;
        ENDWHILE
    ENDPROC

    PROC Settings()
        IF settingsSignal THEN
            CommunitationType := "";
            JointCoordinateSignal := "";
            MoveSignal := FALSE;
```

```
        jointAngles := [0,0,0,0,0,0];
        coordinates := [0,0,0];
        settingsSignal := FALSE;
        coordinatesOutOfRange := FALSE;
        jointsOutOfRange := FALSE;
        MoveAbsJ HOME,v1000,fine,tool0\WObj:=wobj0;
    ENDIF
ENDPROC

PROC tcpUdpJoinMove()
    WHILE JointCoordinateSignal = "JOINT" DO
        IF CommunitationType = "EGM" THEN
            RETURN;
        ENDIF
        WaitUntil MoveSignal;
        restrictionsToJointMove;
        jointAnglesToTarget;
        MoveAbsJ PosUDPTCPJoint,v1000,fine,tool0\WObj:=wobj0;
        MoveSignal := FALSE;
    ENDWHILE
ENDPROC

PROC tcpUdpCoordinateMove()
    IF CommunitationType = "TCP/UDP" THEN
        MoveJ PosUDPTCPCoordinate,v1000,fine,tool0\WObj:=wobj0;
        WHILE JointCoordinateSignal = "COORDINATES" DO
            IF CommunitationType = "EGM" THEN
                RETURN;
            ENDIF
            WaitUntil MoveSignal;
            restrictionsToCoordinateMove;
            coordinatesToTarget;
            MoveJ PosUDPTCPCoordinate,v1000,fine,tool0\WObj:=wobj0;
            MoveSignal := FALSE;
        ENDWHILE
    ENDIF
ENDPROC

FUNC num clamp(num value, num minValue, num maxValue)
    IF value > maxValue THEN
        RETURN maxValue;
    ELSEIF value < minValue THEN
        RETURN minValue;
    ELSE
        RETURN value;
    ENDIF
ENDFUNC

PROC restrictionsToJointMove()
    jointAngles{1} := clamp(jointAngles{1}, -179, 179);
```

```
        jointAngles{2} := clamp(jointAngles{2}, -89, 109);
        jointAngles{3} := clamp(jointAngles{3}, -229, 49);
        jointAngles{4} := clamp(jointAngles{4}, -199, 199);
        jointAngles{5} := clamp(jointAngles{5}, -114, 114);
        jointAngles{6} := clamp(jointAngles{6}, -399, 399);
    ENDPROC

    PROC jointAnglesToTarget()
        PosUDPTCPJoint.robax.rax_1 := jointAngles{1};
        PosUDPTCPJoint.robax.rax_2 := jointAngles{2};
        PosUDPTCPJoint.robax.rax_3 := jointAngles{3};
        PosUDPTCPJoint.robax.rax_4 := jointAngles{4};
        PosUDPTCPJoint.robax.rax_5 := jointAngles{5};
        PosUDPTCPJoint.robax.rax_6 := jointAngles{6};
    ENDPROC

    PROC restrictionsToCoordinateMove()
        IF Sqrt(coordinates{1} * coordinates{1} + coordinates{2} * coordinates{2} +
(coordinates{3} - 352) * (coordinates{3} - 352)) > 800 OR
        Sqrt(coordinates{1} * coordinates{1} + coordinates{2} * coordinates{2} +
(coordinates{3} - 352) * (coordinates{3} - 352)) < 320 THEN
            coordinates{1} := 484.224139982;
            coordinates{2} := 0;
            coordinates{3} := 381.030051802;
            coordinatesOutOfRange := TRUE;
        ENDIF
    ENDPROC

    PROC coordinatesToTarget()
        PosUDPTCPCoordinate.trans.x := coordinates{1};
        PosUDPTCPCoordinate.trans.y := coordinates{2};
        PosUDPTCPCoordinate.trans.z := coordinates{3};
    ENDPROC

    PROC checkingEGMState()
        EGMGetId egmID1;
        egmSt1:=EGMGetState(egmID1);
        TPWrite "EGM state: "\Num:=egmSt1;
    ENDPROC

    PROC egmJoinMove()
        WHILE JointCoordinateSignal = "JOINT" DO
            IF egmSt1 <= EGM_STATE_CONNECTED THEN
                EGMSetupUC ROB_1, egmID1, "default", egmSensor\Joint
\CommTimeout:=COMM_TIMEOUT;
            ENDIF
            EGMStreamStart egmID1;
            EGMActJoint egmID1 \Tool:=tool0 \WObj:=wobj0,
            \J1:=egm_minmax_joint1 \J2:=egm_minmax_joint1 \J3:=egm_minmax_joint1
\J4:=egm_minmax_joint1 \J5:=egm_minmax_joint1 \J6:=egm_minmax_joint1
```

```
        \LpFilter:=100 \MaxSpeedDeviation:=1000;
        EGMRunJoint egmID1, EGM_STOP_HOLD \J1 \J2 \J3 \J4 \J5 \J6 \CondTime:=0.3
\PosCorrGain:=1;
      ENDWHILE
    ENDPROC

    PROC egmCoordinateMove()
      IF CommunitationType = "EGM" THEN
        MoveJ PosUDPTCPCoordinate,v1000,fine,tool0\WObj:=wobj0;
        WHILE JointCoordinateSignal = "COORDINATES" DO
          IF egmSt1 <= EGM_STATE_CONNECTED THEN
            EGMSetupUC ROB_1, egmID1, "default", egmSensor\Pose
\CommTimeout:=COMM_TIMEOUT;
          ENDIF
          EGMStreamStart egmID1;
          EGMActPose egmID1\Tool:=tool0, corr_frame_offs, EGM_FRAME_WORLD,
tool0.tframe, EGM_FRAME_TOOL
          \x:=egm_minmax_lin1 \y:=egm_minmax_lin1 \z:=egm_minmax_lin1
\rx:=egm_minmax_rot1 \ry:=egm_minmax_rot1 \rz:=egm_minmax_rot1
          \LpFilter:=20 \MaxSpeedDeviation:=1000;
          EGMRunPose egmID1, EGM_STOP_HOLD \x \y \z \CondTime:=0.3
\RampInTime:=0.025;
        ENDWHILE
      ENDIF
    ENDPROC

    PROC trapVariablesSetting()
      CONNECT EGMInterrupt WITH EGMInterruptTrap;
      ISignalDO interruptSignal,1,EGMInterrupt;
    ENDPROC

    TRAP EGMInterruptTrap
      IF DInput(EGMStopSignal) = 1 THEN
        RETURN;
      ELSEIF DInput(StopSignal) = 1 THEN
        Stop;
      ENDIF
    ENDTRAP
ENDMODULE
```

Task 2:

```
MODULE Communication
    PERS string CommunitationType;
    PERS string JointCoordinateSignal;
    PERS bool MoveSignal;
    PERS num jointAngles{6};
    PERS num coordinates{3};
    PERS bool coordinatesOutOfRange;
    PERS bool jointsOutOfRange;
    LOCAL VAR string CommType := "TCP";
    LOCAL VAR socketdev clientSocketTCP;
    LOCAL VAR socketdev clientSocketUDP;
    LOCAL VAR socketdev serverSocketTCP;
    LOCAL VAR socketdev serverSocketUDP;
    LOCAL VAR string Message := "";
    LOCAL VAR string IP_ADDRESS := "127.0.0.1";
    LOCAL VAR num PORT := 8888;
    LOCAL VAR num semicolonsPositions{8} := [0,0,0,0,0,0,0,0];
    LOCAL VAR bool analyzingStatus := FALSE;

    PROC main()
        ListenTCP;
        ListenUDP;
        ListenEGM;
    ENDPROC

    PROC ListenTCP()
        IF CommType = "TCP" THEN
            createServerTCP;
            connectWithClientTCP;
            WHILE CommType = "TCP" DO
                SocketReceive clientSocketTCP\Str:=Message;
                analyzingMessageCommType;
                IF CommType = "UDP" OR CommType = "EGM" THEN
                    IF CommType = "UDP" THEN
                        SocketClose clientSocketTCP;
                    ENDIF
                    RETURN;
                ELSEIF CommType = "EGMSTOP" THEN
                    EGMStopSignal := 1;
                    RETURN;
                ELSEIF CommType = "STOP" THEN
                    StopSignal := 1;
                    Stop;
                ELSE
                    CommunitationType := "TCP/UDP";
                    CommType := "TCP";
                ENDIF
                analyzingMode;
                IF JointCoordinateSignal = "JOINT" THEN
```

```
            analyzingJointMessage;
        ELSEIF JointCoordinateSignal = "COORDINATES" THEN
            analyzingCoordinateMessage;
        ENDIF
        IF coordinatesOutOfRange THEN
            SocketSend clientSocketTCP\Str:="COORDINATES_OUT_OF_RANGE";
        ELSEIF jointsOutOfRange THEN
            SocketSend clientSocketTCP\Str:="JOINTS_OUT_OF_RANGE";
        ENDIF
      ENDWHILE
    ENDIF

    ERROR
      IF ERRNO = ERR_SOCK_CLOSED THEN
        TPWrite "Connection closed";
        CommType := "TCP";
        CommunitationType := "";
        MoveSignal := FALSE;
      ENDIF
  ENDPROC

  PROC ListenUDP()
    IF CommType = "UDP" THEN
      createServerUDP;
      WHILE CommType = "UDP" DO
        SocketReceiveFrom serverSocketUDP\Str:=Message, IP_ADDRESS,
PORT\Time:=10;
        analyzingMessageCommType;
        IF CommType = "TCP" OR CommType = "EGM" THEN
          IF CommType = "TCP" THEN
            SocketClose clientSocketUDP;
          ENDIF
          RETURN;
        ELSEIF CommType = "STOP" THEN
          StopSignal := 1;
          Stop;
        ELSE
          CommunitationType := "TCP/UDP";
          CommType := "UDP";
        ENDIF
        analyzingMode;
        IF JointCoordinateSignal = "JOINT" THEN
          analyzingJointMessage;
        ELSEIF JointCoordinateSignal = "COORDINATES" THEN
          analyzingCoordinateMessage;
        ENDIF
        IF coordinatesOutOfRange THEN
          SocketSendTo clientSocketUDP, IP_ADDRESS,
PORT\Str:="COORDINATES_OUT_OF_RANGE";
        ELSEIF jointsOutOfRange THEN
```

```
                SocketSendTo clientSocketUDP, IP_ADDRESS,
PORT\Str:="JOINTS_OUT_OF_RANGE";
            ENDIF
          ENDWHILE
        ENDIF

        ERROR
          IF ERRNO = ERR_SOCK_CLOSED THEN
            TPWrite "Connection closed";
            CommType := "TCP";
            CommunitationType := "";
            MoveSignal := FALSE;
          ENDIF
    ENDPROC

    PROC ListenEGM()
      IF CommType = "EGM" THEN
        analyzingMode;
        CommunitationType := "EGM";
        CommType := "TCP";
      ENDIF
    ENDPROC

    PROC createServerTCP()
      SocketClose serverSocketTCP;
      SocketCreate serverSocketTCP;
      SocketBind serverSocketTCP,IP_ADDRESS,PORT;
      SocketListen serverSocketTCP;
    ENDPROC

    PROC createServerUDP()
      SocketClose serverSocketUDP;
      SocketCreate serverSocketUDP\UDP;
      SocketBind serverSocketUDP,IP_ADDRESS,PORT;
    ENDPROC

    PROC connectWithClientTCP()
      SocketClose clientSocketTCP;
      SocketAccept serverSocketTCP, clientSocketTCP\Time:=10000;
    ENDPROC

    PROC analyzingMessageCommType()
      semicolonsPositions{1} := StrFind(Message,1,";");
      CommType:=StrPart(Message,1,semicolonsPositions{1} - 1);
    ENDPROC

    PROC analyzingMode()
      semicolonsPositions{2} := StrFind(Message,semicolonsPositions{1} + 1,";");
      JointCoordinateSignal:=StrPart(Message,semicolonsPositions{1} +
1,semicolonsPositions{2} - semicolonsPositions{1} - 1);
```

```
    ENDPROC

    PROC analyzingJointMessage()
        semicolonsPositions{3} := StrFind(Message, semicolonsPositions{2} + 1, ";");
        semicolonsPositions{4} := StrFind(Message, semicolonsPositions{3} + 1, ";");
        semicolonsPositions{5} := StrFind(Message, semicolonsPositions{4} + 1, ";");
        semicolonsPositions{6} := StrFind(Message, semicolonsPositions{5} + 1, ";");
        semicolonsPositions{7} := StrFind(Message, semicolonsPositions{6} + 1, ";");
        semicolonsPositions{8} := StrFind(Message, semicolonsPositions{7} + 1, ";");
        analyzingStatus :=
StrToVal(StrPart(Message,semicolonsPositions{2}+1,semicolonsPositions{3}-
semicolonsPositions{2}-1),jointAngles{1});
        analyzingStatus :=
StrToVal(StrPart(Message,semicolonsPositions{3}+1,semicolonsPositions{4}-
semicolonsPositions{3}-1),jointAngles{2});
        analyzingStatus :=
StrToVal(StrPart(Message,semicolonsPositions{4}+1,semicolonsPositions{5}-
semicolonsPositions{4}-1),jointAngles{3});
        analyzingStatus :=
StrToVal(StrPart(Message,semicolonsPositions{5}+1,semicolonsPositions{6}-
semicolonsPositions{5}-1),jointAngles{4});
        analyzingStatus :=
StrToVal(StrPart(Message,semicolonsPositions{6}+1,semicolonsPositions{7}-
semicolonsPositions{6}-1),jointAngles{5});
        analyzingStatus :=
StrToVal(StrPart(Message,semicolonsPositions{7}+1,semicolonsPositions{8}-
semicolonsPositions{7}-1),jointAngles{6});
        Message := "";
        MoveSignal := TRUE;
        WaitUntil MoveSignal = FALSE;
    ENDPROC

    PROC analyzingCoordinateMessage()
        semicolonsPositions{3} := StrFind(Message, semicolonsPositions{2} + 1, ";");
        semicolonsPositions{4} := StrFind(Message, semicolonsPositions{3} + 1, ";");
        semicolonsPositions{5} := StrFind(Message, semicolonsPositions{4} + 1, ";");
        analyzingStatus :=
StrToVal(StrPart(Message,semicolonsPositions{2}+1,semicolonsPositions{3}-
semicolonsPositions{2}-1),coordinates{1});
        analyzingStatus :=
StrToVal(StrPart(Message,semicolonsPositions{3}+1,semicolonsPositions{4}-
semicolonsPositions{3}-1),coordinates{2});
        analyzingStatus :=
StrToVal(StrPart(Message,semicolonsPositions{4}+1,semicolonsPositions{5}-
semicolonsPositions{4}-1),coordinates{3});
        Message := "";
        MoveSignal := TRUE;
        WaitUntil MoveSignal = FALSE;
    ENDPROC
ENDMODULE
```