ROBOCHALLENGE

Interfejs PC robota przemysłowego

Spis treści:

1. Opis sytuacji

Dział utrzymania ruchu planuje zakup nowoczesnego systemu kontroli zrobotyzowanych stanowisk, do podglądu statusu pracy oraz alarmów w poszczególnych gniazdach produkcyjnych. Jako doświadczony programista robotów, proponujesz działowi UR, że sam stworzysz aplikację na komputer PC, do odczytu występujących statusów oraz alarmów na poszczególnych robotach firmy FANUC, wykorzystując bibliotekę Robot Interface. Dział Utrzymania Ruchu nie wierzy jednak, że taki interfejs może stworzyć programista robotów. Masz 40 minut na to, aby udowodnić im, że są w błędzie.

2. Opis zadania

Celem zadania jest uzyskanie połączenia z 4 stanowiskami zrobotyzowanymi, tworząc metody **ConnectRobot(), UpdateStatus(), ReadAlarmList()**, tak, aby pozwalały na poprawną komunikację ze stanowiskami i mogły pobierać dane statystyczne stanowisk. Zadanie jest podzielone na cztery etapy:

1. Uzyskanie połączenia z pojedynczym stanowiskiem zrobotyzowanym.
2. Pobranie danych stanu wykonywanego zadania z jednego stanowiska zrobotyzowanego.
3. Pobranie alarmów z jednego stanowiska zrobotyzowanego.
4. Rozbudowa dotychczasowo utworzonego kodu, tak by można było obsługiwać do czterech stanowisk zrobotyzowanych.

UWAGI

• W projekcie w poszczególnych oknach zaznaczono miejsca, gdzie należy wprowadzić własny kod programu. Nie należy modyfikować istniejącego kodu, jeżeli nie jest to wymagane.

```
/* USER CODE BEGIN */



/* USER CODE END */
```

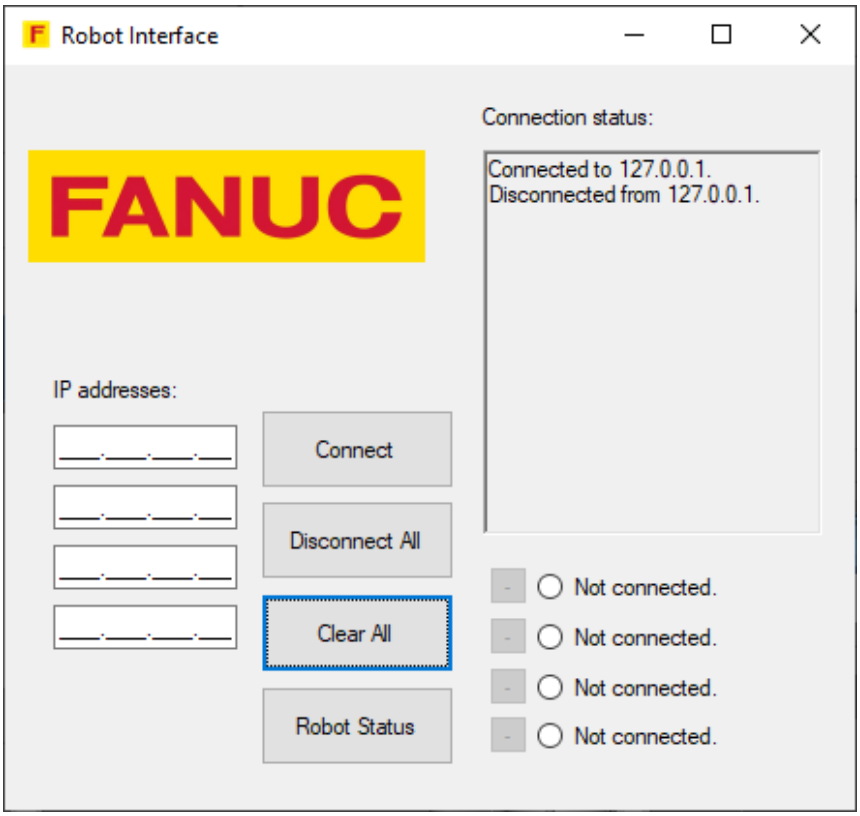
• Adresy IP wybranych robotów zostaną podane przez opiekuna stanowiska

3. Opis interfejsu aplikacji

Aplikacja złożona jest z 3 głównych okien. Pierwsze z nich, okno połączeń (MenuRC.cs) widoczne jest na rys. 1 i składa się z następujących elementów:

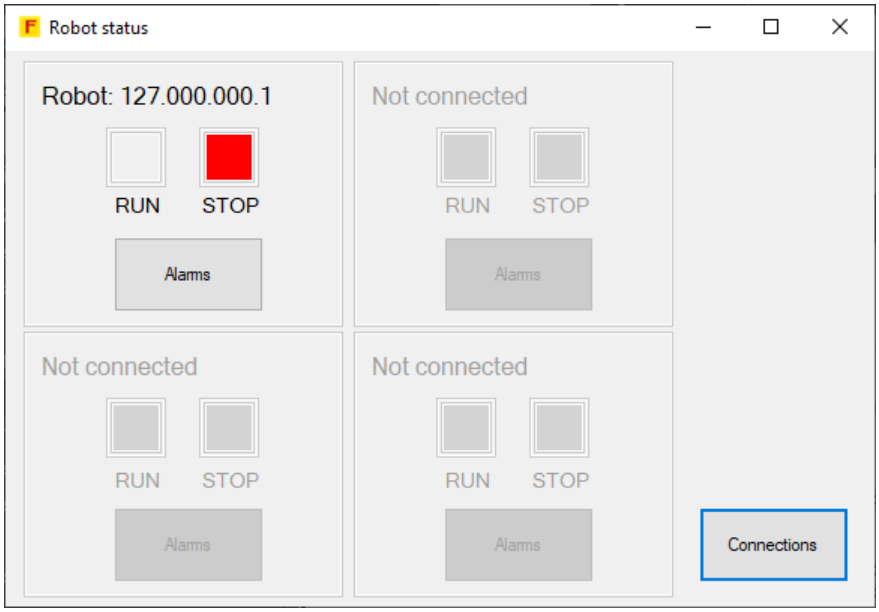
1. Pola **IP addresses** pozwalające na wpisywanie adresów IP stanowisk zrobotyzowanych, z którymi chcemy się połączyć.
2. Przycisk **Connect**, który łączy aplikację ze stanowiskami zrobotyzowanymi podanymi w polach tekstowych.
3. Przycisk **Disconnect All**, który rozłącza aplikację od wszystkich połączonych stanowisk zrobotyzowanych.
4. Przycisk **Clear All**, który czyści zawartość pól tekstowych.

5. Przycisk **Robot Status**, otwierający okno zawierające informacje o stanie pracy poszczególnych stanowisk (RobotStatus.cs).
6. Pole **Connection status**, dające użytkownikowi informację zwrotną dotyczącą połączenia ze stanowiskami zrobotyzowanymi.
7. Listę podłączonych stanowisk zrobotyzowanych.



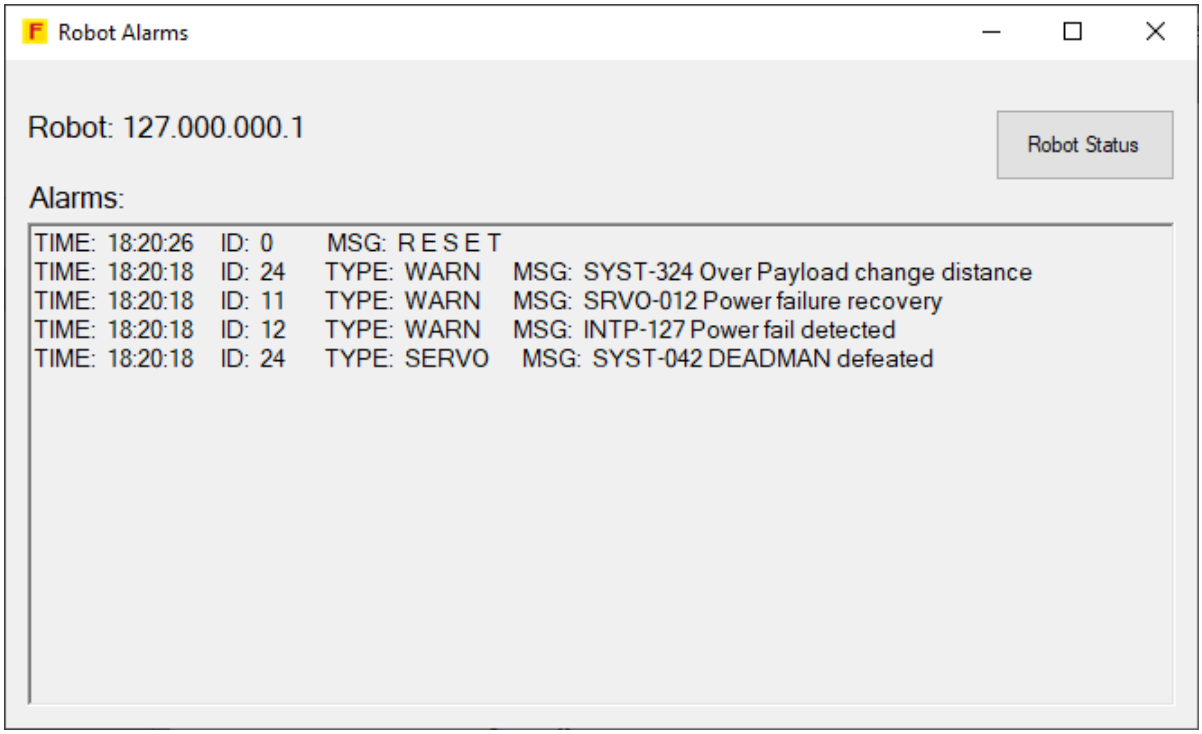
Rysunek 1 Okno połączeń (MenuRC.cs) pozwalające na łączenie i rozłączanie się z robotami oraz dające użytkownikowi informację zwrotną o połączeniu.

Przycisk **Robot Status** otwiera nowe osobne okno statusów (RobotStatus.cs) widoczne na Rys. 2. Widoczne są na nim 4 miejsca na stanowiska zrobotyzowane. Gdy dane slot jest niepodłączony – jego status jest niedostępne.



Rysunek 2 Okno przedstawiające stany wykonywanych przez stanowiska robocze programów (RobotStatus.cs).

Gdy utworzone zostanie połączenie ze stanowiskiem zrobotyzowanym odpowiednie miejsce (slot) stanie się aktywne. Stan wykonywania zadania przez dane stanowisko zrobotyzowane jest sygnalizowane poprzez zapalanie się lampek **RUN** oraz **STOP**. Gdy robot wykonuje program lampka RUN zapala się na zielono, gdy program zostanie zapauzowany lampka **RUN** zaczyna migać na żółto, natomiast zatrzymanie programu sygnalizowane jest zapaleniem się na czerwono lampki **STOP**. Dodatkowo dla każdego z połączonych stanowisk zrobotyzowanych możliwe jest otworzenie okna z alarmami (RobotAlarms.cs). Po jego otworzeniu w polu tekstowym zostaną wypisanie ostatnie oraz nowo pojawiające się alarmy.



Rysunek 3 Okno wyświetlające alarmy uzyskane ze stanowiska zrobotyzowanego. Nowo pojawiające się alarmy są dopisywane od góry listy.

Stanowiska zrobotyzowane są dołączane do pierwszego dostępnego miejsca. Kiedy nie ma żadnego wolnego miejsca nie można dołączyć kolejnych stanowisk zrobotyzowanych.

4. Metody wykorzystywane do komunikacji ze stanowiskiem zrobotyzowanym

W celu komunikacji z robotem stosowane są metody **Connect(), Disconnect()** w celu łączenia lub rozłączania ze stanowiskiem zrobotyzowanym oraz **TimeOutValue()** w celu ustawienia wartości czasu próby połączenia ze stanowiskiem.

C#	bool Connect(string *HostName*)

Specify robot host name (or IP address) as argument.
You need to initialize DataTable before calling this method.

Success returns True, fails returns False.

Rysunek 4 Metoda Connect() służąca do łączenia się ze stanowiskiem zrobotyzowanym

C#	bool Disconnect()

Success returns True, fails returns False.

(*) When disconnected, please delete all FRRJIF objects. If you need to connect again, please re-create objects.

Rysunek 5 Metoda Disconnect() służąca do rozłączania się ze stanowiskiem zrobotyzowanym

C#	int get_TimeOutValue() void set_TimeOutValue(int newValue) or TimeOutValue property

This is time out value for communication error. Default value is 10000 (ms). Minimum value is 100. If you apply small time out value, then you may encounter time out often. The time out value is discard at deleting Core object. Please specify time out again when Core object is created.

(*) When timeout, please disconnect and delete all FRRJIF objects. If you need to connect again, please re-create objects.

Rysunek 6 Metody get_TimeOutValue() i set_TimeOutValue() pozwalające na pobranie i ustawienie wartości parametru

W celu pobrania statusu wykonywania programu oraz alarmów stosowana jest metoda **GetValue()**. Dodatkowo stosowana jest metoda Refresh() w celu aktualizacji tabel danych.

C#	bool GetValue(ref string ProgName, ref short LineNumber, ref short State, ref string ParentProgName, System.Text.Encoding encode)

Argument ProgName will have returned current executing program name.
Argument LineNumber will have returned current executing program line number.
Argument State will have returned execution status. When program stopped, State is 0. When program paused, State is 1. When program executing, State is 2.
Argument ParentProgName will have returned program name that is executed first. If no child program is called, value of ParentProgName is the same as value of ProgName.

(*) DataType for FRRJIF.DataTable.AddTask affects target current executing program. Please refer FRRJIF.DataTable.AddTask.

Success returns True, fails returns False.

Rysunek 7 Metoda GetValue() pozwalająca na pobranie nazwy wykonywanego programu, obecnie wykonywanej linii, stanu wykonywania programu oraz nazwy programu rodzica

C#	*bool **GetValue**(int Count, ref short AlarmID, ref short AlarmNumber, ref short CauseAlarmID, ref short CauseAlarmNumber, ref short Severity, ref short Year, ref short Month, ref short Day, ref short Hour, ref short Minute, ref short Second, ref string AlarmMessage, ref string CauseAlarmMessage, ref string SeverityMessage)*

Specify argument Count as index of target alarm history item. (Specify 1 for the first item.) Argument AlarmID will have returned alarm ID. In case of 'SRVO-001', AlarmID is 11 that represents 'SRVO'. Please see alarm code table in R-J3 reference. If there is no active alarm, AlarmID is zero for active alarm reference.

Argument AlarmNumber will have returned alarm number. In case of 'SRVO-001', AlarmNumber is 1. If there is no active alarm, AlarmNumber is zero for active alarm reference.

Argument CauseAlarmID will have returned cause alarm ID. Some alarm have two alarm messages. The second alarm is cause code. This argument is to read cause code. If there is no cause code, CauseAlarmID is 0.

Argument CauseAlarmNumber will have returned cause alarm Number. This argument is to read cause code alarm number. If there is no cause code, CauseAlarmNumber is 0.

Argument Severity will have return alarm severity. Severity value means as follows:

NONE	128
WARN	0
PAUSE.L	2
PAUSE.G	34
STOP.L	6
STOP.G	38
SERVO	54
ABORT.L	11
ABORT.G	43
SERVO2	58
SYSTEM	123

Argument Year, Month, Day, Hour, Minute, Second will have returned alarm occurred date and time (24 hours format).

Argument AlarmMessage will have returned alarm message. The message is the top line to teach pendant screen includes alarm code like 'SRVO-001'. (Kanji message not supported.)

Argument CauseAlarmMessage will have returned cause code alarm message. (Kanji message not supported)

Argument SeverityMessage will have returned alarm severity string like 'WARN'.

Success returns True, fails returns False.

Rysunek 8 Metoda GetValue() pozwalająca na uzyskanie danych alarmu z pozycji Count na liście alarmów

C#	bool Refresh()

When Refresh method is called, data of DataTable object are refreshed. DataTable object keeps the value until next Refresh method calling. If you want to get the latest robot data, you need to call this method. If you do not call this method, data on data table are not changed.

Success returns True, fails returns False.

Remark

(*) You must call data register methods like AddCurPos before connecting. You must not call data register methods after connecting. If you need to change data table, then you need to disconnect, delete all FRRJIF objects and create data table again.

Rysunek 9 Metoda Refresh() stosowana w celu aktualizacji tablic danych

5. Połączenie ze stanowiskiem zrobotyzowanym

W celu połączenia się ze stanowiskiem zrobotyzowanym należy przygotować metodę **ConnectRobot()**, znajdującą się w oknie **MenuRC.cs** tak, aby przygotowywała niezbędne dane przed połączeniem ze stanowiskiem zrobotyzowanym, a dopiero następnie próbowała się z nim łączyć. W tym celu wewnątrz bloku **try** należy zainicjować nowy obiekt typu **FRRJIf.Core** korzystając z konstruktora **public Core(Encoding encoding)**. Odwołujemy się do zerowego elementu tablicy, ponieważ podczas realizacji pierwszych trzech zadań tworzone metody będą obsługiwać tylko jedno z czterech miejsc na połączone stanowiska zrobotyzowane.

```
mobjCore[0] = new FRRJIf.Core(encode[0]);
```

Przed połączeniem się ze stanowiskiem zrobotyzowanym należy przygotować tablice danych. Służy do tego metoda **PrepareDataTables()**. Nie należy zmieniać kodu wewnątrz tej metody.

```
private void PrepareDataTables(int slot)
```

Na ten moment połączenie jest realizowane tylko z jednym stanowiskiem metodę należy wywołać z parametrem **slot** równym 0.

Kolejnym krokiem jest ustawienie wartości **time out**. W tym celu trzeba zadeklarować zmienną pomocniczą **int timeOut**

Następnie wewnątrz bloku **try** należy ustawić wartość tego parametru poprzez zastosowanie metody **Interaction.GetSetting(),** oraz zmianę uzyskanego w ten sposób typu danych na liczbę całkowitą (**int**).

```
Convert.ToInt32(Interaction.GetSetting(cnstApp, cnstSection, "TimeOut", "-1"));
```

Jeżeli uzyskany czas jest poprawny (czyli większy od zera) możemy ustawić go dla pierwszego miejsca na stanowisko zrobotyzowane.

```
if (timeOut > 0)
 {
    mobjCore[0].set_TimeOutValue(timeOut);
}
```

Po zakończeniu przygotowań możemy połączyć się ze stanowiskiem zrobotyzowanym za pomocą metody widocznej na rys. 4 Connect(). Ponieważ chcemy łączyć się w pierwszym z dostępnych miejsc wywołanie tej metody będzie wyglądać następująco:

```
mobjCore[0].Connect(IP);
```

6. Pobieranie stanu wykonywanego zadania

Pobieranie informacji o stanie wykonywanego sprowadza się do sprawdzenia kilku warunków dotyczących połączenia ze stanowiskiem zrobotyzowanym, odświeżenia tablic danych oraz pobrania szukanych wartości. Metoda UpdateStatus(), którą należy uzupełnić znajduje się w pliku **RobotStatus.cs**.

UWAGA W tej metodzie będziemy odwoływać się do publicznych, statycznych elementów członkowskich klasy **MenuRC**, dlatego przy odwoływaniu się do zmiennych oraz metod należących do tej klasy trzeba określać zarówno zmienną jak i klasę. Np. jeżeli chcemy odwołać się do elementu tablicy isConnected[0] zrobimy to wywołując **MenuRC.isConnected[0]**.

Informacje o połączeniu przechowywane są wewnątrz tablicy **public static bool[] isConnected**. W tej części zadania chcemy sprawdzić czy istnieje połączenie w pierwszym miejscu (w polu tablicy o indeksie 0). Po sprawdzeniu istnienia połączenia wymagane jest odświeżenie danych za pomocą metody **Refresh()**.

```
MenuRC.mobjDataTable[0].Refresh();
```

Brak odświeżenia danych przed próbą ich pobrania zakończy się błędem podczas wykonywania metody **GetValue()**.

Następnie należy spróbować pobrać wartości stanu wykonywania zadania za pomocą metody

```
public bool GetValue(ref string ProgName, ref short LineNumber, ref short State, ref string ParentProgName, Encoding encode);
```

Pierwsze cztery argumenty należy podawać do metody ze słowem kluczowym ref. Ostatnim argumentem powinno być MenuRC.encode[0]. Zmienne niezbędne do przechowywania danych zostały zadeklarowane globalnie.

```
short[] intLine = { 0, 0, 0, 0 };
short[] intState = { 0, 0, 0, 0 };
string[] strParentProgName = { "", "", "", "" };
string[] strProgName = { "", "", "", "" };
```

Na koniec w celu aktualizowania danych podczas działania programu należy dodać wywołanie metody **UpdateStatus()** do wydarzenia **timerStatus_Tick()**.

7. Pobieranie alarmu z listy

Ostatnim zadaniem jest uzupełnienie metody **ReadAlarm()** znajdującej się w pliku **RobotAlarms.cs**. Pozwala ona na pobranie alarmów z listy i wyświetlanie ich w polu tekstowym okna. Metoda dzięki której pobieramy alarm z listy wymaga od nas podania pozycji alarmu na liście, jednak nowe alarmy wpisywane są zawsze na pozycję 1, a pozostałe są przesuwane niżej. Oznacza to, że w celu wypisywania nowych alarmów będziemy musieli pobierać najnowszy alarm z listy i sprawdzać, czy różni się on od poprzedniego.

UWAGA W tej metodzie będziemy odwoływać się do publicznych, statycznych elementów członkowskich klasy **MenuRC**, dlatego przy odwoływaniu się do zmiennych oraz metod należących do tej klasy trzeba określać zarówno zmienną jak i klasę. Np. jeżeli chcemy odwołać się do elementu tablicy isConnected[0] zrobimy to wywołując **MenuRC.isConnected[0]**.

Przed próbą pobrania alarmu z listy trzeba przygotować kilka zmiennych lokalnych i globalnych. 14 zmiennych lokalnych posłuży do odbierania danych oraz wypisywania wiadomości, a 6 globalnych do sprawdzania czy alarm jest nowy.

Wewnątrz metody ReadAlarm() należy przygotować 11 zmiennych typu **short** , które przechowywać będą następujące informacje:

- ID alarmu,
- numer alarmu,
- ID przyczyny alarmu,
- numer przyczyny alarmu,
- stopień alarmu,
- rok,
- miesiąc,
- dzień,
- godzinę,
- minutę,
- sekundę.

Dodatkowo potrzebne są 3 zmienne typu **string** , które przechowywać będą: wiadomość alarmu, wiadomość o przyczynie alarmu oraz typ alarmu.

W celu pobrania alarmu z pozycji Count na liście alarmów należy skorzystać z metody

public bool GetValue(int Count, ref short AlarmID, ref short AlarmNumber, ref short CauseAlarmID, ref short CauseAlarmNumber, ref short Severity, ref short Year, ref short Month, ref short Day, ref short Hour, ref short Minute, ref short Second, ref string AlarmMessage, ref string CauseAlarmMessage, ref string SeverityMessage, Encoding encoding);

Parametr encoding podajemy tak samo jak w poprzednim zadaniu.

Alarm uznajemy za nowy jeżeli różni się on od poprzedniego co najmniej jednym z 6 parametrów: wiadomością alarmu, typem alarmu, ID alarmy, godziną, minutą lub sekundą. W tym celu należy utworzyć 6 zmiennych globalnych przechowujących parametry poprzedniego alarmu (ID, wiadomość, typ, godzinę, minutę, sekundę).

Jeżeli alarm jest uznany za nowy możemy wypisać go do okna. W tym celu skorzystamy z metody PrintAlarms().

private void PrintAlarms(int intID, int intHour, int intMin, int intSec, string strMessage, string strCauseMessage, string strType)

Po wypisaniu alarmu należy zapamiętać wypisany alarm, żeby nie był on wypisywany ponownie.

Ostatnim elementem jest wywołanie metody **ReadAlarm()** wewnątrz wydarzenia timerAlarm_Tick(). Jako parametry należy podać **ref MenuRC.mobjAlarm[robotIndex]** oraz 1.

UWAGA W tym momencie aplikacja powinna działać poprawnie tylko dla jednego połączonego stanowiska zrobotyzowanego. Próby połączenia z dodatkowymi stanowiskami zrobotyzowanymi będą prowadziły do zawieszenia działania aplikacji.

8. Obsługa do czterech stanowisk zrobotyzowanych jednocześnie

Do metody **ConnectRobot()** należy dodać nowy parametr **int** slot, który będzie przedstawiał jedno z miejsc na stanowiska zrobotyzowane. Następnie wewnątrz tej metody musimy zamienić wszystkie odwołania do elementów tablic z [0] na [slot]. Na koniec musimy poprawić wywołania metod wewnątrz pliku poprzez dodanie do nich brakującego parametru i. Miejsca, w których występuje metoda **ConnectRobot()** możemy znaleźć przy pomocy wyszukiwania za pomocą wciśnięcia i przytrzymania klawiszy **ctrl** i **f** oraz wpisania jako szukanego tekstu **ConnectRobot**. Należy pamiętać o kolejności podawanych parametrów.

Operacje te należy powtórzyć dla metody **ReadAlarm()**, z tą różnicą, że jako parametr do wywoływania metody należy dodać trzeci argument **robotIndex**.

W przypadku metody **UpdateStatus()** nie jest konieczne dodawanie parametru do definicji metody. W zamian jednak musimy dodać wewnątrz metody pętlę **for**, która obejmie całość dotychczasowo utworzonej metody oraz zamienić odwołania do elementów tablic z [0] na [slot].

```
for (int slot = 0; slot < 4; slot++)
{
...
}
```

9. System oceniania

1. Realizacja celu głównego – 1 pkt

Celem zadania jest uzyskanie połączenia z 4 wybranymi stanowiskami zrobotyzowanymi.

2. Ukończenie poszczególnych etapów – łącznie 4 pkt

- Poprawne łączenie się z jednym stanowiskiem zrobotyzowanym.
- Poprawne pobieranie statusu wykonywanego zadania ze stanowiska zrobotyzowanego.
- Poprawne odczytywanie listy alarmów ze stanowiska zrobotyzowanego.
- Obsługa wielu stanowisk zrobotyzowanych jednocześnie.

3. Czas wykonania po zapoznaniu się z instrukcją – maksymalnie 3 pkt

- Ukończenie zadania poniżej 30 min (3 pkt)
- Ukończenie zadania poniżej 35 min (2 pkt)
- Ukończenie zadania poniżej 40 min (1 pkt)