Pomiary Automatyka Robotyka, ISSN 1427-9126, R. 24, Nr 2/2020, 77–90, DOI: 10.14313/PAR_236/77

Monitorowanie stanu robota przemysłowego za pomocą aplikacji MFC

Jacek Dunaj, Kamil Bojanek
Sieć Badawcza Łukasiewicz – Przemysłowy Instytut Automatyki i Pomiarów PIAP, Al. Jerozolimskie 202, 02-486 Warszawa

Streszczenie: W artykule przedstawiono sposób realizacji transmisji sieciowej między komputerem PC a robotem przemysłowym. Aplikacje komputera PC zrealizowano w języku C++ z wykorzystaniem bibliotek MFC. Przedstawiono, jak w praktyczny sposób wykorzystać klasę CAsyncSocket do programowania transmisji między komputerem a innym urządzeniem, np. robotem przemysłowym. Druga część artykułu koncentruje się na możliwościach programowych robota w zakresie udostępniania informacji dotyczących jego stanu (m.in. położenia osi, aktualnego wyboru układu współrzędnych i narzędzia). Na przykładzie robota firmy KUKA omówiono sposób odczytywania informacji, kodowania, a następnie transmitowana do współpracującego komputera PC. Przedstawiono także sposób działania i opis dwóch przykładowych aplikacji komputera PC do testowania transmisji sieciowej oraz prezentacji danych odbieranych z robota.

Słowa kluczowe: aplikacja MFC, transmisja siecią Ethernet, biblioteki MFC, klasa CAsyncSocket, robot przemysłowy, Microsoft Visual Studio, KUKA WorkVisual

1. Wprowadzenie

Jednym z zadań stawianych współcześnie przed aplikacjami przemysłowymi jest integracja systemów i sprzężenie maszyny z Internetem. Umożliwia to śledzenie procesu technologicznego oraz generowanie różnych komunikatów prezentowanych na urządzeniach oddalonych takich jak laptopy, smartfony czy sprzęcie wirtualnej rzeczywistości. Na poziomie pojedynczego stanowiska robotowego zadanie to obejmuje m.in. monitorowanie stanów robota oraz informacji generowanej przez jego aplikację. Pod pojęciem stanów robota należy tutaj rozumieć wszystko to, co udostępnia jego oprogramowanie systemowe, a więc w jakim trybie pracy znajduje się robot, jakie są położenia jego osi, jakie są wartości dostępnych zmiennych systemowych itp.

Przemysłowy Instytut Automatyki i Pomiarów PIAP w swoich wdrożeniach przemysłowych wykorzystywał roboty m.in. firmy KUKA i ABB. Wykonywały one czynności bezpośrednio związane z procesami produkcyjnymi takimi jak paletyzacja czy spawanie, ale nie były sprzężone z urządzeniami zewnętrznymi umożliwiającymi prezentację odbieranej informacji. Ich języki programowania zawierają instrukcje umożliwiające transmisję danych siecią Ethernet z wykorzystaniem protokołu TCP/IP. Jednak utrudnieniem pozostaje to, że oprogramowanie realizujące transmisję informacji o stanie robota musi być wykonywane jako rodzaj tzw. „wątku", a więc niezależnie od aktualnego trybu pracy robota (tryb testowy, tryb pracy automatycznej) i nie może ingerować w pracę jego dowolnej aplikacji.

Osobnym zagadnieniem pozostaje, gdzie i w jaki sposób informacja odbierana z robota ma być interpretowana i prezentowana. Praktycznym rozwiązaniem jest wykorzystanie komputera PC z systemem Windows. Jest to uzasadnione tym, że aplikacje przeznaczone dla tego systemu mogą być bardzo rozbudowane pod względem funkcjonalnym, wykorzystując elementy grafiki, zapisu do plików, baz danych i arkuszy kalkulacyjnych, tworzenie statystyk, a także że oprogramowanie narzędziowe do tworzenia aplikacji zawiera wiele gotowych podprogramów bibliotecznych.

Podstawowym zagadnieniem pozostaje jednak realizacja samej transmisji informacji. W powszechnym użyciu są różne aplikacje komputerowe pracujące pod kontrolą systemów operacyjnych Windows, które do wymiany danych wykorzystują sieć Ethernet i protokół TCP/IP. Są to jednak gotowe aplikacje, których nie można modyfikować pod konkretne potrzeby ani wykorzystywać fragmentów ich kodu źródłowego do tworzenia własnych opracowań, ponieważ kod ten z reguły nie jest dostępny, albo pozostaje objęty prawami autorskimi. W ramach prac rozwojowych powadzonych w PIAP mających na celu m.in. poszerzenie oferty na funkcjonalność potencjalnych aplikacji przemysłowych opracowano własne rozwiązania programistyczne, kodowane w języku wysokiego poziomu C/C++. Pozwalają one na budowanie oprogramowania komunikującego się z innymi urządzeniami, w tym z robotami przemysłowymi, za pośrednictwem sieci Ethernet. Za pomocą tych rozwiązań można tworzyć aplikacje, które nie tylko są w stanie transmitować informacje do/z tych urządzeń, ale mogą wykorzystywać możliwości funkcjonalne systemu operacyjnego, pod kontrolą którego pracują.

Autor korespondujący:
Jacek Dunaj, jacek.dunaj@piap.lukasiewicz.gov.pl

Artykuł recenzowany
nadesłany 18.05.2020 r., przyjęty do druku 26.06.2020 r.

W artykule tym opisano w jaki sposób zrealizowano oprogramowanie komunikacyjne z wykorzystaniem dostępnych bibliotecznych funkcji MFC (ang. *Microsoft Foundation Class Library*) [1]. Przedstawiono przykład uniwersalnej aplikacji komputera PC pracującego pod kontrolą systemu operacyjnego Windows (XP, 7, 10) umożliwiającej transmisję informacji między komputerem a innym urządzeniem zewnętrznym (drugi komputer, robot) z wykorzystaniem sieci Ethernet i protokołu TCP/IP. W dalszej części artykułu pokazano, co i w jaki sposób robot przemysłowy, dostępny w PIAP, może udostępniać dla urządzenia zewnętrznego w zakresie danych dotyczącej stanu jego pracy, położenia poszczególnych osi i komunikatów generowanych przez aplikację. Końcowa część artykułu przedstawia kolejną aplikację dla komputera PC, bezpośrednio współpracującą z oprogramowaniem robota w zakresie odbioru, interpretacji i czytelnej prezentacji informacji odbieranej z niego. Przedstawione oprogramowanie dla komputera PC napisano w języku C++ i uruchomiono w środowisku Microsoft Visual Studio. Aplikację robota zrealizowano w języku KRL (ang. *Kuka Robot Language*) i uruchomiono na robocie za pomocą oprogramowania KUKA Work Visual.

2. Klasy w Windows

System operacyjny Windows zawiera wbudowany interfejs programistyczny WinAPI (ang. *Windows Application Programming Interface*), który jest zbiorem standardowych stałych, zmiennych i podprogramów bibliotecznych umożliwiających wykonywanie aplikacji w środowisku tego systemu. Z punktu widzenia programisty wymienione elementy można podzielić na grupy w zależności od funkcji, jakie realizują (np. grupa funkcji do tworzenia i obsługi okna dialogowego, grupa funkcji do obsługi plików). W języku C++ takie grupy funkcji określa się terminem klasa. Każda klasa zdefiniowana w MFC zawiera ściśle określoną liczbę elementów, których nie można modyfikować ani dopisywać do klasy nowych elementów. Język C++ umożliwia na podstawie danej klasy definiować klasy potomne, które „dziedziczą" funkcjonalność klasy źródłowej i do których można dopisywać kolejne elementy. Tę właściwość wykorzystano do tworzenia własnego oprogramowania komunikacyjnego komputer – robot.

3. Komunikacja sieciowa

Komunikacja sieciowa polega na połączeniu dwóch lub więcej urządzeń (komputera, sterownika itp.) za pomocą sieci w celu wymiany informacji. Każdy z końców takiej sieciowej konwersacji nazywany jest gniazdem (ang. *socket*) lub punktem końcowym. Ogólne zasady tworzenia gniazd i ich wykorzystywania do komunikacji sieciowej opisano m.in. w [2]. Aby przez interfejs gniazda mogło dojść do komunikacji sieciowej, a więc do wymiany informacji między dwoma gniazdami wymagane jest sprzężenie lokalnego gniazda z gniazdem współpracującym. W tym celu tworząc aplikację należy najpierw zadeklarować klasę do obsługi gniazda, a następnie zdefiniować to gniazdo za pomocą funkcji **`Create()`** będącej jednym z elementów klasy obsługi gniazda. Definiowanie gniazda polega na wyszczególnieniu kilku parametrów opisujących jego właściwości i uzyskania od oprogramowania sieciowego identyfikatora wskazanego gniazdo. W kolejnym kroku, jeśli wymiana informacji odbywać się będzie w architekturze klient – serwer, to:

- jeśli aplikacja ma działać jako serwer to ustawia swoje lokalne gniazdo w tryb nasłuchu żądania nawiązania połączenia przez klienta wykorzystując funkcję **`Listen()`**,
- jeśli aplikacja ma działać jako **`klient`** to wysyła ze swojego lokalnego gniazda do gniazda współpracującego żądanie nawiązania połączenia wykorzystując funkcję **`Connect()`**.

Inicjatywa nawiązania połączenia wychodzi zawsze od klienta. Wysyłanie informacji przy pomocy funkcji typu **`Send()`** i odbiór informacji przy pomocy funkcji typu **`Receive()`** nie zależą już od tego czy aplikacja działa jako klient czy jako serwer, ponieważ oba punkty konwersacji sieciowej mogą nadawać i odbierać niezależnie od siebie. Podobnie połączenie może być zerwane zarówno przez gniazdo pracujące jako serwer lub jako klient. Wymienione funkcje: **`Connect()`**, **`Listen()`**, **`Send()`** oraz **`Receive()`** są elementami klasy obsługi gniazda. Tak wygląda teoria, w praktyce realizacja tego schematu jest bardziej złożona.

4. Klasa CAsyncSocket i klasa potomna CMyAsyncSocket

Opracowując własne rozwiązania dla komputera PC do komunikacji sieciowej z innymi urządzeniami, w tym z robotami przemysłowymi, zdecydowano się wykorzystać klasę **`CAsyncSocket`** dostępną w bibliotekach MFC [1]. Jest to klasa umożliwiająca dostęp do funkcji sieciowych na bardzo niskim poziomie, skutkujących m.in. wygodą wywołań zwrotnych do powiadamiania o zdarzeniach sieciowych. Dobrze się więc nadaje do tworzenia oprogramowania testowego, szczególnie tam, gdzie z góry nie wiadomo, z jakimi problemami przyjdzie się zmierzyć. Oczywiście możliwości funkcjonalne klasy **`CAsyncSocket`** znacznie przekraczają to co jest potrzebne do zbudowania komunikacji komputer – robot.

Tworząc jakiekolwiek oprogramowanie komunikacyjne należy pamiętać, że nie może się ono ograniczać tylko do funkcji typu „odbierz informację i zapisz ją do bufora odbiorczego" lub „wpisz informację do bufora nadawczego i wyślij ją". Oprogramowanie takie musi także zawierać elementy pozwalające sprawdzić czy transmisja z/do urządzenia współpracującego jest w danym momencie możliwa. Klasa **`CAsyncSocket`** zawiera sześć funkcji bibliotecznych, wywoływanych przez oprogramowanie systemowe (ang. *framework*, znaczenie tzw. bitów maski będzie wyjaśnione w dalszej części artykułu):

1. **`virtual void OnAccept (int ErrorCode)`**;
   – funkcja wywoływana jako powiadomienie, że gniazdo współpracujące odpowiedziało na żądanie nawiązania połączenia. Z funkcją związany jest bit maski **`FD_ACCEPT`**.
2. **`virtual void OnClose (int ErrorCode)`**;
   – funkcja wywoływana jako powiadomienie, że gniazdo współpracujące zamknęło połączenie. Z funkcją związany jest bit maski **`FD_CLOSE`**.
3. **`virtual void OnConnect (int ErrorCode)`**;
   – funkcja wywoływana jako powiadomienie, że gniazdo współpracujące odpowiedziało na wysłane żądanie nawiązania połączenia bez względu na to, czy zostało ono zaakceptowane, czy odrzuciło to żądanie. Z funkcją związany jest bit maski **`FD_CONNECT`**.
4. **`virtual void OnOutOfBandData (int ErrorCode)`**;
   – funkcja wywoływana jako powiadomienie, że gniazdo współpracujące wysłało tzw. informację pozapasmową (ang. out-of-band data). Z funkcją związany jest bit maski **`FD_OOB`**.
5. **`virtual void OnReceive (int ErrorCode)`**;
   – funkcja wywoływana jako powiadomienie, że odebrano informację z gniazda współpracującego. Z funkcją związany jest bit maski **`FD_READ`**.
6. **`virtual void OnSend (int ErrorCode)`**;
   – funkcja wywoływana jako powiadomienie, że możliwe jest wysłanie informacji do gniazda współpracującego. Z funkcją związany jest bit maski **`FD_WRITE`**.

Wymienione funkcje są uruchamiane przez oprogramowanie systemowe z parametrem ***ErrorCode*** określającym kod potencjalnego błędu. Jeśli ***ErrorCode = 0***, to oznacza, że nie stwierdzono błędu podczas rejestracji zdarzenia, którego efektem jest wywołanie funkcji. Taka reakcja na wystąpienie zdarzenia powoduje, że w aplikacji nie można umieścić instrukcji wywołania każdej z tych funkcji w celu sprawdzenia, czy zdarzenie z nią związane rzeczywiście wystąpiło. Żadna z nich nie zwraca bowiem wartości (wszystkie są typu ***void***), nie ma także bezpośredniej możliwości dopisania fragmentów kodu do każdej z nich (np. instrukcji ustalającej wartość dodatkowej flagi). Niedogodność tę można jednak ominąć wykorzystując jedną z możliwości jakie oferuje język C++, poprzez zdefiniowanie klasy potomnej do klasy **CAsyncSocket**.

W środowisku Microsoft Visual Studio dostępne jest narzędzie **Class Wizzard** do tworzenia klas potomnych. Zdefiniowanie przy jego pomocy nowej klasy wymaga określenia jej nazwy (w przypadku aplikacji opisanych w dalszej części tego artykułu użyto nazwy **CMyAsyncSocket**) oraz wskazania bazowej klasy MFC jaką jest **CAsyncSocket**. W wyniku tego narzędzie **Class Wizzard** utworzy i dopisze do aktualnego projektu Visual Studio dwa nowe pliki: **CMyAsyncSocket.cpp** oraz **CMyAsyncSocket.h** zawierające odpowiednio definicję (.cpp) oraz deklarację (.h) nowej klasy. Nazwy wspomnianych plików mogą być dowolne, jednak program domyślnie tworzy je na podstawie nazwy klasy. Następnie w pliku **CMyAsyncSocket.cpp** trzeba dopisać kody „własnych" funkcji **OnAccept()**, ..., **OnSend()** (poniżej przykład funkcji **OnAccept()**):

```
void CMyAsyncSocket::OnAccept
(int nErrorCode)
{
// TUTAJ NALEŻY UMIESCIC WLASNY KOD
// WYKONYWANY PO WYWOLANIU PRZEZ FRAMEWORK
// FUNKCJI OnAccept()
CAsyncSocket::OnAccept (nErrorCode);
}
```

a w pliku **CMyAsyncSocket.h** umieścić deklarację prototypu tej funkcji:

```
void OnAccept (int);
```

4.1. Sygnalizacja błędów w klasach CAsyncSocket i CMyAsyncSocket

Część funkcji zdefiniowanych w klasach **CAsyncSocket** i **CMyAsyncSocket** zwraca wartości typu **BOOL**. Jeśli wartość zwracana przez funkcję jest różna od zera (**TRUE**) to oznacza, że jej wykonanie zakończyło się sukcesem, jeśli zwracaną wartością jest **FALSE**, to oznacza błąd wykonania. W celu zidentyfikowania błędu, bezpośrednio po wywołaniu procedury która zwróciła wartość **FALSE**, należy wywołać bezparametrową funkcję **GetLastError()**. Funkcja ta, także będącą elementem klas **CAsyncSocket** i **CMyAsyncSocket**, zwraca kod błędu. Trzeba jednak pamiętać, że nie wszystkie błędy są błędami krytycznymi, wymagającymi dodatkowej obsługi, ponieważ mogą oznaczać, że w danym momencie nie ma warunków do poprawnego wykonania danej procedury, ale jej kolejne wywołanie może zakończyć się sukcesem.

4.2. Tworzenie punktu końcowego (gniazda) przez aplikację

Po zdefiniowaniu klasy **CMyAsyncSocket** można utworzyć gniazdo lokalne (komputer) i gniazdo oddalone (np. robot) deklarując dwa obiekty:

```
CMyAsyncSocket AsyncLocalSocket;
// gniazdo lokalne
CMyAsyncSocket AsyncRemoteSocket;
// gniazdo oddalone
```

Bez względu na to, czy aplikacja ma pracować jako serwer czy jako klient, następnym krokiem jest zdefiniowanie właściwości lokalnego punktu końcowego (gniazda). Do tego celu służy 4-parametrowa funkcja **Create(..)** obiektu **AsyncLocalSocket**:

```
BOOL BoolVar;
BoolVar = AsyncLocalSocket.Create
(SockPort, SockType, Event,SockAddress);
```

Parametr **SockPort** określa numer portu, parametr **SockAddress** jest ciągiem znakowym określającym adres sieciowy gniazda w formacie „kropkowanym" (np. 128.53.22.14). Jeśli do komunikacji sieciowej ma zostać użyty protokół TCP, to parametr **SockType** musi mieć wartość stałej **SOCK_STREAM** zdefiniowanej w jednym z plików nagłówkowych .h. Parametr **Event** jest sumą logiczną od 0 do 6 bitów maski: **FD_ACCEPT**, **FD_CLOSE**. **FD_CONNECT**. **FD_OOB**. **FD_READ**. **FD_WRITE**. Jeśli dany bit maski wystąpi jako składnik sumy logicznej **Event**, to związana z nim funkcja **OnAccept(), ..., OnSend()** klasy **CMyAsyncSocket** zostanie automatycznie uruchomiona przez oprogramowanie systemowe po zarejestrowaniu zdarzenia odpowiadającej tej funkcji.

Dalsze postępowanie związane z wymianą informacji z urządzeniem współpracującym jest uzależnione od tego, czy aplikacja (lokalny punkt końcowy) ma pracować jako serwer czy jako klient.

4.3. Nawiązanie połączenia gniazda lokalnego z gniazdem oddalonym

Z inicjatywą nawiązania połączenia między dwoma punktami końcowymi sieciowej transmisji występuje zawsze klient. Aby połączenie mogło zostać zrealizowane aplikacja działająca jako serwer musi pozostawać w stanie nasłuchu, a następnie zaakceptować żądanie połączenia od klienta. Ustawienie aplikacji pracującej jako serwer w stan nasłuchu realizuje funkcja **Listen()** obiektu **AsyncLocalSocket** wywoływana instrukcją:

```
BOOL BoolVar;
BoolVar = AsyncLocalSocket.Listen
      (int ConnectionBack);
```

W jej wyniku gniazdo lokalne (punkt końcowy) jest przełączane w tryb, w którym połączenia przychodzące są potwierdzane i ustawiane w kolejce do czasu akceptacji przez proces. Parametr ***ConnectionBack*** określa iłe oczekujących połączeń może być przechowywanych w kolejce oczekiwania. Dozwolona wartość należy do przedziału od 1 do 5.

Funkcja **Listen()** nie blokuje działania aplikacji do czasu zarejestrowania żądania połączenia od klienta, nie musi także być cyklicznie wywoływana aż takie żądanie nadejdzie.

Akceptację połączenia realizuje funkcja **Accept()** obiektu **AsyncLocalSocket** wywoływana instrukcją:

```
BOOL BoolVar;
BoolVar = AsyncLocalSocket.Accept
     (AsyncRemoteSocket,
      &AsyncRemoteSocketAddr,
      &SizeOfSockAddr);
```

Pierwszy parametr jest odwołaniem do obiektu **CMyAsyncSocket AsyncRemoteSocket**, pozostałe parametry są opcjonalne: Drugi jest wskaźnikiem do struktury typu **SOCKADDR**, a trzeci wskaźnikiem do zmiennej typu **int**, do których funkcja **Accept()** wpisuje odpowiednio: informacje o połączeniu (w tym adres gniazda wysyłającego żądanie połączenia) i rozmiar struktury **SOCKADDR** w bajtach.

Jeśli w kolejce nie ma oczekujących połączeń, to funkcja **Accept()** zwraca wartość **FALSE**, a funkcja **AsyncLocalSocket.GetLastError()** błąd o kodzie **WSAEWOULDBLOCK**. Wówczas wywoływanie funkcji **Accept()** można ponawiać aż do zarejestrowania żądania połączenia od klienta lub odczytu błędu innego niż **WSAEWOULDBLOCK**. W aplikacjach opisanych w dalszej części artykułu funkcja **AsyncLocalSocket.Accept()** wywoływana jest w procedurze **OnTimer()** obsługi komunikatów **WM_TIMER** głównego okna dialogowego.

W przypadku aplikacji działającej jako klient do nawiązania połączenia należy użyć funkcji **Connect()** obiektu **AsyncLocalSocket** wywoływanej instrukcją:

```
BOOL BoolVar;
BoolVar = AsyncLocalSocket.Connect
(LPCTSTR RemoteSocketAddress,
UINT RemoteSocketPortNumber);
```

Parametr **RemoteSocketAddress** jest wskaźnikiem (ang. pointer) do ciągu znakowego określającego adres sieciowy gniazda (w tym przypadku urządzenia działającego jako serwer), np. 128.53.22.14, a parametr **RemoteSocketPortNumber** określa numer portu. Jeśli serwer zaakceptuje żądanie połączenia, to funkcja **Connect()** zwraca wartość **TRUE**. Zwracana wartość **FALSE** oznacza, że połączenie nie może zostać nawiązane, a przyczynę odrzucenia żądania można ustalić wywołując funkcję **AsyncLocalSocket.GetLastError()**. Błąd o kodzie **WSAEISCONN** oznacza, że uprzednio nawiązano połączenie z serwerem, natomiast błąd o kodzie **WSAEWOULDBLOCK** oznacza, że sekwencja nawiązywania połączenia w tym momencie nie może zostać ukończona. W tym drugim przypadku wywoływanie funkcji **Connect()** można wielokrotnie ponawiać aż do zaakceptowania nawiązania połączenia przez serwer lub odczytu błędu innego niż **WSAEWOULDBLOCK**. W aplikacjach funkcja **AsyncLocalSocket.Connect()** wywoływana jest w procedurze realizowanej jako wątek (uruchamiany z poziomu głównego okna dialogowego.

4.4. Zerwanie połączenia gniazda lokalnego z gniazdem oddalonym

Programowe zerwanie połączenia gniazda lokalnego z gniazdem oddalonym następuje po wykonaniu funkcji **Close()**. Jednak w zależności od tego czy aplikacja działa jako serwer czy jako klient jest to wykonywane w różny sposób.

- jeśli aplikacja działa jako serwer to zerwanie połączenia z klientem następuje po wywołaniu funkcji **AsyncRemoteSocket.Close()**. W tym przypadku serwer zrywa połączenie tylko z gniazdem, któremu odpowiada obiekt **AsyncRemoteSocket**. Gdyby aplikacja miała także otwarte połączenie z innym gniazdem, np. opisanym obiektem **AsyncRemoteSocket2**, to połączenie z nim nadal byłoby otwarte,
- jeśli aplikacja działa jako klient, to zerwanie połączenia z serwerem następuje po wywołaniu funkcji **AsyncLocalSocket.Close()**. W tym przypadku aby powtórnie nawiązać połączenie z serwerem musi zostać powtórzona sekwencja wykonywana funkcjami **Create()** i **Connect()**.

4.5. Sygnalizacja nawiązania i zerwania połączenia

Oprogramowanie systemowe sygnalizuje nawiązanie i zerwanie połączenia między dwoma współpracujacymi punktami końcowymi (gniazdami) w następujący sposób:

- jeśli aplikacja pracuje jako serwer to nawiązanie połączenia z klientem jest sygnalizowane wywołaniem funkcji **AsyncLocalSocket.OnAccept()** i dodatkowo wywoływaniem funkcji **AsyncRemoteSocket.OnSend()**. Zerwanie połączenia skutkuje automatycznym uruchomieniem funkcji **AsyncRemoteSocket.OnClose()**.
- jeśli aplikacja pracuje jako klient to nawiązanie połączenia z serwerem jest sygnalizowane wywołaniem funkcji **AsyncLocalSocket.OnConnect()** i dodatkowo wywoływaniem funkcji **AsyncLocalSocket.OnSend()**. Zerwanie połączenia skutkuje automatycznym uruchomieniem funkcji **AsyncLocal.OnClose()**.

W obu przypadkach funkcja **OnSend()** nie powinna być używana do oceny, czy połączenie zostało faktycznie nawiązane, ponieważ jej podstawowe przeznaczenie jest inne. Należy pamiętać, że wywołanie każdej z funkcji wymienionych powyżej jest uzależnione od wartości parametru **Event** funkcji **AsyncLocalSocket.Create()** omówionej w punkcie dotyczącym tworzenia gniazda.

4.6. Odbiór informacji od gniazda współpracującego

Po nawiązaniu połączenia odbiór informacji w gnieździe lokalnym odbywa się na podobnej zasadzie jak działają procedury uruchamiane w trybie przerwaniowym. Polega to na tym, że w momencie odbioru informacji oprogramowanie systemowe inicjuje wykonywanie funkcji **OnReceive()** aplikacji równolegle do realizacji samej aplikacji:

- jeśli aplikacja pracuje jako serwer to odbiór informacji od klienta jest sygnalizowane wywołaniem funkcji **AsyncRemoteSocket.OnReceive()**.
- jeśli aplikacja pracuje jako klient to odbiór informacji z serwera jest sygnalizowane wywołaniem funkcji **AsyncLocalSocket.OnReceive()**.

Samo wywołanie funkcji **OnReceive()** nie zapewnia jednak dostępu do odebranej informacji. Właściwy odczyt wykonuje się przy pomocy funkcji **Receive()**, także będącej elementem klas **CAsyncSocket** i **CMyAsyncSocket**. Jej parametrami są wskaźnik (ang. pointer) do bufora odbiorczego **ReceiveBuffer** gdzie odczytana informacja ma zostać umieszczona oraz wymiar tego bufora. Sama funkcja zwraca liczbę bajtów zapisanych przez nią w buforze odbiorczym.

Wspomniano, że funkcja **OnReceive()** jest wykonywana współbieżnie z samą aplikacją. Istnieje więc zagrożenie, że aktualna zawartość bufora odbiorczego **ReceiveBuffer** zostanie utracona w wyniku kolejnego uruchomienia **OnReceive()** zanim aplikacja ją odczyta i zinterpretuje. W przypadku aplikacji omówionych w dalszej części tego artykułu problem ten rozwiązano wykorzystując dodatkowy bufor **ApplicationBuffer** obsługiwany przez dwa indeksy: **IndeksZapisu** i **IndeksOdczytu** oraz **LicznikWypełnienia**. Obsługa tego bufora odbywa się w następujący sposób: Po wykonaniu funkcji **Receive()** zawartość bufora **ReceiveBuffer** zostaje przepisana bajt po bajcie do **ApplicationBuffer** od komórki o indeksie **IndeksZapisu**, jednocześnie zwiększając **IndeksZapisu** i **LicznikWypełnienia** o liczbę przepisanych bajtów. Aplikacja odczytuje zawartość **ApplicationBuffer** od komórki o indeksie **IndeksOdczytu** jednocześnie zwiększając **IndeksOdczytu** i zmniejszając **LicznikWypełnienia** o liczbę odczytanych bajtów. Zwiększanie obu indeksów **IndeksZapisu** i **IndeksOdczytu** odbywa się do rozmiaru bufora **ApplicationBuffer**, następnie są one zerowanie i zliczanie powtarza się.

4.7. Wysyłanie informacji do gniazda współpracującego

Funkcję **OnSend()** klas **CAsyncSocket** i **CMyAsyncSocket** oprogramowanie systemowe wywołuje jako powiadomienie o możliwości wysłania informacji do gniazda współpracują-

cego. Samą transmisję inicjuje funkcja **Send()**, której parametrami są wskaźnik (ang. *pointer*) do bufora nadawczego **TransmitBuffer** zawierającego dane oraz liczbę bajtów do wysłania. W momencie wywołania **OnSend()** aplikacja może nie mieć żadnych danych do transmisji, a więc dołączanie do procedury **OnSend()** instrukcji **Send()** wiąże się ze sprawdzaniem dodatkowych warunków o aktualnym wypełnieniu bufora **TransmitBuffer**. Zamiast tego wygodniej jest sterować wysyłką informacji poprzez wartość dodatkowej flagi typu **BOOL**, która byłaby wysterowywana (**TRUE**) przez funkcję **OnSend()**, a zerowana (**FALSE**) przez główną część aplikcji po instrukcji wywołującej **Send()**. Naturalnie wywołanie **Send()** jest możliwe tylko wtedy, gdy wspomniana flaga ma wartość **TRUE** i w buforze **TransmitBuffer** znajdują się dane do transmisji.

Transmisję do gniazda współpracującego aplikacja może realizować tylko wtedy, gdy nawiązano połączenie z docelowym gniazdem oddalonym. W zależności od trybu pracy aplikacji różny jest sposób obsługi tej transmisji:

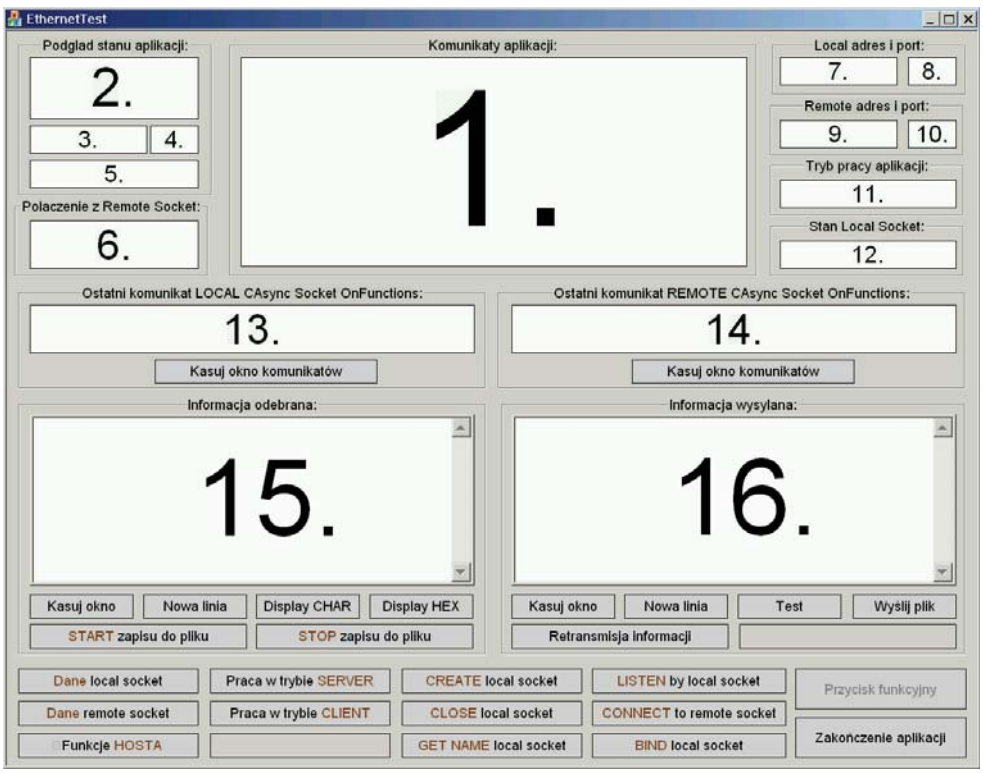
- jeśli aplikacja pracuje jako serwer to możliwość wysyłania informacji do klienta sygnalizuje funkcja **AsyncRemoteSocket.OnSend()**, a samą transmisję wykonuje funkcja **AsyncRemoteSocket.Send()**,
- jeśli aplikacja pracuje jako klient to możliwość wysyłania informacji do serwera sygnalizuje funkcja **AsyncLocalSocket.OnSend()**, a samą transmisję wykonuje funkcja **AsyncLocalSocket.Send()**,

5. Uniwersalna aplikacja EthernetTest.exe do testowania transmisji siecią Ethernet między komputerem PC a dowolnym urządzeniem zewnętrznym

W Przemysłowym Instytucie Automatyki i Pomiarów PIAP zrealizowano kilka aplikacji, które wymagały sprzężenia ze sobą dwóch różnych urządzeń za pomocą sieci Ethernet. Jedną z nich była praca, gdzie punktami końcowymi był robot ABB i kamera wizyjna firmy Sick [8]. W trakcie realizacji podobnych aplikacji częstym problemem jest uruchomienie samej transmisji, ponieważ w przypadku gdy działa ona niewłaściwie zazwyczaj nie od razu wiadomo, którego urządzenia to dotyczy i jakie błędy są tego przyczyną. Dobrą praktyką jest kolejne zastępowanie jednego z urządzeń komputerem PC z poprawnie działającą aplikacją sieciową i przy jej pomocy uruchomienie transmisji na drugim urządzeniu. W związku z tym przydatna jest uniwersalna aplikacja, którą można testować transmisję sieciową między komputerem a urządzeniem zewnętrznym takim jak sterownik robota, kamera wizyjna lub drugi komputer. Aplikacja taka powinna mieć możliwość zamiennej pracy jako serwer i jako klient, a jej kod byłby wzorcem do realizacji innych programów użytkowych, np. monitorujących stan robota. Taką aplikacją jest **EthernetTest.exe**, wykorzystująca elementy klasy **CMyAsyncSocket**. Jej główne okno dialogowe przedstawiono na rysunku 1.

Aplikacja działa w następujący sposób:

- po uruchomieniu analizowany jest dołączony tekstowy plik konfiguracyjny. Zawiera on m.in. informacje o adresie sieciowym i numerze portu komputera, na którym uruchomiono aplikację, a także te same informacje dotyczące urządzenia współpracującego (drugiego komputera lub robota). W pliku konfiguracyjnym jest także informacja, czy aplikacja ma działać jako serwer lub jako klient. Wspomniane parametry mogą być zmieniane z poziomu samej aplikacji (przyciski „**Dane local socket**” i „**Dane remote socket**”, „**Praca w trybie SERVER**”, „**Praca w trybie CLIENT**”),
- po wyświetleniu okna dialogowego przyciskiem „**Create local socket**” należy zdefiniować gniazdo,
- jeśli aplikacja ma działać jako serwer, to należy ją ustawić w tryb nasłuchu kanału transmisyjnego przy pomocy przycisku „**LISTEN by local socket**”. Połączenie będzie nawiązane, jeśli na urządzeniu współpracującym zostanie wykonana funkcja **Connect**,
- jeśli aplikacja ma działać jako klient, to należy wybrać przycisk „**CONNECT to remote socket**”. Połączenie będzie



Rys. 1. Główne okno dialogowe aplikacji EthernetTest.exe do testowania transmisji siecią Ethernet między komputerem PC a dowolnym urządzeniem zewnętrznym (np. robotem)
Fig. 1. The main dialog window of the EthernetTest.exe application for testing Ethernet transmission between a PC and any external device (e.g. a robot)

nawiązane, jeśli uprzednio urządzenie współpracujące zostanie ustawione w stan nasłuchu **Listen**.
— po nawiązaniu połączenia cała odbierana informacja będzie wyświetlana w okienku nr 15. Aby wysłać informację należy wybrać kursorem myszy okienko nr 16, a następnie posłużyć się klawiaturą. Możliwe jest także wysyłanie informacji generowanej przez samą aplikację, wysyłanie informacji odczytanej z pliku tekstowego lub w trybie retransmisji, tzn. to co zostaje odebrane, jest wysyłane do nadawcy.
— przycisk „**CLOSE local socket**" bez względu na wybrany tryb pracy aplikacji zamyka połączenie.

Dla ułatwienia obsługi niektóre przyciski są blokowane, jeśli w danym trybie pracy aplikacji funkcje uruchamiane tymi przyciskami są niedostępne (np. blokowanie przycisku „**LISTEN by local socket**", jeśli aplikacja pracuje jako klient).

Okienko 1:
— komunikaty informacyjne dla operatora wyświetlane przez aplikację **EthernetTest.exe**

Okienko 2:
— komunikat o aktualnym stanie aplikacji **EthernetTest.exe**

Okienko 3:
— odczyt wskazań zegara komputera na którym pracuje aplikacja **EthernetTest.exe**

Okienko 4:
— numer podstanu aplikacji **EthernetTest.exe**

Okienko 5:
— odczyt wskazań kalendarza komputera, na którym pracuje aplikacja **EthernetTest.exe**

Okienko 6:
— informacja o stanie połączenia komputera PC z urządzeniem zewnętrznym (np. z robotem)

Okienko 7:
— adres sieciowy komputera PC

Okienko 8:
— numer portu komputera PC

Okienko 9:
— adres sieciowy zewnętrznego urządzenia współpracującego (np. robota)

Okienko 10:
— numer portu zewnętrznego urządzenia współpracującego (robota)

Okienko 11:
— wybrany tryb pracy aplikacji **EthernetTest.exe** (serwer lub klient)

Okienko 12:
— informacja, czy utworzono kanał transmisyjny do urządzenia zewnętrznego (robota)

Okienko 13:
— komunikat wygenerowany przez ostatnią wykonaną funkcję będącą elementem obiektu lokalnego. Funkcja ta jest automatycznie uruchamiana przez oprogramowanie systemowe (dotyczy sześciu funkcji: **AsyncLocalSocket.OnFunction()**)

Okienko 14:
— komunikat wygenerowany przez ostatnią wykonaną funkcję będącą elementem obiektu oddalonego. Funkcja ta jest automatycznie uruchamiana przez oprogramowanie systemowe (dotyczy sześciu funkcji: **AsyncRemoteSocket.OnFunction()**)

Okienko 15:
— informacja odebrana z urządzenia zewnętrznego, np. z robota. Może być ona wyświetlana w postaci znakowej lub jako kody hexadecymalne poszczególnych znaków. Pod okienkiem umieszczono pięć przycisków:
- **Kasuj okno** – jego wybranie spowoduje wyczyszczenie zawartości okienka,
- **Nowa linia** – jego wybranie spowoduje rozpoczęcie wyświetlania od nowego wiersza,
- **Display char** – odebrana informacja będzie wyświetlana w postaci znakowej,
- **Display hex** – odebrana informacja będzie wyświetlana w postaci kodów hexadecymalnych poszczególnych znaków,
- **Start zapisu do pliku** – odebrana informacja będzie dodatkowo zapisywana do pliku zgodnie z wybranym formatem zapisu, tj. znakowo albo jako kod hexadecymalny poszczególnych znaków,
- **Stop zapisu do pliku** – zatrzymanie zapisu do pliku odbieranej informacji.

Okienko 16:
— informacja wysyłana do urządzenia zewnętrznego, np. do robota. Może być ona wyświetlana w postaci znakowej lub jako kody hexadecymalne poszczególnych znaków. Pod okienkiem umieszczono pięć przycisków:
- **Kasuj okno** – jego wybranie spowoduje wyczyszczenie zawartości okienka,
- **Nowa linia** – jego wybranie powoduje, że wyświetlanie informacja rozpocznie się od nowego wiersza,
- **Test** – aplikacja będzie automatycznie wysyłała ciąg znakowy jak poniżej:

```
123456789 123456789 123456789 123456789
123456789 12345
23456789 123456789 123456789 123456789
123456789 12345
3456789 123456789 123456789 123456789
123456789 12345 1
.....
.............
9 123456789 123456789 123456789
123456789 123456789 123
 123456789 123456789 123456789 123456789
123456789 1234
123456789 123456789 123456789 123456789
123456789 12345
```

  Test ten jest wykonywany w sposób ciągły do monentu aż zostaje zakończony przez operatora przez wybranie przycisku funkcyjnego
- **Wyślij plik** – jego wybranie spowoduje otwarcie okienka dialogowego do wskazania pliku, którego zawartość zostanie wysłana do urządzenia zewnętrznego.
- **Retransmisja** – informacja odbierana z urządzenia zewnętrznego jest wysyłana do nadawcy. Test ten jest wykonywany w sposób ciągły do monentu aż zostaje zakończony przez operatora poprzez wybranie przycisku funkcyjnego

Aplikacja **EthernetTest.exe** może współpracować z oprogramowaniem robota KUKA opisanym poniżej. Jednak nie zawiera ona fragmentów kodu służącego do interpretacji informacji zawartych w przesyłkach z robota, a więc jej działanie ogranicza się tylko do wyświetlenia w okienku nr 15 zawartości przesyłek. Na podstawie tej aplikacji wykonano program **EthernetRobot.exe** interpretujący przesyłki z robota, ale uboższy w zakresie funkcjonalności transmisji sieciowej. Program ten opisano w dalszej części tej publikacji.

6. Przykładowe oprogramowanie dla robota przemysłowego

PIAP dysponuje trzema obotami firmy KUKA: jednym z układem sterowania KRC2 i dwoma z układem sterowania KRC4. Bez względu na układ sterowania sposób programowania tych robotów jest jednakowy. Układy te różnią się jednak zasobami bibliotecznymi jakie są do dyspozycji programisty.

— układ KRC2 zawiera pojedynczy interfejs RS-232 i pojedynczy port Ethernet, ale aby było możliwe wykorzystanie portu Ethernet, to dodatkowo należy osadzić w sterowniku robota pakiet oprogramowania do jego obsługi,
— układ KRC4 nie zawiera interfejsu RS-232, ale standardowo został wyposażony w dwa porty Ethernet oznaczone jako KLI i KONI. Port KLI jest wykorzystywany do współpracy z oprogramowaniem Kuka WorkVisual przeznaczonym do konfigurowania i tworzenia aplikacji na zewnętrznym komputerze PC. Dlatego oprogramowanie systemowe robota zawiera wszystkie funkcje biblioteczne zawarte w pakiecie o nazwie EKI (ang. *Ethernet KRL Interface*) pozwalające na wymianę informacji z dowolnym urządzeniem zewnętrznym przy pomocy sieci Ethernet i protokołu TCP/IP. Na temat portu KONI autorzy pracy nie znaleźli żadnych informacji w dostępnych dla nich podręcznikach firmy KUKA.

Z wymienionych powodów do realizacji tej pracy wybrano jeden z robotów z układem sterowania KRC4, a jako port transmisyjny wykorzystano złącze KLI. Ten ostatni wybór narzucił jednak pewne ograniczenie. Aby oprogramowanie systemowe robota mogło współpracować z oprogramowaniem Kuka WorkVisual port ten musi mieć ustawiony sieciowy adres IP i numer portu ze ścisle określonego zakresu, a dowolna inna aplikacja komputera PC współpracująca z robotem powinna uwzględniać te ustawienia. Przy pomocy panelu programowania robota można modyfikować parametry poza narzucony zakres w sposób „wygodny" dla innej aplikacji, ale wtedy nie ma możliwości korzystania z pakietu Kuka WorkVisual. W trakcie realizacji pracy okazało się, że na zewnętrznym komputerze obie aplikacje, tj. Kuka WorkVisual i program zrealizowany w ramach tej pracy mogą pracować współbieżnie i bezkonfliktowo.

6.1. Zmienne systemowe i ich wykorzystanie w oprogramowaniu robota

Oprogramowanie aplikacyjne robotów Kuka tworzy się w języku KRL (ang. *Kuka Robot Language*). Jak każdy język wysokiego poziomu w swojej składni zawiera on definicje czterech podstawowych typów danych:
— **INT** – jest to typ danych o długości 2 bajtów do definiowania zmiennych typu liczba całkowita ze znakiem,
— **REAL** – jest to typ danych o długości 4 bajtów do definiowania zmiennych typu liczba zmiennoprzecinkowa,
— **BOOL** – jest to typ danych o długości 1 bajtu do definiowania zmiennych logicznych o wartościach **TRUE** lub **FALSE**,
— **CHAR** – jest typ danych o długości 1 bajtu do definiowania zmiennych „znakowych".

Wykorzystując te cztery podstawowe typy danych można definiować tablice wielowymiarowe i zmienne typu „struktura", które najczęściej wykorzystuje się do:
— zdefiniowania położenia osi manipulatora:

```
STRUC AXIS          REAL A1,A2,A3,A4,A5,A6
STRUC E6AXIS        REAL A1,A2,A3,A4,A5,A6,
                    E1,E2,E3,E4,E5,E6
```

Oba wymienione w tym podpunkcie typy zmiennych zawierają informacje, jakie są położenia poszczególnych osi manipulatora (w stopniach), aby robot znalazł się w określonym położeniu. Nie ma tutaj odwołania do żadnego układu współrzędnych prostokątnych i do żadnego narzędzia. Typ **E6AXIS** jest rozszerzeniem typu **AXIS** i odnosi się do przypadku, kiedy robot zawiera od 1 do 6 dodatkowych osi zewnętrznych związanych, np. z torem jezdnym.
— położenia punktu TCP w wybranym układzie współrzędnych prostokątnych:

```
STRUC POS           REAL X,Y,Z, A,B,C, INT S, T
STRUC E6POS         REAL X,Y,Z, A,B,C,E1,E2,
                    E3,E4,E5,E6, INT S,T
```

Oba wymienione w tym podpunkcie typy zmiennych zawierają informacje o współrzędnych punktu TCP i orientacji narzędzia w wybranym układzie współrzędnych prostokątnych. Typ **E6POS** jest rozszerzeniem typu **POS** i odnosi się do przypadku, kiedy robot zawiera od 1 do 6 dodatkowych osi zewnętrznych związanych, np. z torem jezdnym.

Układ sterowania robota KUKA i język KRL zapewniają dostęp do tzw. zmiennych systemowych. Zmienne te są dostępne w trybie programowania „expert" i umożliwiają zintegrowanie programu aplikacyjnego robota z informacjami dostępnymi w jego układzie sterowania. Przy ich pomocy możliwy jest m.in. odczyt aktualnej pozycji robota, odczyt aktualnych pozycji każdej z osi, odczyt aktualnych wartości prędkości i przyspieszeń oraz informacji o tym czy robot znajduje się w pozycji **HOME**. W języku KRL nazwy zmiennych systemowych rozpoczynają się od znaku „**$**". Pełną listę zmiennych systemowych wraz z ich opisem można znaleźć w [4]. W artykule przedstawiono podstawowe informacje na temat zmiennych, które wykorzystano przy realizacji pracy. Zmiennymi tymi są:
— zmienna **$AXIS_ACT** typu **E6AXIS** zawierająca informację o aktualnym położeniu w stopniach wszystkich osi manipulatora. Pozwala ona na przyporządkowanie informacji o położeniu osi do dowolnej zmiennej zadeklarowanej w programie jako zmienna typu **E6AXIS**:

```
E6AXIS _AxisPos       ; deklaracja zmiennej _AxisPos
_AxisPos = $AXIS_ACT  ; przyporządkowanie
```

— zmienna **$H_POS** typu **E6AXIS** zawierająca informację o położeniu w stopniach wszystkich osi manipulatora znajdującego się w wyjściowej pozycji **HOME**. Pozwala ona na przyporządkowanie informacji o położeniu osi robota znajdującego się w pozycji **HOME** do dowolnej zmiennej zadeklarowanej w programie jako zmienna typu **E6AXIS**:

```
E6AXIS _HomePos       ; deklaracja zmiennej _HomePos
_HomePos = $H_POS     ; instrukcja przyporządkowania
```

— zmienna **$POS_ACT** typu **E6POS** zawierająca informację o współrzędnych punktu TCP i orientacji wybranego narzędzia w wybranym układzie współrzędnych prostokątnych. Pozwala ona na przyporządkowanie tej informacji do dowolnej zmiennej zadeklarowanej w programie jako zmienna typu **E6POS**:

```
E6AXIS _CartesianPos     ; deklaracja zmiennej
                           _CartesianPos
CartesianPos = $POSACT   ; przyporządkowanie
```

— zmienna **$ACT_BASE** typu **INT** zawierająca informację o numerze aktualnie wybranego układu współrzędnych. Pozwala ona na przyporządkowanie tej informacji do dowolnej zmiennej zadeklarowanej w programie jako zmienna typu **INT**:

```
INT _NrUkladu           ; deklaracja zmiennej
                          _NrUkladu
_NrUkladu = $ACT_BASE   ; przyporządkowanie
```

— zmienna **$ACT_TOOL** typu **INT** zawierająca informację o numerze aktualnie wybranego narzędzia. Pozwala ona na przyporządkowanie tej informacji do dowolnej zmiennej zadeklarowanej w programie jako zmienna typu **INT**:

```
INT _NrNarzedzia          ; deklaracja zmiennej
                            _NrNarzedzia
_NrNarzedzia = $ACT_TOOL  ; przyporządkowanie
```

— zmienna **$PHGTEMP** typu **INT** zawierająca informację o aktualnej temperaturze we wnętrzu szafy sterowniczej robota. Pozwala ona na przyporządkowanie tej informacji do dowolnej zmiennej zadeklarowanej w programie jako zmienna typu **INT**:

```
INT _TempKCP            ; deklaracja zmiennej
                          _TempKCP
_TempKCP = $PHGTEMP     ; przyporządkowanie
```

— zmienna **$MOVE_OP** typu **ENUM** (typ wyliczeniowy) określająca aktualny tryb pracy robota: **T1**, **T2**, **AUT**, **EX:**

```
SWITCH $MODE_OP
  CASE #T1
     ;  → robot jest w trybie testowym T1
  CASE #T2
     ;  → robot jest w trybie testowym T1
  CASE #AUT
     ;  → robot jest w trybie automatyki AUT
  CASE #EX
     ;  → robot jest w trybie zewnętrznej automatyki EXT
  DEFAULT
     ;  →  robot jest w nieokreślonym trybie pracy (błąd)
ENDSWITCH
```

– zmienna **$IN_HOME** typu **SIGNAL** (lub **BOOL**) określająca czy manipulator robota znajduje się w wyjściowej pozycji **HOME**:

```
IF ($IN_HOME == TRUE) THEN
    ;  → manipulator robota jest w pozycji HOME
ELSE
    ;  → manipulator robota nie jest w pozycji HOME
ENDIF
```

– zmienna **$MOVE_ENABLE** typu **SIGNAL** (lub **BOOL**) określająca czy manipulator robota ma zezwolenie na ruch:

```
IF ($MOVE_ENABLE == TRUE) THEN
    ;  → manipulator robota ma zezwolenie na ruch
ELSE
    ;  → manipulator robota nie ma zezwolenia na ruch
ENDIF
```

– zmienna **$ROBTRAFO[32]** będąca tablicą 32-wymiarową typu **CHAR** zawierajcą nazwę robota:

```
CHAR _RobotName[32]        ; deklaracja zmiennej
                             _RobotName
_RobotName = $ROBTRAFO[]   ; przyporządkowanie
```

Dodatkowo w pliku konfiguracyjnym umieszczono deklarację 50-elementowej tablicy typu **CHAR** o nazwie **UserMessage[50]** i „wstępnie" przyporządkowano jej tekst **„ZAKONCZENIE WYKONYWANIA PROGRAMU"**:

```
DECL CHAR UserMessage[50]
UserMessage[]="ZAKONCZENIE WYKONYWANIA PROGRAMU"
```

Tablica **UserMessage[50]** została zadeklarowana jako zmienna o zasięgu **GLOBAL**, a więc jest dostępna z dowolnego programu aplikacyjnego robota. W związku z tym każda taka aplikacja może umieszczać w niej swój dowolny tekst, ale nie dłuższy niż 50 znaków.

6.2. Plik konfiguracyjny dla pakietu EKI

Aby była możliwa wymiana informacji między robotem a urządzeniem zewnętrznym za pomocą sieci Ethernet i protokołu TCP/IP należy uporzednio skonfigurować połączenie sieciowe i określić format kodowania tej informacji [5]. Odpowiedni plik konfiguracyjny w formacie .XML pod dowolną nazwą obligatoryjnie musi znajdować się w pamięci masowej robota w katalogu:

C:\KRC\Roboter\Config\User\Common\EthernetKRL

W katalogu tym można zapisać wiele plików konfiguracyjnych. O tym, który z nich będzie wykorzystany decyduje parametr funkcji bibliotecznej **EKI_Init()** inicjującej w programie aplikacyjnym transmisję. Parametrem tym jest nazwa pliku konfiguracyjnego. W przykładowym oprogramowaniu dla robota, wykonanym na potrzeby tej pracy, plikiem konfiguracyjnym jest plik o nazwie „**DunajEKI_Client.xml**", a odpowiednie wywołanie funkcji **EKI_Init()** ma postać:

```
DECL EKI_STATUS _RET
.....
RET = EKI_Init („DunajEKI_Client")
```

gdzie **_RET** jest zmienną typu **EKI_STATUS** pod którą zostanie podstawiona wartość zwracana przez funkcję **EKI_Init()**.

Zawartość pliku **„DunajEKI_Client.xml"** zamieszczono poniżej:

```
<?xml version="1.0" encoding="utf-16"?>
<ETHERNETKRL>
   <CONFIGURATION>
      <EXTERNAL>
            <IP>192.168.10.105</IP>
            <PORT>50000</PORT>
      </EXTERNAL>
      <INTERNAL>
        <ENVIRONMENT>Program</ENVIRONMENT>
        <MESSAGES Display="disabled"
                  Logging="disabled" />
        <ALIVE Set_Flag="999" Ping="10" />
        <BUFFERING Limit="1" />
      </INTERNAL>
   </CONFIGURATION>
   <RECEIVE>
      <XML>
        <ELEMENT Tag="Sen/@Type"
                 Type="STRING" />
        <ELEMENT Tag="Sen/EStr"
                 Type="STRING" />
        <ELEMENT Tag="Sen/IPOC"
                 Type="INT" />
        <ELEMENT Tag="Sen/Mode/@PID"
                 Type="INT" />
        <ELEMENT Tag="Sen" Set_Flag="998"
/>
      </XML>
   </RECEIVE>
   <SEND>
      <RAW>
        <ELEMENT Tag="Buffer"
                 Type="STRING" />
      </RAW>
   </SEND>
</ETHERNETKRL>
```

W pliku tym można wyróżnić trzy sekcje:

1. Sekcja **<CONFIGURATION> </CONFIGURATION>** zawierająca informacje o adresie IP i numerze portu urządzenia współpracującego. Przyjęto, że urządzenie współpracujące będzie pracowało w trybie SERVER (sterownik robota jest klientem), w zwiazku z tym w pliku konfiguracyjnym zgodnie z opisem zawartym w [5] pominięto sekcję **<TYPE> </TYPE>**. Należy także zwrócić uwagę na polecenie **<ALIVE />** zawarte w sekcji **<INTERNAL>**. Określa ona numer flagi (w tym przypadku jest to flaga **$FLAG[999]**), która zostaje automatycznie wysterowana przez oprogramowanie systemowe robota (wartość **TRUE**), jeśli istnieje połączenie z urządzeniem zewnętrznym. Parametr **Ping** określa interwał czasowy w sekundach między wysłaniem kolejnego „pingu" monitorujacego połączenie z urządzeniem zewnętrznym.
2. Sekcja **<RECEIVE> </RECEIVE>** zawierająca informacje o formacie przesyłek odbieranych przez robota z urządzenia zewnętrznego. W przykładowym oprogramowaniu wykonanym w ramach tej pracy przyjęto, że urządzenie zewnętrzne (w tym przypadku komputer PC) nie będzie wysyłał do robota żadnej informacji, więc informacja zawarta w tej sekcji może być dowolna, pod warunkiem zachowania jej formalnej poprawności.
3. Sekcja **<SEND> </SEND>** zawierająca informacje o formacie przesyłek z robota do urządzenia zewnetrznego. Przyjęto, że robot będzie wysyłał informację w postaci zbu-

forowanego ciągu znakowego (**<ELEMENT Tag="Buffer" Type="STRING" />**)

6.3. Formaty przesyłek z robota do urządzenia zewnętrznego

Na potrzeby realizacji tej pracy przyjęto następujące założenia co do formatu przesyłek generowanych przez robota:

- każda przesyłka ma format tekstowy i składa się z ciągu znaków ASCII o kodach od **0x20** (spacja) do **0x7E** (znak „**~**"),
- każda przesyłka rozpoczyna się od znaku „**[**" a kończy znakiem „**]**". Dodatkowo po znaku wyznaczajacym koniec przesyłki do bufora nadawczego dodawane są dwa znaki: **carriage return** o kodzie hexadecymalnym **0x0D** i **line feed** o kodzie haexadecymalnym **0x0A**. Znaki te nie są interpretowane przez program aplikacyjny komputera, zostały dodane po to, aby ułatwić wyświetlanie odebranych przesyłek lub ich rejestracje do pliku tekstowego podczas uruchamiania oprogramowania.
- po znaku „**[**" rozpoczynającym przesyłkę występuje dwuliterowy kod przesyłki, a następnie ciąg znakowy z informacją stanowiącą „zawartość" przesyłki.
- formatowanie przesyłek odbywa się w buforze nadawczym zadeklarowanym jako 64-wymiarowa tablica znakowa:

```
DECL CHAR _Buffer[64]
```

- przed wpisaniem przesyłki do bufora nadawczego jego zawartość jest kasowana poprzez jego wypełnienie znakami o kodzie hexadecymalnym **0x00**:

```
INT _IDX
........
FOR _IDX=1 TO 64
   Buffer [IDX] = 0;
ENDFOR
```

- do formatowania przesyłek najczęściej jest wykorzystywana instrukcja **SWRITE**. Format tej instrukcji oraz sposób jej wykorzystania opisano w [7].
- wysłanie zawartości bufora jest realizowane funkcją biblioteczną **EKI_Send()** o następującym formacie:

```
DECL EKI_STATUS _RET
........
_RET=EKI_Send ("DunajEKI_Client", _Buffer[]);
```

gdzie pierwszy parametr „**DunajEKI_Client**" określa nazwę pliku konfiguracyjnego, drugi parametr „**_Buffer[]**" – nazwę bufora nadawczego

6.4. Struktura podprogramu „SendInfoByEthernet()" realizującego transmisję

W rozdziale o zmiennych systemowych robota opisano, w jaki sposób można odczytywać informację o różnych stanach robota KUKA. W tym punkcie opisano jak informacja ta jest formatowana i transmitowana do urządzenia zewnętrznego.

1. Początek podprogramu:

Deklaracje używanych zmiennych:

```
DECL CHAR          _CR
DECL CHAR          _NL
DECL CHAR          _Buffer[64]
DECL CHAR          _RobotName[32]
DECL INT           _NrNarzedzia
DECL INT           _NrUkladu
DECL INT           _TempKCP
DECL INT           _IDX
DECL INT           _Offset
DECL E6AXIS        _AxisPos
DECL E6AXIS        _HomePos
DECL E6POS         _CartesianPos
DECL EKI_STATUS    _RET
DECL STATE_T       _State
```

2. Start podprogramu:

Program wykorzystuje timer `$TIMER[12]` do odliczania 200 milisekundowych interwałów czasowych między wysłaniem kolejnego pakietu przesyłek. Jeśli `$TIMER[12]` zliczył 200 ms, to jest powtórnie uruchamiany, a następnie sprawdzany jest stan flagi `$FLAG[999]`. Wartość tej flagi, zgodnie z deklaracją zawartą w pliku konfiguracyjnym `DunajEKI_Client.xml`, określa czy otwarty jest kanał transmisyjny między robotem a urządzeniem zewnętrznym. W przypadku, gdy kanał jest zamknięty, tzn. jeśli wartość flagi `$FLAG[999]` jest równa `FALSE`, podprogram w pętli kolejno inicjuje jego otwarcie (funkcja `EKI_Init („DunajEKI_Client")`. Zakończenie pętli programowej następuje albo po otwarciu kanału, albo po zliczeniu przez `$TIMER[12]` 200 ms. W tym drugim uznaje się, że nie ma możliwości otwarcia kanału transmisyjnego do urządzenia zewnętrznego i podprogram kończy działanie.

```
IF ($TIMER_FLAG[12] == TRUE) OR
   ($TIMER_STOP[12] == TRUE) THEN
   $TIMER      [12] = -200
   $TIMER_STOP [12] = FALSE

   IF  $FLAG[999] == FALSE THEN
      _RET = EKI_Init  ("DunajEKI_Client")
      WHILE $FLAG[999] == FALSE
        _RET = EKI_Open("DunajEKI_Client")
   IF ($TIMER_FLAG[12] == TRUE) THEN
        _RET = EKI_Close ("DunajEKI_Client")
        _RET = EKI_Clear ("DunajEKI_Client")
            GOTO THE_END
        ENDIF
      ENDWHILE
   ENDIF
```

Jeśli timer `$TIMER[12]` zliczył 200 ms i wartość flagi `$FLAG[999]` wynosi `TRUE`, to można realizować transmisję przesyłek do urządzenia zewnętrznego

3. Podstawienie pod zmienne pomocnicze **_CR i _NL** wartości odpowiednio: **0x0D** (**carriage return**) i **0x0A** (**line feed**)

```
_CR = 13
_NL = 10
```

4. Formatowanie i wysłanie przesyłki zawierającej nazwę robota.

Przesyłka ma format:

[NRxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx]

gdzie parametr „**xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx**" określający nazwę robota jest ustalany na podstawie odczytu zmiennej systemowej **$ROBTRAFO[]**:

```
_RobotName[] = $ROBTRAFO[] ; → Tablica
ROBTRAFO ma wymiar 32
_Offset = 0;
SWRITE (_Buffer[], _State, _Offset,
        "[NR%s]%c%c",_RobotName[],_CR,_NL)
_RET=EKI_Send("DunajEKI_Client", _Buffer[]);
```

5. Formatowanie i wysłanie przesyłki zawierającej dwie informacje:
   - o ustawieniu manipulatora robota w pozycji **HOME**,
   - o ustawionym zezwoleniu na ruch manipulatora.

Przesyłka ma format:

[FLxy]

gdzie:

– **x** jest informacją że manipulator jest w pozycji **HOME** (**1-TAK**, **0-NIE**)
– **y** jest informacją, że jest zezwolenie na ruch manipulatora (**1-TAK**, **0-NIE**)

Wartości tych parametrów są ustalane na podstawie odczytu zmiennych systemowych **$IN_HOME** i **$MOVE_ENABLE**:

```
_Buffer[1] = "["
_Buffer[2] = "F"
_Buffer[3] = "L"
IF ($IN_HOME == TRUE) THEN
   _Buffer[4] = "1"
ELSE
   _Buffer[4] = "0"
ENDIF
IF ($MOVE_ENABLE == TRUE) THEN
   _Buffer[5] = "1"
ELSE
   _Buffer[5] = "0"
ENDIF
_Buffer[6] = "]"
_Buffer[7] = _CR
_Buffer[8] = _NL
_RET=EKI_Send("DunajEKI_Client", _Buffer[]);
```

6. Formatowanie i wysłanie przesyłki zawierającej informację o wybranym trybie pracy robota.

Przesyłka ma format:
[MOxxxxxxx]
gdzie tekstowy parametr „**xxxxxxx**" o zmiennej długości może mieć następujace wartości:
– „**T1**" – robot jest w testowym trybie pracy T1,
– „**T2**" – robot jest w testowym trybie pracy T2,
– „**AUT**" – robot jest w trybie automatyki,
– „**EXT_AUT**" – robot jest w trybie zewnętrznej autoamtyki (sterowanie przez PLC),
– „**???**" – błąd.

Wartość parametru „**xxxxxxx**" jest ustalana na podstawie odczytu zmiennej systemowej **$MODE_OP**:

```
_Offset = 0;
SWITCH $MODE_OP
   CASE #T1
      SWRITE (_Buffer[], _State, _Offset,
              "[MOT1]%c%c", _CR, _NL)
   CASE #T2
      SWRITE (_Buffer[], _State, _Offset,
              "[MOT2]%c%c", _CR, _NL)
   CASE #AUT
      SWRITE (_Buffer[], _State, _Offset,
              "[MOAUT]%c%c", _CR, _NL)
   CASE #EX
      SWRITE (_Buffer[], _State, _Offset,
              "[MOEXT_AUT]%c%c", _CR, _NL)
   DEFAULT
      SWRITE (_Buffer[], _State, _Offset,
              "[MO???]%c%c", _CR, _NL)
ENDSWITCH
_RET=EKI_Send("DunajEKI_Client", _Buffer[]);
```

7. Formatowanie i wysłanie przesyłki zawierającej trzy informacje:
– o numerze aktualnie wybranego układu współrzędnych prostokątnych,
– o numerze aktualnie wybranego narzędzia,
– o tempreraturze we wnętrzu szafy sterowniczej KCP robota.

Przesyłka ma format:
[DAxxxx:yyyy:zzzz]
gdzie:
– „**xxxx**" jest informacją o numerze aktualnie wybranego układu współrzędnych prostokątnych,
– „**yyyy**" jest informacją o numerze aktualnie wybranego narzędzia,
– „**zzzz**" jest informacją o tempreraturze we wnętrzu szafy sterowniczej KCP robota,
Parametry te są 4-cyfrowymi liczbami dziesiętnymi z zerami wiodącymi, oddzielonymi od siebie dwukropkiem. Ich wartości są ustalane na podstawie odczytu zmiennych systemowych **$ACT_BASE**, **$ACT_TOOL** i **$PHGTEMP**:

```
_NrUkladu    = $ACT_BASE
_NrNarzedzia = $ACT_TOOL
_TempKCP     = $PHGTEMP
_Offset = 0;
SWRITE (_Buffer[], _State, _Offset,
        "[DA%+04d:%+04d:%+04d]%c%c",
        _NrUkladu, _NrNarzedzia, _TempKCP,
        _CR, _NL)
_RET = EKI_Send("DunajEKI_Client", _Buffer[]);
```

Jeśli odczytana wartość zmiennej systemowej **$ACT_BASE** lub/i **$ACT_TOOL** jest ujemna, to oznacza, że nie wybrano żadnego układu współrzędnych prostokątnych lub/i nie wybrano żadnego narzędzia.

8. Formatowanie i wysłanie przesyłki zawierającej informację o położeniu punktu TCP i kątach orientacji aktualnie wybranego narzędzia w aktuualnie wybranym układzie współrzędnych prostokątnych.

Przesyłka ma format:
[KR±xxxx.xxx:±yyyy.yyy:±zzzz.zzz:±aaaa.aaa:±bbbb.bbb:±cccc.ccc]
gdzie:
– ± oznacza znak liczby,
– **xxxx.xxx** – wartość współrzędnej **X**,
– **yyyy.yyy** – wartość współrzędnej **Y**,
– **zzzz.zzz** – wartość współrzędnej **Z**,
– **aaaa.aaa** – wartość kąta **A**,
– **bbbb.bbb** – wartość kąta **B**,
– **cccc.ccc** – wartość kąta **C**.
Wszystkie parametry, oddzielone od siebie dwukropkiem, są 8-cyfrowymi liczbami zmiennoprzecinkowymi ze znakiem, kropką dziesiętną i zerami wiodącymi. Ich wartości ustala się na podstawie odczytu zmiennej systemowej **$POS_ACT**:

```
IF ((_NrUkladu > 0) and
    (_NrNarzedzia > 0)) THEN
   _CartesianPos = $POS_ACT
   _Offset = 0;
   SWRITE (_Buffer[], _State, _Offset,
           "[KR %+08.3f:%+08.3f:%+08.3f:
                %+08.3f:%+08.3f:%+08.3f]
           %c%c",
     _CartesianPos.X, _CartesianPos.Y,
     _CartesianPos.Z,
     _CartesianPos.A, _CartesianPos.B,
     _CartesianPos.C, _CR, _NL)
   _RET=EKI_Send("DunajEKI_Client", _Buffer[]);
ENDIF
```

Zmienna systemowa **$POS_ACT** odczytywana jest tylko wtedy, gdy uprzednio ręcznie za pomocą panelu sterowania bądź w aplikacji robota wybrano numer układu współrzędnych i numer narzędzia. W przeciwnym razie oprogramowanie systemowe robota wygeneruje błąd.

9. Formatowanie i wysłanie przesyłki zawierającej informację o aktualnym położeniu w stopniach każdej z sześciu osi manipulatora osi robota

   Przesyłka ma format:
   [AP±aaaa.aaa:±bbbb.bbb:±cccc.ccc:±dddd.ddd:±eeee.eee:±ffff.fff]
   gdzie:
   - ± oznacza znak liczby,
   - **aaaa.aaa** – wartość w stopniach aktualnego położenia osi **A1** manipulatora robota,
   - **bbbb.bbb** – wartość w stopniach aktualnego położenia osi **A2** manipulatora robota,
   - **cccc.ccc** – wartość w stopniach aktualnego położenia osi **A3** manipulatora robota,
   - **dddd.ddd** – wartość w stopniach aktualnego położenia osi **A4** manipulatora robota,
   - **eeee.eee** – wartość w stopniach aktualnego położenia osi **A5** manipulatora robota,
   - **ffff.fff** – wartość w stopniach aktualnego położenia osi **A6** manipulatora robota.

   Wszystkie parametry, oddzielone od siebie dwukropkiem, są 8-cyfrowymi liczbami zmiennoprzecinkowymi ze znakiem, kropką dziesiętną i zerami wiodącymi. Ich wartości ustala się na podstawie odczytu zmiennej systemowej **$AXIS_ACT**:

```
_AxisPos = $AXIS_ACT
_Offset = 0;
SWRITE (_Buffer[], _State, _Offset,
        "[AP% +08.3f:%+08.3f:%+08.3f:
         %+08.3f:%+08.3f:%+08.3f]
         %c%c",
         _AxisPos.A1, _AxisPos.A2,
        _AxisPos.A3,_AxisPos.A4,
         _AxisPos.A5, _AxisPos.A6,_CR, _NL)
_RET = EKI_Send("DunajEKI_Client", _Buffer[]);
```

10. Formatowanie i wysłanie przesyłki zawierającej informację o położeniu w stopniach każdej z 6 osi manipulatora osi robota pozostającego w pozycji **HOME**.

    Przesyłka ma format:
    [HP±aaaa.aaa:±bbbb.bbb:±cccc.ccc:±dddd.ddd:±eeee.eee:±ffff.fff]
    gdzie:
    - ± oznacza znak liczby,
    - **aaaa.aaa** – wartość w stopniach położenia osi **A1** manipulatora robota w pozycji **HOME**,
    - **bbbb.bbb** – wartość w stopniach położenia osi **A2** manipulatora robota w pozycji **HOME**,
    - **cccc.ccc** – wartość w stopniach położenia osi **A3** manipulatora robota w pozycji **HOME**,
    - **dddd.ddd** – wartość w stopniach położenia osi **A4** manipulatora robota w pozycji **HOME**,
    - **eeee.eee** – wartość w stopniach położenia osi **A5** manipulatora robota w pozycji **HOME**,
    - **ffff.fff** – wartość w stopniach położenia osi **A6** manipulatora robota w pozycji **HOME**.

    Wszystkie parametry, oddzielone od siebie dwukropkiem, są 8-cyfrowymi liczbami zmiennoprzecinkowymi ze znakiem, kropką dziesiętną i zerami wiodącymi.. Ich wartości ustala się na podstawie odczytu zmiennej systemowej **$H_POS**:

```
_HomePos = $HPOS
_Offset = 0;
SWRITE (_Buffer[], _State, _Offset,
        "[HP% +08.3f:%+08.3f:%+08.3f:
         %+08.3f:%+08.3f:%+08.3f]
         %c%c",
         _HomePos.A1, _HomePos.A2,
        _HomePos.A3,_HomePos.A4, _HomePos.A5,
         _HomePos.A6, _CR, _NL)
_RET = EKI_Send("DunajEKI_Client", _Buffer[]);
```

11. Formatowanie i wysłanie przesyłki zawierającej komunikat wygenerowany przez dowolną aplikację robota.

    Przesyłka ma format:
    [RMxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx]
    gdzie parametr „**xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx**” o długości do 50 znaków jest odczytywany z tablicy **RobotEthernetMessage[50]** zadeklarowanej jako zmienna globalna. Tablicę tę może wypełniać dowolna aplikacja robota

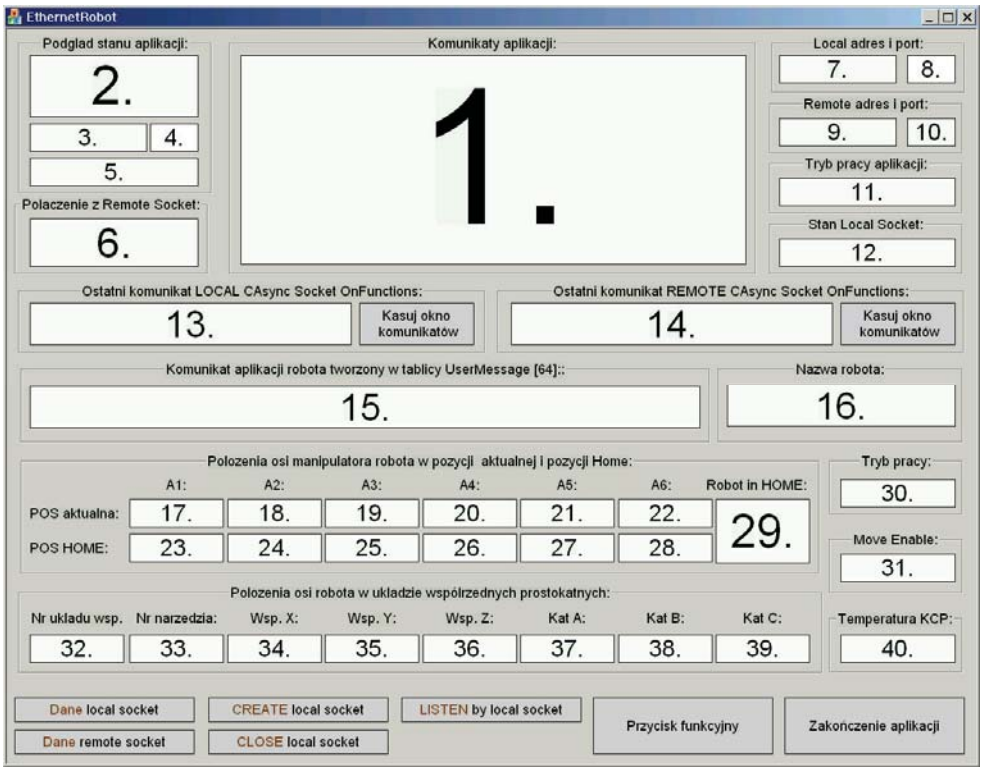
```
_Offset = 0;
SWRITE (_Buffer[], _State, _Offset,
      "[RM%s]%c%c",RobotEthernetMessage[],
        _CR, _NL)
_RET=EKI_Send("DunajEKI_Client", _Buffer[]);
```

6.5. Wywołanie podprogramu SendInfoByEthernet() realizującego transmisję

Podprogram **SendInfoByEthernet()** realizujący formatowanie i transmisję informacji z robota może być uruchamiany z poziomu jego dowolnej aplikacji. Jednak w takim przypadku cała informacja dotyczyć będzie momentu jego uruchomienia, a więc np. podczas realizacji instrukcji ruchu cała transmisja jest wstrzymana. Jeśli nie uruchomiono żadnej aplikacji wywołującej **SendInfoByEthernet()**, to robot nie realizuje transmisji do urządzenia zewnętrznego. Oprogramowanie systemowe robota firmy KUKA ma jednak wbudowany mechanizm umożliwiający obejście tego problemu. Jest nim tzw. interpreter submitów [6]. Jego działanie polega na realizowaniu „w tle” dodatkowego programu aplikacyjnego, wykonywanego równolegle i niezależnie od „właściwej” aplikacji, a także wtedy, gdy aplikacji tej nie uruchomiono. Program interpretrera submitów może zawierać wszystkie instukcje dostępne w języku KRL za wyjątkiem instrukcji ruchu, może też korzystać ze wszystkich zasobów sprzętowych i programowych robota. Powinien on działać tak jak program sterownika PLC tzn. nie oczekiwać w pętli na spełnienie określonego warunku, tylko być wykonywany w sposób ciągły od pierwszej do ostatniej instrukcji. Jest on więc rodzajem „wątku” wykonywanym niezależnie od aktualnej aplikacji robota i równolegle do niej. W pamięci masowej robota Kuka jest on zapisywany pod nazwą **sps.sub**. Jeśli zostanie w nim zawarta instrukcja wywołująca podprogram **SendInfoByEthernet()**, to odczyt i transmisja informacji opisanej w poprzednim rozdziale jest realizowana niezależnie od aktualnego trybu pracy robota i wybrania (lub niewybrania) aplikacji do realizacji. Podprogram **SendInfoByEthernet()** można zapamiętać w dowolnym katalogu pamięci masowej robota, ale dla zachowania przejrzystości zapisu – zaleca się katalog **\KRC\R1\System**.

7. Aplikacja EthernetRobot.exe do wizualizacji informacji odbieranej z robota

Wykorzystując elementy aplikacji **EthernetTest.exe** oraz klasę **CMyAsyncSocket** zaprojektowano i uruchomiono aplikację **EthernetTest.exe** współpracującą z opisanym oprogramowaniem robota KUKA. Interpretuje i dekoduje ona



Rys. 2. Główne okno dialogowe aplikacji EthernetRobot.exe do wizualizacji informacji odbieranej z robota
Fig. 2. Main dialog window of the EthernetRobot.exe application for visualization of information received from the robot

przesyłki odebrane z robota, a następnie wyświetla je w oknie dialogowym (rys. 2):

Aplikacja działa w podobny sposób jak **EthernetTest.exe**. Ponieważ może pracować tylko jako serwer, to w głównym oknie dialogowym pominięto te przyciski obsługi, które są wymagane w trybie pracy klient. Także informacja wyświetlana w okienkach od 1 do 14 jest identyczna jak w przypadku analogicznych okienek aplikacji **EthernetTest.exe**.

Zadaniem aplikacji **EthernetRobot.exe** jest odbiór i dekodowanie przesyłek z robota, a następnie wyświetlanie zawartej w nich informacji na ekranie monitora.

Okienko 15:
– komunikat wpisany do tablicy **RobotEthernetMessage[]** zdefiniowanej w pliku **KRC\R1\SYSTEM\$config.dat** w pamięci masowej robota i przesłany do komputera nadrzędnego przesyłką **[RMxxx....xxx]**

Okienko 16:
– nazwa robota odczytana ze zmiennej systemowej **$ROBTRAFO[]** i przesłana do komputera nadrzędnego przesyłką **[NRxxx....xxx]**

Okienka 17, 18, 19, 20, 21, 22:
– informacja o aktualnej pozycji w stopniach osi manipulatora robota, odpowienio: **A1**, **A2**, **A3**, **A4**, **A5** i **A6**. Jest ona odczytywana ze zmiennej systemowej **$AXIS_ACT** i przesłana do komputera nadrzędnego przesyłką **[AP...]**

Okienka 23, 24, 25, 26, 27, 28:
– informacja o pozycji w stopniach osi robota manipulatora, odpowienio: **A1**, **A2**, **A3**, **A4**, **A5** i **A6** w przypadku gdy znajduje się on w pozycji **HOME**. Jest ona odczytywana ze zmiennej systemowej **$H_POS** i przesłana do komputera nadrzędnego przesyłką **[HP...]**

Okienko 29:
– informacja, czy manipulator robota znajduje się w pozycji **HOME**. Jest ona odczytywana ze zmiennej systemowej **$IN_HOME** i przesłana do komputera nadrzędnego przesyłką **[FL...]**

Okienko 30:
– informacja o wybranym trybie pracy robota. Jest ona odczytywana ze zmiennej systemowej **$MODE_OP** i przesłana do komputera nadrzędnego przesyłką **[MO...]**

Okienko 31:
– informacja czy robot ma zezwolenie na ruch manipulatora. Jest ona odczytywana ze zmiennej systemowej **$IN_HOME** i przesłana do komputera nadrzędnego przesyłką **[FL...]**

Okienko 32:
– informacja o numerze aktualnie wybranego układu współrzędnych prostokątnych. Jest ona odczytywana ze zmiennej systemowej **$ACT_BASE** i przesłana do komputera nadrzędnego przesyłką **[DA...]**

Okienko 33:
– informacja o numerze aktualnie wybranego narzędzia. Jest ona odczytywana ze zmiennej systemowej **$ACT_TOOL** i przesłana do komputera nadrzędnego przesyłką **[DA...]**

Okienka 34, 35, 36, 37, 38, 39:
– informacja o współrzędnych punktu **TCP** i kątach orientacji aktualnie wybranego narzędzia w aktualnie wybranym układzie współrzędnych prostokątnych. Jest ona odczytywana ze zmiennej systemowej **$POS_ACT** i przesłana do komputera nadrzędnego przesyłką **[KR...]**

Okienko 40:
– informacja o temperaturze wewnątrz szafy sterowniczej **KCP** robota. Jest ona odczytywana ze zmiennej systemowej **$PHGTEMP** i przesłana do komputera nadrzędnego przesyłką **[DA...]**.

8. Uwagi końcowe

Przedstawione w tym artykule rozwiązania programowe związane z wymianą informacji robot – komputer przy pomocy sieci Ethernet nie były dotychczas stosowane w aplikacjach przemysłowych zrealizowanych przez PIAP. Opisane oprogramowanie

dotyczy przesyłu danych, wybranych w sposób dowolny i niekoniecznie trafnie jeśli chodzi o ich aktualną stronę praktyczną. Głównym celem tej pracy było jednak opracowanie i zrealizowanie niezawodnie działających funkcji transmisyjnych, które będzie można stosować w przyszłych potencjalnych aplikacjach. Wydaje się, że cel ten osiągnięto. W dalszych zamierzeniach przewiduje się sprzężenie robota z aplikacjami smartfonowymi i urządzeniami wirtualnej rzeczywistości takimi jak okulary Microsoft HoloLens.

Podziękowanie

Publikację opracowano m.in. na podstawie wyników IV etapu programu wieloletniego: „Poprawa bezpieczeństwa i warunków pracy", finansowanego w latach 2017–2019 w zakresie badań naukowych i prac rozwojowych przez Ministerstwo Nauki i Szkolnictwa Wyższego/Narodowe Centrum Badań i Rozwoju. Koordynator programu: Centralny Instytut Ochrony Pracy – Państwowy Instytut Badawczy".

Bibliografia

1. Microsoft Visual C++ 6.0 MFC Library Reference.
2. Leinecker R.C., Archer T., – *Visual C++ 6 Vademecum profesjonalisty*, Wydawnictwo HELION 2000.
3. KUKA System Software KR C2 / KR C3: Expert Programming – Release 5.2, KUKA Roboter GmbH 2003.
4. KUKA System Software KR C: System Variables – KUKA Roboter GmbH.
5. „KUKA System Technology …: KUKA.Ethernet KRL 2.1" for KUKA System Software 8.2, 2012 KUKA Roboter GmbH.
6. „KUKA System Software KR C: Submit Interpreter" Operation and Programming, 2005 KUKA Roboter GmbH
7. „KUKA System Software: CREAD/CWRITE" Programming CREAD/CWRITE and relatred statements, 2007 UKA Roboter GmbH.
8. Dunaj J., *Positioning of Industrial Robot Using External Smart Camera Vision*, R. Szewczyk and M. Kaliczyńska (eds.), Recent Advances in Systems, Control and Information Technology, Advances in Intelligent Systems and Computing 543, DOI: 10.1007/978-3-319-48923-0_35.

Monitoring the Status of an Industrial Robot Using the MFC Application

Abstract: The article presents how to implement a network transmission between a PC and an industrial robot. PC applications were implemented in C ++ using MFC libraries. It shows how to use the CAsyncSocket class in a practical way to program transmission between a computer and another device, e.g. an industrial robot. The second part of the article focuses on the robot's programming capabilities in providing information on its state (positions of axes, currently selected coordinate system and tool, etc.). The way of reading information, coding and then transmitting to a cooperating PC is discussed on the example of a KUKA robot. Also presented are the mode of operation and description of two sample PC applications for testing network transmission and presentation of data received from the robot.

Keywords: MFC application, Ethernet transmission, MFC libraries, CAsyncSocket class, industrial robot, Microsoft Visual Studio, Kuka WorkVisual

mgr inż. Jacek Dunaj

jacek.dunaj@piap.lukasiewicz.gov.pl
ORCID: 0000-0001-6812-1253

W 1980 r. ukończył studia na Wydziale Elektrycznym Politechniki Warszawskiej, od 1985 r. jest zatrudniony w Sieci Badawczej Łukasiewicz – PIAP. Specjalizuje się w programowaniu różnego rodzaju sprzętu: mikroprocesorów, kontrolerów, sterowników i robotów przemysłowych, systemów wizyjnych a także komputerów PC programowanych w języku asemblera i C/C++ w środowisku różnych systemów operacyjnych. Współautor oprogramowania dla kilku urządzeń opracowanych w PIAP, a także wielu wdrożeń przemysłowych, w szczególności wymagających współpracy kilku różnych urządzeń automatyki i wykorzystania oprogramowania biurowego (baz danych, arkuszy kalkulacyjnych).

mgr Kamil Bojanek

kamil.bojanek@piap.lukasiewicz.gov.pl
ORCID: 0000-0002-4448-3302

Urodzony w Pruszkowie w 1983 r. Dyplom magistra Marketingu uzyskał w 2008 r. w Wyższej Szkole Pedagogicznej w Warszawie. Kolejno ukończył studia podyplomowe w 2010 r. w Szkole Głównej Gospodarstwa Wiejskiego - Metody i Techniki Menedżerskie – Pre-MBA oraz w 2012 r. Wyższa Szkoła Ekonomii i Innowacji Lublinie – studia podyplomowe „Menadżer Promocji Nauki". Pracuje w Sieci Badawczej Łukasiewicz – PIAP jako główny specjalista w dziale Sprzedaży i Marketingu. Kierownik projektu „Serwis Przemysłowy PIAP".