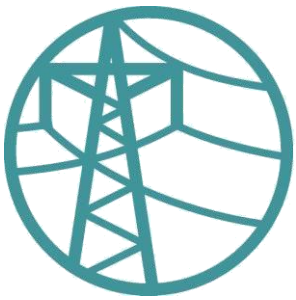


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Katedra Informatyki i Automatyki

Patryk Duda

Koncepcja systemu akwizycji i wizualizacji danych z symulatora lotu ALSIM 200 MCC.

Praca dyplomowa inżynierska

Opiekun pracy:
dr inż. Sławomir Samolej

Rzeszów, 2016

Spis Treści

1. Wstęp

Niniejsza praca inżynierska ten ma na celu przybliżenie, jak działa symulator lotu Alsim 200 MCC, oraz pokazać koncepcję utworzenia systemu komunikacji z symulatorem w celu zapisywania i zarchiwizowania danych. Dodatkowo, zarchiwizowane dane mają być wizualizowane w formie wykresu liniowego.

Koncepcja ta zakłada, że komunikacja z symulatorem lotu będzie odbywać się przez protokół UDP. Schemat działania tego protokołu, jak również lista danych, które symulator wysyła (a więc należy je odebrać) został dokładnie opisany w drugim rozdziale pracy. Jest on również podstawą do przedstawiania koncepcji stworzenia systemu odbierania danych wraz archiwizacją i wizualizacją na opisywany tu symulator lotu, to jest ALSIM 200 MCC.

Trzeci rozdział pracy składa się z trzech podpunktów. Pierwszy z nich skupia się na części archiwizacyjnej, a więc opisuje jak dane odebrane z symulatora można gromadzić. Została przedstawiona przykładowa baza danych, wraz z diagramem związków encji, która może gromadzić dane z symulatora, jak również jedna z wielu opcji przenoszenia danych z aplikacji odbierającej dane z symulatora do samej bazy danych. Przykłady również uwzględniają możliwe operacje na danych, aby jak najbardziej móc zautomatyzować cały proces archiwizacji danych, wliczając w to przeniesienie danych do pliku tekstowego.

Drugi podpunkt skupia się na samej aplikacji, która jest w stanie, korzystając z protokołu UDP, komunikować się z utworzonym serwerem i przesyłać dane. Jest to więc przykład prostego połączenia klient-serwer, korzystającego z protokołu UDP. Oprócz dokładnego opisania tego przykładowego programu, zaproponowano tez zmiany, które uwzględnią nie tylko komunikację, lecz archiwizację już odebranych danych.

Trzeci, ostatni podpunkt skupia się na wizualizacji danych. Oprócz dokładnej analizy przykładowej aplikacji zdolnej tworzyć wykresy liniowe z danych zapisanych w pliku

tekstowym, wspomniano też o innych programach zdolnych przeprowadzać takie operacje. Również tu zaproponowano możliwe modyfikacje aplikacji, a dokładniej jej możliwe przebudowanie, aby dostosować ją do wyświetlania wykresów z danych zebranych podczas archiwizacji danych.

Praca jest studium wykonalności systemu archiwizacji i wizualizacji danych pochodzących z symulatora lotu. Jest to więc koncepcja różnych możliwości i opcji, które można zastosować, aby móc się połączyć z symulatorem lotu, poprawnie odebrać dane, zarchiwizować je, a na końcu wizualizować je w formie wykresu liniowego. Zamiarem autora było zgromadzenie i analiza programów oraz technik, które po odpowiednich modyfikacjach można będzie zintegrować w system spełniając postawione wymagania.

2. Strumień danych generowanych przez symulator ALSIM 200 MMC. [1*]

Symulator ALSIM 200 MMC to jeden z najpopularniejszych symulatorów lotu tej firmy. Głównym powodem popularności tego symulatora lotu jest jego szeroka możliwość konfiguracji – potrafi symulować samoloty jednosilnikowe, dwusilnikowe, a nawet lekkie odrzutowce. Instrumenty pokładowe są w pełni skomputeryzowane, w dodatku aerodynamika jest odwzorowana bardzo realistycznie. Sprawia to, że zdarzenia w symulatorze lotu nie odbiegają wiele od tego, jak samolot zachowałby się w rzeczywistości. Symulator AL200 MCC służy głównie do symulowania lotów na niższych, ograniczonych wysokościach. Podczas takiej symulacji pilot skupiony jest głównie na instrumentach pokładowych.

Całe oprogramowanie w symulatorze składa się z sześciu programów, które komunikują się między sobą. Są to:

-IO.exe, który służy do łączenia elektronicznych komponentów z pozostałą piątka programów (a więc odpowiada za interfejs wejścia-wyjścia).

-Pannes.exe, który służy do symulowania systemów awioniki.

-PosteInstructeur.exe, który służy do kontrolowania położenia samolot, jak również wiatr i awarie, które mogą zajść na symulowanym modelu.

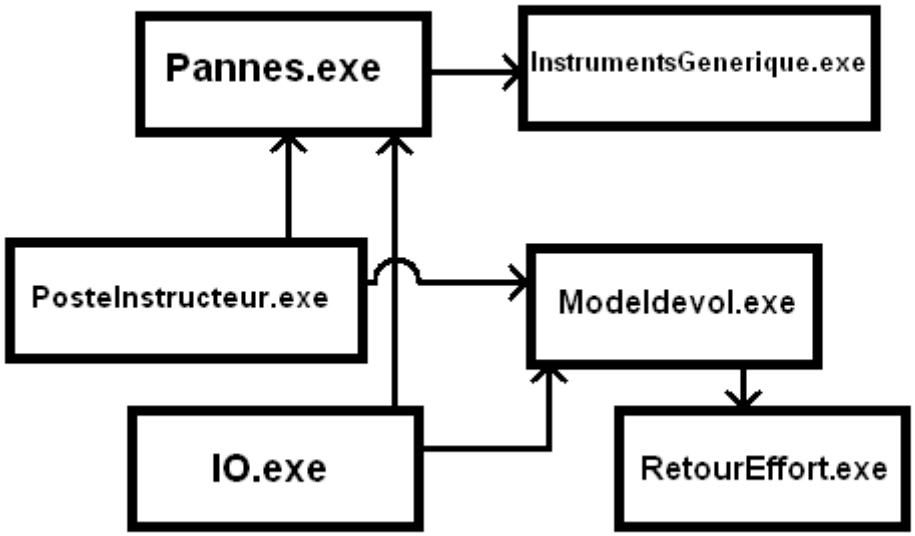
-Modeledevol.exe, który służy do obliczania parametrów związanych z silnikiem symulowanego modelu, jak również parametry samego lotu.

-RetourEffort.exe, który odpowiada za nadzór systemu kontroli przeciążeń.

-InstrumentsGeneriques.exe, który odpowiada za rysowanie instrumentów pokładowych.

Programy te komunikują się w schemacie przedstawionym na Rys. 2.1.

Dane te są przesyłane między komputerami (każdy program jest uruchamiany na osobnym komputerze) dzięki lokalnej sieci komputerowej UDP/IP/Ethernet. Wszelka wymiana danych odbywa się rozgłoszeniowo. Jest to kluczowe dla konfiguracji symulatora, jak i tworzenia komunikacji. Liczba danych, jaka jest przesyłana między komputerami, jest bardzo duża i inne modele wymiany danych (np. TCP) mogłyby się okazać za wolne. Obecny system odbiera wszystkie dane, jednak nie ze wszystkich będzie korzystał. Nie każda bowiem informacja jest interesująca tudzież istotna dla przebiegu symulacji. Odebrane pakiety danych można jednak zidentyfikować. Każdy pakiet, oprócz danych, zawiera też czterobajtowy identyfikator, zwany ID. To właśnie po nim się identyfikuje i sprawdza, czy te dane są istotne, czy tez można je pominąć.



Rys.2.1 Schemat komunikacji programów składających się na symulator lotu ALSIM 200 MMC.

ID	DATA

Rys. 2.2 Struktura pakietu danych, który jest wysyłany przez oprogramowanie ALSIM 200 MCC.

Cały pakiet to 255 bajtów, gdzie ID zajmuje 4 bajty, a DATA (a więc informacje przesyłane w tym pakiecie) pozostałe 251 bajtów. (por rys. 2.2).

Dane, jakie mogą być przesyłane przez symulator lotu dotyczą różnych parametrów samolotu. Są one zapisane w strukturach, które z kolei zawierają poszczególne parametry. Jedną z ważniejszych struktur jest struktura SPosition, która odpowiada za położenie samolotu. Jej kod znajduje się poniżej.

```
struct  SPosition
{
   long lIDEngin;
   long lEtatEngin;
   double mat[4][3];
   double dPosNeutreProfondeur;
   double dPosNeutreDirection;
   double dPosNeutreInclinaison;
   float dFacteurGainProfondeur;
   float dFacteurGainDirection;
   float dFacteurGainInclinaison;
   float dVitesseGain;
   double hpaQFE;
   double ftHeight;
   double dAttitude;
   double dBank;
   double dMagneticHeading;
   double ftpmInertialVerticalSpeed;
   double ftpmVerticalSpeed;
   double ktAirSpeed;
   double ktMaxAirSpeed;
   double dSideSplip;
   double dBille;
   double dAiguille;
};
```

Parametrów w tej strukturze jest bardzo dużo, najważniejsze jednak to:

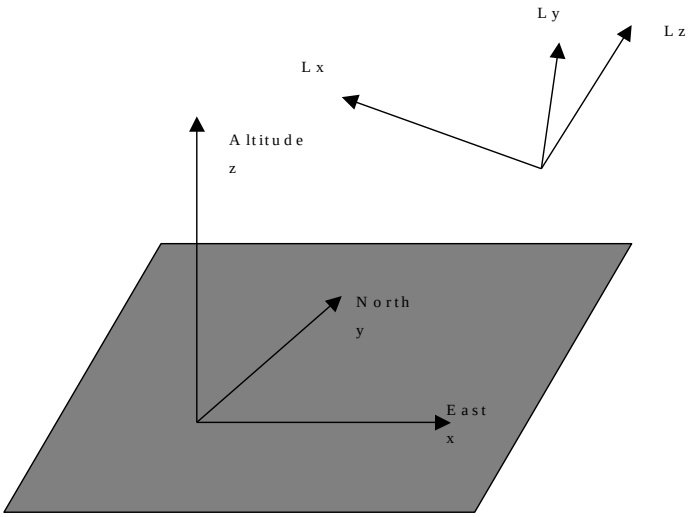
- `double mat[4][3]`, macierz ta (macierz o nazwie mat) odpowiada za położenie samolotu w przestrzeni, to jest współrzędne geograficzne, jak i wektory kierunkowe samolotu. Geograficzne współrzędne samolotu zawarte są w komórkach macierzy [3][0],[3][1],[3][2] (kolejno długość, wysokość i szerokość geograficzna). Z kolei wektory kierunkowe samolotu są zawarte w aż dziewięciu komórkach macierzy. Układ tych wektorów kierunkowych wygląda tak:

-Ly [0][0],[1][0],[2][0] odpowiada za wektor Ly zrzutowany kolejno na oś x,y i z.

-Lx [0][2],[1][2],[2][2] odpowiada za wektor Lx zrzutowany kolejno na oś x,y i z.

-Lz [0][1],[1][1],[2][1] odpowiada za wektor Lz zrzutowany kolejno na oś x,y i z.

Na rysunku 2.3 przedstawiono, jak poszczególne wektory odnoszą się względem kierunków geograficznych.



Rys. 2.3 Układ współrzędnych samolotu względem kierunków geograficznych (źródło [2]).

-`ftHeight`, wysokość samolotu. Jednostką są stopy.

-`dAttitude`, położenie samolotu, względem horyzontu. Jednostką są stopnie.

-`dBank`, przechylenie samolotu. Jednostką są stopnie.

-`dMagneticHeading`, kurs samolotu, podany względem bieguna magnetycznego Ziemi.

-`ftpmInertialVerticalSpeed`, prędkość wznoszenia się samolotu. Jest na wyświetlana na instrumentach w kokpicie. Jednostką są stopy na sekundę.

-`ftVerticalSpeed`, chwilowa prędkość wznoszenia. Jednostką są stopy na sekundę.

-`ktAirSpeed`, prędkość samolotu, która jest odczytywana bezpośrednio z instrumentów. Prędkość ta jest dostarczana przez system statycznych rurek pilota, a więc zawiera błędy pozycji oraz instrumentów. Jednostką jest węzeł (jest to jednostka prędkości, gdzie jeden węzeł jest równy jednej mili morskiej na godzinę).

-`dBille`, odchylenie nosa samolotu względem linii horyzontu. Jednostką są stopnie.

Drugą duża strukturą jest struktura SOtgFlightParameters, która zawiera dodatkowe parametry lotu. Jest ona przedstawiona poniżej:

```
struct SQtgFlightParameters
{
   double msDt;
   double dBank;
   double dAttitude;
   double dMagneticHeading;
   double dTrueHeading;
   double dMagneticCourse;
   double dTrueCourse;
   double ftPressureAltitude;
   double ftHeight;
   double hpaQFE;
   double hpaQNH;
   double ktTAS;
   double ktIAS;
   double fpmVz;
   double dIncidence;
   double dDerapage;
   double dDeltaIsa;
   double dWindDirection;
   double ktWind;
   double dProfondeur;
   double dInclinaison;
   double dDirection;
   double dTrimProfondeur;
   double dTrimDirection;
   double dTrimInclinaison;
   double percCoupleMoteurProfondeur;
   double percCoupleMoteurInclinaison;
   double percCoupleMoteurDirection;
   double dFlaps;
   double dGear;
};
```

Ważniejsze parametry tej struktury to:

-`dBank`, przechylenie samolotu. Jednostką są stopnie.

-`dAttitude`, położenie samolotu (dokładniej odchylenie) względem horyzontu. Jednostką są stopnie.

-`dMagneticHeading`, kurs samolotu względem bieguna magnetycznego Ziemi. Jednostką są stopnie.

-`dTrueHeading`, geograficzny kurs samolotu. Różni się on od kursu względem bieguna magnetycznego. Różnica ta zależna jest od położenia samolotu na Ziemi (na

przykład, dla terytorium Polski geograficzny kurs różni się od kursu względem bieguna magnetycznego Ziemi o cztery dodatnie stopnie). Jednostką są stopnie.

-dMagneticCourse, kurs względem bieguna magnetycznego. Pomimo, że jest to bardzo podobna wartość do tej zawartej w zmiennej dMagneticHeading, to jednak może się ona różnić. Ma to związek z wiatrem. Różnica ta pojawia się np. gdy samolot leci bokiem. Jednostką są stopnie.

-dTrueCourse, Zmienna uwzględniająca różnicę między geograficznym kursem samolotu a kursem samolotu względem bieguna magnetycznego Ziemi. Wyjściową nie jest jednak zmienna dMagneticHeading, a dMagneticCourse. Jednostką są stopnie.

-ftPressureAltitude, wysokość ciśnieniowa. Wartość ta jest uwzględniana na podstawie różnicy ciśnienia atmosferycznego na wysokości, gdzie jest samolot, a punktem odniesienia (którym najczęściej jest poziom morza)Jednostką są stopy.

-ftHeight, wysokość lotu samolotu. Należy przy tym sprawdzić poziom odniesienia, zwłaszcza jeśli jest inny niż poziom morza. Jednostką są stopy.

-hpaQFE, ciśnienie, mierzone względem ciśnienia na poziomie odniesienia. Poziomem odniesienia w tym przypadku jest lotnisko startu samolotu. Jednostką są hektopaskale.

-hpaQNH, ciśnienie, mierzone względem ciśnienia na poziomie odniesienia. Poziomem odniesienia w tym przypadku jest poziom morza. Jednostką są hektopaskale.

Różnicą między hpaQFE a hpaQNH jest fakt, że w przypadku hpaQFE wysokościomierz barometryczny wskazuje wysokość względną, a w przypadku hpaQNH – bezwzględną.

-ktTAS, prawdziwa prędkość samolotu (uwzględniany więc jest nie tylko wektor prędkości samolotu, ale i wektor prędkości wiatru). Jednostką są węzły.

-fpmVz, prędkość wznoszenia się samolotu. Jednostką są stopy na minutę.

-dWindDirection, kierunek wiatru. Jednostką są stopnie.

-dWindSpeed, prędkość wiatru. Jednostką są węzły.

Jak widać, niektóre zmienne, jak dBank, powtarzają się w obu głównych strukturach. Ważne więc jest, aby poprawnie identyfikować zmienne odbierane z sieci rozgłoszeniowej.

Inne, mniejsze struktury, również znajdują się w symulatorze lotu. Jedną z nich jest struktura SWind, która odpowiada za wiatr. Jej wygląd pokazany jest poniżej:

```
struct SWind
{
  double dWindLevel1;
  double dWindLevel2;
  double dWindAngle[3];
  double dWindSpeed[3];
};
```

Obecne w niej parametry to:

- dWindlevel1, górna granica pierwszej strefy wiatru.

- dWindlevel2, górna granica drugiej strefy wiatru.

- dWindAngle[3], niewielka tablica, która zawiera kierunki wiatru dla każdej z trzech stref wiatru.

- dWindSpeed[3], niewielka tablica, która zawiera prędkości wiatru dla każdej z trzech stref wiatru.

Inna, pomniejszą (lecz jednak ważną) strukturą jest SOfe, która zawiera informacje o ciśnieniu. Również ona przedstawiona jest poniżej:

```
struct SQfe
{
  double hpaQFE;
  double hpaQNH;
  double ftHauteurSol;
};
```

-hpaQFE, ciśnienie, mierzone względem ciśnienia na poziomie odniesienia. Poziomem odniesienia w tym przypadku jest lotnisko startu samolotu. Jednostką są hektopaskale.

-`hpaQNH`, ciśnienie, mierzone względem ciśnienia na poziomie odniesienia. Poziomem odniesienia w tym przypadku jest poziom morza. Jednostką są hektopaskale.

-`ftHauteurSol`, wysokość samolotu nad poziomem gruntu, który jest pod samolotem. Jednostką są stopy.

Oczywiście struktur jest tu bardzo wiele, jak np. `SDoubleGroup`, czy też `SIOButton`, jednak nie ma potrzeby omawiania ich wszystkich.

Warto wspomnieć, że niektóre wiadomości nie są przesyłane w strukturach, lecz jako pojedyncze zmienne typu double. Są to np. informacje o poziomie chmur czy też oblodzeniu.

Wszystkie te informacje są odbierane ze strumienia danych, jaki przesyłany jest między programami składowymi symulatora lotu, jak i pomiędzy serwerem a klientem (czyli nasza komunikacja z symulatorem). Dane te, odpowiednio odebrane, można używać do wielu celów. Koncepcja ta jednak skupi się na odpowiednim pokazaniu tych danych, jak i zapisaniu ich.

3. Koncepcja modułu programowego do archiwizacji wizualizacji danych.

3.1 Schemat bazy danych do przechowywania parametrów lotu.

Dane odbierane z symulatora lotu są liczne, a więc trudne do posegregowania. Jest to szczególnie istotne, ponieważ w wielu przypadkach skupiamy się na danych odbiegających od normy (na przykład zbyt niskie ciśnienie, za mała wysokość i tym podobne). Istotny więc będzie taki system segregowania danych, który pozwoli szybko znaleźć konkretne dane wśród wszystkich odebranych.

Jednym z popularnych systemów odnajdywania i segregacji danych jest utworzenie bazy danych. Poprawnie utworzona baza danych pozwala, dzięki odpowiednim zapytaniom SQL, szybko odnaleźć niemal dowolne dane i niemal dowolną wartość (lub uzyskać informację o braku takiej wartości wśród danych). Utworzenie bazy danych zaczyna się od utworzenia diagramu ERD[3]. Jest to szczególnie istotny punkt, ponieważ nawet drobny błąd w koncepcji projektowania bazy danych (a więc w diagramie ERD), może potem

spowodować poważne problemy, w tym potrzebę przebudowania niemal całego systemu bazy danych.

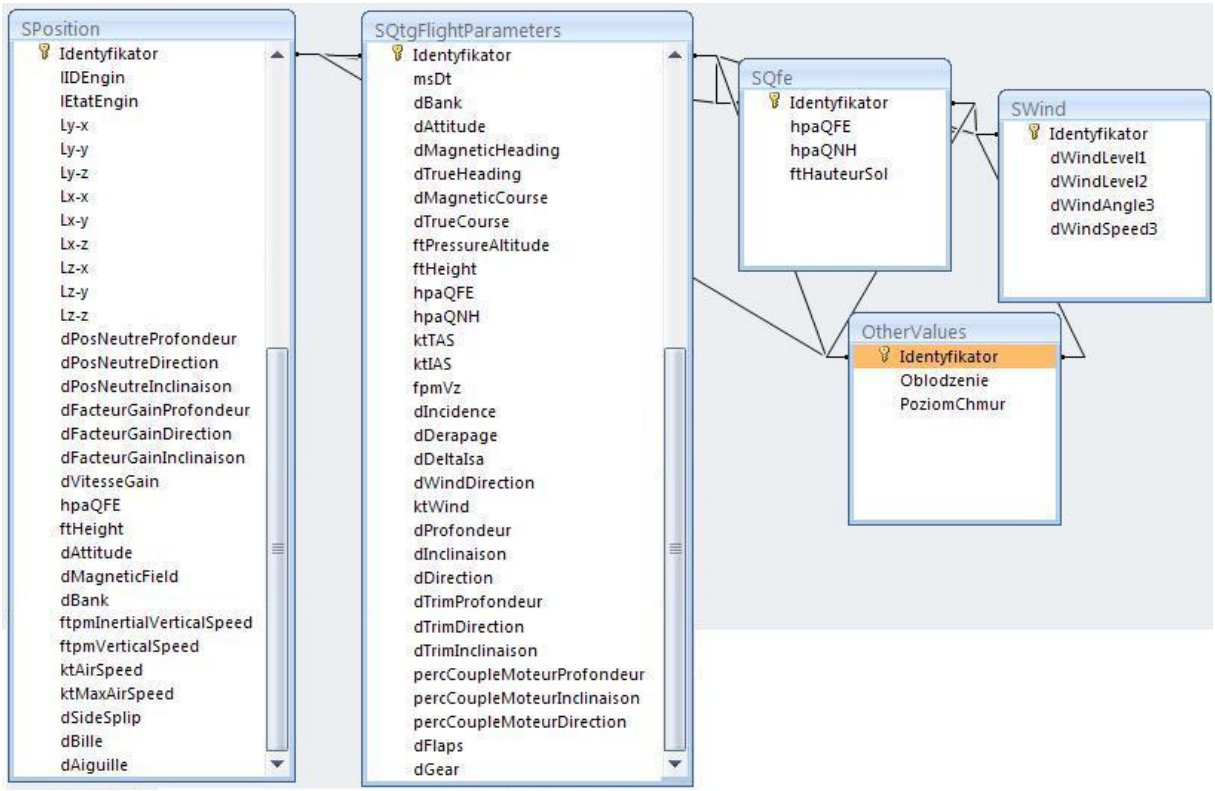
Na szczęście, w tym przypadku utworzenie takiego diagramu nie jest trudne, jako że między wartościami nie ma wiele zależności (to jest, wartości w większości przypadków są niezależne od siebie). Należy jednak zaznaczyć, że z powodu wielu typów wartości, tabele będą mieć bardzo dużo kolumn. Utworzenie takiej bazy danych może więc być czasochłonne, pomimo swojej prostoty.

Koncepcyjnie, najprościej potraktować każdą strukturę omówioną w rozdziale drugim jako tabelę, natomiast każdą wartość jako kolumnę. Każda tabela wymaga jednak klucza głównego. Może to być cokolwiek, musi jednak mieć unikatową wartość (wartości pod żadnym względem nie mogą się powtarzać). Dla pewności warto więc utworzyć dodatkową kolumnę – czas. Pewnym bowiem jest, że w danym czasie może zaistnieć tylko jedna wartość. Innym przykładowym kluczem głównym może być kolumna ID, która będzie przyjmować inkrementowaną o jeden liczbę całkowitą, zaczynając od 1 (to jest, 1,2,3 i tak dalej). Samo ID jest mniej czytelne niż podanie czasu danego zdarzenia, jednak z powodu dużej ilości odbieranych danych, podanie dokładnego czasu (który najłatwiej pobrać z zegara systemowego) może być problematyczne. Zastosowanie ID pozwala więc całkowicie obejść ten problem.

Na rys. 3.1-1 jest przedstawiony diagram ERD dla bazy danych, która można zastosować (przykład z Identyfikatorem jako kluczem głównym). Należy pamiętać, że:

-Przykład z Datą i godziną jako kluczem głównym jest prawie identyczny – wystarczy zmienić Identyfikator na Datę. Decyzję tą należy jednak podjąć już w fazie projektowania.

-Aby zgromadzić wszystkie dane, należy zamieścić wszystkie zmienne pobrane z symulatora samolotu. Mimo że ten diagram uwzględnia najważniejsze dane, nie są to jednak wszystkie dane. Schemat wstawiania tabel do bazy danych jest jednak taki sam: nawą tabeli jest struktur, gdzie są dane, a kolumnami są same dane. Należy też pamiętać o identyfikatorze/dacie.



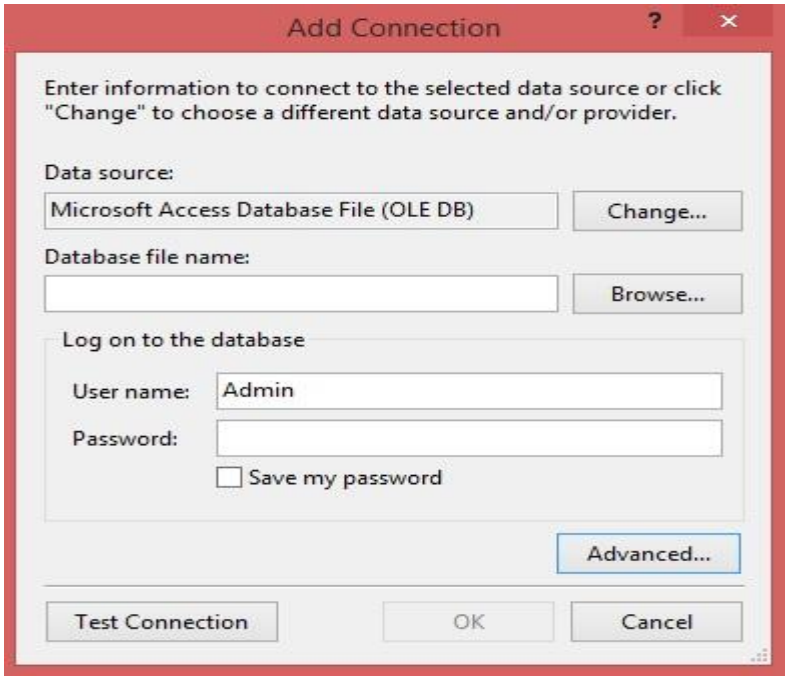
Rys. 3.1-1. Diagram ERD dla bazy danych przechowującej odbierane dane (przykład z Identyfikatorem jako kluczem głównym).

Co można zauważyć na Rys. 3.1-1, to fakt, że niektóre dane się powtarzają (np. zmienna hpaQFE występuje w dwóch tabelach). Choć nie jest to optymalne rozwiązanie, nie jest również błędne. Uniknie to też problemów, gdy pomimo, że zmienna dotyczy tej samej wartości, w obu strukturach będzie mieć inną wartość. Nie ma też problemu, by zmienne i tabele nazywały się inaczej niż ich odpowiednik w programie. Przykładem jest tabela OtherValues, gdzie będą wstawione wartości, które nie są obecne w żadnej strukturze, a jedynie jako pojedyncze wartości. Innym widoczny zjawiskiem jest relacja między wszystkimi tabelami na poziomie zmiennej Identyfikator. Wynika to z tego, że bardzo istotne jest, aby jeden identyfikator dotyczył **wszystkich** danych w bazie danych, które w danej chwili zostały odebrane. (np. dane o identyfikatorze 10000 w tabeli SPosition i dane o identyfikatorze 10000 w tabeli SQfe to dane z tej samej chwili). Identyfikator dotyczy bowiem każdego osobnego przesłania danych (w przypadku schematu z datą – danego momentu, w którym dane wysłano). Nawet jeśli przesłano tylko jedną daną, takie zdarzenie ma wszędzie osobny identyfikator. Nie jest to jednak problemem, albowiem komórki gdzie mają być wpisane dane można zostawić puste lub identyczne jak z poprzedniego przesłania danych. W przypadku z użyciem daty i godziny zamiast identyfikatora, również taka relacja

powinna być utworzona, aby program wiedział, że dane te przyszły w tym samym czasie, a więc są od siebie zależne.

W przypadku tworzenia bazy danych dobrze jest unikać w nazwach tabel i kolumn spacji, jak i polskich znaków. Może to bowiem tworzyć spore problemy w przypadku używania zapytań SQL (te z kolei służą do szukania informacji w bazie danych) lub w przypadku działania programów nieobsługujących polskich znaków (takie zjawisko jest jednak rzadkością).

Aby dane odebrane trafiły do bazy danych tak, jak powinny, należy najpierw utworzyć bazę danych (np. w programie Microsoft Access), z właściwymi tabelami i kolumnami, jak i typami danych w kolumnach (na szczęście dla nas, wszystkie dane są liczbami zmiennoprzecinkowymi). Należy również utworzyć wszystkie relacje. Taka baza danych może jednak być pusta (to jest, nie mieć żadnych danych). Następnie należy utworzyć połączenie między bazą danych a aplikacją odbierającą dane. W przypadku programowania w Microsoft Visual Studio ten proces zaczyna się od utworzenia tzw. Data Connection (to jest, połączenie z danymi z bazy danych). Nie jest to trudne, wystarczy podać ścieżkę do bazy danych i podać typ pliku, z którym się połączymy. Poniżej jest przedstawione okno tworzenia połączenia z danymi bazy danych:



Rys. 3.1-2 Okno dodawania połączenia z baza danych.

Poza wybraniem typu pliku z jakim ma się połączyć (można bowiem łączyć się też np. z Windows SQL Server czy serwerami bazy danych tworzonymi w Oracle), należy podać ścieżkę do pliku z bazą danych. Lokalizacja pliku jest o tyle istotna, że przeniesienie pliku może bowiem być problematyczne. Trzeba bowiem podać ścieżkę za każdym razem, nie tylko przy tworzeniu połączenia, ale również w kodzie samego programu. Przeniesienie pliku może więc spowodować potrzebę przebudowywania dużej ilości kodu.

Dzięki takiej operacji teraz wszystkie aplikacje mogą korzystać z tej bazy danych. Należy jednak połączenie to uwzględnić w kodzie (tak, aby aplikacja wiedziała, że ma się połączyć właśnie z ta bazą danych).

Połączenie to odbywa się dzięki interfejsowi OleDB. Jest to system połączenia baza danych-program obecny w Microsoft Visual Studio. Na początku należy użyć funkcji OleDBConnect. Służy ona do tego, aby utworzyć połączenie między bazą danych a programem. To w tym momencie program wie, że pozostałe operacje (a więc operacje SQL) będą odbywały się właśnie na tej, a nie innej bazie danych.

Kod takiego połączenia wygląda następująco:

```
System.Data.OleDb.OleDbConnection nazwa_zmiennej_OleDbConnect = new
System.Data.OleDb.OleDbConnection(@"Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=ściezka do pliku z bazą danych ");
```

Od teraz zmienna cnUsers odpowiada za połączenie i niektóre operacje będą się do tej zmiennej odwoływać. Również od tego momentu można pisać polecenia SQL. Najczęściej używane polecenia SQL to SELECT (wybiera informacje z bazy danych, według podanych kryteriów), INSERT INTO (wpisz informacje do bazy danych według określonych kryteriów), DELETE (działa podobnie jak SELECT, jednak znalezionych informacji nie wybiera, lecz usuwa). Komendy wpisane błędnie nie wykonają się. Przykładowym błędem może być usuwanie obowiązkowych danych (niektóre dane w wierszu mogą być obowiązkowe do wypełnienia. To, czy są obowiązkowe czy nie, definiuje się podczas fazy tworzenia bazy danych) czy też wpisanie błędnego formatu danych (np. próba wpisania tekstu tam, gdzie podajemy tylko liczby).

Następnie tworzy się zapytanie SQL. Koncepcja automatycznej archiwizacji zakłada, że pobrane dane nie będą w żaden sposób modyfikowane, a jedyna związana z nimi operacja

to archiwizacja. Interesuje więc nas zapytanie INSERT INTO, która służy do wstawienia danych.

Kod pozwalający zastosować komendę INSERT INTO wygląda następująco:

```
System.Data.OleDb.OleDbCommand nazwa_zmiennej_komendySQL = new
System.Data.OleDb.OleDbCommand("INSERT INTO nazwa_tabeli(dane, które chcemy
wstawić do tabeli, oddzielone przecinkiem) VALUES (@nazwa zmiennych z
wartościami, które chcemy wstawić, oddzielone przecinkiem)",
nazwa_zmiennej_OleDbConnect);
```

Można też zamiast parametrów podać stałe wartości, jednak wtedy wstawiana wartość będzie zawsze taka sama.

Następnie należy przypisać parametry. Parametry są o tyle istotne, jeśli konieczne jest szukanie/usuwanie/wstawianie według określonych parametrów, które mogą być zmienne w czasie (np. manualne szukanie danych po ID). W przypadku naszych założeń, a więc gromadzenia danych z symulatora lotu i archiwizowania ich, parametrem będzie zmienna odebrana z symulatora lotu.

Sam przykład kodu do zainicjowania parametru wygląda następująco:

```
    sSelect.Parameters.AddWithValue("@nazwa zmiennej w zapytaniu SQL",
zmienna_z_danymi, które mają trafić do tabeli);
```

Parametrów powinno być tyle, ile zdefiniowaliśmy zmiennych w zapytaniu SQL powyżej. Nie ma przeciwwskazań, aby zmienną z danymi były dane pobrane z symulatora lotu. Przykładowy program pobiera jednak dane z symulatora lotu w postaci tekstu (patrz podrozdział 3.2), wymagana więc będzie konwersja z typu danych string na float tudzież double.

Jeden parametr – ID, nie będzie pobierany z symulatora lotu. Jego wartość ma być inkrementowana o 1 (to jest, każdy kolejny wiersz ma wartość ID o jeden większą niż poprzedni). Należy więc, po każdym dopisaniu kolejnego wiersza, zwiększyć wartość zmiennej odpowiedzialnej za ID o 1:

```
    sSelect.Parameters.AddWithValue("@Identyfikator", ID);

    ID++;
```

Należy jednak uprzednio podać wartość początkową równą 1 (lub 0, jeśli inkrementacja zachodzi przed wpisaniem wiersza do bazy danych). Takie rozwiązanie może powodować problemy przy wyłączeniu programu. Jego ponowne uruchomienie spowoduje powrót wartości ID na 1, a więc program będzie próbował tworzyć wiersze z istniejącym ID, co może powodować błędy. Aby temu zapobiec, można czyścić bazę danych przed dodawaniem danych (zapytaniem SQL DELETE), lub odczytać ostatnią wartość ID z bazy danych (ostatnia jest również maksymalną wartością) i użyć jak początkową wartość ID. Można to przykładowo dokonać zapytaniem SQL SELECT, aby wyszukał wszystkie obecne wiersze (wystarczy podać mu gwiazdkę jako kryterium, po którym ma wyszukiwać). Dzięki temu, gdy zwrócimy ilość wyszukanych wierszy i przypiszemy ją do zmiennej, uzyskamy informację o tym, ile jest wierszy w bazie danych. Ta informacja pozwala nam się więc dowiedzieć, jakie jest największe ID (jako że każdy kolejny wiersz ma ID większe o 1). Kod do przeprowadzenia takiej operacji może wyglądać następująco

```
int Y;

System.Data.OleDb.OleDbCommand nazwa_zmiennej_komendySQL);  = new
System.Data.OleDb.OleDbCommand("SELECT * FROM nazwa_tabeli WHERE
ID=@IdInkrementowane", nazwa_zmiennej_OleDbConnect);

System.Data.OleDb.OleDbDataAdapter nazwa_zmiennej_adaptera = new
System.Data.OleDb.OleDbDataAdapter(nazwa_zmiennej_komendySQL);
//OleDbAdapter jest potrzebny, aby móc uzyskać wartość, którą zwraca
komenda SELECT.

  DataSet zmienna_datasetu = new DataSet();//Tworzenie nowego DataSetu

  nazwa_zmiennej_adaptera.Fill(zmienna_datasetu); //DataSet zostaje
wypełniony danymi z bazy danych. Również taka operacja jest potrzebna, aby
uzyskać wartość, którą zwraca zapytanie SQL SELECT.

  Y = zmienna_datasetu.Tables[0].Rows.Count; //Tutaj, do zmiennej Y,
zostanie przypisana ilość odnalezionych wierszy.
```

Tak przygotowany system jest w stanie samodzielnie gromadzić dane w bazie danych. Pozostaje problem automatycznego generowania pliku tekstowych z bazy danych. Najłatwiejszym rozwiązaniem jest stworzenie pętli, która będzie odbierać kolejne wartości z bazy danych, a następnie zapisywać te dane do liku tekstowego. Rozwiązanie to jest jednak możliwe tylko wtedy, gdy jako klucz główny zastosujemy nie datę, lecz identyfikator. Daty

nie można bowiem równomiernie inkrementować, ponieważ dane nie będą pobierane z systemu regularnie.

Wykonanie tych wszystkich operacji może być niemalże niemożliwe, jeśli nie ma kolumny nazwanej ID. Jeśli jednak zależy nam na obecności daty przy każdym odebraniu danych z symulatora lotu, można ustawić ID jako klucz główny, a datę jako zwykłą kolumnę.

Wszystkie te operacje, a więc przykładowe programy i zapytania SQL, należy zastosować w programie zbierającym dane z symulatora. Przykładowy program, który odbiera dane z symulatora lotu, został omówiony w rozdziale 3.2

3.2 Koncepcja serwera pakietów UDP zgodnego z WinSock 2.

Prawdopodobnie najważniejszą częścią projektowanej aplikacji jest utworzenie połączenia, które pozwoli odbierać dane z sieci komputerowej poprzez protokół UDP[4]. Choć nie jest on tak bezpieczny jak podobne mu protokoły (np. TCP/IP), jest zdecydowanie szybszy. Wynika to z tego, że jedynym zabezpieczeniem w przypadku przesyłania pakietów jest suma kontrolna, poza tym nie czeka się na potwierdzenie, czy pakiet dotarł do celu.

Sam protokół UPD działa w sposób następujący:

-Najpierw tworzony jest serwer. Serwer ten, po otworzeniu, otwiera połączenie (to jest, socket) i czeka na klientów.

-Gdy klient połączy się z serwerem, wtedy serwer tworzy nowe połączenie, identyczne do poprzedniego. Nowe połączenie jest socketem między serwerem a klientem, nikt inny nie ma więc do niego dostępu.

-Następnie dane są wymieniane tak długo, aż jedna ze stron (najczęściej klient) nie zamknie połączenia.

-Gdy połączenie się zakończy, socket do połączenia z tym klientem znika, a serwer znów wyszukuje klientów.

-Gdy serwer przestanie działać, całe połączenie zanika i operację trzeba powtarzać od początku.

Serwer jest w stanie obsługiwać wiele klientów naraz, jednak każdy z nich ma osobny socket. Oznacza to, że każde połączenie z klientem jest osobne i klienci mogą nie wiedzieć, że łączą się z tym samym serwerem.

Choć programowanie połączenia przez protokół UDP zaczęło się na systemach UNIX, również do Windowsa szybko utworzono rozwiązania pozwalające używać tego protokołu. Biblioteki to obsługujące są zawarte w specyfikacji WinSock. Obecna tutaj koncepcja i przykładowy program testowy korzysta z rozwiązań zawartych w WinSock 2[5], który jest przeznaczony do nowszych systemów operacyjnych.

Podczas realizacji studium wykonalności systemu akwizycji i wizualizacji danych symulatora znaleziono gotowe rozwiązanie aplikacji ilustrującej technikę tworzenia systemu komunikacji w oparciu o protokół UDP [6]. W dalszej części rozdziału wskazano najważniejsze fragmenty programu i omówiono ich działanie.

Sam program testowy jest typowym programem typu .exe. Składa się on jednak z dwóch części – klienta i serwera. Dopiero uruchomienie obu programów pozwala sprawdzić połączenie. Do testów zastosowano połączenie na adresie 127.0.0.1, który jest adresem IP przeznaczonym do połączeń wewnątrzsystemowych (to jest, bez łączenia się z inną maszyną).

Program testowy jest prawdopodobnie najważniejszą częścią całego systemu akwizycji i wizualizacji danych, dlatego jego kod zostanie poniżej dokładnie omówiony, zaczynając od programu tworzącego serwer:

```
 /*
    Simple UDP Server
    Silver Moon ( m00n.silv3r@gmail.com )
*/
```

Program testowy został utworzony przez Silver Moon. Na początku programu jest więc zamieszczona o tym informacja, wraz z kontaktem mailowym do twórcy tegoż programu

```
#include<stdio.h>
#include<winsock2.h>
#pragma comment(lib,"ws2_32.lib") //Winsock Library
#define BUFLEN 512  //Max length of buffer
```

```
#define PORT 8888    //The port on which to listen for incoming data
```

Przed rozpoczęciem głównej cześci programu, czyli int main(), do programu dołączono pliki nagłówkowe bibliotek „stdio” oraz „winsock2”. Za pomocą makrodefinicji zdefiniowano długość bufora (`#define BUFLEN 512`) wraz z numerem portu, na którym serwer będzie nasłuchiwał klientów (`#define PORT 8888`). Maksymalna długość bufora wynosi tutaj 512 znaków, zaś port ma numer 8888.

Zastosowana dyrektywa kompilatora - `#pragma comment(lib,"ws2_32.lib")` służy do wskazania na konkretny plik w bibliotece. W przeciwnym przypadku, program może nie odczytać pliku z bibliotek WinSock2, lub odczytać biblioteki dotyczące innych wersji Winsocka. Plik `ws2_32.lib` odwołuje się do WinSock2 w wersji 32-bitowej, co obecnie jest najpopularniejszym rozwiązaniem. Główna funkcja programu ma postać:

```
int main()
{
    SOCKET s;
    struct sockaddr_in server, si_other;
    int slen , recv_len;
    char buf[BUFLEN];
    WSADATA wsa;

    slen = sizeof(si_other) ;
```

Następnie, już w ciele funkcji main(), zaczyna się deklaracja zmiennych, które będą używane w programie. Niektóre zmienne mają standardowe typy (int, char, struct), jednak inne zmienne mają unikatowe typy, wymagające głębszego omówienia.

Pierwsza jest zmienna s typu SOCKET. Ta zmienna, nieco później, zostanie całym socketem, który będzie używany do komunikacji między serwerem a klientem. Jednak zanim będziemy mogli taki socjet utworzyć, musimy zainicjalizować całą bibliotekę WinSock. Do tego będzie potrzebna zmienna wsa o typie WSADATA. Będzie ona gromadzić informacje o WinSocku, gdy już WinSock zostanie zainicjowany.

Na końcu użyto też operatora rozmiaru sizeof. Służy on po to, aby zmienna slen miała tą samą wartość, co rozmiar struktury si_other. Struktura si_other z kolei przechowuje informacje o kliencie, takie jak jego adres IP czy port. Można więc, dzięki operatorowi sizeof, pobrać informację o ilości znaków wysłanych przez klienta i tą właśnie informację przechowuje zmienna slen.

```
printf("\nInitialising Winsock...");
    if (WSAStartup(MAKEWORD(2,2),&wsa) != 0)
    {
        printf("Failed. Error Code : %d",WSAGetLastError());
        exit(EXIT_FAILURE);
    }
printf("Initialised.\n");
```

Po wypisaniu informacji, że WinSock jest inicjalizowany, pojawia się warunek if. W warunku jest uruchamiana funkcja WSAStartup, wraz z dwoma argumentami (funkcja uruchomi się niezależnie od spełnienia tego warunku). Funkcja ta służy do inicjalizacji WinSocka. Pierwszy argument mówie funkcji, jaka wersja WinSocka będzie inicjalizowana; w tym przypadku jest inicjalizowana wersja 2.2. Druga zmienna to wskaźnik zmiennej, gdzie dodatkowe informacje będą przechowywane (czyli wcześniej wspomniana zmienna wsa). Warunek sprawdza, czy zaistniał błąd w inicjalizacji. W przypadku gdy błąd zaistniał, następuje wypisanie błędu wraz z jego kodem, a następnie wyjcie z programu. Jeśli natomiast błąd nie zaistniał, zostaje wypisana informacja o udanej inicjalizacji WinSocka.

```
if((s = socket(AF_INET , SOCK_DGRAM , 0 )) == INVALID_SOCKET)
    {
        printf("Could not create socket : %d" , WSAGetLastError());
    }
    printf("Socket created.\n");
```

W kolejnym warunku typu if uruchamiana jest funkcja socket, która pozwala nam socket utworzyć. Posiada ona trzy parametry, gdzie pierwsza to nazwa rodziny protokołów,

druga to typ przesyłania danych, zaś trzecia to rodzaj protokołu (dla serwera wpisywane jest tu 0). Ten socket obsługuje protokoły Internetu (AF_INET), które przesyłają informacje data gramowo (SOCK_DGRAM).

Jeśli warunek zostaje spełniony, oznacza to obecność błędu podczas tworzenia socjeta. Wypisywana jest stosowna informacja wraz z numerem błędu. W przeciwieństwie do poprzedniej części programu (inicjalizacja WinSocka), istnienie błędu nie powoduje zamknięcia programu. Jeśli natomiast warunek nie jest spełniony, zostaje wypisana informacja o pomyślnym utworzeniu socketu.

```
  //Prepare the sockaddr_in structure
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons( PORT );
```

Po utworzeniu socketa, tworzona jest struktura sockaddr_in. Struktura ta zawiera dane o serwerze i w jaki sposób się łączy (a więc np. jego IP czy port) z interfejsem sieciowym.

```
//Bind
    if( bind(s ,(struct sockaddr *)&server , sizeof(server)) == SOCKET_ERROR)
    {
        printf("Bind failed with error code : %d" , WSAGetLastError());
        exit(EXIT_FAILURE);
    }
    puts("Bind done");
```

Następna funkcja, bind, służy do przypisania adresu i portu do danego socketu. Dzięki temu wiadomo, jaki adres i port socket posiada. Pierwszy argument to zmienna, na której jest utworzony socket (a więc s), drugi argument to struktury, które mają adres własny (czyli struktura przygotowana wcześniej – sockaddr_in), wpisany jako wskaźnik. Trzeci argument to rozmiar struktury, która zawiera adres własny.

Gdy przypisanie adresu się nie uda, warunek jest spełniony i tak jak w poprzednich przypadkach wypisywany jest komunikat o błędzie wraz z numerem błędu. Tym razem

jednak następuje wyjście z programu. Jeśli natomiast przypisanie adresu się uda, wypisywana jest informacja o sukcesie.

```
//keep listening for data
    while(1)
    {
        printf("Waiting for data...");
        fflush(stdout);

        //clear the buffer by filling null, it might have previously
received data
        memset(buf,'\0', BUFLEN);
```

Po przywiązaniu adresu do socketa, zaczyna się nasłuchiwanie klienta. Moment ten się kończy dopiero przy wyłączeniu programu, stąd pętla nieskończona while(1). Początkowo, następuje wypisanie informacji o czekaniu na informacje. Następnie, funkcja memset, gdzie czyszczony jest bufor pamięci (dokładniej, to jest wypełniany znakami nowej linii). Pierwszy argument wskazuje na to, gdzie jest pierwsza komórka pamięci (w tym przypadku poczatek bufora), drugi argument to znak jakim ma być zmienna wypełniona (wspomniany znak pusty: \0), zaś trzecia podaje wielkość pamięci, która ma być wypełniona (w tym przypadku rozmiar bufora, wynoszący 512). Dzięki temu usuwa się dane, które mogły zostać z poprzedniego przesyłania danych.

```
        //try to receive some data, this is a blocking call
        if ((recv_len = recvfrom(s, buf, BUFLEN, 0, (struct sockaddr *)
&si_other, &slen)) == SOCKET_ERROR)
        {
            printf("recvfrom() failed with error code : %d" ,
WSAGetLastError());
            exit(EXIT_FAILURE);
        }
```

Tutaj jest odbieranie danych, wraz ze sprawdzeniem, czy się udało. Służy do tego funkcja recvfrom, która ma aż pięć argumentów. Pierwszy z nich to zmienna odpowiedzialna

za wcześniej utworzony socket (a więc s), drugi to bufor na otrzymywane dane (zmienna buf), zaś trzecim jest długość bufora (zmienna BUFLEN). Czwarty argument to flaga wpływająca na działanie funkcji (tutaj zastosowano standardowe działanie, a więc wpisano 0). Piąty argument to wskaźnik na strukturę, gdzie są przechowywane dane o połączeniu (używana już wcześniej struktura sockaddr, tym razem jednak odnosi się do si_other, a więc dane klienta o połączeniu). Ostatni argument to wskaźnik wskazujący na rozmiar wiadomości od klienta (zdefiniowane na początku programu slen).

Jak w poprzednich warunkach if, jeśli zostaje spełniony, oznacza to błąd. Oznacza to, że zostanie wypisana o tym informacja z numerem błędu, wraz z wyłączeniem programu. Jeśli warunek if nie jest spełniony, nic się nie dzieje (a wiec zostaje wykonywany dalszy kod).

```
  //print details of the client/peer and the data received

        printf("Received packet from %s:%d\n", inet_ntoa(si_other.sin_addr), ntohs(si_other.sin_port));

        printf("Data: %s\n" , buf);
```

Jeśli dane zostały odebrane poprawnie, dwie funkcje printf wypisują odebrane dane, jak i informację skąd dane zostały odebrane (to jest, ID i port klienta).

```
//now reply the client with the same data

        if (sendto(s, buf, recv_len, 0, (struct sockaddr*) &si_other, slen) == SOCKET_ERROR)

        {

            printf("sendto() failed with error code : %d" , WSAGetLastError());

            exit(EXIT_FAILURE);

        }

}
```

Po odebraniu danych, wysyłane są one powrotem do klienta, dzieki funkcji send to. Parametry są podobne do funkcji recvfrom, poza trzecim. Trzeci parametr to wartość, jaką poprzednio wykonana funkcja recvfrom przyjęła i została zapisana w zmiennej recv_len. Również tutaj, w przypadku błędu, zostaje wysłana wiadomość o błędzie, wraz z jego numerem, jak i następuje zamknięcie programu.

Ostatnia klamra zamyka wcześniej tworzoną pętlę nieskończoną While(1).

```
    closesocket(s);
    WSACleanup();
    return 0;
}
```

Gdy program zakończy działanie, należy zamknąć socket funkcją closesocket (parametr to socket, jaki ma zostać zamknięty), jak również odpalić bezparametrową funkcję WSACleanup(), która porządkuje ustawienia systemowe po działaniu całego WinSocka. Dzięki temu niepotrzebne operacje i pliki w rejestrze zostaną usunięte. 0 to wartość, którą zwraca główna funkcja programu main, przy użyciu instrukcji „return 0".

Kod dotyczący programu, który ma klient nie różni się zbyt znacznie od kod programu serwera. Przede wszystkim jest mniej złożony, albowiem nie trzeba się martwić o zakładanie serwera. Mimo to między tymi programami zachodzi kilka znacznych różnic.

```
/*
    Simple udp client
    Silver Moon (m00n.silv3r@gmail.com)
*/
#include<stdio.h>
#include<winsock2.h>

#pragma comment(lib,"ws2_32.lib") //Winsock Library

#define SERVER "127.0.0.1"  //ip address of udp server
#define BUFLEN 512  //Max length of buffer
#define PORT 8888   //The port on which to listen for incoming data
```

Na początku definiowane są zmienne i biblioteki. Jedyną różnicą jest `#define SERVER "127.0.0.1"` gdzie przechowywane jest IP serwera (w tym przypadku program służy do testów, a więc IP jest stałe i wynosi 127.0.0.1.

```
int main(void)
{
    struct sockaddr_in si_other;
    int s, slen=sizeof(si_other);
    char buf[BUFLEN];
    char message[BUFLEN];
    WSADATA wsa;
```

Również tu większość zmiennych jest taka sama. Jest tu jednak tylko jedna struktura (`struct sockaddr_in si_other`), jako że klient nie potrzebuje informacji o połączeniu z serwerem (potrzebuje tylko IP, które jest już zdefiniowane). Jest też nowa zmienna – tablica znaków `char message[BUFLEN]` , gdzie będzie przechowywana wiadomość wysyłana do serwera.

```
  //Initialise winsock
    printf("\nInitialising Winsock...");
    if (WSAStartup(MAKEWORD(2,2),&wsa) != 0)
    {
        printf("Failed. Error Code : %d",WSAGetLastError());
        exit(EXIT_FAILURE);
    }
    printf("Initialised.\n");
```

Również tu trzeba inicjalizować WinSocka. Wszystko jednak zachodzi identycznie jak w przypadku serwera.

```
//create socket
    if ( (s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == SOCKET_ERROR)
    {
        printf("socket() failed with error code : %d" , WSAGetLastError());
        exit(EXIT_FAILURE);
```

```
}
```

Również po stronie klienta socket musi zostać utworzony. Wszystko odbywa się tak, jak po stronie serwera, z paroma drobnymi różnicami. Ostatni argument funkcji socket nie wynosi 0, lecz IPPROTO_UDP. Zdefiniowany jest więc rodzaj protokołu dla socketa. Drugą drobną różnicą jest fakt, że w programie nie jest wypisywany fakt o tym, że socket został poprawnie utworzony.

```
//setup address structure
    memset((char *) &si_other, 0, sizeof(si_other));
    si_other.sin_family = AF_INET;
    si_other.sin_port = htons(PORT);
    si_other.sin_addr.S_un.S_addr = inet_addr(SERVER);
```

Również w przypadku klienta, przygotowuje się strukturę sockaddr_in. Wcześniej trzeba jednak wypełnić pamięć. Pamięć zmiennej si_other wypełniana jest zerami. Poza tym adres jest wypełniany adresem serwera (`inet_addr(SERVER)`).

```
//start communication
    while(1)
    {
        int i;
            printf("Enter message : ");
        gets(message);
```

Klient również uruchamia pętlę nieskończoną. Wykonywane są jednak w niej inne zadania. Początkowo jest inicjalizowana zmienna i (typu int), następnie wypisywane jest `"Enter message : "` (a więc od tego momentu klient może wpisać wiadomość, która serwer może potencjalnie odebrać). Następnie funkcja gets pozwala nam wpisywać dane, które zostaną umieszczone w zmiennej message. Funkcja działa tak długo, aż nie podamy znaku nowej linii (wpisujemy go, wciskając Enter).

```
//send the message
        if (sendto(s, message, strlen(message) , 0 , (struct sockaddr *)
&si_other, slen) == SOCKET_ERROR)
        {
```

```
            printf("sendto()    failed    with    error    code    :    %d"    ,
WSAGetLastError());

            exit(EXIT_FAILURE);

        }
```

Klient również używa funkcji sendto, wraz ze sprawdzeniem czy nie wystąpił błąd. Funkcja ma podobne parametry co ta używana w serwerze. Jednak widać, że przesyłana jest wiadomość wpisana dzięki funkcji gets (`message, strlen(message)`). Reszta kodu jest identyczna jak w przypadku wersji używanej na serwerze.

```
//receive a reply and print it

            //clear the buffer by filling null, it might have previously
received data

        memset(buf,'\0', BUFLEN);

      Również tu, pamięć bufora jest wypełniana znakami nowej linii.

  if (recvfrom(s, buf, BUFLEN, 0, (struct sockaddr *) &si_other, &slen) ==
SOCKET_ERROR)

        {

            printf("recvfrom()    failed    with    error    code    :    %d"    ,
WSAGetLastError());

            exit(EXIT_FAILURE);

        }
```

Funkcja ta, recvfrom, wraz z warunkiem, jest identyczna jak w przypadku serwera. Różnicą jest fakt, że jest ona wykonywana po wysyłaniu informacji do serwera, nie na odwrót. Kolejnością jest bowiem wysłanie informacji od klienta do serwera, a następnie odesłanie tej informacji z serwera do klienta.

```
puts(buf);

    }


    closesocket(s);

    WSACleanup();


    return 0;

}
```

Na końcu, to co jest w buforze jest wypisywane (a więc powinno się wypisać to, co wysłaliśmy do serwera). Instrukcje wykonywane poza pętlą nieskończoną while(1) również jest identyczna jak w przypadku serwera.

Zestaw programów testowych do ustalenia podstawowej komunikacji z zastosowaniem protokołu WinSock pracuje jako dwie aplikacje konsolowe. Uruchamia się je poprzez wskazanie właściwych plików wykonywalnych .exe. Należy jednak pamiętać, aby najpierw uruchomić serwer, a potem klienta (co prawda klient się uruchomi bez serwera, jednak wysłanie dowolnej wiadomości spowoduje błąd programu i jego zamknięcie).

Aby przetestować program, przesłano dwie wiadomości przez klienta do serwera. Najpierw, uruchamiamy program Ws_Server1.exe (program odpowiedzialny za serwer). Jeżeli serwer, działa poprawnie, program będzie wyglądał jak na zrzucie ekranu poniżej:

Rys. 3.2-1 Działanie programu Ws_Serwer1.exe zaraz po jego uruchomieniu.

Z kolei klient po uruchomieniu będzie wyglądał inaczej. Główną różnicą jest brak niektórych informacji (np. o utworzeniu socketu), jak i prośba o podanie wiadomości:

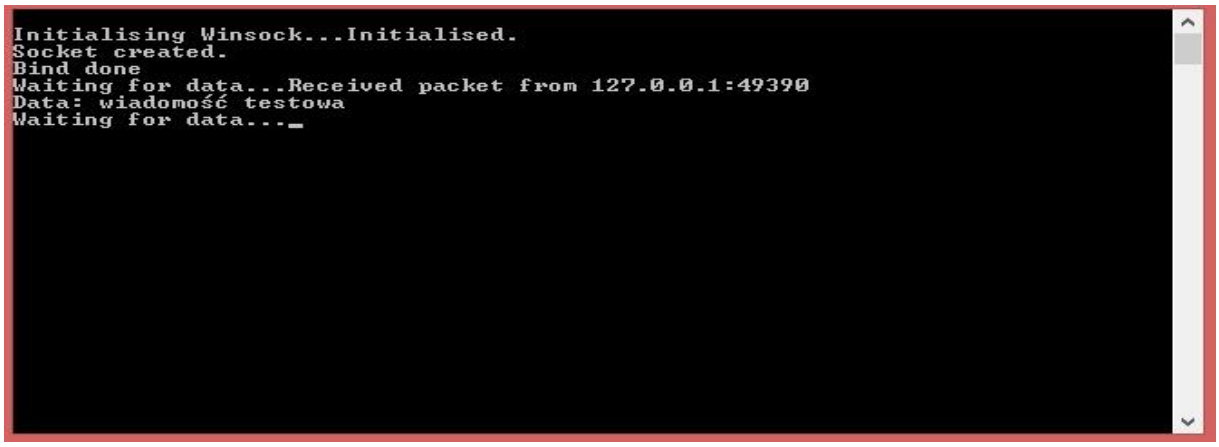
Rys. 3.2-2 Działanie programu Ws_Client1.exe zaraz po jego uruchomieniu.

Po wykonaniu tych czynności można przesłać wiadomość. Spróbujmy wysłać więc wiadomość „wiadomość testowa” przez klienta i sprawdźmy, jak zachowają się oba programy.

Po stronie klienta, po wysłaniu wiadomości, dostajemy ją powrotem, po czym pojawia się ponownie prośba o wysłaniu wiadomości. Nie ma bowiem ograniczeń co do ilości wysłanych wiadomości:

Rys. 3.2-3 Działanie programu Ws_Client1.exe zaraz po wysłaniu wiadomości.

Z kolei po stronie serwera dociera wiadomość od klienta. Uprzednio jednak jest wyświetlana informacja, skąd ta wiadomość przyszła. Po odebraniu wiadomości dalej trwa oczekiwanie na wiadomość:

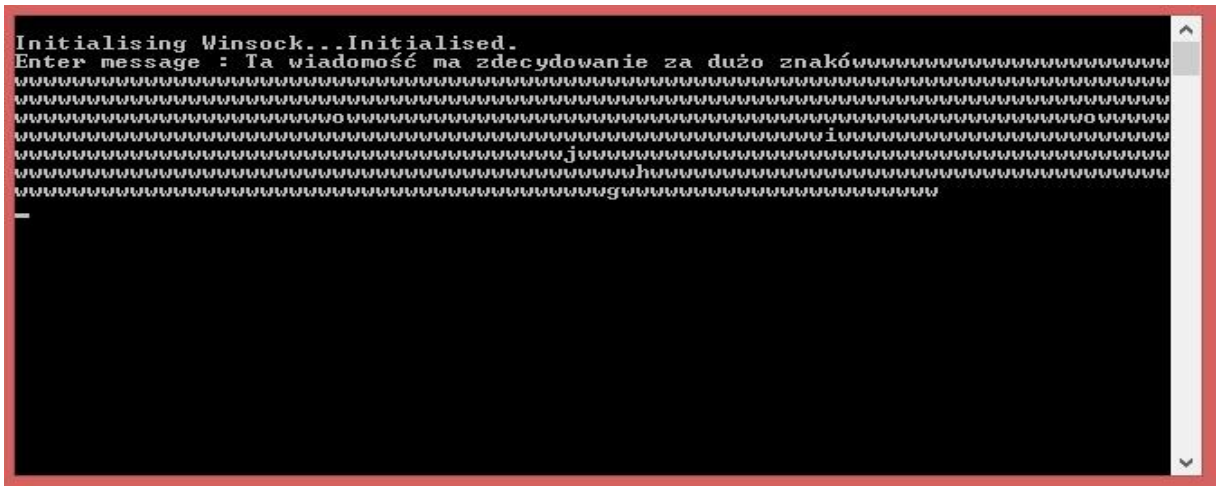


Rys. 3.2-4 Działanie programu Ws_Serwer1.exe zaraz po wysłaniu wiadomości.

Jak widać, wiadomość przyszła z klienta o IP równym 127.0.0.1, gdzie portem jest 49390.

Należy jednak pamiętać, że zbyt długie wiadomości, nie będą poprawnie dostarczone. Zostaną one wysłane, jednak serwer zgłosi błąd, a program go obsługujący się wyłączy. Wiadomość nie może jednak mieć więcej niż 512 znaków. Program klient również przestaje poprawnie pracować, jako że po naciśnięciu Enter nie można już wpisać kolejnych wiadomości:

Na rys. 3.2-5 pokazano zrzut ekranu z zachowania programu po próbie wysłania wiadomości. Zrzut ekranu jest tylko po stronie klienta – program po stronie serwera wyłącza się zbyt szybko, aby zrzut ekranu mógł zostać wykonany:



Rys. 3.2-5 Działanie programu Ws_Client1.exe zaraz po wysłaniu zbyt długiej wiadomości.

Widać więc, że podkreślenie, pojawiające się zwykle na ostatnim znaku programu, jest linię niżej, a nie zaraz po zakończeniu wiadomości. Była więc podjęta próba wysłania wiadomości. Z kolei nie pojawiła się wiadomość Enter message, co oznacza, że nie można wpisać nowej wiadomości ani ją wysłać (nie można pojąć nawet próby wysłania wiadomości, co można zrobić pomimo braku obecności serwera).

W przypadku użycia tego programu jako komunikację z symulatorem lotu, ten błąd jest czysto teoretyczny - przesyłane zmienne są liczbami i żadna z nich nie zawiera ponad 512 cyfr. Z powodu braku zauważenia innych poważnych błędów (jak na przykład problem z szybkością typem wiadomości – nie stoi nic na przeszkodzie by bardzo szybko wysyłać bezznakowe wiadomości), można uznać, że program w takiej formie nadaje się do komunikacji w symulatorze.

Opisany powyżej program, choć można zastosować do odbierania informacji z symulatora lotu, tych danych nie gromadzi w żaden sposób. Należy więc zmodyfikować ten program, aby to było możliwe.

Najprościej jest użyć rozwiązań pokazanych w rozdziale 3.1 (OleDB). Dzięki temu można użyć instrukcji SQL, dokładniej instrukcji INSERT INTO. Jako argumenty instrukcji SQL podaje się dane odebrane z symulatora (przykładowy program zapisuje już dane do zmiennej, zmienna ta jest jednak nadpisywana przy każdym kolejnym odebraniu danych). Należy jednak pamiętać, aby zapisać dane do bazy danych (a więc wywołać komendę SQL), zanim kolejna dana zostanie odebrana (aby nie nadpisać odebranych informacji). Nie jest to jednak problemem, albowiem każda zmienna jest odebrana po każdym wykonaniu się pętli while. Wystarczy więc wpisać kod odpowiedzialny za wpisywanie danych do bazy danych w tejże pętli. Program przez to będzie wykonywał się znacznie wolniej, jednakowoż żadna dana nie zostanie pominięta.

Tak zapisane dane można zastosować do wizualizacji danych (na przykład poprzez wykres liniowy). Koncepcja, wraz z przykładowym programem zdolnym do wykonania takiej wizualizacji, została omówiona w następnym rozdziale – 3.3

3.3 Koncepcja modułu programowego do wizualizacji danych.

Ostatnią częścią, jaką należy zrealizować, to moduł do wizualizacji danych. Dane, które zostaną odebrane, a następnie przeniesione do bazy danych, należy potem

wizualizować. Sama wizualizacja będzie polegać na tworzeniu wykresów liniowych, gdzie osią poziomą będzie czas, a osią pionową – dowolna zmienna mierzona (na przykład szybkość wznoszenia się samolotu). Dlatego tym bardziej zaleca się, aby w części projektowania bazy danych zamiast identyfikatora użyć systemowej daty i godziny (patrz rozdział 3.1). Dzięki temu tworzenie wykresów będzie łatwiejsze, ponieważ będzie można łatwiej przypisać dane do konkretnego czasu. W przypadku używania identyfikatora, trzeba będzie przypisać czas do każdej danej, lub podać w przybliżeniu różnicę czasową między dwiema najbliższymi danymi (a więc np. między danymi o ID 999 a danymi o ID 1000), mierząc średni czas odbierania danych. Może to jednak być kłopotliwe, gdy dane z różnych przyczyn będą odbierane wolniej niż zwykle przez dłuższy czas. Wtedy dane na wykresie mogą nie odpowiadać rzeczywistości.

Dla pewności, można w wykresie używać nie danych po kolei, lecz np. co dziesiątej. (tzw. próbkowanie). Wynika to z tego, że dane odbierane są bardzo szybko i nawet uwzględnianie tylko co dziesiątej informacji oznacza, że mamy do czynienia z ułamkami sekund. Dzięki takiemu rozwiązaniu nie tracimy zbyt wiele informacji, a jednocześnie wykres bardziej odpowiada rzeczywistości.

Tworzenie wykresów bezpośrednio z bazy danych może być bardzo trudne. Można jednak eksportować tabele z bazy danych, aby były przystosowane do innych programów (a tym samym innych formatów). Microsoft Access umożliwia eksportowanie tabel między innymi do Excela, plików tekstowych, Microsoft Worda (format RTF), PDF czy XML. Tworzenie wykresów z danych w tych formatach może okazać się łatwiejsze.

Wspomniany wcześniej Microsoft Excel[7] umożliwia tworzenie wykresów, jednak do tworzenia precyzyjnych wykresów do symulatorów lotu jest mało popularny. Częściej używanymi rozwiązaniami jest MatLab[8], który jest narzędziem umożliwiającym bardzo wiele. Jego główną zaletą, w porównaniu do wykresów z Excela, jest możliwość dowolnego skalowania wykresu. Jest to istotne, ponieważ wykres może zawierać bardzo wiele danych i bez skalowania niemożliwe będzie zobaczenie np. ile dane wynosiły w danej minucie. Wykres tez można dowolnie podpisać, utworzyć legendę, czytelniejszy jest też wykres, gdy względem jednej osi pokazanych jest wiele danych (głównie dzięki przybliżeniu, które pozwala uniknąć „zlania się” danych, a więc niemożności odczytania wartości z powodu nałożeniu się wykresów). Wykres tez można modyfikować, zmieniając mu kolor, grubość, zmieniając typ punktów (np. kropki, krzyżyki) i tym podobne. Danych nie trzeba też

wpisywać bezpośrednio do Matlaba – Matlab umożliwia tworzenia wykresu z danych odczytanych z pliku tekstowego.

W przypadku, gdy chcemy użyć darmowych rozwiązań, można użyć np. darmowego programu Gnuplot[9], który został stworzony do rysowania wykresów. W nim również można tworzyć wykres, używając danych z pliku tekstowego. Można tez przybliżyć wykres, jak i rysować określonym stylem, podobnie jak w Matlabie. Należy jednak pamiętać, że program ma nieco inną składnię i program który zadziała w Matlabie nie zadziała w Gnuplot, jak i na odwrót.

W koncepcji wizualizacji danych można tez zastosować moduł programowy przygotowany w oknie Windowsowym. Okno to jednak znacznie różni się od okien z programu do komunikacji. Jest on bowiem o wiele bardziej rozbudowany, bardziej również przypomina typowe okna znane przeciętnemu użytkownikowi (to jest, posiada menu w lewym górnym rogu, sam program jest też graficzny, a nie tekstowy). Oczywiście wykres można przybliżyć, jak i oddalić (w menu programu jest też przycisk 100%, pozwalający wyświetlić cały wykres w oknie, nie marnując przy tym miejsca), można też wizaulizację wykresów włączyć/wyłączyć, aby porównać różne dane lub zwiększyć czytelność wykresu. Moduł ten rysuje wykres na podstawie danych pobranych z pliku tekstowego (format .txt).

Moduł programowy ten został przygotowany przez pana dr. Sławomira Samoleja. Kod tegoż programu (głównie części odpowiedzialne za tworzenie wykresu) zostaną omówione poniżej.

```
#define STRICT          // żądanie ścisłego przestrzegania typów danych
                                // w programie
#include <windows.h>
#include <windowsx.h> // udostępnienie rozszerzonych funkcji API
#include <math.h>   // modf fabs floor
#include <stdio.h>
#include <stdlib.h>
#include "resource.h"


#define NUM     10
```

Bibliotek jest tu więcej niż w przypadku programu testowego do komunikacji. Zdefiniowano też żądanie ścisłego przestrzegania typów danych. Oznacza to, że typ danych nigdy nie zmieni się samodzielnie (to jest, przez kompilator).

Następnie jest wiele zmiennych, które będą używane do tworzenia wykresu. Główne typy to float (najmniejszy typ liczbowy pozwalający zapisywać liczby niecałkowite), *FILE (wskazuje na zmienne, gdzie dokona się wstępna analiza danych odczytanych z pliku, jak również samo odczytanie pliku), POINT (struktura danych do przechowywania dwuwymiarowych współrzędnych punktów na ekranie) i BOOL (zmienne logiczne mające wartość TRUE i FALSE. Odpowiadają one za włączanie i wyłączanie wizualizacji poszczególnych wykresów). Oprócz tego są też zmienne typu int, double, czy też char.

Po zadeklarowaniu zmiennych, następuje utworzenie specjalnej głównej funkcji programu windows- `int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)`. Jest to funkcja typu main odpowiadająca za tworzenie okna i jego poprawne działanie (możliwość zmiany rozmiaru, minimalizację, menu w lewym górnym rogu, tytuł okna i tym podobne).

Po definicji funkcji WinMain w programi zamieszczon definicję głównej procedury okna: `LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)`. Tutaj jest zamieszczona cała obsługa okna, a więc co się dzieje gdy go uruchomimy, wciśniemy poszczególne klawisze na klawiaturze, gdy klikniemy na element okna kursorem myszy, jak również to co ma się wydarzyć podczas rysowania elementów i gdy program czeka na dalsze instrukcje. Poszczególne części programu wykonują się, gdy zaistnieje dane zdarzenie (program więc, poza czekaniem, nic nie robi dopóki nie wykonamy na nim żadnej akcji).

Główna część program opiera się na instrukcji switch, która się nie kończy, dopóki program nie zostanie zamknięty (a więc dopóki program nie zostanie zamknięty, któryś z przypadków może zaistnieć i może zostać wykonany odpowiadający mu kod). Pierwszym zdefiniowanym przypadkiem jest `case WM_CREATE`. Jest on wykonywany, gdy program zostanie uruchomiony.

```
// Otworzyć pliki i dokonać wstępnej analizy danych
            f_czas=fopen("czas.txt","r");
            f_w_zad=fopen("w_zad.txt","r");
            f_ster=fopen("sterow.txt","r");
            f_odp_uk=fopen("wyj_obj.txt","r");
```

Najpierw, dzięki instrukcji fopen, otwierane są pliki, które zawierają dane do odczytania wykresu. Ten przykładowy moduł zawiera cztery wykresy, a więc czyta dane z czterech plików. Niepełna podana ścieżka jest wskazówką, aby pliki tekstowe znalazły się w folderze, gdzie jest przechowywany program.

```
// policzyć ile jest danych w pliku
      while(!feof(f_czas))
      {
            fscanf(f_czas,"%f\n",&x0); // odczytaj kolejną daną
            data_counter++; // wylicz ile jest danych w zestawie
      }
```

Następnie dane w pliku są liczone, dzięki pętli while (while korzysta z pliku tekstowego odpowiedzialnego za czas na wykresie). Zmienna data_counter co iterację pętli jest inkrementowana, dzięki czemu można policzyć ile danych jest w zestawie, a tym samym się dowiedzieć ile punktów wykres ma zawierać.

```
// zaallokowac pamiec pod tablice z danymi:
                        m_czas=  new  float[data_counter];
                        m_w_zad= new  float[data_counter];
                        m_ster=  new  float[data_counter];
                        m_odp_uk=new  float[data_counter];
                        m_uchyb= new  float[data_counter];
                        p_w_zad= new  POINT[data_counter];
                        p_ster=  new  POINT[data_counter];
                        p_odp_uk=new  POINT[data_counter];
                        p_uchyb =new  POINT[data_counter];
```

Zmienna data_counter jest potrzebna, aby wiedzieć, ile pamięci potrzebują nowe tablice zmiennych typu `float` i `POINT`. Wykres może bowiem składać się z niemalże dowolnej ilości punktów i możliwość zarezerwowania pamięci z pominięciem obliczeń, ile pamięci trzeba zarezerwować, jest nie tylko niepraktyczna, ale i również niemalże niemożliwa. W dodatku program mógłby wyrzucić błąd lub nie wykreślić całego wykresu w przypadku zbyt małej pamięci.

```
//wczytanie danych do tablic:

fseek(f_czas,0L,SEEK_SET); // ustaw znacznik odczytu pliku ponownie na poczatek
```

funkcja fseek ustawia znacznik odczytu pliku na początek. Dzięki temu plik z danymi, które mają zostać wizualizowane, jest czytany od nowa.

```
//data_counter=10000;

            for(i=0;i<data_counter;i++)

                  {

                        fscanf(f_czas,"%f\n",&m_czas[i]);

                        fscanf(f_w_zad,"%f\n",&m_w_zad[i]);

                        fscanf(f_ster,"%f\n",&m_ster[i]);

                        fscanf(f_odp_uk,"%f\n",&m_odp_uk[i]);

                        m_uchyb[i]=m_w_zad[i]-m_odp_uk[i];

                  }
```

Tutaj następuje wczytanie danych do tablic. Pętla for wykonuje się tyle razy, ile wynosi zmienna data_counter. Dzięki czemu wczytane zostaną wszystkie dane i jednocześnie tablice nie będą mieć pustych znaków, co jest optymalnym rozwiązaniem.

```
//wyznaczenie wartosci niezbednych do skalowania wykresów:

                        max_w_zad=min_w_zad=m_w_zad[0];

                        max_ster=min_ster=m_ster[0];

                        max_odp_uk=min_odp_uk=m_odp_uk[0];

                        max_uchyb=min_uchyb=m_uchyb[0];


                  for(i=1;i<data_counter;i++)

                  {
```

```
                    if(m_w_zad[i]>max_w_zad) max_w_zad=m_w_zad[i];
                    if(m_w_zad[i]<min_w_zad) min_w_zad=m_w_zad[i];
                    if(m_ster[i]>max_ster) max_ster=m_ster[i];
                    if(m_ster[i]<min_ster) min_ster=m_ster[i];
                    if(m_odp_uk[i]>max_odp_uk) max_odp_uk=m_odp_uk[i];
                    if(m_odp_uk[i]<min_odp_uk) min_odp_uk=m_odp_uk[i];
                    if(m_uchyb[i]>max_uchyb) max_uchyb=m_uchyb[i];
                    if(m_uchyb[i]<min_uchyb) min_uchyb=m_uchyb[i];
                }
```

Tutaj są wyznaczane wartości maksymalne i minimalne wartości zadanej, sterowania, uchybu i odpowiedzi. Również ta pętla wykonuje się tyle razy, ile wynosi data_counter. Należy bowiem sprawdzić każdą daną, która będzie obecna w wykresie.

```
// wyznaczenie wartosci mksymalnej dla przebiegu:
            if(max_w_zad>max_odp_uk) max_y=max_w_zad;
                  else  max_y=max_odp_uk;

            if(min_w_zad<min_odp_uk) min_y=min_w_zad;
                  else  min_y=min_odp_uk;
```

Następnie jest wyznaczana wartość maksymalna i minimalna dla przebiegu. Są używane zmienne, które zostały uzyskane w procesie wyznaczania wartości maksymalnych, minimalnych i tym podobne.

```
// ustalenie, że wybrano w-zad i ster:
                              bool_w_zad=TRUE;
                              bool_odp_uk=TRUE;
```

Tutaj zmieniono dwie zmienne typu BOOL na TRUE. Odpowiadają one kolejno wizualizacji wartości zadanej i wizualizacji odpowiedzi układu. Gdyby te zmienne były ustawione na FALSE, wykresy byłyby poprawnie policzone, lecz nie zostałyby wyświetlone.

Kolejną częścią kodu jest tworzenie tzw. gridu, a więc siatki. Okno z wizualizacją posiada bowiem siatkę, która pomaga odczytywanie informacji z wykresu. Nie wpływa to jednak na sam wykres, tylko na czytelność okna z wykresem.

Po utworzeniu gridu odczytywana jest pierwotna wartość wmax i wmin. Jest to ostatnia operacja, jaka wykona się wraz z uruchomieniem programu. Od tego momentu program nic nie wykonuje, dopóki nie wykonamy na nim akcji.

Następnym typem akcji jest WM_SIZE. Obsługa teg komunikatu jest uruchamiana, gdy zostaje zmieniony rozmiar okna z programem. Za każdym razem, gdy jest zmieniany rozmiar okna, jego bieżące wymiary są zachowywane w odpowiednich zmiennych stosowanych później do ustalania proporcji elementów graficznych w oknie programu.

Drugą najważniejszą akcją jest WM_PAINT. To właśnie ona odpowiada za to, co się dzieje, gdy następuje zmiana w oknie. To tutaj są wykorzystywane obliczenia wykonane podczas uruchamiania programu (WM_CREATE), a więc następuje rysowanie wykresu.

Cały proces jest złożony i wymaga wielu obliczeń. Sprowadza się jednak do trzech podstawowych operacji:

- Najpierw tworzony jest tzw. metaplik (jest to plik, który zostanie wypełniony danymi, użyty, a następnie usunięty podczas wykonania się całego kodu w instrukcji WM_PAINT). Format tego pliku to .wmf, a jego nazwa to hdcmeta.

-Następnie następuje pobranie uchwytu fontów, sprowadza się do przypisania do zmiennej czcionki o zmiennym rozmiarze (variable font), dzięki czemu tekst w programie będzie mógł zmienić rozmiar w zależności od rozmiaru okna.

-Kolejnym etapem jest wywołanie funkcji CreatePen, a dokładniej `SelectObject(hdcMeta,CreatePen(PS_DOT,1,RGB(0,0,0)));`), która pozwala rysować elementy w określonym kolorze i w określonym stylu (wybrano kolor czarny; RBG równe 0 i kropki; PS_DOT, tutaj więc odbywa się rysowanie wspomnianej wcześniej siatki. Jest to też pierwsze miejsce w programie, gdzie używany jest stworzony wcześniej metaplik.

-Po narysowaniu siatki następuje naniesienie tekstu na okno (a więc podpisanie odpowiednimi wartościami osi x i y).

-Gdy siatka zostanie ukończona, zachodzi rysowanie wykresu. Po wstępnych obliczeniach, sprawdzane jest, które zmienne odpowiedzialne za widoczność wykresu są ustawione na TRUE. Jeśli są, następuje rysowanie wykresu. Jeśli nie, wykres nie zostaje wykreślony. W przypadku gdy żaden wykres nie ma być widoczny, aktywuje się zabezpieczenie i program wraca do stanu początkowego. (samo zabezpieczenie jest jednak

zakodowane znacznie później, poza instrukcjami). Kod każdego rysowania wykresu jest prawie identyczny. Różnią się zdefiniowanym kolorem w funkcji odpowiedzialnej za aktywowanie rysowania (a więc różna jest wartość kolejnych kolorów w RGB).

-Na końcu, gdy wykresy zostaną już nakreślone, metaplik zostaje kasowany, następuje tez koniec operacji w instrukcji WM_PAINT.

Kolejną instrukcją jest WM_LMOUSEDOWN. Obsługuje ona zdarzenia, gdy trzymamy przycisk myszy.

```
ptBeg.x = ptEnd.x = LOWORD (lParam) ;
                ptBeg.y = ptEnd.y = HIWORD (lParam) ;
                DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                SetCapture (hwnd) ;
                fBlocking = TRUE ;
                return 0 ;
```

Kod tejże instrukcji, zamieszczony powyżej, jest jedną z dwóch części kodu odpowiedzialnego za zaznaczanie obszaru, który następnie zostanie powiększony. Ważną zmienną jest tutaj fBlocking, która jest tu ustawiana na TRUE. Wraz z instrukcjami obecnymi w WM_MOUSEMOVE pozwala ruchem myszy zaznaczyć dowolny obszar w oknie programu.

```
case WM_MOUSEMOVE :
if (fBlocking)
                    {
                    DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                    ptEnd.x = LOWORD (lParam) ;
                    ptEnd.y = HIWORD (lParam) ;
                    DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                    }
                return 0 ;
```

Wspomniana wcześniej instrukcja WM_MOUSEMOVE wykonuje swój kod, wyłącznie, gdy fBlocking ma wartość TRUE. Oznacza to, że gdy lewy przycisk myszy nie

jest przytrzymany, przemieszczanie kursora po programie nie wykonuje żadnej akcji. Należy jednak pamiętać, że każdy ruch kursorem w obszarze programu powoduje sprawdzenie kodu instrukcji WM_MOUSEMOVE.

Kolejną instrukcją, która również wykonuje się tylko wtedy, gdy zmienna fblocking ma wartość TRUE, jest WM_LBUTTONUP. Instrukcja ta jest wykonywana, gdy lewy przycisk myszy nie jest trzymany. Cały kod instrukcji jest długi i odpowiada za przybliżenie zaznaczonego wcześniej obszaru (zaznaczanie obszaru jest zaprogramowane w instrukcjach WM_MOUSEMOVE i WM_LBUTTONDOWN). To tutaj też jest wykonywany ostatni etap przybliżenia co widać w ustawieniu wartości zmiennej fBlocking na FALSE.

Następnie jest zdefiniowana instrukcja WM_DESTROY. Jest ona aktywowana, gdy program zostaje w dowolny sposób wyłączony. Usuwa ona niektóre zmienne, gdzie są przechowywane obliczenia do wykresów.

Ostatnia instrukcja, która zostanie tu omówiona, to WM_COMMAND. Jest ona istotna, albowiem odpowiada za menu na górze okna programu, jak również za jego funkcjonowanie. Tworzenie menu również opiera się na dużej instrukcji switch, gdzie poszczególne case odpowiadają za jedną pozycję w menu.

Poszczególne pozycje w menu to:

-Plik, gdzie z kolei dostępne opcje to Koniec i Skopiuj do Schowka

-100% (pozwala oddalić obraz tak, aby cały wykres by widoczny w oknie)

-Dane, gdzie z kolei są dostępne opcje wartość zadana, odpowiedź układu, uchyb i sterowanie.

-O… (informacje o programie).

Opcja Koniec jest najłatwiejsza do zaprogramowania i wygląda następująco:

```
case ID_PLIK_KONIEC:
      SendMessage (hwnd, WM_CLOSE, 0, 0L) ;
```

Opcja ta służy wyłącznie do zamykania programu. Program można jednak zamknąć klikając w X w prawym górnym rogu programu. Oba sposoby wyłączenia programu wywołają instrukcję WM_DESTROY

Kolejną opcją w menu Plik jest Kopiuj do schowka. Pozwala ona skopiować cały obraz w oknie programu do schowka, co można potem wkleić do dowolnego programu graficznego, jak np. Microsoft Paint. Jest to opcja o tyle praktyczna, że inne metody kopiowania i wklejania obrazu pozwalają skopiować albo obraz całego okna (to jest, całego okna programu ,wraz z tytułem i menu), tudzież obraz całego pulpitu)

Opcja 100% pozwala widzieć całość wykresów (dokładniej to obszar całej osi czasu, w tym przypadku od 0 do 1). Jest to też jedyna opcja pozwalająca oddalić obraz. Używając myszki, można jedynie obraz przybliżać.

Opcja O… wyświetla informacje o programie w oknie wiadomości (MessageBox) typowym dla programów Windows. Takie okno wygląda następująco:

Rys. 3.3-1 Okno wyświetlające się po wybraniu opcji O… z menu programu do wyświetlania wykresów.

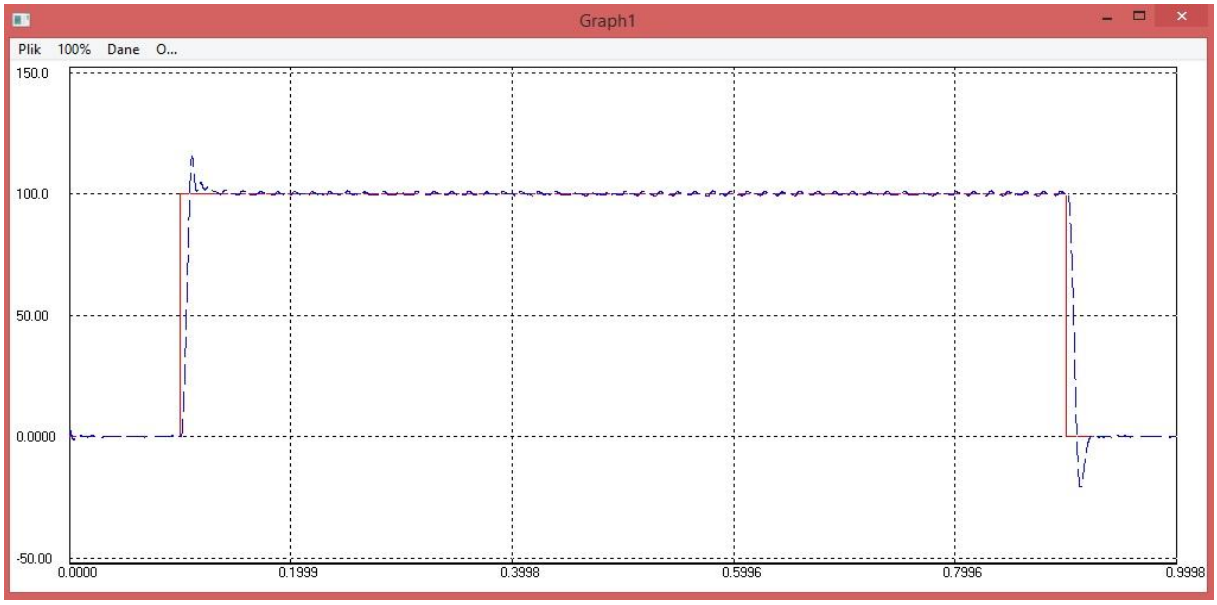
Jest to więc proste okno informacyjne, gdzie jedyną opcją jest wciśnięcie OK.

Pozostałe opcje, dostępne w menu Dane, odpowiadają za widoczność wykresów. Ich wciśnięcie spowoduje wyświetlenie lub schowanie wykresu, co odpowiada zmianie wartości odpowiedzialnej za widoczność wykresu na TRUE lub FALSE. Opcje w menu można zaznaczać i odznaczać.

Po zakończeniu instrukcji WM_COMMAND nie ma juz więcej instrukcji. Pozostała część kodu odpowiada za konwersję wartości float do tablicy tekstowej, za rysowanie granic obszaru, który ma zostać powiększony (aby było widać, jaki obszar zaznaczamy), jak również

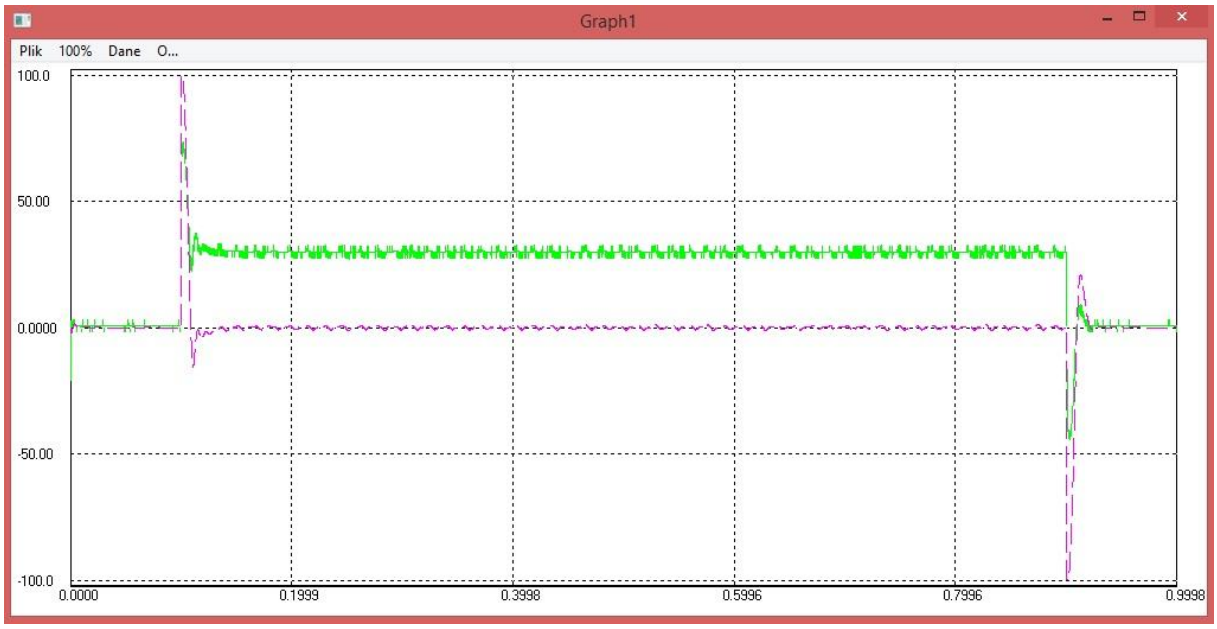
za wyznaczanie globalnego maksymalnego x i y. Zaprogramowane też jest zabezpieczenie powodujące, że w przypadku odznaczenia wszystkich wykresów, okno wraca do stanu początkowego (a więc tego, który był obecny zaraz po odpaleniu programu). Jest to ważne, albowiem rysowanie siatki pomocniczej, jak i podpisanie osi x i y jest zależne od tego, jaki przebieg mają wyświetlane wykresy.

Na rysunku 3.3-2 pokazano program w działaniu, zaczynając od postaci programu zaraz po jego uruchomieniu.



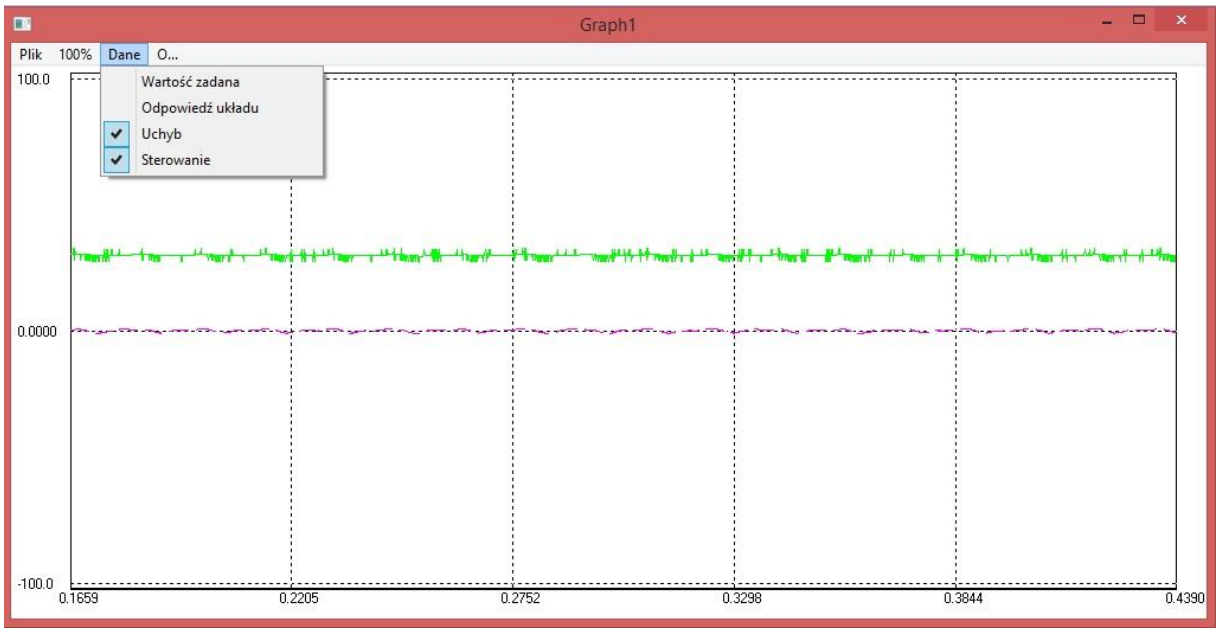
Rys. 3.3-2 Okno analizowanego programu do wykreślania wykresów, zaraz po odpaleniu programu (stan początkowy).

Domyślnie są wyświetlane dwa z czterech wykresów: wartość zadana i odpowiedź układu. Można jednak wyświetlić dowolną kombinację wykresów, pod warunkiem że przynajmniej jeden wykres zostanie wyświetlony. Widać też, że siatka jest tak dobrana, aby wykres był w całości widoczny. Wykres wykracza bowiem momentami poza wartość 100. Gdy jednak nie będzie tego robił, siatka się zmieni, co widać na rys. 3.3-3.



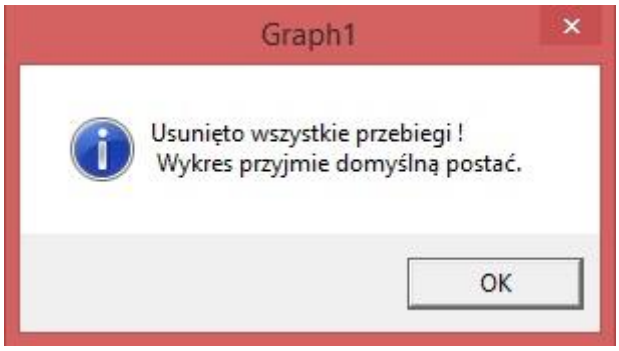
Rys. 3.3-3 Okno analizowanego programu do wykreślania wykresów, przy wyświetlaniu przebiegu sterowania i uchybu.

Obraz można przybliżać niemal dowolnie. Można nawet przybliżać obszar, gdzie wykres nie jest rysowany (ponieważ nie ma takich wartości). Wartości na osiach zostaną dostosowane do przybliżonego obszaru, co widać na zrzucie ekranu poniżej:



Rys. 3.3-4 Okno analizowanego programu do wykreślania wykresów, przy przybliżeniu przebiegu sterowania i uchybu, wraz z otwarciem menu Dane.

Gdy wybierze się opcję niewyświetlania żadnego wykresu, program, zanim wróci do stanu początkowego, wyświetli stosowny komunikat, taki jak na zrzucie ekranu poniżej:



Rys. 3.3-5 Okno wyświetlające się po wybraniu po niewybraniu żadnego wykresu do wyświetlania.

Po dokładnej analizie programu wiadomo więc, jak go zmodyfikować, aby był przystosowany do wyświetlania wartości odebranych z symulatora lotu. Wiadomo już, że program odczytuje wartości do wykresu z pliku tekstowego. Trzeba jednak pliki tekstowe odpowiednio nazwać i przenieść do folderu z programem, aby zostały poprawnie odczytane (można ewentualnie podać pełną ścieżkę do folderu z danymi do wykresu). Plik tekstowy CZAS.TXT, który służy do tworzenia osi czasu, jak i jej podpisywania, można zastąpić kluczem głównym z bazy danych (identyfikator). Nie jest to jednak konieczne, wystarczy bowiem pamiętać, że wartość 0 odpowiada chwili, kiedy zaczęło się odczytywanie wartości, a 1 – chwili końcowej odczytywania wartości.

Problemem może być fakt, że ilość odczytywanych wartości może być inna od tego, ile wartości jest zdefiniowanych w pliku tekstowym CZAS.TXT. Zastosowanie liczb z identyfikatora, zamiast pliku tekstowego CZAS.TXT może więc być wygodniejszym rozwiązaniem. Można jednak policzyć ilość wszystkich pobranych wartości z tabeli bazy danych (stosowną funkcją SQL, patrz 3.1). i zastosować tą informację do stworzenia pliku z poprawnymi wartościami na miejsce pliku tekstowego CZAS.TXT. Jest to jednak czasochłonne, a zautomatyzowanie tego procesu, bardzo trudne.

Po wykonaniu tej operacji teoretycznie nie trzeba już nic modyfikować, zakładając że wprowadzimy tylko cztery wykresy do programu. Pozostają jedynie modyfikacje kosmetyczne, jak zmiana menu Dane na podpisanie, co ten wykres przedstawia (np. prędkość wznoszenia się samolotu). Jeżeli jednak chcemy wyświetlać więcej niż cztery wykresy naraz (a dane zgromadzone zdecydowanie pozwalają stworzyć więcej wykresów), potrzebna jest przebudowa programu.

Przebudowa programu polega głównie na zwiększeniu ilości zmiennych i zwiększeniu ilości obliczeń. Dokładny schemat przebudowania to (Schemat opiera się na kolejności, w jakiej należy dokonać zmian, zakładając że będzie się zmiany dodawać od początku programu do końca. Należy jednak dodać wszystkie zmiany, aby program działał poprawnie).

-Zwiększyć ilość zmiennych typu fopen, aby otworzyć więcej plików tekstowych.

-Zaalokować dodatkową pamięć dla każdego wykresu, a więc dodatkowe zmienne typu float i point.

-Wczytać dane z nowych plików tekstowych do tablic.

-Wyznaczyć maksimum i minimum dla danych z nowych plików tekstowych.

-Dodać nowe zmienne typu BOOL, które będą odpowiadać za widoczność dodatkowych wykresów.

-Dodać nowe rysowania przebiegów dla dodatkowych wykresów.

-Przy zamknięciu programu, usunąć zmienne odpowiedzialne za dodatkowe wykresy.

-Dodać nowe opcje w menu Dane, pozwalające wyświetlić/schować dodatkowe wykresy.

-Przebudować wyznaczanie globalnych maksymalnych x i y, tak aby uwzględnić dodatkowe wykresy.

Ostatni punkt jest szczególny, jako że rozpatrywany jest każdy przypadek widoczności wykresów (a więc przypadek gdy widać wykres pierwszy, przypadek gdy widać wykres drugi, przypadek gdy widać wykres pierwszy i drugi i tak dalej). Oznacza to, że dodanie nowych wykresów znacznie zwiększa ilość przypadków, jakie należy rozpatrzyć (dokładnie ilość przypadków to ilość wykresów do kwadratu, a więc dla czterech wykresów jest to 16 przypadków). Pomimo że kod dla poszczególnych przypadków różni się nieznacznie, cała operacja modyfikacji może potrwać bardzo długo w przypadku dodawania dużej ilości wykresów.

Ostatnim zagadnieniem, jaki warto poruszyć, jest automatyzacja całego procesu wyświetlania wykresów, dokładniej zapisywania nowych plików tekstowych (a więc nowe dane), a następnie korzystania z nich. Najprostszym rozwiązaniem jest nadpisywanie

obecnych plików tekstowych. Wystarczy w odpowiednim folderze zapisać nowy plik tekstowy o takiej samej nazwie. Dzięki czemu odpada problem ze zmienianiem ścieżki do nowego pliku, jako ze nowy plik ma ta samą nazwę, a więc ma tą samą ścieżkę. Utrudnia to jednak archiwizację danych, jako że stare dane są nadpisane, a więc zniszczone. Można to częściowo obejść, gdy nowe dane będą zawierać nowe pobrane informacje, ale również i te starsze. Przy dłuższym korzystaniu z systemu może jednak nastąpić sytuacja, że danych będzie zbyt wiele, aby program był w stanie je obsłużyć w satysfakcjonującym czasie. Wtedy obecne dane należy zarchiwizować, a następnie upewnić się, że nowy plik tekstowy, z którego będą odczytane dane, będzie zawierał znacznie mniej danych.

4. Zakończenie

Praca ta miała za zadanie stworzyć koncepcję systemu, który pozwoli odebrać dane z symulatora lotu, następnie je zarchiwizować i wizualizować. Cel pracy zrealizowano poprzed analizę strumieni danych generowanych przez symulator lotu oraz przez zgromadzenie i omówienie gotowych aplikacji, które można w przyszłości zintegrować (po ew. dostosowaniu kodów źródłowych) w założony system. Przeprowadzona na wstępie analiza całego symulatora i danych z niego odbieranych pozwoliła wyszczególnić główne problemy z utworzeniem poprawnej koncepcji dla takowego zadania, jak na przykład duża ilość odbieranych danych jak i ich różnorodność (to jest, dane potrafią dotyczyć różnych informacji na temat samolotu i jego otoczenia).

Drugim etapem prac nad koncepcją systemu było stworzenie przykładowej bazy danych, jak i jego zaprojektowanie w taki sposób, aby uwzględnić różnorodność danych. Kolejnym krokiem było przybliżenie systemu połączenia baza danych-program OleDB, dostępnego w Microsoft Visual Studio. Zostały też podane przykładowe funkcje i kod, które pozwalają typowym programom okienkowym nie tylko łączyć się z bazą, ale i na niej operować; to jest dodawać, usuwać i wyszukiwać danych w niej obecnych.

Trzecim etapem prac było przybliżenie przykładowego, znalezionego w Internecie programu do prostej komunikacji serwer-klient korzystającego z protokołu UDP. Jego analiza pozwoli na skonstruowanie podobnego programu dostosowanego już do właściwego symulatora (jako że symulator również komunikuje się z klientem, w tym przypadku samolotem, poprzez protokół UDP). Zostały też podane przykładowe możliwości modyfikacji programu tak, aby uwzględniał archiwizację przesyłanych danych, jak i przesyłanie danych z bazy danych do pliku tekstowego.

Czwartym etapem prac było pokazanie kilku możliwości tworzenia wykresów liniowych. Choć wspomniano o możliwościach korzystania z już gotowych programów, tak komercyjnych jak i darmowych, główną częścią była analiza programu okienkowego, przygotowanego przez promotora tejże pracy. Analiza ta pozwala stwierdzić, czy, jak również jakie modyfikacje należy przeprowadzić, aby program dostosować do tworzenia wykresów liniowych na bazie danych odebranych z symulatora lotu.

Autor za własny wkład pracy uważa:

- (rozdział 2) Przygotowanie opracowania, dotyczącego symulatora lotu ALSIM 200 MCC oraz struktur danych z parametrami lotu przez niego generowanego, na bazie pracy inżynierskiej „Wizualizacja parametrów lotu samolotu na bazie symulatora lotu ALSIM 200 MCC" z 2008 roku, autorzy: Kumecki Jacek i Piotr Pryga.

- (rozdział 3.1) Stworzenie diagramu ERD przykładowej bazy danych, zdolnej do przechowywania danych z symulatora lotu.

- (rozdział 3.1) Przedstawienie przykładu połączenia baza danych-program OleDB, jak również podanie przykładów zastosowania możliwości tego połączenia w praktyce.

- (rozdział 3.2) Analiza programu „Simple UDP Server" przygotowanego przez Silver Moon, który jest przykładem połączenia serwer-klient korzystającego z protokołu UDP.

- (rozdział 3.2) Opisanie protokołu UPD, jak również przedstawienie możliwej modyfikacji programu „Simple UDP Server", aby uwzględniał zadanie archiwizacji odbieranych danych.

-(rozdział 3.3) Wstępne przedstawienie możliwości dwóch programów mogących tworzyć wykresy liniowe z wcześniej pozyskanych danych

-(rozdział 3.3) Analiza programu „Wykresy3" przygotowanego przez promotora tejże pracy inżynierskiej, dr inż. Sławomira Samoleja, który jest przykładową aplikacją okienkową zdolną realizować zadanie wizualizacji danych zapisanych w pliku tekstowym o formacie .txt.

-(rozdział 3.3) Przedstawienie możliwych modyfikacji „Wykresy3", aby dostosować go do wizualizacji danych odebranych z symulatora lotu.

.

Załączniki

[1]. Zaprojektowana baza danych: Baza Danych (przykład do pracy inżynierskiej).accdb

[2]. Projekt „Simple UDP Server” składający się z dwóch aplikacji – Ws_Server1.exe i Ws_Client.exe, wraz z ich kodem źródłowym.

[3] Aplikacja „Wykresy3” wraz z jej kodem źródłowym.

Bibliografia

[1*] rozdział przygotowany przy pomocy pracy inżynierskiej „Wizualizacja parametrów lotu samolotu na bazie symulatora lotu ALSIM 200 MCC” z 2008 roku, przygotowaną przez Kumeckiego Jacka i Piotra Prygę.

[2] „Receiving network messages”, Alsim Dokument nr TECH\OKL\RT\01.

[3] Garcia-Mlina H, Ullman J. D.,Widom J., “Systemy baz danych. Kompletny podręcznik, Wydanie II, Helion 2011.

[4] Tanenbaum A. S., Wetherall D. J., „Sieci komputerowe”, Helion 2012.

[5] Toth V. „Programowanie Windows 98/NT”, Helion 1999.

[6]http://www.binarytides.com/udp-socket-programming-in-winsock/

[7]http://msdn.microsoft.com/en-us/library/office/ee658205(v=office.14).aspx

[8]http://www.mathworks.com/products/matlab/

[9]http://www.gnuplot.info

POLITECHNIKA RZESZOWSKA im. I. Łukasiewicza Rzeszów, 2016
Wydział Elektrotechniki i Informatyki
Katedra Informatyki i Automatyki

STRESZCZENIE PRACY DYPLOMOWEJ INŻYNIERSKIEJ

KONCEPCJA SYSTEMU AKWIZYCJI I WIZUALIZACJI

DANYCH Z SYMULATORA LOTU ALSIM 200 MCC.

Autor: Patryk Duda, nr albumu: EA-DI-132945 Opiekun: dr inż. Sławomir Samolej
Słowa kluczowe: archiwizacja danych, symulator lotu, sieci komputerowe.

Przedstawienie koncepcji utworzenia systemu komunikacji między symulatorem lotu a samolotami, wraz z archiwizacją danych z symulatora i ich wizualizacją. Pokazanie możliwości archiwizacji danych, z użyciem baz danych i interfejsu OleDB. Opisanie przykładowego programu zdolnego do komunikacji poprzez protokół UDP. Analiza przykładowego programu zdolnego wizualizować odebrane dane.

RZESZOW UNIVERSITY OF TECHNOLOGY Rzeszow, 2016
Faculty of Electrical and Computer Engineering
Department of Computer and Control Engineering

DIPLOMA THESIS (MS) ABSTRACT

CONCEPT OF SYSTEM OF ACQUISITION AND VISUALISATION OF DATA FROM ALSIM 200 MCC SIMULATOR

Author: Patryk Duda, code: EA-DI -132945 Supervisor: Sławomir Samolej, PhD, Eng.
Key words: data acquisition, flight simulator, computer networks.

Showing the concept of making a system of communications between simulator and planes, with acquisition of data from simulator and data visualisation. Showing the capabilities of data acquisition with usage of database and OleDB interface. Description of ex ample program capable of communications using UDP protocol. Analysis of example program capable of visualisation of received data.