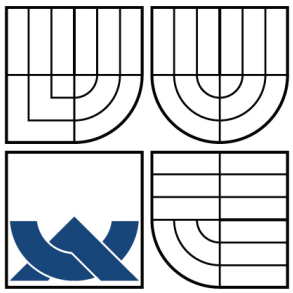
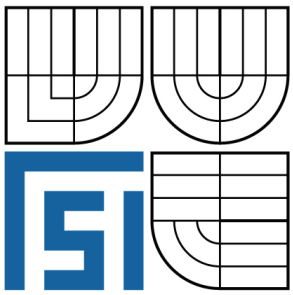


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA STROJNÍHO INŽENÝRSTVÍ

ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

FACULTY OF MECHANICAL ENGINEERING

INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS

DIAGNOSTICKÁ APLIKACE ZALOŽENÁ NA TWINCAT

INDUSTRIAL TWINCAT BASED DIAGNOSTICS APPLICATION

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — ĽUBOMÍR ZIGO

AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ KREJSA, Ph.D.

SUPERVISOR

BRNO 2010

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Ústav mechaniky těles, mechatroniky a biomechaniky
Akademický rok: 2009/2010

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

student(ka): Ľubomír Zigo

který/která studuje v **bakalářském studijním programu**

obor: **Mechatronika (3906R001)**

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma bakalářské práce:

Diagnostická aplikace založená na TwinCAT

v anglickém jazyce:

Industrial TwinCAT based diagnostics application

Stručná charakteristika problematiky úkolu:

Cílem práce je vytvořit průmyslově cílenou diagnostickou aplikaci běžící na průmyslovém PC a využívající software TwinCAT. TwinCAT neobsahuje diagnostický modul, což v praxi způsobuje problémy při poruchách. Řešením tohoto stavu a předmětem práce je aplikace která monitoruje proměnné TwinCAT, vyhodnocuje je a následně detekuje a lokalizuje poruchy.

Cíle bakalářské práce:

1. Nastudovat TwinCAT .NET interface
2. Navrhnout diagnostickou aplikaci, schopnou načítat příslušné proměnné (stavy senzorů, pohonu, atd.), detekovat a diagnostikovat poruchu, upozornit obsluhu
3. Aplikaci implementovat a odladit

Seznam odborné literatury:
firemní dokumentace TwinCAT.NET

Vedoucí bakalářské práce: Ing. Jiří Krejsa, Ph.D.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2009/2010.

V Brně, dne 22.10.2009

L.S.

prof. Ing. Jindřich Petruška, CSc. Ředitel ústavu	doc. RNDr. Miroslav Doupovec, CSc. Děkan fakulty

Anotácia

Táto práca popisuje postup tvorby diagnostickej aplikácie pre gumárenský lis riadený systémom Beckhoff TwinCAT. Aplikácia v pravidelných intervaloch monitoruje lis a v prípade poruchy oznámi jej príčinu obsluhe na monitore riadiaceho automatu.

Diagnostická aplikácia je vytvorená vo vyššom programovacom jazyku Visual C++ vo vývojovom prostredí Microsoft Visual Studio. So systémom TwinCAT komunikuje prostredníctvom knižnice štandardu .NET Framework TwinCAT::Ads, ktorá je súčasťou systému. Poruchové texty sa načítavajú z databázy prostredníctvom rozhrania Microsoft OLE DB s využitím jazyka SQL.

Kľúčové slová

Diagnostika, TwinCAT, Visual C++, Microsoft OLE DB, SQL

Annotation

This bachelor´s thesis describes designing of diagnostics application for press controlled by Beckhoff TwinCAT. Application monitors press regularly and informs staff in case of breakdown.

Diagnostics application is created in high-level programming language Visual C++ in Microsoft Visual Studio environment. Application communicates with TwinCAT via .NET Framework library TwinCAT::Ads, which is part of TwinCAT. Breakdown texts are read from database via Microsoft OLE DB interface using SQL.

Keywords

Diagnostics, TwinCAT, Visual C++, Microsoft OLE DB, SQL

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne na základe pokynov a rád vedúceho práce a že som všetky použité literárne a internetové zdroje uviedol v zozname použitej literatúry.

V Brne dňa: 26.5.2010

.....

Ľubomír ZIGO

Poďakovanie

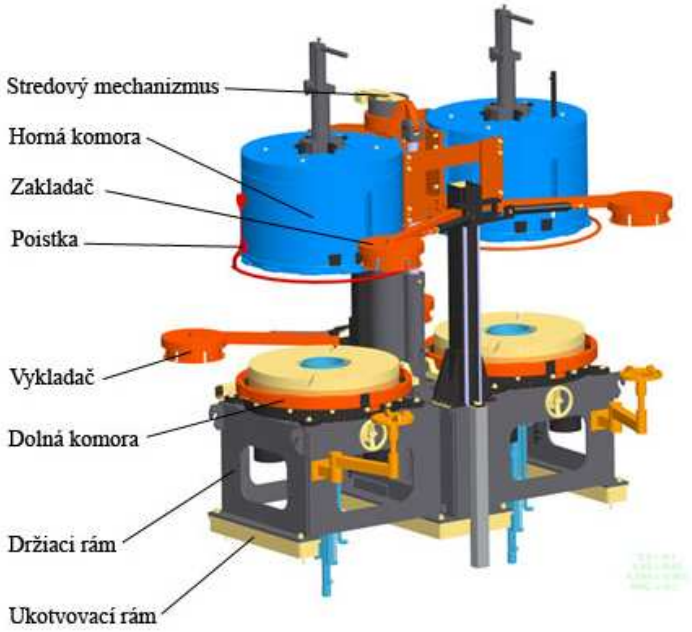
Chcel by som poďakovať všetkým, ktorí mi pomohli pri vypracovaní tejto bakalárskej práce. Predovšetkým by som rád poďakoval vedúcemu svojej práce Ing. Jiřímu Krejsovi Ph.D. za rady a pomoc a taktiež svojmu otcovi Ing. Ľubomírovi Zigovi za praktické rady z oblasti gumárenského priemyslu.

Obsah

II. Praktická časť

1. Úvod

V súčasnosti je výroba pneumatík plne automatizovaný proces, ktorý prebieha na gumárenských lisoch. Jedným z takýchto lisov je aj model Matador CP45. Keďže tento stroj je pomerne drahé zariadenie, základnou požiadavkou je, aby pracoval neustále bez zbytočných prestojov, ktoré sú spôsobené poruchami lisu. Tieto prestoje treba čo najviac skrátiť, aby tak neklesala výrobná kapacita stroja. Výroba jednej pneumatiky trvá 5-10 minút podľa jej veľkosti. Preto aj krátke prestoje predstavujú veľké straty, keďže výrobná prevádzka obsahuje viac ako sto lisov.



Obr.1 Model gumárenského lisu Matador CP45.[7]

Gumárenský lis je pomerne zložité zariadenie a preto aj nájdenie príčiny prestoja nie je vždy jednoduché. Niekedy sa podarí obsluhe nájsť príčinu rýchlo, ale vo väčšine prípadov to zaberie viac času. Veľmi pritom záleží na skúsenostiach a znalostiach obsluhy. V prípade, že obsluha nie je schopná nájsť príčinu, musí zavolať špecialistu údržby. Tento postup je ale časovo náročný a spôsobuje veľké prestávky vo využití zariadenia. Pritom často ide o prestoje, ktoré by sa dali odstrániť veľmi rýchlo, keby mala obsluha nejakú správu o príčine.

Riešením tohto stavu a cieľom bakalárskej práce je diagnostická aplikácia, ktorá na obrazovke riadiaceho systému zobrazí správu, určujúcu príčinu nespustenia alebo prerušenia automatického cyklu stroja. Táto aplikácia pozostáva z dvoch častí. Prvá časť sa doplní priamo do riadiaceho softvéru stroja vo forme nových programových blokov. Táto časť bude sledovať činnosť stroja a v prípade poruchy nastaví príslušnú poruchovú premennú. Druhá časť bude samostatná aplikácia vytvorená vo vyššom programovacom jazyku (Visual C++) a bude realizovať vizualizáciu poruchových premenných na obrazovke vo forme poruchového okna s textami všetkých vzniknutých porúch. Vizualizačná aplikácia komunikuje s riadiacou aplikáciou prostredníctvom klienta vytvoreného pomocou knižnice štandardu Microsoft .NET Framework. Táto knižnica je k dispozícii na webových stránkach dodávateľa riadiaceho softvéru firmy Beckhoff. Na stránke sú k dispozícii aj vzorové príklady, ktoré boli základom pre

vytvorenie tejto aplikácie. Zobrazované texty zodpovedajúce príslušnej poruchovej premennej sú uložené v databáze vytvorenej vo formáte Microsoft Access 2003. Táto forma uloženia textov umožní pracovníkom údržby voľne editovať a dopĺňať poruchové texty. Na komunikáciu s touto databázou sa použilo rozhranie Microsoft OLE-DB a na vytvorenie dotazov do databázy sa využil databázový jazyk SQL.

Táto práca sa skladá z teoretickej a praktickej časti. V teoretickej časti sú zhrnuté informácie a poznatky potrebné pri tvorbe diagnostickej aplikácie. V praktickej časti je popísaný postup jej tvorby.

I. Teoretická časť

2. Diagnostika

Diagnostika je hľadanie a odstraňovanie porúch pri spustení alebo pracovnom cykle stroja. Poruchy môžu mať viacero príčin, v základe ich môžeme rozdeliť na softvérové a hardvérové. Softvérové sú spôsobené zlým nastavením alebo chybou programu. Hardvérové sú spôsobené zlým konštrukčným riešením alebo opotrebením materiálu. V niektorých prípadoch môže byť hardvér i softvér v poriadku, ale problém môže nastať v ich komunikácii. Ďalej môžeme poruchy rozdeliť na tie, ktoré vznikli neodbornou obsluhou stroja a tie, ktoré vzniknú z iných príčin, napríklad obmedzená životnosť komponentov.

Základným cieľom diagnostiky je nájsť miesto, kde nastala porucha a jej príčinu. Potom už je jej odstránenie jednoduché.

2.1 Druhy diagnostiky

V našom prípade budeme diagnostikovať 3 druhy porúch:

- *Diagnostika štartu* – pred spustením stroja do automatického cyklu ho musí obsluha dať do základného stavu. Pre skúsenú obsluhu to nie je problém, ale novej obsluhe to zaberie viac času. Nová diagnostika odbremení obsluhu od nutnosti dokonale poznať základný stav stroja. Obsluhe bude stačiť stlačiť tlačidlo automatického cyklu a všetky nesplnené podmienky sa zobrazia na zobrazovacej jednotke v okne vizualizačnej aplikácie. Nastavenie základného stavu tak miesto obsluhy skontroluje doplnená časť aplikácie a výsledok oznámi obsluhe vo výpise. Tým sa podstatne skráti čas na spustenie stroja do automatického cyklu.

- *Diagnostika pohybov* – činnosť stroja pozostáva z vykonávania určitých pohybov pomocou vzduchových alebo hydraulických valcov. Vykonanie každého pohybu sa kontroluje príslušnými snímačmi, ktoré určujú vykonanie ďalšieho pohybu. V prípade, že pohyb sa vykoná, ale nie je potvrdený snímačom, činnosť stroja nepokračuje ďalej. Doba tejto nečinnosti závisí len od obsluhy, kedy si tento stav všimne. Keďže obsluha obsluhuje obyčajne viac ako desať lisov niekedy to trvá dosť dlho. Doplnená diagnostická časť riadiaceho programu monitoruje každý pohyb a ak tento nie je vykonaný v určenom čase, zobrazí sa poruchová správa, ktorá na to upozorní obsluhu.

- *Diagnostika poruchy snímačov* – pohyblivé časti strojov majú na oboch koncoch pohybu snímač. Z tejto dvojice snímačov môže byť zopnutý vždy len jeden. Ak by snímače na opačných koncoch pohybu boli zopnuté naraz, znamená to, že nastala porucha jedného snímača z tejto dvojice. Spustenie automatického cyklu v takomto stave by mohlo viesť k havárií stroja. Aplikácia túto poruchu nájde a oznámi ju v poruchovom okne a pokiaľ nebude odstránená nedovolí spustiť automatický cyklus.

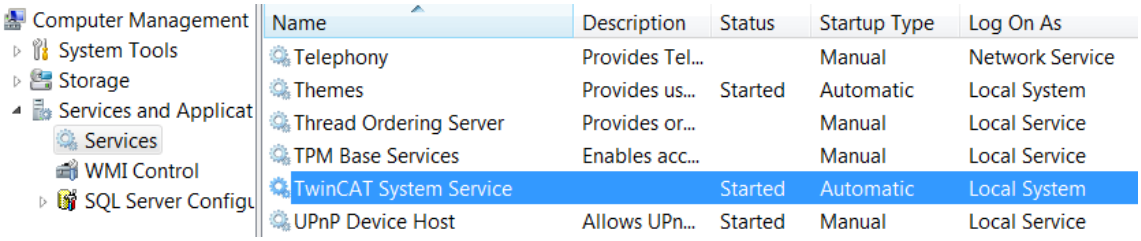
3. Beckhoff TwinCAT

Beckhoff TwinCAT systém dokáže premeniť štandardné PC s operačným systémom Windows na real-time riadiaci systém, ktorý obsahuje PLC systém, NC systém pre ovládanie servo pohonov a vývojové prostredie. [1]

Systém TwinCAT pozostáva z dvoch častí:

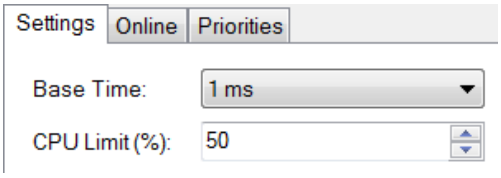
- Runtime - real-time riadiaci systém
- Vývojové prostredie pre tvorbu a konfiguráciu projektu

Prvá časť Runtime je služba Windows pod menom TwinCAT System Service (*TCATSysSrv.exe*), ktorá sa spúšťa ešte pred prihlásením do Windows.

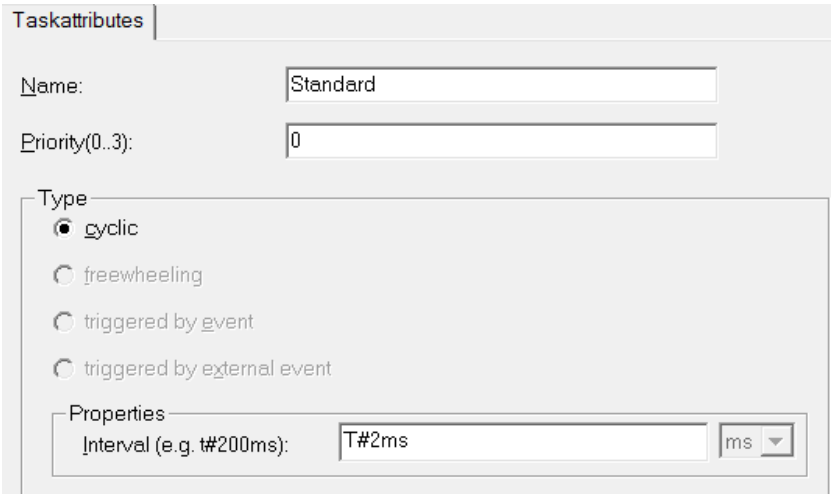


Obr.2 TwinCAT System Service ako služba Windows.

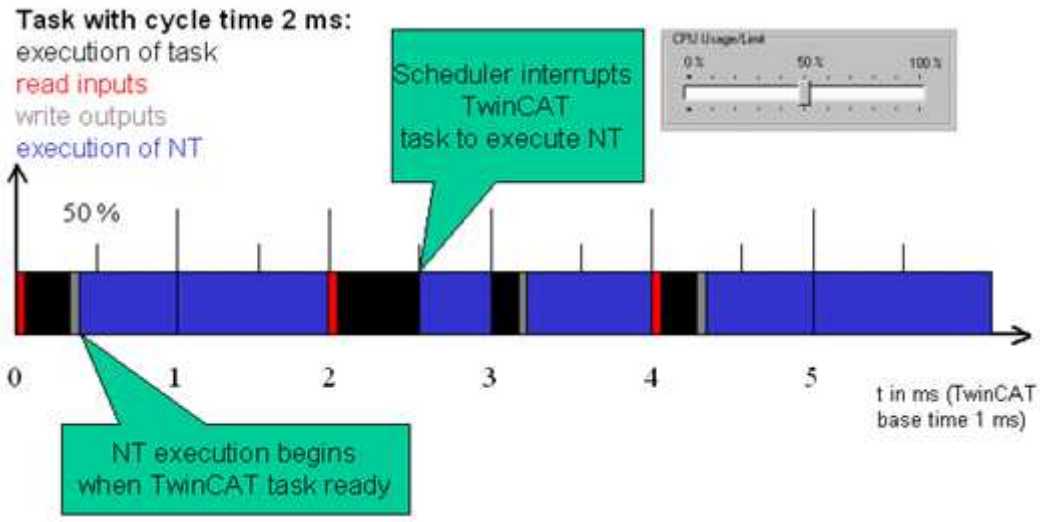
Princíp spolupráce medzi systémom TwinCAT a operačným systémom Windows
V nastaveniach systému sa nastaví čas (Base Time), ktorý určuje kedy sa bude systém spúšťať. Na zobrazenom obrázku to je 1 ms, čo znamená, že operačný systém každú 1 ms odovzdá riadenie systému TwinCAT. Ďalej sa nastaví limit využitia CPU. V tomto prípade je to 50%.



Obr.3 Nastavenie Base time a CPU Limit.



Obr.4 Nastavenie vykonávacieho intervalu.



Obr.5 Princíp rozdelenia riadenia medzi TwinCAT a operačný systém.[1]

Po štarte systému TwinCAT začína každú milisekundu činnosť systém TwinCAT a to načítaním okamžitých stavov vstupov, pokračuje spracovaním úloh projektu, končí zápisom výstupov. Po ukončení je činnosť odovzdaná operačnému systému. Ak systém TwinCAT nestihne ukončiť úlohy do nastavenej doby (50%), preruší svoju činnosť, odovzdá činnosť operačnému systému a počká na nasledujúci cyklus, kedy dokončí svoju činnosť.

Druhá časť systému TwinCAT je vývojové prostredie, ktoré slúžia na vytvorenie projektu. Každý projekt v TwinCATe sa skladá z dvoch častí, softvérovej a hardvérovej. Softvérová časť sa vytvára v aplikácii PLC Control a je uložená v súbore *MenoProjektu.pro*. Hardvérová časť sa vytvára v aplikácii System Manager a je uložená v súbore *MenoProjektu.tsm*. Tieto aplikácie spustíme zo systémovej lišty.

Obr. 6 Menu pre spustenie aplikácii a konfiguráciu systému.

3.1. TwinCAT PLC Control

PLC Control je aplikácia na vytváranie softvérovej časti projektu. Programy vytvorené touto aplikáciou môžu pracovať na viacerých hardvérových platformách a takisto môžeme pri ich vytváraní využiť viacero jazykov. TwinCAT PLC Control podporuje všetky programovacie jazyky definované normou IEC 61131-3 .

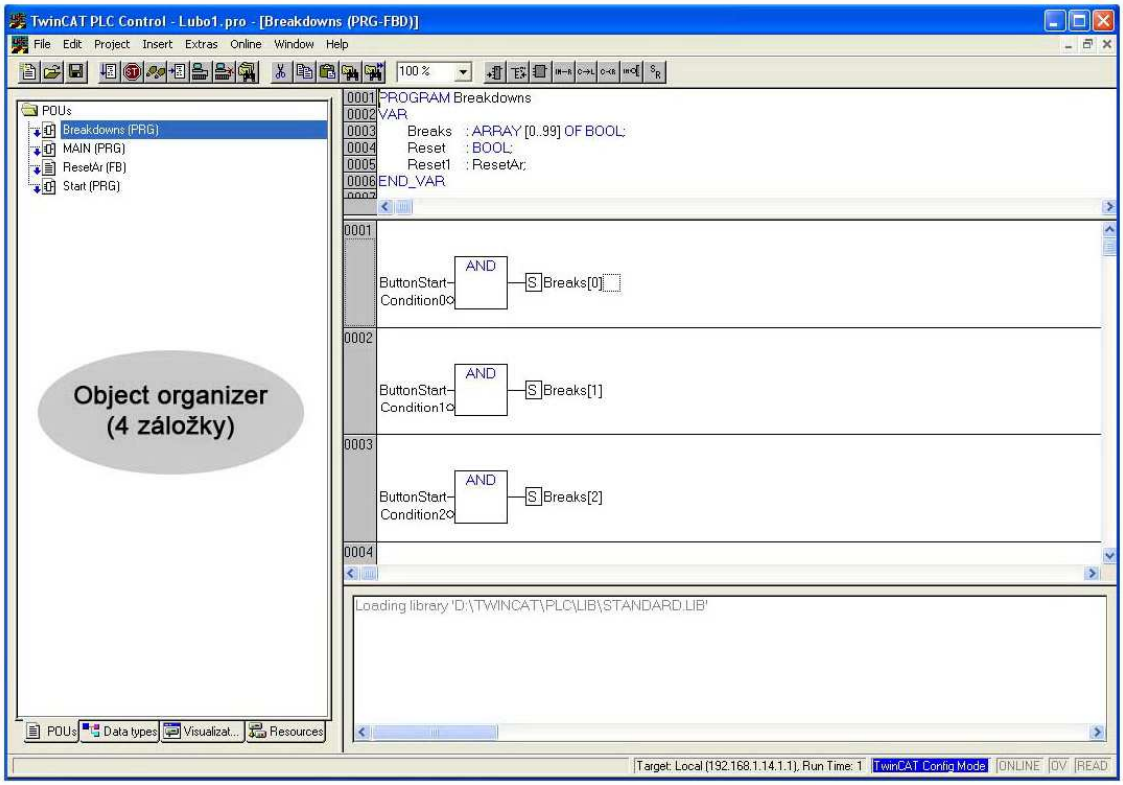
Textové jazyky:

- Instruction List (IL)
- Structured Text (ST)

Grafické jazyky:

- Function Block Diagram (FBD)
- Ladder Diagram (LD)
- Continous Function Chart (CFC)
- Sequential Function Chart (SFC)

Pri otvorení treba vybrať z ponuky hardvérovú platformu (PC, BC, CX) a programovací jazyk. Potom sa spustí samotné okno aplikácie PLC Control, kde vytvárame PLC program.



Obr.7 PLC Control s programom v jazyku FDB.

Na ľavej strane sa zobrazí Object Organizer so štyrmi záložkami.
Záložka *POUs* obsahuje všetky užívateľské programové jednotky: programy, funkčné bloky, funkcie, akcie.
Záložka *Data types* obsahuje užívateľské dátové typy.
Záložka *Visualisations* obsahuje pomocné vizualizácie.
Záložka *Resources* obsahuje globálne premenné, zoznam použitých knižníc, konfiguráciu alarmov, konfiguráciu úloh a pomocné funkcie na monitorovanie premenných.

Pravá časť je rozdelená na tri časti.
Prvá slúži na deklarovanie (lokálne premenné, počiatočné stavy premenných)
Druhá sa skladá zo sietí (network), ktoré obsahujú riadiace funkcie, ktoré tvoria samotný PLC program.
Tretia časť slúži na zobrazovanie informácii pre programátora (veľkosť programu, upozornenia, chyby).

Pre moju prácu sú veľmi dôležitou časťou PLC programu premenné, pretože tie budem monitorovať. Premenné programu sa delia na globálne a lokálne. Vstupy a výstupy sa deklarujú ako globálne premenné :

```
VAR_GLOBAL
        Vstup1   AT %I*     : BOOL;        (bez adresy)
        Vstup1   AT %I20.1 : BOOL;        (pri požadovanej adrese)
        Vstup2   AT %I*     : BOOL;
        Vystup1  AT %Q*  : BOOL;
        Vystup2  AT %Q*  : BOOL;
END_VAR
```

3.1.1 Spustenie programu

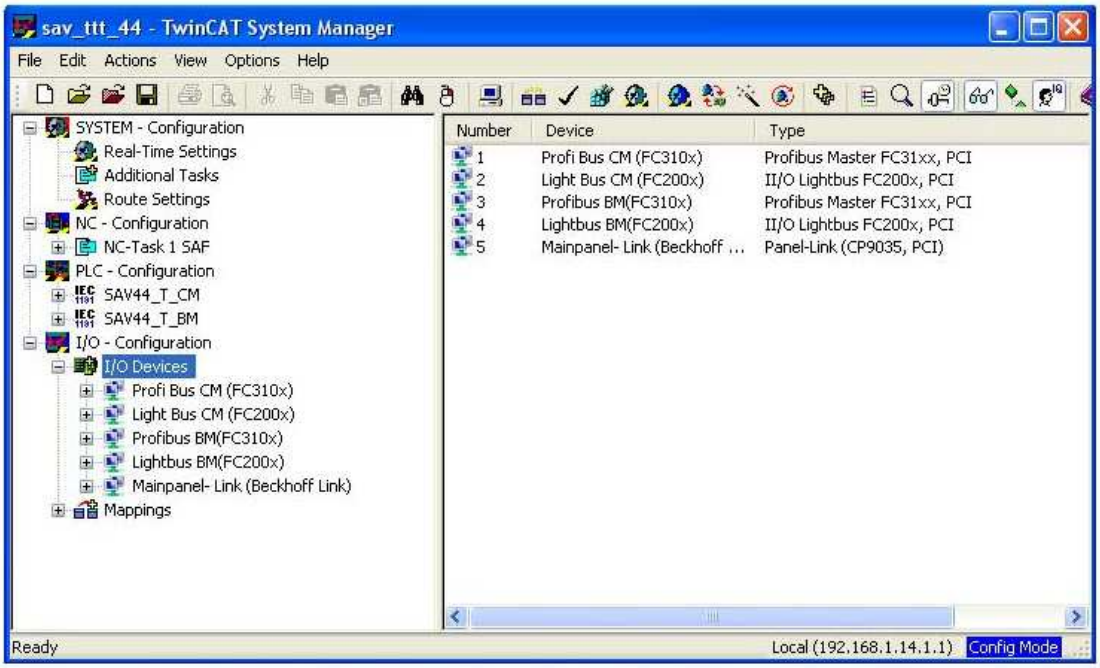
Po vytvorení všetkých potrebných programových jednotiek sa program skompiluje voľbou *Project – Rebuild All*. Ak je kompilovanie úspešné, vytvorí sa súbor *MenoProjektu.tpy*, ktorý obsahuje informáciu o vstupoch a výstupoch. Projekt sa zapíše do Boot adresára voľbou *Online – Create Bootproject*, čím sa pôvodný projekt prepíše. Vytvorený projekt sa zapíše do PLC voľbou *Online – Login*. Projekt sa spustí voľbou *Online – Run (F5).* Projekt sa vykonáva ako v skutočnom PLC a možno ho ladiť.

3.2. TwinCAT System Manager

System Manager je aplikácia na vytváranie hardvérovej časti projektu. Slúži na prepojenie projektov vytvorených v PLC Control s ovládaným zariadením. Jednotlivým premenným priradí reálne stavy snímačov, pohonov, atd. prostredníctvom komunikačného rozhrania.

3.2.1 Pripájanie externých zariadení

Prvým krokom pri spájaní PLC programu s reálnym zariadením je určenie PLC programu, s ktorým budeme v System Manageri pracovať. Ďalej musíme určiť rozhranie, cez ktoré bude prebiehať komunikácia a následne sa vyberú jednotlivé zariadenia.



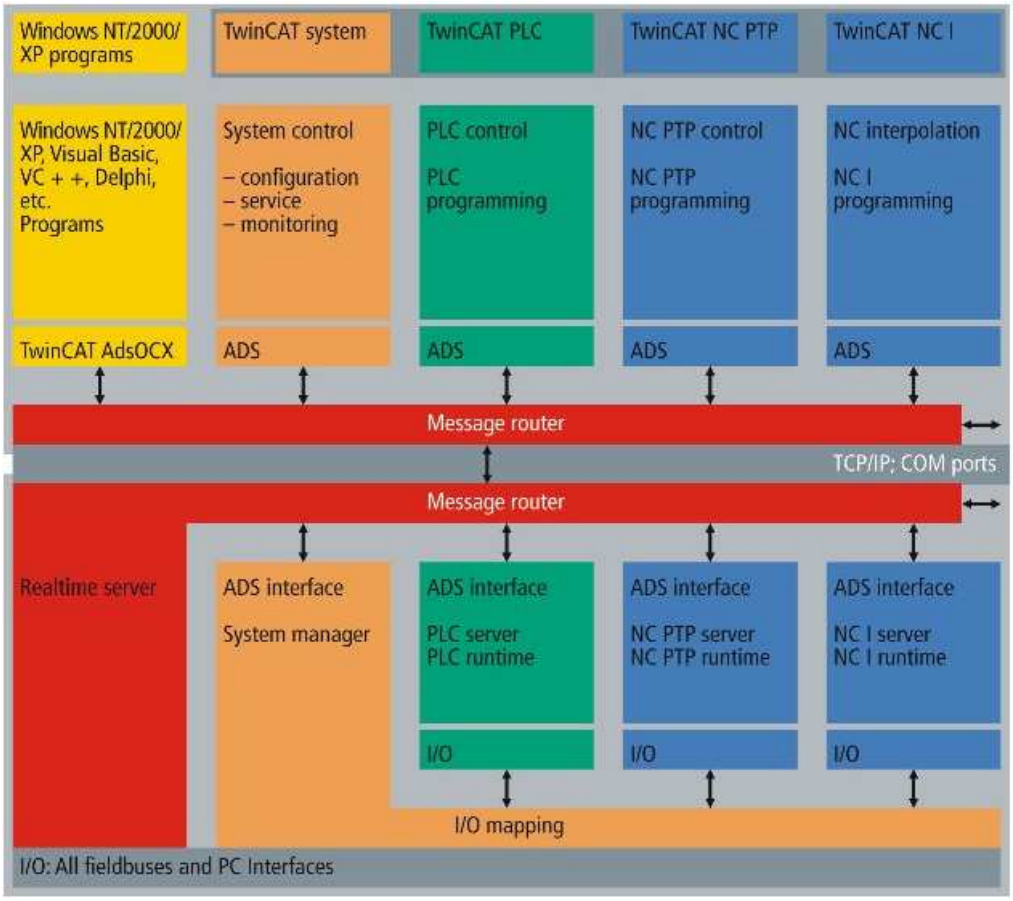
Obr.8 Okno System Managera.

K položke *PLC Configuration* treba voľbou *Append PLC Project* priradiť súbor *MenoProjektu.tpy*. K položke *I/O Configuration – I/O Devices* sa voľbou *Append Device* vyberie komunikačné rozhranie (Lightbus, Profibus, Ethernet, EtherCAT...). K tomuto rozhraniu sa voľbou *Append Box* postupne pridávajú jednotlivé zariadenia zo zobrazeného katalógu. Servopohony sa pridávajú do položky *NC Configuration* voľbou *Append task*. K tejto položke sa postupne pridajú jednotlivé osy voľbou Append axis. Jednotlivé osi sa môžu parametrizovať na zobrazenej obrazovke vpravo. V online režime je možné zariadenia načítať voľbou *Scan Devices*. Po vytvorení hardvérovej konfigurácie, treba prilinkovať logické vstupy a výstupy s fyzickými. Vyberie sa každý jednotlivý vstup a výstup a voľbou *Linked to* sa mu priradí skutočný vstup alebo výstup. Hotová konfigurácia za zapíše voľbou *Actions – Activate Configuration*.

4. TwinCAT ADS Koncept

Architektúra systému TwinCAT dovoľuje pracovať s jednotlivými programovými modulmi (napr. TwinCAT PLC, NC, ...) ako so samostatnými zariadeniami. Pre každú úlohu existuje programový modul (server alebo klient). Servery sú programové jednotky, ktoré vykonávajú úlohy, ale správajú sa ako hardvérové zariadenia. Preto o nich hovoríme ako o virtuálnych zariadeniach implementovaných v softvéri. Klienti sú programy, ktoré vyžadujú služby serverov (napr. vizualizácia). Toto umožňuje systému TwinCAT rozširovanie, pretože stále môžeme pridávať nové servery a klientov pre ďalšie úlohy ako osciloskopy, PID regulátory atd.[1]

Správy medzi týmito objektmi sa posielajú cez ADS rozhranie prostredníctvom tzv. message routera. Ten riadi a distribuuje všetky správy v systéme i správy cez TCP/IP pripojenia. Toto dovoľuje všetkým serverom a klientom vzájomne posielať príkazy, dáta, správy, stavové informácie atd.



Obr.9 TwinCAT device koncept založený na ADS rozhraní.[1]

ADS - PortNr	ADS Device description
100	Logger
110	Eventlogger
300	IO
301	Additional Task 1
302	Additional Task 2
500	NC
801	PLC RuntimeSystem 1
811	PLC RuntimeSystem 2
821	PLC RuntimeSystem 3
831	PLC RuntimeSystem 4
900	Camshaft controller
10000	System Service
14000	Scope

Tab.1 Zoznam ADS zariadení vytvorených na PC po inštalácii systému TwinCAT.[1]

Diagnostická aplikácia bude komunikovať s premennými zariadenia 801, čo je PLC Runtime System 1 lokálneho PC .

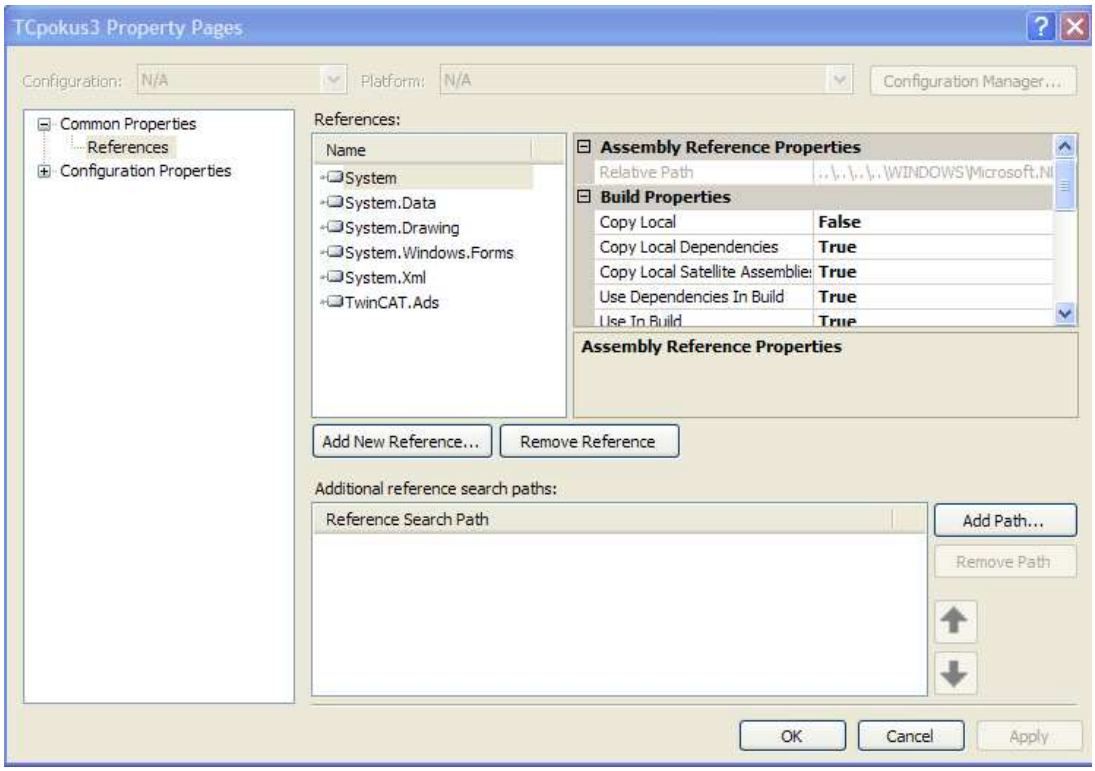
4.1 TwinCAT::ADS.NET knižnica

Systém TwinCAT ma viaceré rozšírenia. Jedným z nich je knižnica štandardu Microsoft .NET Framework, ktorá umožňuje prenos dát medzi TwinCATom a ďalšími aplikáciami.

Ak chceme túto knižnicu využívať v našej aplikácii vytvorenej vo Visual Studiu treba ju zahrnúť medzi používané knižnice. Potom môžeme v našej aplikácii využívať všetky triedy a funkcie, ktoré táto knižnica obsahuje.

Knižnicu pripojíme nasledovným spôsobom:
Pridáme ju do References našej aplikácie – pravým tlačidlom klikneme na názov projektu, zvolíme References, otvorí sa nám nasledujúce okno. Tam klikneme na Add New Reference a vyberieme našu knižnicu (zadáme cestu k nej).
Do kódu medzi používané namespace pridáme našu knižnicu a doplníme nasledovný riadok:

```
using System::IO, TwinCAT::Ads;
```



Obr.10 Okno References pre pripájanie knižníc.

Toto umožní prístup ku všetkým typom definovaným v TwinCAT::Ads knižnici bez ďalšieho uvádzania jej mena. Jadrom tejto knižnice je trieda TcAdsClient, ktorá umožňuje používateľovi komunikovať s ADS zariadeniami. Na začiatku musíme vytvoriť objekt tejto triedy, ktorý sa pripojí k ADS zariadeniu metódou Connect.

4.2 Použité triedy a metódy

Knižnica TwinCAT.Ads obsahuje tieto triedy:

- AdsBinaryReader
- AdsBinaryWriter
- AdsDatatypeNotSupportedException
- AdsErrorException
- AdsInvalidNotificationException
- AdsNotificationErrorEventArgs
- AdsNotificationEventArgs
- AdsNotificationExEventArgs
- AdsStateChangedEventArgs
- AdsStream
- AdsSymbolVersionChangedEventArgs
- AmsAddress
- AmsNetId
- AmsRouterNotificationEventArgs
- TcAdsClient
- TcAdsSymbolInfo
- TcAdsSymbolInfoCollection
- TcAdsSymbolInfoLoader

Pre vytvorenie vizualizačnej časti aplikácie boli použité len niektoré triedy.

4.2.1. Trieda TcAdsClient

Táto trieda je jadrom knižnice *TwinCAT.Ads*. Prvým krokom je vytvorenie objektu tejto triedy, je to tzv. komunikačný klient, ktorý nám umožní prácu s ADS zariadením, v našom prípade je to *PLC Control – Runtime 1*.

Vytvorenie ADS klienta volaním jeho konštruktora. [2]

```
TcAdsClient  ^tcClient = gcnew TcAdsClient();
```

Metóda *Connect* – prostredníctvom tejto metódy sa komunikačný klient aplikácie spojí s Ads zariadením

Vytvorenie spojenia ADS klienta s ADS zariadením PLC Control – Runtime 1. [2]

```
tcClient -> Connect(safe_cast<int>(AmsPort::PlcRuntime1));
```

Metóda *CreateVariableHandle* – touto metódou sa vytvorí spojenie s konkrétnou premennou v ADS zariadení. V našom prípade je to pole *Poruchy*.

Vytvorenie spojenia ADS klienta s premennou ADS zariadenia. [2]

```
hVarR = tcClient ->CreateVariableHandle(“Poruchy”);
```

Metóda *DeleteVariableHandle* – je to opak metódy CreateVariableHandle, ukončí spojenie s konkrétnou premennou v Ads zariadení.

Zrušenie spojenia ADS klienta s premennou ADS zariadenia. [2]
```
tcClient ->DeleteVariableHandle(hVarR);
```

4.2.2. Trieda AdsStream

Táto trieda slúži vytváranie komunikačných buffrov, tzv. streamov. Prostredníctvom týchto buffrov bude prebiehať všetka komunikácia medzi TwinCATom a diagnostickou aplikáciou.

Vytvorenie komunikačného buffera s veľkosťou *lenght*. [2]
```
dataStream = gcnew AdsStream(lenght);
```

4.2.3. Trieda AdsBinaryReader

Prostredníctvom tejto triedy vytvoríme jej objekt, je to tzv. čítač. Tento objekt načítava údaje z PLC Control prostredníctvom buffra triedy AdsStream. Použitím niektorej z metód tejto triedy načítame premenné rôznych dátových typov:

- Read
- ReadBoolean
- ReadByte
- ReadInt32
- ReadPlcString
- ReadPlcTIME

Vytvorenie čítača načítavajúceho údaje cez buffer dataStream. [2]
```
adsRead = gcnew AdsBinaryReader(dataStream);
```

Načítanie premennej typu Boolean do poľa. [2]
```
pole[i] = adsRead->ReadBoolean();
```

4.2.4. Trieda AdsBinaryWriter

Prostredníctvom tejto triedy vytvoríme jej objekt, je to tzv. zapisovač. Tento objekt zapisuje do PLC Control prostredníctvom buffra triedy AdsStream.
Použitím niektorej z metód tejto triedy môžeme zapisovať premenné rôznych dátových typov:

- Write
- WritePlcString

Vytvorenie zapisovača zapisujúceho dáta cez buffer dataStream. [2]

```
adsWrite = gcnew AdsBinaryWriter(dataStream);
```

Zápis hodnoty *true* do premennej typu Boolean. [2]

```
adsRead->Write(true);
```

5. Práca s databázami

Existuje mnoho druhov databáz, ale v dnešnej dobe sú najpoužívanejšie tzv. relačné databázy. V týchto databázach sú dáta zoradené do jednej alebo viacerých tabuliek. Každá tabuľka sa skladá z riadkov a stĺpcov. Každý riadok obsahuje všetky informácie o jednej položke, každý stĺpec obsahuje jednu informáciu o každej položke.

V praxi sa veľké relačné databázy zvyčajne skladajú z veľkého množstva tabuliek. Každá tabuľka má zvyčajne len pár polí a množstvo záznamov (má veľa riadkov, málo stĺpcov). Dôvodom je zlepšenie dotazovacej schopnosti. S viacerými tabuľkami s menej poľami sa dá pracovať rýchlejšie ako s jednou tabuľkou s veľkým množstvom polí.

Relačné databázy môžeme vytvárať a upravovať viacerými spôsobmi. Na trhu je veľké množstvo databázových manažérov, ktoré umožňujú široké možnosti pri práci s databázami. Samozrejmosťou je pridávanie a vymazávanie buniek, updatovanie polí v záznamoch, ďalej sú tam zahrnuté možnosti na obmedzenie práce, závislé na oprávneniach daného používateľa. Rovnako ako môžeme pristupovať k informáciám z každej tabuľky, môžeme kombinovať záznamy z dvoch alebo viacerých tabuliek do novej tabuľky na základe ich vzťahov a získavať z nich informácie. Kombinovanie tabuliek týmto spôsobom nazývame spájanie tabuliek (table join). Na programovanie týchto operácií v relačných databázach môžeme využiť jazyk SQL, ktorý je podporovaný väčšinou databázových manažérov. Preložené z [3].

5.1 Základy SQL

SQL znamená Structured Query Language. Je to relatívne jednoduchý jazyk, vytvorený špeciálne pre pristupovanie a upravovanie informácií v relačných databázach.

Vytvorila ho spoločnosť IBM pre interne účely, ale v dnešnej dobe je celosvetovo používaný. SQL sám o sebe neexistuje ako programový balík, zvyčajne je súčasťou iného prostredia, či už je to databázový manažér alebo programovacie prostredie, ako napr. VisualBasic.NET, Java alebo C++. Prostredie obsahujúce SQL zabezpečuje činnosti ako vstup/výstup a komunikácia s operačným systémom, pričom SQL sa využíva na dotazovanie databázy.

SQL má príkazy pre získavanie, triedenie a updatovanie záznamov v tabuľke, pridávanie a mazanie záznamov i polí, spájanie tabuliek a ďalšie možnosti pre vytváranie a úpravu databáz. SQL príkazy sa zvyčajne, ale nie nevyhnutne píšu s bodkočiarkou (ako príkazy v C++) a kľúčové slová sa píšu veľkými písmenami.[3]

5.1.1 Získavanie dát s využitím SQL

Na získavanie dát používame príkaz SELECT. Tento príkaz má ale oveľa širšie využitie. Výsledkom príkazu SELECT je vždy tzv. recordset. Je to sada dát vytvorená podľa parametrov príkazu SELECT. Dáta recordsetu sú zoradené do tabuľky, s pomenovaním stĺpcov podľa pôvodnej tabuľky a riadkami, ktoré boli vybrané podľa podmienok v príkaze SELECT. Recordset vytvorený príkazom SELECT môže obsahovať len jeden záznam, môže však byť aj prázdny.

Zrejme najjednoduchšou operáciou je sprístupnenie všetkých záznamov z jednej tabuľky. Pokiaľ databáza obsahuje tabuľku Products, všetky jej záznamy môžeme získať nasledujúcim SQL príkazom:

```
SELECT * FROM Products;
```

Znak * znamená, že chceme pracovať so všetkými poľami databázy. Parameter nasledujúci za slovom FROM definuje tabuľku, s ktorou pracujeme. Záznamy získané príkazom SELECT nie sú ničím obmedzené, takže sa týmto spôsobom dostaneme ku všetkým druhom dát.
Ak pracujeme len s niektorými poľami, môžeme ich vybrať použitím názvov týchto polí oddelených čiarkami na mieste hviezdičky v predchádzajúcom príklade:

```
SELECT ProductID, UnitPrice FROM Products;
```

Týmto príkazom načítame z tabuľky Products iba stĺpce ProductID a UnitPrice.

Ak názov poľa obsahuje medzery, treba ho napísať do úvodzoviek, alebo hranatých zátvoriek.

```
SELECT "Product ID”,”Unit Price” FROM Products;
SELECT [Product ID],[Unit Price] FROM Products;
```

V prostredí C++ je vhodnejšie využívať hranaté zátvorky, pretože úvodzovky sa používajú pre označovanie reťazcov. Preložené z [3].

5.2 Microsoft OLE DB

OLE DB je rozhranie slúžiace na efektívne získavanie dát z rôznych zdrojov. Toto rozhranie sa skladá z viacerých komponentov. Pomocou neho môžu aplikácie vytvorené vo Visual Studiu získavať dáta z databáz, textových súborov a podobne.

OLE DB využíva model poskytovateľov a spotrebiteľov. Spotrebiteľ zadá požiadavku na dáta, poskytovateľ spracuje túto požiadavku a odovzdá potrebne dáta späť spotrebiteľovi. Poskytovateľ musí byt pripravený na všetky požiadavky, ktoré môže spotrebiteľ zadať. Tieto požiadavky musia v ňom byť implementovane.

Podľa definície je spotrebiteľom každý systém alebo aplikácia, ktoré získavajú akékoľvek dáta cez rozhranie OLE DB. Toto rozhranie je implementovane v poskytovateľovi. Poskytovateľ je akýkoľvek programový komponent, ktorý využíva OLE DB rozhranie na sprístupnenie dát a odovzdávanie ďalším objektom, tj. spotrebiteľom.[4]

5.2.1 Model OLE DB

OLE DB pozostáva z viacerých komponentov.

Komponenty na pripojenie ku zdroju dát a ich zobrazenie:

- Data sources
- Sessions
- Commands
- Rowsets

Komponenty pre pracú so zobrazenými dátami:

- Accessors
- Transactions
- Enumerators

5.2.2 Programovanie v architektúre OLE DB

K dátam môžeme pristupovať tiež programovo prostredníctvom sprostredkovateľa OLE DB v jazyku Visual C++. OLE DB je rozhranie pre prístup k dátam, ktoré komunikuje so zdrojom dát kompatibilnými s technológiou OLE DB a umožňuje načítať dáta, pracovať s nimi a aktualizovať ich. To sa dá urobiť definovaním pripojovacieho reťazca vo vlastnosti ConnectionString metódy Open objektu Connection a predaním informácii o pripojení sprostredkovateľa OLE DB.[6]

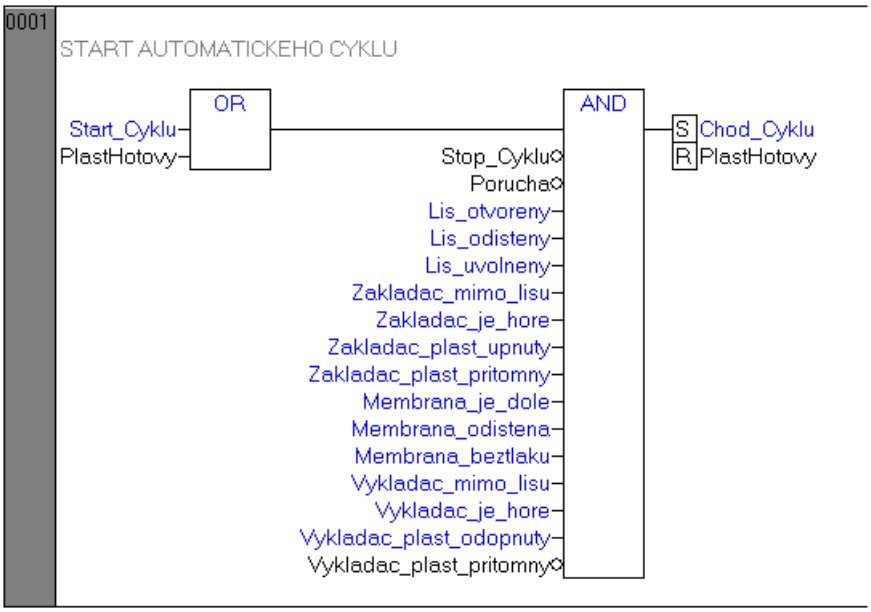
6. PLC Program

PLC program riadi automatický cyklus lisu na výrobu pneumatík. Po spustení sa začnú vykonávať jednotlivé funkcie. Jedná sa o sled činností, ktorých výsledkom je hotový plášť. Zakladač s polotovarom sa presunie do lisu, kde sa polotovar presunie na membránu. Do membrány sa pustí malá para, ktorá polotovar čiastočne vyformuje do tvaru plášťa. Zakladač sa presunie mimo lisu a ten sa zatvorí, potom zaistí a následne sa začne stláčať. Keď je lis zatvorený a zaistený malá para sa vypne a do lisu sa pustí para s vyšším tlakom, ktorou sa polotovar vulkanizuje po dobu 20s. Po tomto čase sa para uzavrie, lis sa odistí, otvorí a vykladač vyberie z neho hotový plášť. Tým sa lis vráti do pôvodného stavu a je pripravený na výrobu ďalšieho plášťa.

Tento program je vytvorený v prostredí PLC Control. Hotový program sa skompiluje a nahrá sa na PLC Runtime1, kde sa spustí a riadi lis.

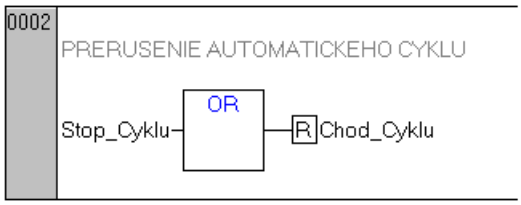
6.1 Popis súčasného stavu

Automatický cyklus sa spustí len v prípade, že sú splnené všetky podmienky vo funkcii, ktorá riadi spustenie automatického cyklu. Po poruche alebo inom prerušení sa cyklus spustí tlačidlom Štart. Ak cyklus úspešne skončí, ďalší cyklus spustí snímač, ktorý sleduje dokončenie plášťa, pričom nie je potrebné použiť tlačidlo pre spustenie.



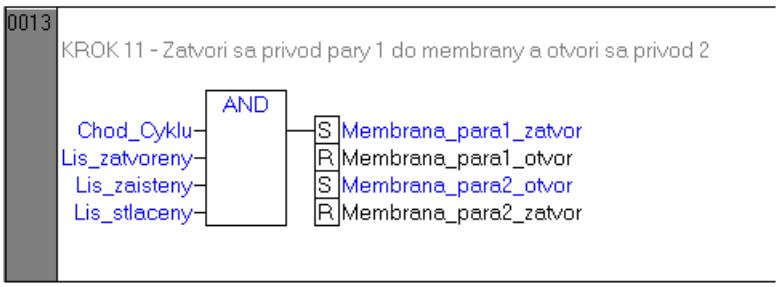
Obr.11 Funkcia Štart automatického cyklu.

Automatický cyklus sa preruší stlačením tlačidlo Stop_Cyklu. Po doplnení diagnostickej časti, sa chod cyklu preruší vždy po náleze poruchy.



Obr.12 Funkcia pre prerušenie cyklu.

V prípade, keď sa cyklus dostane do stavu, že lis je zatvorený, zaistený a stlačený, malá para sa vypne a spustí sa para s vyšším tlakom, ktorou sa polotovar vulkanizuje.



Obr.13 Funkcia pre prepnutie pary.

Táto kapitola bola spracovaná podľa [7].

II. Praktická časť

7. Doplnenie PLC programu o diagnostickú časť

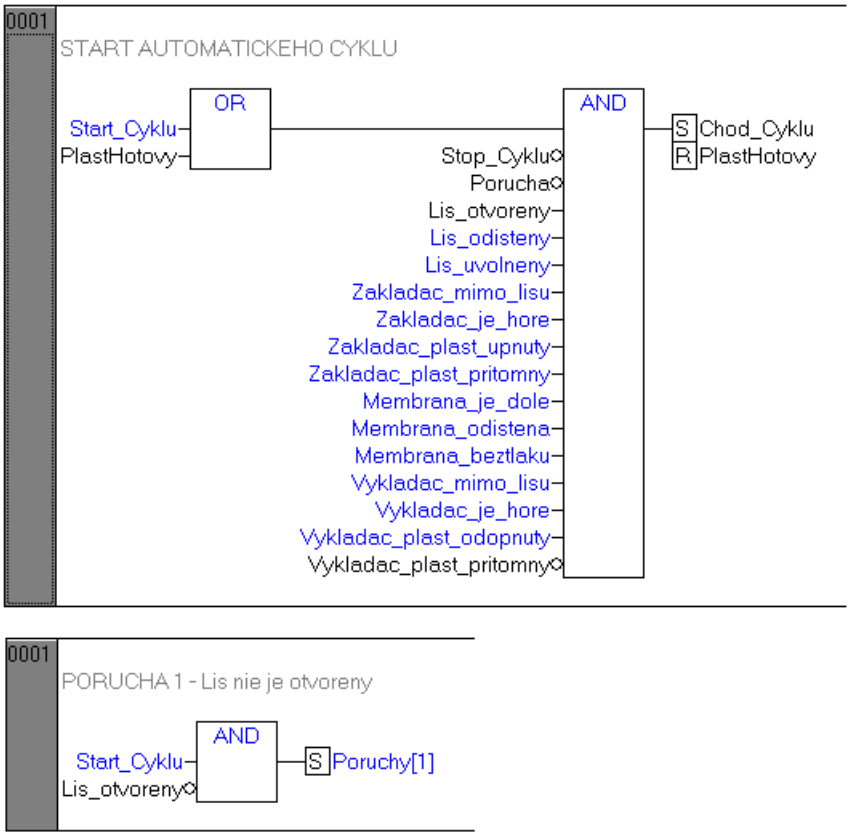
Pôvodný PLC program obsahuje automatický cyklus, ktorý riadi lis. V tomto programe nie je vyriešená diagnostika porúch. Doplnenie diagnostiky je podmienkou pre zobrazovanie týchto porúch prostredníctvom diagnostickej aplikácie. Diagnostika sa skladá z funkcii, ktoré ovládajú pole porúch (pole Poruchy). Pri poruche nastavia daný prvok poľa porúch na hodnotu *True*. Toto pole ďalej spracováva diagnostická aplikácia a takto zobrazuje obsluhe, kde nastala porucha.

V PLC programe nastávajú 3 druhy porúch, takže aj diagnostika sa skladá z 3 častí.

7.1. Diagnostika štartu

Lis sa spustí po stlačení tlačidla *Štart* len ak sú splnené podmienky, ktoré sú pre spustenie nutné. Ak nie je niektorá podmienka splnená, lis sa nespustí. Pri spustení ďalšieho cyklu už nie je potrebné stlačiť tlačidlo *Štart*, ďalší cyklus spustí premenná *PlastHotovy*. Tento proces sa opakuje, kým nenastane porucha.

Príklad: Obsluha stlačí tlačidlo Štart. Automatický cyklus sa nespustí, pretože lis nie je úplne otvorený. Zakladač pri pohybe do lisu by do neho mohol naraziť. Diagnostická funkcia to vyhodnotí ako poruchu a tá neumožní spustenie lisu.

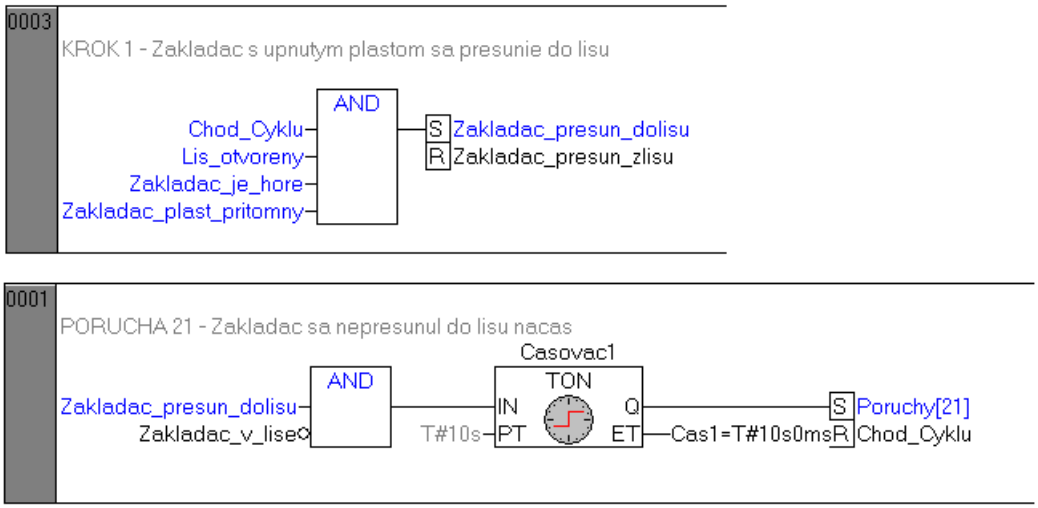


Obr.14 Funkcia pre diagnostiku štartu.

7.2. Diagnostika pohybov

Porucha môže nastať pri pohybe ktorejkoľvek časti lisu. Zisťovanie týchto porúch vyriešime časovačom, ktorý bude kontrolovať, či sa daný pohyb vykoná v časovom limite. Príslušný časovač vždy nastavíme na čas o niečo dlhší ako trvanie pohybu a tento bude kontrolovať, či sa pohyb ukončil načas.

Príklad: Zakladač sa začne presúvať do lisu a súčasne ešte nie je v lise, takže je splnená podmienka *AND*, čím sa spustí časovač. Keď bude zakladač v lise poruší sa podmienka a časovač sa vypne. Ak sa zakladač nedostane do lisu v priebehu 10s, aplikácia to vyhodnotí ako poruchu a príslušný prvok poľa porúch nastaví na *True* a súčasne preruší automatický cyklus.

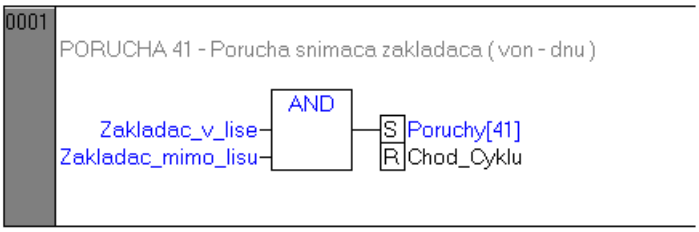


Obr.15 Funkcia pre diagnostiku pohybov.

7.3. Diagnostika poruchy snímačov

Poruchu stroja môže byť spôsobená aj poruchou niektorého snímača. Dva snímače, ktoré sú v koncových bodoch pohybu jednej časti nemôžu byť zopnuté naraz. Ak sa to stane, znamená to, že jeden z týchto snímačov je nefunkčný a treba ho vymeniť. Tento stav je nebezpečný pretože môže spustiť nesprávny úkon, ktorého následkom môže byť poškodenie zariadenia.

Príklad: Môže nastať stav, že zakladač bude v lise, ale poškodený snímač ukáže, že je mimo lisu. Toto môže spôsobiť zatvorenie lisu a poškodenie zakladača zatvárajúcim sa lisom. Zobrazená funkcia spôsobí prerušenie automatického cyklu.



Obr.16 Funkcia pre diagnostiku snímačov.

8. Vizualizácia

Prostredníctvom doplnenia diagnostickej časti v PLC programe je možné nájsť poruchu. Príčina poruchy je síce známa, ale aby mohla byť odstránená treba ju zobraziť pre obsluhu stroja, ktorá nemá prístup do programu PLC Control a nedokáže ju nájsť priamo v ňom.

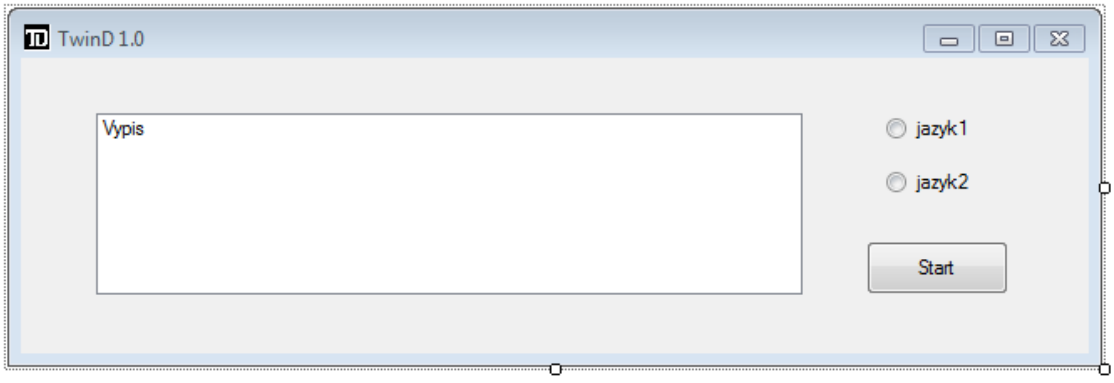
Pre tento účel je vytvorená diagnostická aplikácia, ktorá nájdenú poruchu zobrazí obsluhe. Pri chode automatického cyklu bude aplikácia minimalizovaná v systémovej lište. Ak nastane porucha, aplikácia sa maximalizuje, pričom zobrazí kde nastala porucha. Po odstránení poruchy a jej potvrdením obsluhou sa aplikácia opäť minimalizuje do pôvodného stavu. Obsluha lisu bude schopná samostatne odstraňovať poruchy okamžite, čo skráti prestoje vo výrobe na minimum. Diagnostická aplikácia je vytvorená v programovacom prostredí Microsoft Visual Studio v jazyku Visual C++.

8.1 Design GUI

Aplikáciu s názvom *TwinD* vytvoríme prostredníctvom ako nový projekt typu Windows Forms Applications, čo je šablóna pre vytváranie programov s využitím štandardu .NET Framework. Do tejto šablóny vložíme potrebné prvky.

- List Box pre výpis poruchy
- Tlačidlá pre voľbu jazyka (jazyk1, jazyk2)
- Tlačidlo pre spustenie samotnej diagnostiky
- Časovač

Tlačidlá pre voľbu jazyka nemajú priamo určený text, tento text sa tiež načíta z databázy pri spustení aplikácie. Toto umožní meniť týmto spôsobom používané jazyky bez potreby zasahovať do zdrojového kódu aplikácie. Jazyky stačí prepísať v databáze poruchových textov.



Obr.17 Design GUI.

8.2 Nastavenie správania aplikácie

Po spustení aplikácie obsluha vyberie jazyk, v ktorom chce zobrazovať poruchy. Tlačidlom Štart sa spustí samotná diagnostika lisu, pričom aplikácia sa zminimalizuje. Takisto sa z databázy načítajú všetky poruchové texty vo vybranom jazyku. Pri poruche sa aplikácia zobrazí z lišty a obsluhe oznámi príčinu poruchy. Po odstránení poruchy sa monitorovanie lisu opäť spustí tlačidlom Štart.

V prípade poruchy sa aplikácia zobrazí v zadanej veľkosti:

```
this->WindowState = System::Windows::Forms::FormWindowState::Normal;
this->ClientSize = System::Drawing::Size(x,y);
```

Po odstránení poruchy sa aplikácia zminimalizuje:

```
this->WindowState = System::Windows::Forms::FormWindowState::Minimized;
```

8.3 Komunikácia s TwinCATom

Diagnostická aplikácia v pravidelných intervaloch načíta z TwinCATu pole porúch. Všetky prvky tohto poľa typu *Boolean* sú primárne nastavené do stavu *False*. Pri poruche sa jeden z prvkov poľa nastaví do stavu *True*. Diagnostická aplikácia vyhodnotí, ktorý prvok poľa zmenil stav a na základe podmienok vyhodnotí, kde nastala porucha. Po potvrdení poruchy obsluhou lisu aplikácia tento prvok poľa vráti späť do stavu *False* a pokračuje v monitorovaní.

Aplikácia bude načítavať pole, ktorého veľkosť je nastavená na 100 prvkov a pri poruche zresetuje konkrétny prvok, kde nastala porucha. Nie je pravdepodobné, že nastanú 2 poruchy v jednom okamžiku. Na komunikáciu s TwinCATom využijeme knižnicu TwinCAT::Ads .NET, ktorá je súčasťou dodávaného softvéru.

8.3.1 Konštanty a premenné pre komunikáciu

Komunikáciu môžeme rozdeliť na 2 časti – čítanie a zápis. Pre každú z týchto častí potrebujeme rozdielne konštanty a premenné, pretože prenášané údaje majú rozdielnu veľkosť, takisto aj počet čítaných a zapisovaných premenných je rozdielny, preto aj buffery pre tieto činnosti musia mať rozdielnu veľkosť.

```
static const Int32 numberR = 10;            // pocet citanych premennych
static const Int32 numberW = 1;             // pocet zapisovanych premennych
static const Int32 varWidth = 1;            // sirka premennej pre Boolean = 1Byte
static const Int32 lenghtR = numberR * varWidth; // dlzka bufera pre citanie
static const Int32 lenghtW = numberW * varWidth; // dlzka bufera pre zapis
Int32 hVarR;                                // pomocna premenna pre AdsBinaryReader
Int32 hVarW;                                // pomocna premenna pre AdaBinaryWriter
static String^ PLCvarR =L".Poruchy";        // meno citanej premennej v PLC
static String^ PLCvarW =L".Reset";          // meno zapisovanej premennej v PLC
TcAdsClient^ tcClient;                      // deklaracia ADS klienta
AdsStream^ dataStreamR;                     // deklaracia bufra na citanie
AdsStream^ dataStreamW;                     // deklaracia bufra na zapis
AdsBinaryReader^ adsRead;                   // deklaracia citaca z bufra
AdsBinaryWriter^ adsWrite;                  // deklaracia zapisovaca do bufra
```

8.3.2 Vytvorenie komunikácie

Najdôležitejšou časťou knižnice TwinCAT::Ads je trieda *TcAdsClient*. Pri vytváraní komunikácie vytvoríme nový objekt tejto triedy. Je to tzv. komunikačný klient, ktorý sa metódou *Connect* pripojí k systému TwinCAT na Runtime1, kde beží náš program. Runtime1 beží na porte 801.

Vytvorenie noveho objektu triedy TcAdsClient [2]

```
tcClient = gcnew TcAdsClient;
```

Pripojenie tcClienta na port 801 priamo [2]

```
tcClient -> Connect(801);
```

Pripojenie tcClienta na port 801 cez adresu Runtime1 [2]

```
tcClient -> Connect(safe_cast<int>(AmsPort::PlcRuntime1));
```

8.3.3 Načítavanie

Pomocou metódy *CreateVariableHandle* sa spojíme s načítavanou premennou PLCvarR. Vytvoríme nový objekt triedy AdsStream, buffer dataStreamR s veľkosťou lenghtR. Ďalej vytvoríme nový objekt triedy AdsBinaryReader, čítač adsRead, ktorý bude načítavať dáta z TwinCATu a zapisovať do dataStreamR.

Vytvorenie spojenia s premennou v PLC [2]
```
hVarR = tcClient->CreateVariableHandle(PLCvarR);
```

Vytvorenie dátoveho buffera na čítanie [2]
```
dataStreamR = gcnew AdsStream(lenghtR);
```

Vytvorenie čítača z buffera [2]
```
adsRead = gcnew AdsBinaryReader(dataStreamR);
```

Jednoduché načítanie (tcClient prečíta *hVarR* a zapíše do buffera *dataStreamR*) [2]
```
tcClient -> Read (hVarR, dataStreamR);
```

Načítanie poľa typu Boolean
```
tcClient->Read(hVarR, dataStreamR);
dataStreamR->Position = 0;
for (int i = 0; i < polePor->Length ; i++)
      polePor[i] = adsRead->ReadBoolean();
```

8.3.4 Zápis

Pomocou metódy *CreateVariableHandle* sa spojíme so zapisovanou premennou PLCvarW. Vytvoríme nový objekt triedy AdsStream, buffer dataStreamW s veľkosťou lenghtW. Ďalej vytvoríme nový objekt triedy AdsBinaryWriter, zapisovač adsWrite, ktorý bude zapisovať dáta z buffera dataStreamW do TwinCATu.

Vytvorenie spojenia s premennou v PLC [2]
```
hVarW = tcClient->CreateVariableHandle(PLCvarW);
```

Vytvorenie dátového buffera na zápis [2]
```
dataStreamW = gcnew AdsStream(lenghtW);
```

Vytvorenie zapisovača do buffera [2]
```
adsWrite = gcnew AdsBinaryWriter(dataStreamW);
```

Jednoduchý zápis (tcClient zapíše obsah dataStreamW do premennej hVarW) [2]
```
tcClient -> Write(hVarW, dataStreamW);
```

Zápis do premennej typu Boolean
```
dataStreamW->Position = 0;
adsWrite->Write(true);
tcClient->Write(hVarW, dataStreamW);
```

8.4. Načítavanie porúch z databázy

Diagnostická časť PLC programu prostredníctvom doplnených diagnostických funkcii vyhodnotí, kde nastala porucha. Aby mohla byť porucha odstránená obsluhou lisu, je potrebné obsluhe oznámiť, kde porucha nastala. Aplikácia teda musí pre každú možnú poruchu zobraziť správny text, miesto a príčinu poruchy. Všetky tieto texty sú uložené v databáze poruchových textov. Databáza je vytvorená vo formáte Microsoft Access 2003. Aplikácia teda musí túto databázu načítať a pre každú vzniknutú poruchu zobraziť daný poruchový text na monitore automatu. Na vytvorenie spojenia medzi diagnostickou aplikáciou a databázou poruchových textov využijeme rozhranie Microsoft OLE DB a jazyk SQL.

8.4.1 Vytvorenie spojenia

Vytvorenie spojenia s databázou využíva rozhranie Microsoft OLE DB, preto musíme jeho knižnicu pridať medzi používané *namespaces*.

```
using namespace System::Data::OleDb;
```

Do reťazca *databaza* zadáme názov databázy s ktorou budeme pracovať. V prípade, že nie je v rovnakej zložke ako program, treba zadať celú programovú cestu.
Do reťazca *provider* zadáme pripojovací reťazec (*connection string*) pre formát Microsoft Access 2003. Pripojovacie reťazce pre rôzne typy súborov nájdeme na stránke http://www.connectionstrings.com.
Do reťazca *connection* zadáme poskytovateľa spojenia (reťazec *provider*) a databázu, s ktorou pracujeme (reťazec *databaza*).
Do reťazcov *queryIndex*, *queryEnglish*, *querySlovak* zadáme SQL príkazy, ktorými z databázy načítame potrebné dáta.

```
String^ databaza = L"Poruchy1.accdb";           // databaza s poruchami
String^ databaza = L"C:\\Users\Documents\Poruchy1.accdb";  //cesta k databaze
String^ provider = L"Microsoft.Jet.OLEDB.4.0"; // database provider for Access 2003
connection = L"Provider=" + provider + "; Data Source=" + databaza; // connection string
queryIndex = L"SELECT Index FROM Texty";            // query string pre pole Index
queryEnglish = L"SELECT English FROM Texty";        // query string pre pole English
querySlovak = L"SELECT Slovak FROM Texty";          // query string pre pole Slovak
```

Aby sme mohli v diagnostickej aplikácii pracovať s databázou musíme vytvoriť potrebné premenné na uloženie dát. Pole *pIndex* slúži na načítanie indexov porúch, pole *pText* slúži na načítanie poruchových textov.

```
static array<String^>^ pIndex = gcnew array<String^>(numberR); // pole cisla poruchy
static array<String^>^ pText = gcnew array<String^>(numberR);  // pole textu poruchy
```

Vytvorili sme všetky potrebné náležitosti pre vytvorenie spojenia s databázou, na samotné spojenie použijeme funkciu *ReadOLEdataToArray*.

8.4.1.1 Funkcia ReadOLEdataToArray

Táto funkcia vytvorí samotné spojenie s databázou, čo umožní využívať dáta z databázy v našej aplikácii. Využije pritom reťazec *connection*, ktorý bol popísaný s predošlej časti. Funkcia vytvorí spojenie so zdrojovou databázou, vytvorí príkaz pre načítavanie dát a otvorí spojenie metódou *Open*. Ďalej vytvorí nový objekt triedy OleDbDataReader, čítač *reader*, ktorý načítava údaje z dátového prúdu metódou *ExecuteReader*. Pomocou cyklu *while* načíta všetky potrebné prvky do daného pola a nakoniec zatvorí dátový prúd.

```
private: System::Void ReadOLEdataToArray(String^ connectionString, String^ queryString,
array<String^>^ vArray)

        {
        // vytvorenie spojenia so zdrojom zadanym v spojovacom retazci
        OleDbConnection^ connection = gcnew OleDbConnection(connectionString);

        // vytvorenie prikazu
        OleDbCommand^ command = gcnew OleDbCommand(queryString, connection);

        // otvorenie spojenia
        connection->Open();

        // vytvorenie citaca z datoveho prudu napojeneho na datovy zdroj
        OleDbDataReader^ reader = command->ExecuteReader();

        // citanie riadkov zadaneho pola zdrojovej databazy premennej pole
        Int32 i = 0;
        while (reader->Read())
                {
                vArray[i] = reader[0]->ToString();
                i = i+ 1;
                }
        // zatvorenie datoveho prudu
        reader->Close();
        }
```

Funkcia bola prevzatá z [4].

8.4.2 Načítanie dát z databázy

Načítanie daného poľa databázy prebehne pomocou funkcie *ReadOLEdataToArray*, kde zadáme parametre, ktoré chceme načítať. V tomto prípade načítame indexy a poruchové texty v Jazyku1. Parameter *connection* udáva spojenie, z ktorého sa dáta načítajú. Parametre *queryIndex* a *queryJazyk1* udávajú SQL príkaz, ktorým sa vyberú potrebné dáta. Parametre *pIndex* a *pText* udávajú názvy polí, do ktorých budú dáta uložené.

```
ReadOLEdataToArray(connection, queryIndex, pIndex);
ReadOLEdataToArray(connection, queryJazyk1, pText);
```

8.4.3 Výpis

Výpis poruchových textov bude realizovaný prostredníctvom tzv. *listBoxu* s názvom *Vypis*, ktorý je súčasťou GUI aplikácie. Pri spustení prvok *listBox* vyprázdnime metódou *Clear*. Prvá funkcia vypíše do *listBoxu Vypis* všetky prvky uložené v poliach *pIndex* a *pText*, táto funkcia sa ale v samotnej aplikácii nevyužíva. Tam sa využíva druhá funkcia, ktorá vypíše len konkrétnu poruchu, ktorá nastala pri chode stroja.

```
// pomocny vypis vsetkych poruchovych textov
    private: System::Void Output(System::Void)
             {
                Vypis->Items->Clear();
                for(int i = 0 ; i < pIndex->Length ; i++)
                    Vypis->Items->Add(L"   " + pIndex[i]+ L"   " + pText[i]);
             }

// vypis poruchoveho textu do okna Vypis
    private: System::Void vypisPoruchy(System::Int32 index)
             {
                Vypis->Items->Add(pIndex[index]+ L" : " + pText[index]);
             }
```

9. Záver

Cieľom bakalárskej práce bolo vytvorenie diagnostickej aplikácie pre gumárenský lis riadený systémom Beckhoff TwinCAT. Základom bolo naštudovanie spôsobu komunikácie s TwinCATom prostredníctvom knižnice štandardu .NET Framework a jej implementácia pre správne fungovanie diagnostickej aplikácie.

V práci som popísal postup tvorby tejto aplikácie vytvorenej podľa potrieb priemyselnej výroby, v ktorej sa táto aplikácia bude používať. Pri tvorbe aplikácie som využil znalosti z programovania v prostredí Visual C++, programovania PLC, práce s databázami a spojil ich do komplexnej diagnostickej aplikácie, ktorá skráti prestoje vo výrobe a tým zvýši výrobnú efektivitu.

Aplikácia v pravidelných intervaloch monitoruje lis tým, že načítava pole porúch z programu PLC Control. Ak niekde nastane porucha, aplikácia ju oznámi obsluhe prostredníctvom monitoru riadiaceho automatu. Obsluha tak okamžite uvidí príčinu poruchy a podľa toho zvolí ďalší postup s cieľom minimalizovať dobu prestoja. Toto umožní aj menej skúsenej obsluhe odstrániť príčinu poruchy a následného prestoja vo výrobe a pracovníkom údržby tiež ušetrí čas pri odstraňovaní porúch, lebo nebudú musieť prácne hľadať príčinu poruchy.

Aplikácia bola vytvorená prostredníctvom vývojového prostredia Microsoft Visual Studio v jazyku Visual C++. Je určená pre diagnostiku riadiaceho systému Beckhoff TwinCAT. Na počítačoch bez tohto systému nie je funkčná. Aplikácia bola otestovaná na operačných systémoch Windows XP a Windows 7 s nainštalovaným systémom TwinCAT, kde pracovala podľa požiadaviek. Po spustení aplikácie obsluha vyberie jazyk, v ktorom chce aplikáciu používať. Tlačidlom Štart sa spustí monitorovanie lisu, pričom samotná aplikácia sa minimalizuje. Pri poruche sa aplikácia zobrazí, s tým že okamžite oznámi príčinu poruchy. Zobrazovanie porúch funguje v oboch jazykoch správne.

Aplikácia je určená pre použitie v priemyselnej výrobe, kde sa už určitý čas využíva. Zainteresovanými ľuďmi bola ohodnotená ako veľmi jednoduchá na používanie, ale zároveň funkčná a užitočná pre plynulosť výroby. Táto práca tak splnila svoj cieľ.

Literatúra

[1] Beckhoff Information System, 2007

[2] Beckhoff TwinCAT ADS .NET Component: Examples, 2007
www.beckhoff.com

[3] Horton I.: Ivor Horton's Beginning Visual C++ 2008, 2008
www.wrox.com

[4] Microsoft Developer Network
http://msdn.microsoft.com

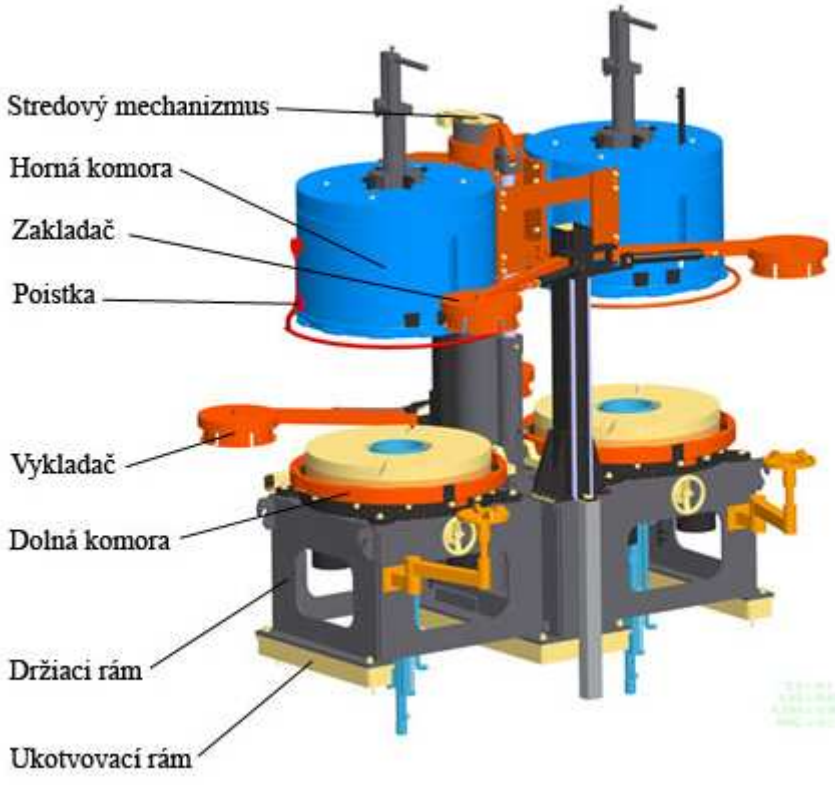
[5] The Connection String Reference
http://www.connectionstrings.com

[6] Domovská stránka lokality Office Online
http://office.microsoft.com

[7] Firemná dokumentácia Conti Machinery, 2006 - 2010

Prílohy

Príloha č.1: Schéma lisu Matador CP45



Obr. 18 Lis otvorený.[7]

Obr. 19 Lis zatvorený.[7]

Príloha č.2: Zdrojový kód diagnostickej aplikácie

```
#pragma once


namespace TwinD {

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
using namespace TwinCAT::Ads;               //z - pripojenie kniznice TwinCAT::Ads
using namespace System::Data::OleDb;        //z - pripojenie kniznice OleDb

/// <summary>
/// Summary for Form1
///
/// WARNING: If you change the name of this class, you will need to change the
///          'Resource File Name' property for the managed resource compiler tool
///          associated with all .resx files this class depends on.  Otherwise,
///          the designers will not be able to interact properly with localized
///          resources associated with this form.
/// </summary>
public ref class Form1 : public System::Windows::Forms::Form
{
public:
        Form1(void)
        {
                InitializeComponent();
                //
                //TODO: Add the constructor code here
                //
        }

protected:
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        ~Form1()
        {
        if (components)
                {
                        delete components;
                }
        }
private: System::Windows::Forms::ListBox^  Vypis;
protected:

protected:
private: System::Windows::Forms::Button^  buttonStart;
private: System::Windows::Forms::RadioButton^  radioButton1;
private: System::Windows::Forms::RadioButton^  radioButton2;
private: System::Windows::Forms::Timer^  timerRead;
private: System::ComponentModel::IContainer^  components;

private:
        /// <summary>
        /// Required designer variable.
        /// </summary>

//z premenne pre komunikaciu s TwinCAT

static const Int32 numberR = 100;                   //z pocet citanych premennych
static const Int32 numberW = 1;                     //z pocet zapisovanych premennych
static const Int32 varWidth = 1;                    //z sirka premennej pre Boolean = 1Byte
static const Int32 lenghtR = numberR * varWidth;    //z dlzka bufera pre citanie
static const Int32 lenghtW = numberW * varWidth;    //z dlzka bufera pre zapis

Int32 hVarR;                                        //z pomocna premenna pre AdsBinaryReader
Int32 hVarW;                                        //z pomocna premenna pre AdsBinaryWriter

static String^ PLCvarR =L".Poruchy";                //z meno citanej premennej v PLC
static String^ PLCvarW =L".Reset";                  //z meno zapisovanej premennej v PLC
```

```
TcAdsClient^ tcClient;                              //z deklaracia ADS klienta
AdsStream^ dataStreamR;                             //z deklaracia bufra na citanie
AdsStream^ dataStreamW;                             //z deklaracia bufra na zapis
AdsBinaryReader^ adsRead;                           //z deklaracia citaca z bufra
AdsBinaryWriter^ adsWrite;                          //z deklaracia zapisovaca do bufra


//z premenne pre komunikaciu s databazou

static array<String^>^ pIndex = gcnew array<String^>(numberR); //z pole cisel poruch
static array<String^>^ pText = gcnew array<String^>(numberR);  //z pole textov poruch

String^ connection;                 //z spojenie s databazou
String^ queryIndex;                 //z SQL prikaz - dotaz na pole Index
String^ queryJazyk1;                //z SQL prikaz - dotaz na pole English
String^ queryJazyk2;                //z SQL prikaz - dotaz na pole Slovak


//z ostatne premenne

static array <Boolean>^ polePor = gcnew array <Boolean>(numberR);   //z pole na nacitanie PLC
static Int32 readTime = 1;          //z cas nacitavania v sekundach

//z koniec deklaracie premennych


#pragma region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
{
        this->components = (gcnew System::ComponentModel::Container());
        System::ComponentModel::ComponentResourceManager^  resources = (gcnew
        System::ComponentModel::ComponentResourceManager(Form1::typeid));
        this->Vypis = (gcnew System::Windows::Forms::ListBox());
        this->buttonStart = (gcnew System::Windows::Forms::Button());
        this->radioButton1 = (gcnew System::Windows::Forms::RadioButton());
        this->radioButton2 = (gcnew System::Windows::Forms::RadioButton());
        this->timerRead = (gcnew System::Windows::Forms::Timer(this->components));
        this->SuspendLayout();
        //
        // Vypis
        //
        this->Vypis->FormattingEnabled = true;
        this->Vypis->ItemHeight = 16;
        this->Vypis->Location = System::Drawing::Point(60, 41);
        this->Vypis->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
        this->Vypis->Name = L"Vypis";
        this->Vypis->Size = System::Drawing::Size(561, 132);
        this->Vypis->TabIndex = 0;
        //
        // buttonStart
        //
        this->buttonStart->Location = System::Drawing::Point(673, 134);
        this->buttonStart->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
        this->buttonStart->Name = L"buttonStart";
        this->buttonStart->Size = System::Drawing::Size(113, 39);
        this->buttonStart->TabIndex = 1;
        this->buttonStart->Text = L"Start";
        this->buttonStart->UseVisualStyleBackColor = true;
        this->buttonStart->Click += gcnew System::EventHandler(this,
        &Form1::buttonStart_Click);
        //
        // radioButton1
        //
        this->radioButton1->AutoSize = true;
        this->radioButton1->Location = System::Drawing::Point(689, 41);
        this->radioButton1->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
        this->radioButton1->Name = L"radioButton1";
        this->radioButton1->Size = System::Drawing::Size(69, 21);
        this->radioButton1->TabIndex = 2;
        this->radioButton1->TabStop = true;
        this->radioButton1->Text = L"jazyk1";
        this->radioButton1->UseVisualStyleBackColor = true;
```

```
        this->radioButton1->CheckedChanged += gcnew System::EventHandler(this,
        &Form1::radioButton1_CheckedChanged);
        //
        // radioButton2
        //
        this->radioButton2->AutoSize = true;
        this->radioButton2->Location = System::Drawing::Point(689, 80);
        this->radioButton2->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
        this->radioButton2->Name = L"radioButton2";
        this->radioButton2->Size = System::Drawing::Size(69, 21);
        this->radioButton2->TabIndex = 3;
        this->radioButton2->TabStop = true;
        this->radioButton2->Text = L"jazyk2";
        this->radioButton2->UseVisualStyleBackColor = true;
        this->radioButton2->CheckedChanged += gcnew System::EventHandler(this,
        &Form1::radioButton2_CheckedChanged);
        //
        // timerRead
        //
        this->timerRead->Tick += gcnew System::EventHandler(this, &Form1::timerRead_Tick);
        //
        // Form1
        //
        this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
        this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
        this->ClientSize = System::Drawing::Size(851, 217);
        this->Controls->Add(this->radioButton2);
        this->Controls->Add(this->radioButton1);
        this->Controls->Add(this->buttonStart);
        this->Controls->Add(this->Vypis);
        this->Icon = (cli::safe_cast<System::Drawing::Icon^  >(resources-
        >GetObject(L"$this.Icon")));
        this->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
        this->Name = L"Form1";
        this->StartPosition = System::Windows::Forms::FormStartPosition::CenterScreen;
        this->Text = L"TwinD 1.0";
        this->Load += gcnew System::EventHandler(this, &Form1::Form1_Load);
        this->ResumeLayout(false);
        this->PerformLayout();

            }
#pragma endregion

//z Koniec generovaneho kodu, zaciatok vlastnej aplikacie

private: System::Void Form1_Load(System::Object^  sender, System::EventArgs^  e)
        {
                NastavenieDatabazy();
                NastavenieTwinCAT();
                this->Text = L"TwinD Offline";
                ReadOLEdataToArray(connection, queryIndex, pIndex);
                ReadOLEdataToArray(connection, queryJazyk1, pText);
                this->radioButton1->Text = pText[55];
                this->radioButton2->Text = pText[56];
         }


private: System::Void radioButton1_CheckedChanged(System::Object^  sender, System::EventArgs^
e)      {
                ReadOLEdataToArray(connection, queryIndex, pIndex);
                ReadOLEdataToArray(connection, queryJazyk1, pText);
        }


private: System::Void radioButton2_CheckedChanged(System::Object^  sender, System::EventArgs^
e)      {
                ReadOLEdataToArray(connection, queryIndex, pIndex);
                ReadOLEdataToArray(connection, queryJazyk2, pText);
        }
```

```
private: System::Void buttonStart_Click(System::Object^  sender, System::EventArgs^  e)
	{
		NastavenieCasovaca();
		this->Text = L"TwinD Online";
		Vypis->Items->Clear();              //z zmazanie listBoxu Vypis
		dataStreamW->Position = 0;
		adsWrite->Write(true);
		tcClient->Write(hVarW, dataStreamW);
	}


private: System::Void timerRead_Tick(System::Object^  sender, System::EventArgs^  e)
	{
		tcClient->Read(hVarR, dataStreamR);  //nacitanie premennej do bufra
		dataStreamR->Position = 0;           //nastavenie bufra
		//naplnenie textoveho pola z bufra
		for (int i = 0; i < polePor->Length ; i++)
		 {
		  polePor[i] = adsRead->ReadBoolean();
		  if (polePor[i] == true)
		    {
		     timerRead->Enabled = false;
		     // zobrazenie okna aplikacie v zadanej velkosti
		     this->Text = L"TwinD Offline";
		     this->WindowState = System::Windows::Forms::FormWindowState::Normal;
		     this->ClientSize = System::Drawing::Size(851, 217);
		     vypisPoruchy(i);
		    }
		 }

		if (timerRead->Enabled == true)
		 {
		  this->WindowState = System::Windows::Forms::FormWindowState::Minimized;
		 }
	}


//z ---------- Komunikacia s databazou Access 2003 ----------

private: System::Void NastavenieDatabazy(System::Void)
	{
		//z cesta k PoruchovaDatabaza.mdb
		String^ databaza = L"PoruchyZ.mdb";

		//z provider for Access 2003 (.mdb)
		String^ provider = L"Microsoft.Jet.OLEDB.4.0";

		//z connection string
		connection = L"Provider=" + provider + "; Data Source=" + databaza;

		//z query string pre pole Index
		queryIndex = L"SELECT Index FROM Texty";

		//z query string pre pole Jazyk1
		queryJazyk1 = L"SELECT Jazyk1 FROM Texty";

		//z query string pre pole Jazyk2
		queryJazyk2 = L"SELECT Jazyk2 FROM Texty";
	}
```

```
//z ---------- Komunikacia s TwinCAT ----------

private: System::Void NastavenieTwinCAT(System::Void)
	{
		//z vytvorenie ADS klienta
		tcClient = gcnew TcAdsClient();

		try
			{
			//z vytvorenie spojenia s ADS zariadenim (PLC Runtime1)
			tcClient->Connect(safe_cast<int>(AmsPort::PlcRuntime1));

			//z vytvorenie spojenia s premennou v PLC na citanie
			hVarR = tcClient->CreateVariableHandle(PLCvarR);

			//z vytvorenie spojenia s premennou v PLC na zapis
			hVarW = tcClient->CreateVariableHandle(PLCvarW);

			//z vytvorenie datoveho bufra na citanie
			dataStreamR = gcnew AdsStream(lenghtR);

			//z vytvorenie citaca z bufra
			adsRead = gcnew AdsBinaryReader(dataStreamR);

			//z vytvorenie datoveho bufra na zapis
			dataStreamW = gcnew AdsStream(lenghtW);

			//z vytvorenie zapisovaca do bufra
			adsWrite = gcnew AdsBinaryWriter(dataStreamW);
			}

		catch (Exception^ err)		//z osetrenie poruchy
			{
			MessageBox::Show(err->Message);	//z zobrazenie priciny poruchy
			}
	}


//z ---------- Nastavenie casovaca ----------

private: System::Void NastavenieCasovaca(System::Void)
	{
		timerRead->Enabled = true;
		timerRead->Interval = readTime * 1000; //z prevod sekundy na tisiciny
	}


//z ---------- ReadOLEdataToArray ----------

private: System::Void ReadOLEdataToArray(String^ connectionString, String^ queryString,
array<String^>^ vArray)
	{
		//z vytvorenie spojenia so zdrojom zadanym v spojovacom retazci
		OleDbConnection^ connection = gcnew OleDbConnection(connectionString);

		//z vytvorenie prikazu
		OleDbCommand^ command = gcnew OleDbCommand(queryString, connection);

		//z otvorenie spojenia
		connection->Open();

		//z vytvorenie citaca z datoveho prudu napojeneho na datovy zdroj
		OleDbDataReader^ reader = command->ExecuteReader();

		//z nacitanie databazy do pola
		Int32 i = 0;
		while (reader->Read())
		 {
		  vArray[i] = reader[0]->ToString();
		  i = i+ 1;
		 }

		//z zatvorenie datoveho prudu
		reader->Close();
	}
```

```
//z ---------- Vypis poruchy ----------

private: System::Void vypisPoruchy(System::Int32 index)
	{
		Vypis->Items->Add(pIndex[index]+ L" : " + pText[index]);
	}


//z ---------- Ukoncenie aplikacie ----------

private: System::Void Form1_FormClosing(System::Object^  sender,
System::Windows::Forms::FormClosingEventArgs^  e)
	{
		try
			{
			tcClient->DeleteVariableHandle(hVarR);
			tcClient->DeleteVariableHandle(hVarW);
			}

		catch (Exception^ err)
			{
			MessageBox::Show(err->Message);
			}
	}
 };
}
```