ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Katedra počítačů

Programové vybavení automatického testeru LIN

Software for Automated LIN Tester

Diplomová práce

Studijní program: Elektrotechnika a informatika (magisterský), strukturovaný
Studijní obor: Výpočetní technika

Vedoucí práce: doc Ing. Jiří Novák, Ph. D.

Miloslav Klášterka

Praha 2015

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Miloslav Klášterka**

Studijní program: Elektrotechnika a informatika (magisterský), strukturovaný
Obor: Výpočetní technika

Název tématu: **Programové vybavení automatického testeru LIN**

Pokyny pro vypracování:

Navrhněte a v jazyce C# implementujte uživatelské rozhraní automatizovaného testeru vozidlových řídicích jednotek s rozhraním LIN pro OS Windows.
Programové vybavení musí poskytovat následující funkce:
- import popisných souborů ve formátech .ldf a .cdb
- zobrazení LIN clusteru s parametry jednotlivých jednotek, zpráv a signálů
- uložení veškerých parametrů v instancích C++ tříd využívajících knihovny STL
- serializace parametrů do XML souboru
- návrh a implementace volání jednotlivých testů implementovaných v jazyce C++

Seznam odborné literatury:

Nagel, C.,Evjen, B., Glyn, J., Skinner, M., Watson, K.: Profesional C#2008, Computer Press 2013, ISBN 978-8025124017
McConnel, S.: Dokonalý kód, Computer Press 2006, ISBN 80-251-0849-X

Vedoucí: doc.Ing. Jiří Novák, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

L.S.

doc. Ing. Filip Železný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 28. 2. 2014

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 4. 1. 2015 .....

Poděkování

Tímto bych chtěl poděkovat vedoucímu své diplomové práce, doc. Ing. Jiřímu Novákovi, Ph. D., za pomoc, trpělivost a veškerý čas, který mi během tvorby této práce věnoval.

Anotace

Cílem této práce je vytvořit uživatelské rozhraní automatizovaného testeru vozidlových řídicích jednotek s rozhraním LIN pro operační systém Windows. Toto rozhraní je implementováno v jazyce C#, nicméně využívá původní knihovny vytvořené v jazyce C++ pro práci s LIN a CAN sběrnicí. Práce je tedy věnována především problémům interoperability jazyků C# a C++ s využitím jazyka C++ CLI. Dále je popsán standard sběrnice LIN a jeho definiční formát LDF, který je využit jako zdrojový soubor pro tvorbu testů. Nakonec je představeno a popsáno realizované řešení, které již bylo předáno testovacímu pracovišti firmy Škoda Auto a. s.

Anotation

Goal of this work is to design and develop a Windows GUI (Graphical User Interface) for automated tester of ECUs (Electronic Control Unit) equipped with a LIN communication interface. GUI is implemented in C#, it should nevertheless use original libraries supporting LIN and CAN communication written in C++. Work is mostly dedicated to interoperability issues between C# and C++ using a C++ CLI language. Besides, the LIN technology including the LDF definition file format and its use for tester setup are described. At the end a final product (already used by Škoda Auto testing staff) is briefly described.

Obsah

1 Úvod 1

2 Základy C# 3

2.1 Architektura .NET 3

2.2 .Net a správa paměti 5

2.3 Jazyk C# 6

2.3.1 Jmenné prostory 6

2.3.2 Třída 7

2.3.3 Proměnné 7

2.3.4 Vlastnost třídy 8

2.3.5 Události a delegáti 8

2.3.6 Atributy 9

2.3.7 Serializace objektů do XML 10

2.3.8 Background Worker 11

2.4 Winforms a datové vazby 12

2.5 Volání nativního kódu z .NET 13

2.6 Využití jazyka C++ CLI jako wrapperu 14

3 Testování řídicích jednotek 16

3.1 Sestava testovacího pracoviště 16

3.1.1 Vnější jednotka 17

3.1.2 Karta AG-CAN 17

3.2 Popis standardu LIN 18

3.2.1 Struktura sítě LIN 18

3.2.2 Koncepce návrhu sítě LIN 19

3.2.3 Rámce 20

3.2.4 Signály a diagnostické zprávy 20

3.2.5 Časovací tabulka 21

3.3 LIN Description File 21

3.3.1 Globální definice 21
3.3.2 Definice uzlů 22
3.3.3 Atributy uzlu 23
3.3.4 Definice signálů 24
3.3.5 Definice diagnostického signálu 25
3.3.6 Definice rámců 25
3.3.7 Definice časovací tabulky 27
3.3.8 Dodatečné informace 28
4 Implementace aplikace 30
4.1 Požadavky na aplikaci 30
4.2 Ovládání aplikace 31
4.2.1 Nový projekt 31
4.2.2 Uložení a načtení projektu 32
4.2.3 Spuštění testů 32
4.2.4 Vizualizace LDF 33
4.2.5 Ovládání přístrojů 34
4.2.6 Manuální konfigurace vnější jednotky 35
4.3 Popis implementace 35
4.3.1 Rozvržení zdrojového kódu 35
4.3.2 Implementace wrapperů 35
4.3.3 Třídy popisující testy 39
4.3.4 Import a struktura LDF 40
4.3.5 Implementace serializace XML 42
4.3.1 Logování souboru 44
4.4 Formuláře uživatelského rozhraní 46
4.4.1 Plovoucí panely 46
4.4.2 Hlavní formulář 47
4.4.3 Panel Testů 48

4.5 Ladění ve smíšeném módu 51
5 Výsledky testů 53
5.1 Funkční testy 53
5.2 Integrační testy 53
5.3 Zákaznické testování 54
6 Závěr 55
7 Zdroje 56
8 Přílohy 58
8.1 Ukázka definice souboru LDF 58

1 Úvod

V dnešní době se setkáváme s elektronickými zařízeními na každém kroku. Obvyklým místem, kde se můžeme setkat s vyšším počtem zařízení, která nám mají usnadňovat a zpříjemňovat život, jsou například interiéry moderních automobilů. Dnešní vozy jsou doslova napěchované elektronikou tohoto druhu, ať už se jedná např. o automatické stahování oken, elektronické otáčení zrcátek, zapínání a otáčení světel, ovládání klimatizace, odemykání dveří nebo různých druhů signalizací, jako jsou signalizace otevřených dveří či nerozsvícených světel během jízdy. Tyto periferie se vyznačují tím, že sice zvyšují komfort při užívání vozu, nejedná se však o kritické systémy, jejichž případná nefunkčnost by přímo ohrožovala život nebo zdraví cestujících v autě. Pro jejich správnou funkci je důležité, aby mohly komunikovat jak spolu navzájem, tak s nadřazenou vozidlovou řídící jednotkou. V automobilovém průmyslu je jedním ze standardů pro komunikaci těchto periférii síť využívající protokol Local Interconnect Network, dále jen LIN. Výhodou této sériové sítě je především nižší cena komponent pro připojení oproti ostatním průmyslovým sběrnicím.

Pomocí LIN sběrnice může komunikovat až 16 zařízení v tzv. clusteru, kde jedno ze zařízení vždy plní úlohu mastera, tj. komunikačního prvku, který zajišťuje vysílání požadavků na přenos dat i synchronizačních informací a zprostředkovává komunikaci mimo cluster. Při návrhu a výrobě všech zařízení spolupracujících s touto sběrnicí je nutné, aby důsledně dodržovala definovaný standard, komunikovala pouze, když jsou na řadě, a nezpůsobovala tak neočekávané kolize. Je tedy důležité všechny tyto komponenty před umístěním do automobilu pečlivě otestovat.

Úkolem diplomové práce je navrhnout grafické uživatelské rozhraní programu, který bude sloužit k testování řídicích jednotek a jejich komunikace s ostatními zařízeními pomocí sběrnice LIN. Program navazuje na předešlou verzi obdobného testeru, který však sloužil jako tester řídicích jednotek sběrnice CAN. Původní rozhraní, jež bylo vyvíjeno před několika lety, využívá dnes již poměrně zastaralou technologii pro tvorbu formulářových aplikací Microsoft Foundation Classes 8.0. Přestože již existuje několik novějších verzí této knihovny, jejich problém tkví na dnešní dobu v poměrně komplikovaném kódování a tím pádem i horší udržovatelnosti kódu. Proto vznikl požadavek na přepracování GUI do nějaké jiné, modernější platformy. Po několika konzultacích s vedoucím práce jsme pro

toto řešení zvolili jako novou platformu dynamicky se rozvíjející Framework Microsoft .Net s využitím jazyka C#. Při implementaci bude ovšem nutné zachovat stávající knihovny testeru CAN a integrovat je do nové aplikace. Dále bude nutné umožnit načítání souboru formátu LIN Definition File, dále jen LDF, který je součástí standardu LIN a popisuje cluster jednotek, zpráv a signálů. Informace z tohoto souboru je nutné přehledným způsobem prezentovat uživateli pomocí grafického rozhraní. Uživatelem definované parametry a nastavení bude nutné ukládat na disk do XML souboru tak, aby změny byly perzistentní po vypnutí aplikace.

Souběžně s implementací programu vznikaly i nové testy, jež přímo definují různé scénáře testování LIN jednotek. Jejich realizace se zhostil Ing. Tomáš Grus ve spolupráci s vedoucím práce, doc. Ing. Jiřím Novákem, Ph.D. Výsledný program tak zahrnuje kompletní implementaci testovacího pracoviště jednotek LIN.

2 Základy C#

Během tvorby programu byla řešena především integrace modulů vytvářených v jazycích C++ a C#. Popis obou jazyků by překročil rozsah a rámec této práce, proto je následující kapitola věnována pouze základní syntaxi těchto jazyků v kontextu problematiky diplomové práce.

2.1 Architektura .NET

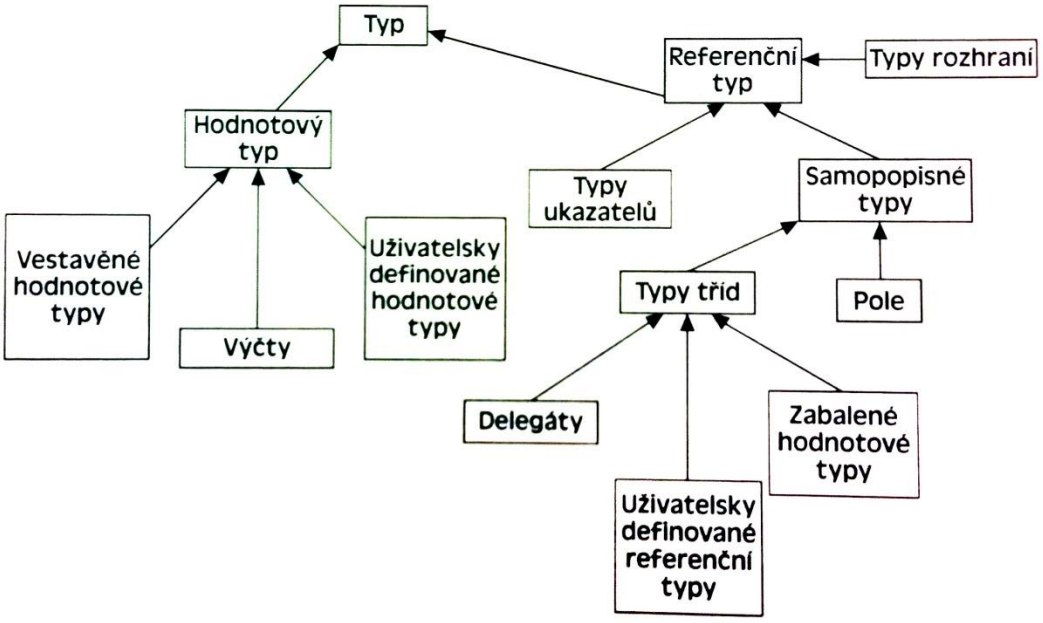
Jazyk C# je sice samostatný programovací jazyk, je ale speciálně navržen a určen pro použití s platformou .NET Framework. Všechny jeho rysy jsou totožné s rysy této platformy a nedá se o něm hovořit odděleně bez podrobnějšího popisu již zmíněného prostředí .Net. Tento Framework, vyvinutý společností Microsoft, je navržen tak, aby umožňoval vývoj, zavádění a spouštění distribuovaných aplikací nezávislých nejen na programovacím jazyku, ve kterém jsou vytvářeny, ale i nezávisle na platformě, na které budou výsledné aplikace spouštěny. Díky speciálně navržené architektuře by se tak mělo předcházet nutnosti zbytečně přepisovat jednou hotové zdrojové kódy z různých jazyků a systémů. Překlad v prostředí .Net se provádí ve dvou krocích:

V první fázi se program překládá do jazyka symbolických adres CIL (Common Intermediate Language), což je nízkoúrovňový jazyk s jednoduchou syntaxí, plnící stejnou úlohu jako bajtový kód jazyka Java. Během kompilace se ke kódu CIL přiloží ještě metadata obsahující tabulky s informace jak o všech typech a členských prvcích definovaných ve zdrojovém kódu, tak všech typech a členských prvcích, na které zdrojový kód odkazuje. Výsledný modul se označuje jako řízený (managed) modul. Jeho hlavní výhodou je, že je pro jeho další provádění nezávislý na programovacím jazyku, ve kterém byl původně napsán. Kromě překladače jazyka C# existuje celá řada překladačů pro ostatní jazyky, např. pro C++, Visual Basic, JScript, J# (překladač jazyka Java), Perl a pro dalších asi dvacet jazyků. Překladač C++ CLI je v práci hojně využíván a bude v textu popsán později.

Ve druhé fázi se kód překládá z jazyka CIL do zdrojového kódu specifického pro cílovou platformu pomocí modulu CLR (Common Language Runtime). Díky tomu je umožněna zmíněná platformní nezávislost, jelikož soubor s CIL instrukcemi je možné přenést na libovolnou platformu s interpretem CLR, a závěrečný převod na binární kód

příslušné platformy uskutečnit až při prvním spuštění programu. Pro zvýšení výkonu využívá technologie .Net ještě metodu Just-In-Time, kdy při spuštění nepřekládá aplikaci celou v jednom průchodu, ale jednotlivé části aplikace vždy až při příslušném volání. Po prvním překladu jsou výsledné části nativního kódu uloženy, není proto nutné je překládat znovu.

Všechny jazyky, které chtějí kompilovat pomocí výše zmíněného postupu, musejí dodržovat specifikaci CTS (Common Type System), která předepisuje předem definované datové typy, jež jsou dostupné v jazyku CIL. Zjednodušeně řečeno CTS určuje, jak se např. proměnná typu integer z jazyka Visual Basic, která je 32 bitová a znaménková, převede do datového typu int32 jazyka CIL a následně s ním půjde zavolat metoda implementovaná v jazyku C#. Z toho vyplývá potřeba silné typové kontroly všech zúčastněných jazyků. CTS popisuje ovšem mnohem více datových typů a zavádí jejich hierarchické členění, které je ukázáno na obr. 2.1 Hierarchie typů dle CTS. To důsledně rozlišuje dva základní typy: hodnotový a referenční. Rozdíl je celkem patrný už z názvu. Proměnné hodnotového typu přímo ukládají příslušná data, zatímco proměnné referenčního typu obsahují pouze adresu. V terminologii C++ se proměnné hodnotového typu dají přirovnat k přístupu k proměnné pomocí ukazatele.



Obr. 2.1 Hierarchie typů dle CTS. Zdroj: NAGEL, CH. et al.*C# 2008 Programujeme profesionálně,* s. 51.

Programovací jazyky se často značně liší. Aby bylo možné vytvářet typy, jež budou v rámci jiných programovacích jazyků snadno použitelné, je nutné použít pouze ty prvky programovacího jazyka, které budou použitelné i v ostatních jazycích. Za tímto účelem byl definován standard pro společnou jazykovou specifikaci CLS (Common Specification Language). Jedná se o minimální množinu funkcí, které musí autoři překladače implementovat, aby bylo možné generovat kód kompatibilní s CLR. CLS definuje pravidla, která musí být dodržena pouze u navenek viditelných typů a metod, mají-li být dosažitelné z libovolného jiného jazyka podporujícího CLS.

Všechny výše zmíněné standardy byly shrnuty a označeny jako CLI (Common Language Infrastructure). Jelikož prostředí .NET a jazyk C# dodržují tento standard beze zbytku, mohou spolupracovat se všemi jazyky, které zahrnují rozšíření CLI. Proto se jazyk C# jeví jako vhodný adept pro tvorbu GUI, které umožní využívat původní kódy napsané v jazyku C++ a prezentovat je uživatelům. Navíc díky architektuře .NET nebude v budoucnu problém do výsledné aplikace přispívat i jinými kódy z jazyků podporujících standard CLI.[1]

2.2 .Net a správa paměti

Veškerá dynamicky přidělená paměť na platformě .Net je sice stejně jako ve většině ostatních jazycích přidělena haldě, na rozdíl od jazyka C++ je ale tato halda řízená, neboli „managed“. Udržuje ji modul CLR a mohou ji tak používat všechny aplikace .NET. Běhový systém .Net se při přidělování paměti spoléhá na automatickou správu paměti, využívá k tomu automatickou správu paměti (Garbage collector). Ten řeší problém, jak získat zpět paměť, o kterou spuštěné aplikace žádají. K tomu využívá systém počítání referencí, kdy ke každému objektu udržuje čítač s počtem ukazatelů, které na něj odkazují. Vždy když vznikne odkaz na objekt, systém si inkrementuje vnitřní čítač o hodnotu 1 a poté, co odkaz zanikne, se čítač naopak dekrementuje. Když je čítač nastaven na hodnotu 0, systém ví, že takovýto objekt může odstranit z paměti. Toto odstranění se ovšem neprovádí okamžitě, avšak pouze v určitých časových intervalech, nebo když systém detekuje nedostatek volné operační paměti. Tuto kontrolu může programátor vynutit i přímo v programu pomocí volání příslušných funkcí. Implementaci automatické správy

[1] NAGEL, CH. et al.C# 2008 Programujeme profesionálně, s. 43-53.

paměti v prostředí .Net značně zjednodušuje fakt, že díky návrhu jazyka CIL je celé prostředí silně typově kontrolované.[2]

2.3 Jazyk C#

Popis celé definice všech možných konstruktů v C# by byl značně zdlouhavý a pro práci ne zcela účelný, proto je omezen pouze na ty konstrukty jazyka, které jsou často používány v diplomové práci.

2.3.1 Jmenné prostory

Jak bylo zmíněno výše, C# při kompilování přidává do výsledného předkompilovaného modulu metadata, obsahující kompletní informace o kompilované třídě. Tyto informace slouží překladači i během linkování. V kódu se díky tomu nemusí vytvářet hlavičkové soubory, ani se nemusí explicitně vkládat části kódu do sebe příkazem include. Vývojové prostředí tedy umožňuje využít jakoukoli třídu nebo typ, který je přítomný v .Net, nebo připojený k aplikaci. Těch je ale značné množství. Proto je vhodné je rozdělit do menších celků, k čemuž jsou určeny jmenné prostory (namespace).

Jmenné prostory slouží k logickému uspořádání tříd a jiných typů. Typ umístěný do jmenného prostoru získává dlouhý název složený z názvu jmenného prostoru, v němž je deklarován, zapsaného ve tvaru řady jmen oddělených tečkami, a je ukončen jménem třídy. Jmenné prostory lze také vnořovat do jiných jmenných prostorů a vytvářet tak hierarchickou strukturu.

Jelikož jsou názvy jmenných prostorů někdy značně dlouhé, jejich vypisování před každou třídu by bylo značně únavné. C# naštěstí umožňuje názvy tříd zkrátit pomocí příkazu using. Stačí na začátku souboru zadat toto klíčové slovo, za něj jmenný prostor a ve zbývající části souboru můžeme odkazovat na typy z příslušného jmenného prostoru pouze pomocí názvu typu. Příklad užití namespace a using je následující:[3]

```
using System;
using System.Windows.Forms;

namespace LINTest{
        namespace LINTest.GUI{
        }
}
```

2 NAGEL, CH. et al.C# 2008 Programujeme profesionálně, s. 52-53.

[3] NAGEL, CH. et al.C# 2008 Programujeme profesionálně, s.100-103.

2.3.2 Třída

Základní konstrukt moderních objektově orientovaných programovacích jazyků jako je C# je třída. Na rozdíl od C++ jsou všechny proměnné a funkce povinně umístěné uvnitř třídy, a to i v případě statických metod a funkcí. Dále je na rozdíl od C++ nutné uvádět modifikátory přístupů jakou jsou public, private, protected a internal před každou metodou a funkcí. Třída může být potomkem pouze jedné bázové třídy, ale může dále implementovat libovolný počet rozhraní. Příklad definice třídy:

```
public class MyClass : MyParrent, MyInterface {
        public MyClass(){ }      //konstruktor
        ~MyClass(){} //destruktor
        public void ClassMethod(){ }
        private int classMember;
}
```

2.3.3 Proměnné

Jazyk C# klade důraz na bezpečnost vykonávaného kódu. Stručně řečeno, kompilátor vyžaduje, aby každá proměnná byla inicializována nějakou výchozí hodnotou, než na ni bude v kódu odkázáno, jinak ohlásí chybu. Díky tomu se nemůže stát, že se z paměti načtou nějaké nesmyslné hodnoty.

Jak je vidět z obr. 2.1 Hierarchie typů dle CTS, proměnné jsou v jazyce C# děleny na hodnotové (datové) a referenční. Referenční datové typy jsou obdobou ukazatelů v jazyce C++. Níže je uvedena jednoduchá ukázka definice a deklarace obou typů proměnných. U hodnotového typu se nijak neliší od jazyka C++. U referenční proměnné je důležité nezapomenout na vytvoření příslušného objektu, např. pomocí klíčového slova new.[4]

```
int Count = 0; //definice a deklarace hodnotové proměnné
File f;   //deklarace referenční proměnné
f = new File();   //definice
Directory dir = new Direcotry(); //definice s deklarací
```

[4] NAGEL, CH. et al.C# 2008 Programujeme profesionálně, s. 74-79.

2.3.4 Vlastnost třídy

Vlastnost třídy je vlastně obdoba členské metody, která se ale z vnějšku třídy jeví jako proměnná. Často se využívá jako rozhraní tam, kde se vrací nebo nastavují proměnné třídy. V C++ nahrazuje často používané „gettery“ a „settery“, tj. metody, které pouze vrací nebo nastavují privátní proměnné třídy. Příklad takové vlastnosti je zobrazen níže. Poslední řádek v příkladu ukazuje zjednodušený zápis, kdy překladač automaticky vygeneruje stejný kód jako v předcházející části.

```
private String m_name;
public String Name {
        get { return m_name; }
        set { m_name = value; }
}
public Int32 lndex { get; set; } // zkrácený zápis
```

2.3.5 Události a delegáti

Delegáti slouží v situacích, kdy chceme předávat metody jiným metodám. Na rozdíl od ukazatelů na funkci jazyka C++, které odkazují pouze na místo v paměti, kde není známo, nač ve skutečnosti ukazují a jaké jsou parametry a návratové typy, jsou delegáti typově bezpeční. Překladač hlídá počet a typy parametrů funkce, na kterou delegát ukazuje, a typ návratové hodnoty. Příklad níže ukazuje definici delegátu, vytvoření instance delegátu a dva způsoby zavolání funkce, na kterou ukazuje. Delegáti se často používají v součinnosti s událostmi (Event).

```
private delegate string GetAString();

static void Main(){
        GetAString firstStringMethod = new GetAString(x.ToString);
        String strFirst = firstStringMethod.Invoke();
        String strSecond = firstStringMethod();
}
```

Myšlenka události je obecně založena na tom, že často existuje kód, který je nutné při výskytu určité události informovat. Tato situace je běžná především při programování grafických uživatelských rozhraní. K tomu definuje jazyk C# události. Příklad definice události, včetně metody, která událost vyvolá, je uveden níže.

```
public delegate void ChangedDelegate();
public event ChangedDelegate Change;

private void OnChanged (){
        if(Change != null){
                Changed();
        }
}
```

Pokud třída deklaruje událost, může se jiná třída registrovat, jako tzv. předplatitel události. To provede tak, že přidá instanci delegáta do seznamu delegátů této události, pomocí operátoru „+=“. Metoda, na níž tato instance delegátu ukazuje, bude volána vždy při výskytu této události. Pokud již třída nechce být informována o výskytu události, může zavolat operátor „–=“. Příklad níže ukazuje registraci k odběru události Change, jež byla definována výše.[5]

```
public class Subsriber{
        private ClassChange publisher= new ClassChange();
        public void Subsriber(){
                publisher. Change += new ChangedDelegate(Publisher_Changed);
                //zkrácený zápis
                publisher.Change += Publisher_Changed;

        }
        void Publisher_Changed (){
                Console.WriteLine(“SomethingChanged”)
        }
}
```

2.3.6 Atributy

Atributy v jazyce C# nabízejí mechanismus, jak asociovat dodatečné informace s kódem (třídou, datovým typem, metodou, vlastností, atd.). Prostředí .Net nabízí sadu atributů, na jejichž základě pak překladač může určitým způsobem uzpůsobit proces překladu. Platforma .Net také dovoluje definovat vlastní atributy, které lze pak využívat pomocí systému reflexe. Atribut se využívá tak, že se napíše do hranaté závorky před

[5] NAGEL, CH. et al.C# 2008 Programujeme profesionálně, s. 239-267.

volaný objekt, se kterým se má asociovat. K němu se pak do kulatých závorek dopisují parametry tak, jak jsme zvyklí u volání metody. Příklad volání atributu je následující:[6]

```
[Obsolete("Tato třída by se nemělá dále používat", false)]
public class OldClass{
}
```

2.3.7 Serializace objektů do XML

Serializace znamená trvalé uložení objektů na disk. Poté může jiná část aplikace, nebo dokonce jiná aplikace objekt deserializovat a tím ho dostat do stejného stavu, v jakém se nacházel před serializací. K tomu definuje platforma .Net třídu XmlSerializer, ve jmenném prostoru System.Xml.Serialization. Ta pomocí metody Serialize serializuje veškeré veřejné vlastnosti a datové složky třídy. Naopak pomocí metody Deserialize vrátí instanci třídy se všemi veřejnými vlastnostmi a datovými složkami třídy do stavu, v němž byly před serializací. Serializaci lze navíc ovlivňovat atributy s různými parametry, jako je například [XmlIgnore], jež zamezí serializaci daného objektu, nebo [XmlAttribute], jež explicitně nastavuje serializaci objektu do atributu, nebo [XmlElement], jež explicitně nařídí překladači serializovat objekt jako element. Příklad třídy, jež provádí serializaci a deserializaci sebe sama je níže.[7]

```
using System.Xml;
using System.Xml.Serialization;
using System;
public class LocalSettings
{
        [XmlAttribute("NameOfSetting")]
        public String Name { get; set; }
        public int Count { get; set; }
        [XmlIgnore]
        public String Comment{ get; set; }

        public void Save()
        {
                XmlSerializer xml = new XmlSerializer(typeof(LocalSettings));
                XmlWriter writer;
                writer = new XmlTextWriter(localSettingPath, Encoding.UTF8);
                xml.Serialize(writer, this);
        }
```

[6] NAGEL, CH. et al.C# 2008 Programujeme profesionálně, s. 429-436.
[7] NAGEL, CH. et al.C# 2008 Programujeme profesionálně, s. 1050-1058.

```
        private static LocalSettings Load()
        {
                XmlSerializer xml = new XmlSerializer(typeof(LocalSettings));
                XmlReader xmlReader = new XmlTextReader(localSettingPath);
                LocalSettings ret = (LocalSettings)xml.Deserialize(xmlReader);
                xmlReader.Close();
                return ret;
        }
}
```

2.3.8 Background Worker

Jednou z implementací vzoru asynchronních událostí je komponenta background worker. Tato komponenta se využívá především na formulářových aplikacích. Umožňuje jednoduchým způsobem spouštět operace v separátním vlákně a informovat formulářovou aplikaci o změně a dokončení tohoto vlákna. To umožňuje spouštění časově náročných operací, které by jinak po dobu svého běhu zamezily zpracování ostatních událostí, a tím pádem i aktualizaci a překreslování formuláře. K tomuto účelu jsou implementovány události DoWork a ProgressChange.

Příklad použití třídy BackgroundWorker je uveden níže. Kód, který je umístěn v metodě bgw_DoWork, běží odděleně v jiném vlákně. Z tohoto vlákna ale nelze upravovat prvky na hlavním formuláři. Proto je nutné předávat informace druhému vláknu metodou ReportProgress. Obsluha této metody, v našem příkladě bgw_ProgressChanged, je už prováděna původním vláknem formulářové aplikace a může tak měnit formulářové prvky.[8]

```
public class MainForm : Form
{
        private MainForm(object sender, EventArgs e)
        {
                InitializeComponent();
                BackgroundWorker bgw = new BackgroundWorker();
                bgw.DoWork += bgw_DoWork;
                bgw.WorkerReportsProgress = true;
                bgw.ProgressChanged += bgw_ProgressChanged;
                bgw.RunWorkerAsync();
        }
```

[8] NAGEL, CH. et al.C# 2008 Programujeme profesionálně, s. 647-653.

```
        void bgw_ProgressChanged(object sender, ProgressChangedEventArgs e)
        {
                progressBar1.Value(e.ProgressPercentage);
        }

        void bgw_DoWork(object sender, DoWorkEventArgs e)
        {
                for(int i = 0;i<100;i++){
                        //do somthing usefull
                        bgw.ReportProgress(i);
                }
        }
}
```

2.4 Winforms a datové vazby

Windows forms (zkráceně WinForms) je knihovna platformy .Net, která umožňuje vývojářům programovat efektní a inteligentní klientské aplikace. Vývojové prostředí MS Visual Studio umožňuje pomocí této knihovny vytvářet nové formuláře (též okna nebo dialogy) jednoduchým způsobem: přetažením ovládacího prvku ze sady nástrojů do okna vizuálního návrhu. Na pozadí vytváří kód jazyka C#, který lze procházet nebo s ním dále pracovat.[9]

Datová vazba (databinding) je mechanismus, který umožňuje propojit vlastnosti ovládacích prvků se zdrojem dat. Když se poté členská proměnná změní, aktualizuje se i vlastnost ovládacího prvku. To platí i opačně: jestliže aktualizujeme text v poli, projeví se tato změna ve zdroji dat. Příklad níže ilustruje vytvoření třídy, která obsahuje textové pole txtName a vytvoření datové vazby její vlastnosti Text s vlastností Name definované v třídě DataClass.[10]

```
using System;
using System.Windows.Forms;

namespace GUI
{
        public class DataClass
        {
                public String Name { get; set; }
        }
```

[9] NAGEL, CH. et al.C# 2008 Programujeme profesionálně, svazek 2, s. 23-29.
[10] NAGEL, CH. et al.C# 2008 Programujeme profesionálně, svazek 2, s. 92-93.

```
        public partial class Form1 : Form
        {
                public Form1()
                {
                        InitializeComponent();
                        DataClass dataInstance = new DataClass();
                        txtName.DataBindings.Add("Text", dataInstance, "Name");
                }
        }
}
```

2.5 Volání nativního kódu z .NET

Na začátku této kapitoly je popis .Net a toho, jak je docíleno interoperability modulů z různých jazyků. V této práci jsou popsány dvě konkrétní metody přístupu k nativnímu kódu. Obě metody lze využít, avšak každá má své klady a zápory. První metoda volání nativního kódu je realizována pomocí atributu DllImport. Druhá možnost vychází z existence překladače C++ firmy Microsoft o rozšíření CLI, které umožňuje vytvářet řízené třídy běžící na řízené haldě, jež jsou schopny přímo pracovat s nativními objekty a metodami na haldě neřízené. Umožňuje to vytvořit zapouzdřovací třídu (wrapper) a pomocí ní volat vše potřebné v neřízeném (unmanaged) kódu. O rozšíření C++ CLI je celá další kapitola, ve které je druhá metoda podrobně popsána.

Pomocí atributu DllImport lze přímo vystavit metodu, která je umístěna v jiné neřízené dynamické knihovně napsané v jazyku C++. Je nutné znát a opsat přesnou signaturu volané metody. Je zde ale problém s tzv. „name magling“.[11] Kompilátory vždy při překladu přikládají k metodám specifické symboly, aby zajistily unikátnost jmen v různém kontextu. Bohužel kompilátory od různých firem vkládají jiné symboly a výsledný název metody zkompilované v jiném překladači nebo na jiné počítačové architektuře není dopředu znám. Při volání knihoven kompilovaných jinak než kompilátorem společnosti Microsoft je nutné uvést název metody včetně těchto symbolů.[12]

[11] Name mangling. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-12-28]. Dostupné z: http://en.wikipedia.org/wiki/Name_mangling ,

[12] Using the DllImport Attribute. MICROSOFT. *Microsoft Development Network* [online]. 2014 [cit. 2014-12-28]. Dostupné z: http://msdn.microsoft.com/en-us/library/aa984739%28v=vs.71%29.aspx

Naštěstí ke zjištění symbolů i metod volání existují různé nástroje, kterými se dá prozkoumávat již existující zkompilovaná dynamická knihovna. Visual Studio společnosti Microsoft k tomuto účelu dodává utilitu dumpbin.exe.

Pomocí této metody lze volat funkce obsahující jako parametr strukturu nebo třídu, i když s některými omezeními plynoucími z možností C#. Jelikož je řízená třída vždy předávána odkazem, nelze využít volání metody, která má jako parametr třídu předávanou hodnotou. A naopak u struktury, jež je hodnotový typ, nelze předat ukazatele na ukazatele na strukturu.[13] Příklad volání je následující:

```
[DllImport("msvcrt.dll", EntryPoint=" printf", CallingConvention=CallingConvention.Cdecl,
SetLastError="true")]
public static extern int printf(String format, int i, String s);
```

Kvůli výše zmíněným problémům je metoda DllImport vhodná především v případě, kdy nejsou dostupné zdrojové kódy dané knihovny a kdy se nepředpokládají časté změny v cílové knihovně. Nevýhodou je, že neumožňuje některá volání, na což by se měl brát zřetel při psaní nativního kódu. Dále je pak složitější ladění, pokud je potřeba kompilovat nativní kód z řízeného běhového prostředí.

2.6 Využití jazyka C++ CLI jako wrapperu

Wrapperem se v informační technologii obecně myslí jakýkoliv prostředek, jehož hlavním účelem je zpřístupňovat zdroje druhých stran bez anebo s minimem výpočetních úprav. Jeho účel je pouze vylepšovat pohodlí, bezpečnost nebo kompatibilitu.

Jak bylo uvedeno v podkapitole 2.1 Architektura .NET, CLI je otevřená specifikace, která popisuje tvorbu spustitelného kódu a běhové prostředí, které dovoluje souběžné užití více vysokoúrovňových programovacích jazyků, aniž by byly přepsány pro specifickou architekturu. Implementace tohoto rozšíření se díky firmě Microsoft dočkal i jazyk C++.

V našem případě vytváříme wrapper především z důvodu kompatibility mezi řízeným a neřízeným kódem. Jazyk C++ CLI obsahuje možnosti, jak pracovat s objekty a ukazateli v neřízené i řízené části paměti (haldě). Pro práci s řízenou částí paměti definuje nové operátory.

[13] Passing Structures MICROSOFT. *Microsoft Development Network* [online]. 2014 [cit. 2014-12-28]. Dostupné z: http://msdn.microsoft.com/en-us/library/awbckfbz.aspx

Operátor ^ je do jisté míry obdobou operátoru * pro práci s ukazateli v C++, odkazuje ale na objekt v řízené části paměti. Operátor gcnew vytváří instance objektů v řízené paměti. Operátor % dereferencuje ukazatel na objekt, stejně jako operátor & v jazyce C++.[14]

Následující zdrojový kód ukazuje deklaraci jednoduchého wrapperu, který zabaluje třídu MyClass obsahující metody GetName a SetName.

```
public ref class MyClassWrapper
{
public:
        MyClassWrapper (MyClass  *ptrClass);
        ~ MyClassWrapper ();
        property String^ Name;

private:
        MyClass *wrapped;
};
```

Při volání konstruktoru takového wrapperu je poté nutné dodat ukazatel na původní existující objekt zabaleného typu, v daném případě MyClass* ptrClass. Nutno podotknout, že tento konstruktor lze zavolat pouze z neřízeného kódu. Při destrukci objektu wrapperu by se nemělo zapomenout zapouzdřený objekt dealokovat voláním delete, pokud se tak neděje jinde v programu. Ukázka definice wrapperu je uvedena níže.

```
MyClassWrapper::~TestCaseResultWrapper(){
        delete wrapped;
}

MyClassWrapper::TestCaseResultWrapper(MyClass* ptrClass) {
        wrapped = ptrClass;
}

String^ TestCaseResultWrapper::Name::get(){
        return marshal_as<String^>(wrapped->GetName());
}
void TestCaseResultWrapper::Name::set(String^ value){
        wrapped->SetName(marshal_as<std::string>(value));
}
```

[14] 14 NAGEL, CH. et al.C# 2008 Programujeme profesionálně, svazek 2, s. 710-715.

3 Testování řídicích jednotek

Úkolem této práce je vytvořit GUI, nikoliv návrh HW ani implementace testů pro řídicí jednotky. Celá aplikace však přímo z GUI s jednotlivými přístroji úzce spolupracuje, proto celé pracoviště určené k testování jak LIN, tak CAN jednotek krátce představíme dle příručky programátora[15].

3.1 Sestava testovacího pracoviště

Testovací pracoviště je složeno ze standardně vyráběných přístrojů, diagnostického přístroje, CANoe a dále dvou přístrojů vyrobených na katedře měření ČVUT – „vnější jednotky“ a AG-CAN. Celá sestava je zobrazena na obr. 3.1 Sestava testovacího pracoviště. V následující tabulce je seznam jednotlivých komponent.

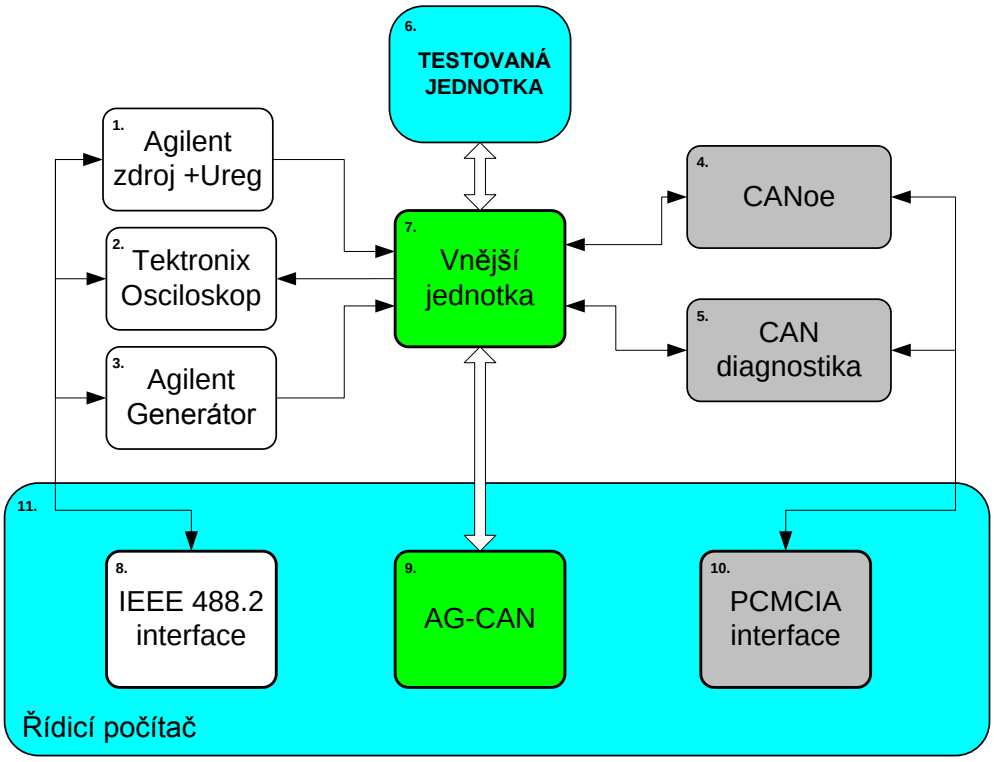
Přístr.	**Popis**	**Výrobce**	**Typové označení**
1.	Agilent zdroj +Ureg	Agilent	E3632A
2.	Tektronic Osciloskop	Tektronix	TDS 3032A
3.	CANoe	Vector Informatic GmbH	CANcardX
4.	CANdiagnostika	Softing	EICcard2/HW v1.10
5.	Testovaná jednotka		
6.	Vnější jednotka	FEL-ČVUT K13138	TEST-CAN v3.0
7.	IEEE 488		
8.	AG-CAN	FEL-ČVUT K13138	AG-CAN v3.0
9.	PCMCIA		
10.	Řídicí počítač	PC sestava s procesorem P4.	

Tab 3.1 Seznam přístrojů testovacího pracoviště. Zdroj: NOVÁK, J. *CAN Test Příručka programátora v. 2.1.*

Řízení „**vnější jednotky**“ je ovládáno pomocí **AG-CAN**u. **AG-CAN** je zásuvná karta typu PCI, ve které je implementován inteligentní CAN/LIN generátor se záchytným systémem. V „**řídicím počítači**“ je zasunut **AG-CAN**, GPIB rozhraní pro ovládání GPIB přístrojů a PCMCI rozhraní pro ovládání **CANdianostik**y a **CANoe**. Řízení testů je

[15] NOVÁK, J. CAN Test Příručka programátora v. 2.1.

ovládáno programem v „řídicím počítači“, který průběh testů zaznamenává formou protokolu o měření. [16]



Obr. 3.1 Sestava testovacího pracoviště. Zdroj: NOVÁK, J. *CAN Test Příručka programátora v. 2.1*.

3.1.1 Vnější jednotka

Vnější jednotka je realizována dvoustranným plošným spojem, který je osazen převážně relé ve funkci spínačů a přepínačů. Vhodně sepnutá relé (pomocí konfigurace z karty AG-CAN) cíleně propojují měřicí přístroje a napájecí zdroje s měřenou jednotkou. Plošný spoj obsahuje CPLD obvod XILINX, který slouží ke konfiguraci relé, k sériové komunikaci s AG-CANem a k ovládání budičů CAN a LIN. Dále je zde obvod pro měření vnitřního odporu rozhraní CAN a LIN, obvod pro detekci úrovně napájecího napětí testované jednotky a výkonový operační zesilovač pro generování startovacích průběhů.[17]

3.1.2 Karta AG-CAN

Fyzicky se jedná o PCI kartu umístěnou v „**řídicím počítači“**. Karta je osazena čipem PCI rozhraní typu PCI9030, programovatelným hradlovým polem Altera, čtveřicí čipů SJA1000 a obvody rozhraní pro komunikaci s „**vnější jednotkou**“. Programovatelné hradlové pole propojuje PCI čip s čipy SJA1000 a dále je v něm implementován

[16] NOVÁK, J. CAN Test Příručka programátora v. 2.1.

[17] Tamtéž.

inteligentní CAN/LIN generátor, dvojice obvodů UART s podporou protokolu LIN a CAN/LIN trigrovací jednotky se záchytným časovým systémem.[18]

3.2 Popis standardu LIN

Sběrnice LIN je sériová asynchronní sběrnice používající ke komunikaci jednovodičové spojení připojených zařízení. Je navržena pro použití v automobilové technice s ohledem na minimální cenové náklady spojené s její aplikací. S tím je spojena i nižší komunikační rychlost, typicky 19200 bit/s a použití standardního rozhraní UART/SCI. Nemá za cíl nahradit v automobilech dnes hojně používanou spolehlivou, robustní a rychlou sběrnicí CAN, ale má pokrýt množinu aplikací, pro které je použití sběrnice CAN zbytečně nákladné, nebo zatím nebyly z cenových důvodů napojeny na elektronický řídicí systém automobilu. Cena vynaložená na propojení s lokální sítí automobilu je několikrát nižší než u CANu. I když se může zdát tato částka vzhledem k ceně vozu zanedbatelná, je třeba si uvědomit, že elektronika je dnes v automobilech téměř všude a každé ušetřené euro při sériové výrobě, kdy se vyrábí statisícové série, je velmi důležité. Výrobce totiž zvyšuje užitnou hodnotu produktu při zachování stejné ceny.

Současné aplikace vycházejí převážně z oblasti automobilového průmyslu. Jedná se především o ovládání a polohování zrcátek, stahování oken, ovládání zámků dveří a střešního okna, polohování sedadel, ovládání klimatizace, stěračů nebo osvětlení. LIN zde realizuje propojení čidel, ovladačů, akčních členů a indikátorů. Tyto jednotky pak mohou být jednoduše napojeny na síť automobilu a stávají se dostupné pro všechny typy diagnostiky při servisních pracích.[19]

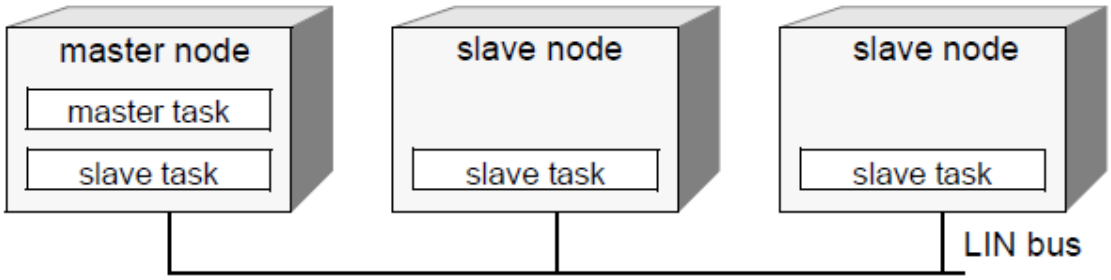
3.2.1 Struktura sítě LIN

Zařízení připojená do sítě pomocí jednoho LIN rozhraní se označují jako uzel (node). Ty se dále shlukují do clusterů, v nichž pak vždy jeden z uzlů plní úlohu jak mastera, tak slave; zbylé uzly zastávají pouze úlohu slave. Master rozhoduje, kdy a který rámec bude

[18] NOVÁK, J. CAN Test Příručka programátora v. 2.1.

[19] SUTORÝ, Tomáš. LIN (Local Interconnect Network). Elektrorevue [online]. 2004, roč. 2004, č. 12 [cit. 2014-12-19]. Dostupné z: http://www.elektrorevue.cz/clanky/04012/index.html

přenesen po sběrnici. Úkolem slave je dodat data přenášená v těchto rámcích. Jeden uzel může být členem více clusterů. [20]



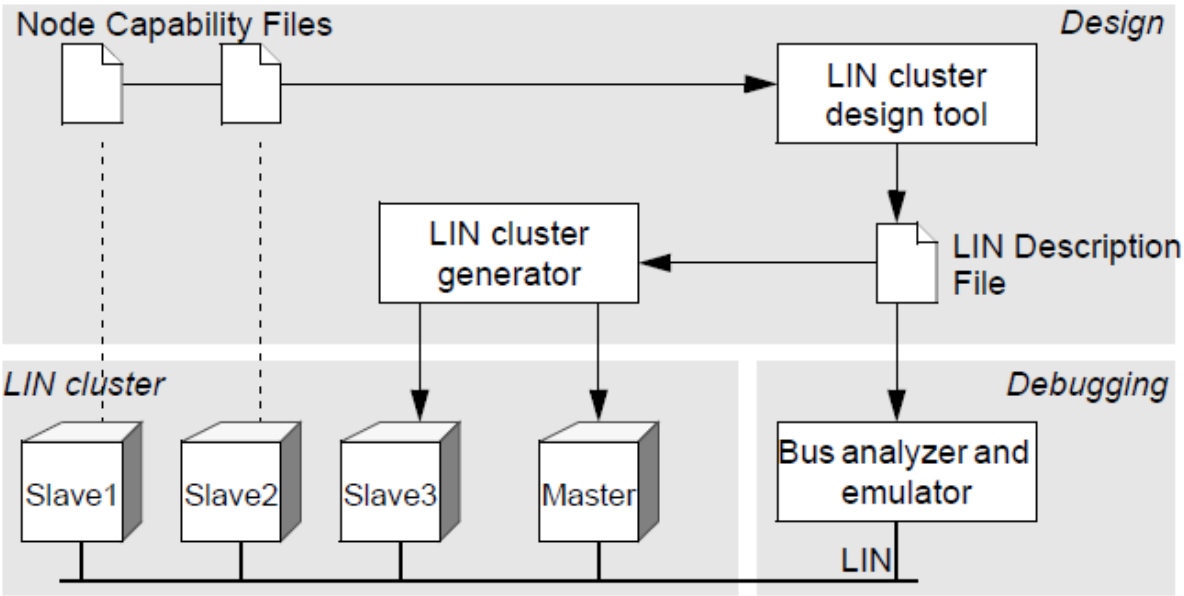
Obr. 3.2 Příklad clusteru tvořeného jedním master uzlem a dvěma slave uzly. Zdroj: LIN Consortium: *LIN Specification Package Revision 2.1*

3.2.2 Koncepce návrhu sítě LIN

LIN umožňuje využití řady návrhových a vývojových nástrojů pro urychlení, vývoj a zajištění spolehlivosti clusteru. K tomu dopomáhá definice jazyka pro popis schopností uzlu (node capability language specification), na jehož základě výrobce zařízení dodá soubor se všemi možnostmi uzlu, NCF (Node capability File). Ty je dále možné parsovat návrhovým nástrojem LIN clusteru. Na jejich základě vznikne LDF soubor (LIN Description File). LDF je dále parsováno LIN cluster generátorem, který automaticky nastaví příslušné funkce jednotlivých nodů. LIN soubor lze dále využít nástroji pro analýzu a ladění clusteru LIN. Celý proces je zobrazen na obr. 3.3 Proces návrhu a tvorby sítě LIN.[21]

20 LIN Consortium: LIN Specification Package Revision 2.1

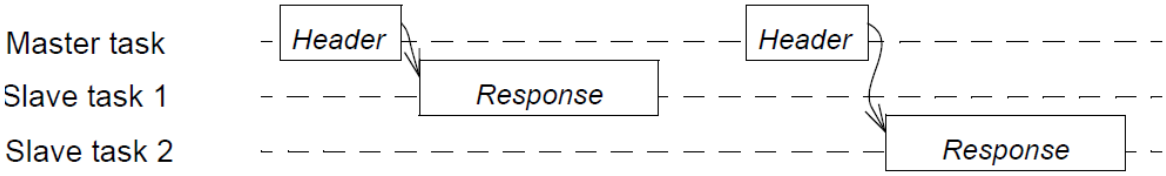
21 Tamtéž.



Obr. 3.3 Proces návrhu a tvorby sítě LIN, Zdroj: LIN Consortium: *LIN Specification Package Revision 2.1Specification Package Revision 2.1.*

3.2.3 Rámce

Rámce se skládají z hlavičky (header) vysílané masterem a z odpovědi (response) dodané příslušným slave uzlem. Hlavička se skládá z přerušovacího a synchronizačního pole následovaného identifikátorem rámce, který jednoznačně identifikuje účel rámce. Úkol slave, dodávání odpovědí asociovaných s identifikátorem, zobrazuje obr. 3.4 Tvorba rámce na sběrnici LIN. Odpověď se skládá z datového pole a pole kontrolního součtu.



Obr. 3.4 Tvorba rámce na sběrnici LIN, Zdroj: LIN Consortium: *LIN Specification Package Revision 2.1Specification Package Revision 2.1.*

3.2.4 Signály a diagnostické zprávy

V rámci mohou být přenášeny dva typy dat: signály nebo diagnostické zprávy. Signál je skalární hodnota nebo pole dat, které je zabaleno do datového pole rámce. Signály jsou vždy umístěny na stejné pozici v datovém poli ve všech rámcích se stejným identifikátorem. Diagnostické zprávy jsou přenášeny pomocí dvojice rámců

s identifikátory 60 a 61. Interpretace datového pole závisí nejenom na datech samotných, ale také na stavech komunikujících uzlů.

3.2.5 Časovací tabulka

Master vysílá hlavičky na základě časovací tabulky (schedule table), která specifikuje rámce a intervaly mezi začátkem a koncem rámců. Aplikace master uzlu může obsahovat více časovacích tabulek a vybírat mezi nimi dle stavu a potřeb aplikace.

3.3 LIN Description File

LDF popisuje celý cluster; obsahuje všechny informace nutné k monitorování clusteru. Tyto informace lze využít i k emulování jednoho nebo více uzlů, pokud jsou tyto uzly fyzicky nedostupné. Syntaxe LDF souboru je dostatečně jednoduchá k tomu, aby se dala upravovat pomocí textového editoru, nicméně lze využít i různé počítačem řízené vývojové nástroje. Aplikace je vyvíjená pro LDF verze 2.0, nicméně starší verze standardu jsou zpětně kompatibilní. Využití NCF, zmíněné v podkapitole 3.2.2 Koncepce návrhu sítě LIN, představují nejjednodušší cestu k automatickému vygenerování LDF. Jelikož později v aplikaci s informacemi z LDF pracujeme, ve zkratce popíši význam jednotlivých elementů.

Všechny elementy v LDF souboru mají přesně dané pořadí, v jakém se mohou vyskytovat. Pokud se jedná o element, jenž v sobě obsahuje více informací, následuje na témž řádku složená otevírací závorka „{“. Pro přehlednost není uvedena celá definice souboru najednou, ale po jednotlivých elementech; vnitřky elementů jsou vynechány a uvedeny později. Povinné elementy jsou zvýrazněny tučným písmem. Text uzavřený v hranatých závorkách [] se může vyskytovat jednou i vícekrát.[22]

3.3.1 Globální definice

```
LIN_description_file;
LIN_protocol_version = <char_string>;
LIN_language_version = <char_string>;
LIN_speed = <real_or_integer> kbps;
Channel_name = <char_string>;
Nodes {
}
```

[22] LIN Consortium: LIN Specification Package Revision 2.1

```
composite  {
}
Signals {
}
Diagnostic_signals {
}
Frames {
}
Sporadic_frames {
}
Event_triggered_frames  {
}
Diagnostic_frames {
}
Node_attributes {
}
Schedule_tables {
}
Signal_encoding_types {
}
Signal_representation  {
}
```

Globální definice popisuje obecné vlastnosti clusteru. Zahrnuje:

- LIN_protocol_version – verze použitého protokolu LIN.
- LIN_language_version – verze použitého jazyka LIN, tato práce popisuje verzi 2.0.
- LIN speed definition – definuje bitovou rychlost clusteru. Pohybuje se od 1 do 20 kbit/s.
- Channel postfix – postfix pro všechny názvy v LDF. Řeší možné konflikty v názvech, pokud je master node členem více jak jednoho clusteru.[23]

3.3.2 Definice uzlů

Sekce definice uzlů identifikuje jména všech účastnících se uzlů. Pro master uzel navíc specifikuje časovou základnu a jitter. Definice v této sekci vytvářejí množinu identifikátorů, všechny musí být unikátní.

[23] LIN Consortium: LIN Specification Package Revision 2.1

```
Nodes {
        Master: <node_name>,  <time_base> ms, <jitter> ms;
        Slaves: <node_name>, <node_name>, <node_name>,…;
}
```

- time_base – nejmenší časová jednotka použitá v clusteru, implementovaná masterem. Časování rámců v časovací tabulce je založeno na této hodnotě. Obvykle 5 nebo 10 ms.
- jitter – specifikuje rozdíl mezi minimálním a maximálním časovým zpožděním začátku hlavičky rámce.[24]

3.3.3 Atributy uzlu

Atributy uzlu dodávají všechny nutné informace o chování jednotlivých uzlů slave.

```
Node_attributes {
        [<node_name> {
                LIN_protocol = <char_string> ;
                configured_NAD = <integer> ;
                initial_NAD = < integer> ;
                product_id = <integer>, <integer> , <integer>;
                response_error = <identifier> ;
                fault_state_signals = <identifier> [,<identifier> ];
                P2_min = <real_or_interger> ms ;
                ST_min = <real_or_interger> ms ;
                configurable_frames {
                        [<identifier> = <identifier> ;]
                }
        }]
}
```

Všechny identifikátory <node_name> musí být definovány jako uzly slave z předešlé kapitoly.

- LIN_protocol – verze protokolu LIN, který tento uzel podporuje.
- configured_NAD – specifikuje síťovou adresu, kterou je uzel adresován uvnitř požadavku. Používá se i k řešení konfliktu. Musí být unikátní v rámci clusteru.
- initial_NAD – výchozí adresa uzlu. Pokud není definována, použije se configured_NAD.

[24] LIN Consortium: LIN Specification Package Revision 2.1

- product_id – Skládá se z supplier_id (identifikátoru výrobce), function_id (identifikátoru funkcionality produktu dané výrobcem) a varianty (identifikátor verze výrobku). Dohromady tvoří unikátní identifikátor výrobku.
- response_error – identifikátor signálu, používaného jednotkou k indikaci chyb komunikace.
- P2_min – minimální doba, kterou potřebuje slave na odeslání odpovědi od příjmu posledního rámce diagnostického požadavku.
- ST_min – minimální doba, kterou potřebuje slave k přípravě další části odpovědi na diagnostický rámec.
- configurable_frames - seznam všech rámců zpracovávaných uzlem.[25]

3.3.4 Definice signálů

Sekce definice signálů obsahuje identifikátory všech signálů v clusteru a jejich vlastností.

```
Signals {
        [ <signal_name>: <signal_size>, <init_value>, <published_by>
                                [, <subscribed_by>] ;]
}
```

- signal_name – jméno signálu
- signal_size – velikost signálu
- init_value – výchozí hodnota signálu je užívána všemi odběrateli signálu, dokud nepřijmou rámec s tímto signálem
- published_by – identifikátor uzlu, který signál vysílá
- subscribed_by – seznam identifikátorů uzlů, které signál přijímají.[26]

25 LIN Consortium: LIN Specification Package Revision 2.1

26 Tamtéž.

3.3.5 Definice diagnostického signálu

Jelikož se informace o vysílacích/příjímacích uzlech nevztahují na diagnostické signály, mají vlastní definici.

```
Diagnostic_signals {
        MasterReqB0: 8, 0 ;
        MasterReqB1: 8, 0 ;
        MasterReqB2: 8, 0 ;
        MasterReqB3: 8, 0 ;
        MasterReqB4: 8, 0 ;
        MasterReqB5: 8, 0 ;
        MasterReqB6: 8, 0 ;
        MasterReqB7: 8, 0 ;
        SlaveRespB0: 8, 0 ;
        SlaveRespB1: 8, 0 ;
        SlaveRespB2: 8, 0 ;
        SlaveRespB3: 8, 0 ;
        SlaveRespB4: 8, 0 ;
        SlaveRespB5: 8, 0 ;
        SlaveRespB6: 8, 0 ;
        SlaveRespB7: 8, 0 ;
}
```

3.3.6 Definice rámců

V LDF souboru se vyskytují definice čtyř typů rámců: nepodmíněný (unconditional), sporadický (sporadic), událostí spouštěný (event triggered) a diagnostický (diagnostic). Každý má trochu jinou definici.

Nepodmíněné rámce

```
Frames {
        [<frame_name>: <frame_id>, <published_by>, <frame_size> {
                [<signal_name>, <signal_offset> ;]
        }]
}
```

- frame_name – název rámce
- frame_id – identifikátor rámce, v rozsahu 0 až 59
- published_by – identifikátor uzlu, který rámec vysílá

- frame_size – velikost rámce v rozsahu 1 až 8 bajtů
- signal_name – seznam názvů signálů publikovaných v rámci
- signal_offset – specifikuje pozici LSB (Least Significant Bit) signálu v rámci.

Sporadické rámce

```
Sporadic_frames {
        [<sporadic_frame_name>: <frame_name> ([, <frame_name>]) ;]
}
```

- sporadic_frame_name – název rámce;
- frame_name – identifikátor názvu rámce, z množiny definovaných nepodmíněných rámců.

Událostí vyvolané rámce

```
Event_triggered_frames {
        [<event_trig_frm_name>:
                <collision_resolving_schedule_table>,
                <frame_id>
                [, <frame_name>] ;]
 }
```

- event_trig_frm_name – identifikátor událostí vyvolaného rámce
- collision_resolving_schedule_table – identifikátor časovací tabulky z množiny definovaných časovacích tabulek
- frame_id – identifikátor rámce, číslo v rozsahu 0 až 59
- frame_name – identifikátor názvu rámce, z množiny definovaných nepodmíněných rámců.

Diagnostické rámce

MasterReq a SlaveResp je rezervované klíčové slovo pro diagnostické rámce. Nesmí se využívat jinde jako název identifikátorů rámce.

```
Diagnostic_frames {
        MasterReq: 60 {
                MasterReqB0, 0;
                MasterReqB1, 8;
                MasterReqB2, 16;
                MasterReqB3, 24;
                MasterReqB4, 32;
                MasterReqB5, 40;
                MasterReqB6, 48;
                MasterReqB7, 56;
        }
        SlaveResp: 61 {
                SlaveRespB0, 0;
                SlaveRespB1, 8;
                SlaveRespB2, 16;
                SlaveRespB3, 24;
                SlaveRespB4, 32;
                SlaveRespB5, 40;
                SlaveRespB6, 48;
                SlaveRespB7, 56;
        }
}
```

3.3.7 Definice časovací tabulky

Klíčovou vlastností protokolu LIN je užití časovacích tabulek. Ty zaručují, že sběrnice nebude přetížena. Jsou také klíčovou komponentou, která garantuje periodicitu signálu. Deterministické chování je možné díky faktu, že všechny přenosy v LIN clusteru jsou inicializovány masterem. Zodpovědností mastera je, že všechny rámce mají dostatek času k uskutečnění přenosu. Definice časovací tabulky je následující:

```
Schedule_tables {
        [<schedule_table_name> {
                [<command>    delay <frame_time> ms ;]
        }]
}
```

- schedule_table_name – identifikátor časovací tabulky
- command – posloupnost příkazů, které časovací tabulka provádí
- frame_time – specifikuje trvání časového slotu daného příkazu.

3.3.8 Dodatečné informace

Následující sekce dodává do LDF souboru další informace, které nemají vliv na chování LIN clusteru. Obsahují ale návod, jak prezentovat data přenášená pomocí LIN. Tato definice je nepovinná.

Typy kódování signálů (Signal encoding types)

Typ kódování signálu je určen k popisu toho, jakou informaci signál reprezentuje a jaké hodnoty je měřítko signálu. Typ kódování signálu lze využít v nástrojích pro analýzu provozu na sběrnici LIN, které díky němu mohou prezentovat uložený provoz ve srozumitelné formě.

Typ kódování signálu může být definován čtyřmi různými způsoby podle toho, jaký typ hodnoty reprezentuje. Může reprezentovat: logickou hodnotou (logical_value), fyzickou hodnotou (physical_value), BCD hodnotou (bcd_value) a ASCII hodnotu (ascii_value). Znak | v definici níže reprezentuje výběr jedné z možností.

```
Signal_encoding_types {
        [<signal_encoding_type_name> {
                [logical_value, <signal_value> , <text_info>) ; |
                physical_value, <min_value>, <max_value>, <scale>, <offset> (, <text_info>) ; |
                bcd_value ; |
                ascii_value ;]
        }]
}
```

- signal_encoding_type_name – identifikátor názvu typu kódování
- signal_value – hodnota signálu, v rozsahu 0 až 65535
- text_info – textová poznámka k hodnotě signálu
- min_value – minimální přípustná hodnota, v rozsahu 0 až 65535
- max_value – maximální přípustná hodnota, v rozsahu 0 až 65535. Musí být stejná nebo vyšší než min_value
- scale – měřítko hodnoty
- offset – posunutí hodnoty

V případě, že je fyzická hodnota přijata pomocí LIN v rozsahu minimální a maximální přípustné hodnoty, výsledná hodnota se vypočítá následovně:

Fyzická hodnota = (měřítko × přijatá hodnota) + posunutí

Reprezentace signálů (Signal representation)

Deklarace reprezentace signálu je použita k asociování signálů s odpovídajícím typem kódování signálů.

```
Signal_representation {
        [<signal_encoding_type_name>: <signal_name> [, <signal_name> ];]
}
```

- signal_encoding_type_name – název typu signálu
- signal_name – název signálu

4 Implementace aplikace

4.1 Požadavky na aplikaci

Zadání práce je následující: „Navrhněte a v jazyce C# implementujte uživatelské rozhraní automatizovaného testeru vozidlových řídicích jednotek s rozhraním LIN pro OS Windows.“

Programové vybavení musí poskytovat následující funkce:

1. import popisných souborů ve formátech .ldf a .cdb
2. zobrazení LIN clusteru s parametry jednotlivých jednotek, zpráv a signálů
3. uložení veškerých parametrů v instancích C++ tříd využívajících knihovny STL
4. serializace parametrů do XML souboru
5. návrh a implementace volání jednotlivých testů implementovaných v jazyce C++

Kromě výše uvedeného zadání bylo během tvorby aplikace vzneseno několik funkčních požadavků od vedoucího práce.:

6. po vypnutí uložit veškerá nastavení tak, aby po zapnutí aplikace vypadala stejně;
7. možnost manuální konfigurace vnější jednotky a ovládání přístrojů testovacího pracoviště;
8. zobrazovat výsledky z testování a logy jednotlivých testů. Změny v logu aktualizovat na vyžádání knihovny pomocí zpětného volání (callbacku).

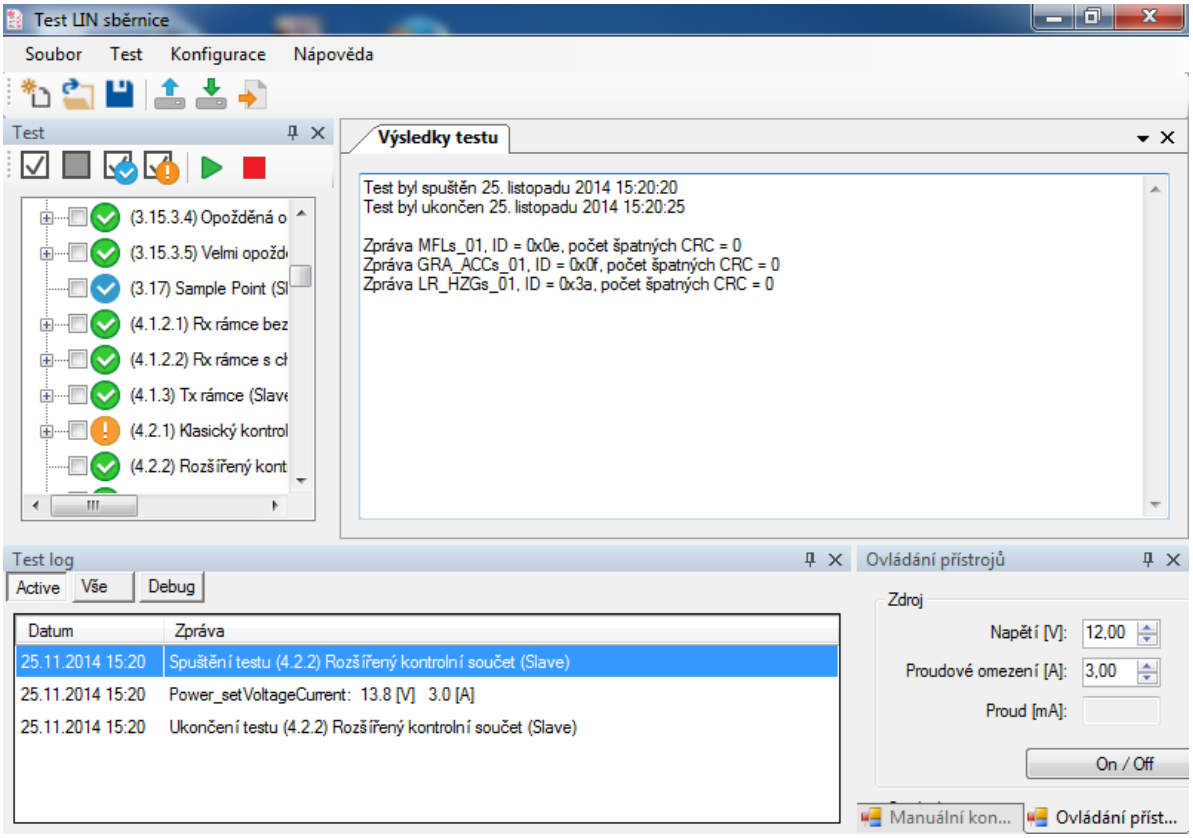
Dále několik nefunkčních požadavků:

9. navrhnout rozvrstvení aplikace tak, aby bylo možné přidávat další testovací UseCase a měnit strukturu objektů bez jakýchkoliv změn GUI, tj. pouze úpravou kódu C++;
10. navrhnout uživatelské prostředí tak, aby se příliš nelišilo od původní verze, na kterou jsou uživatelé zvyklí;
11. využít vývojové prostředí Microsoft Visula Studio 2013;
12. umožnit jednoduché ladění zdrojového kódu (debug) celého projektu.

O realizaci těchto požadavků pojednává celá kapitola 4.

4.2 Ovládání aplikace

Hlavní okno aplikace je členěno do několika panelů. Při prvním spuštění je vhodné tyto panely zorganizovat dle preference uživatele. Úkon stačí provést pouze jednou, neboť se nastavení automaticky uloží. Panely lze skrývat a zobrazovat pomocí menu konfigurace → okno a výběru příslušného panelu.



Obr. 4.1 Hlavní okno aplikace, Zdroj: vlastní zpracování

4.2.1 Nový projekt

Pro založení nového testovacího projektu stačí vybrat soubor nový, dále vybrat validní soubor typu LDF. Po načtení souboru se zobrazí dialog pro výběr jednotky, jež se má testovat. Jelikož některé informace nutné k běhu testů nejsou obsaženy přímo v LDF, je nutné je doplnit v okně konfigurace viz podkapitola 4.2.4 Vizualizace LDF. Pokud je vybrána jako testovaná jednotka master, zobrazí se dialog pro vybrání souboru typu DBC.

Před spuštěním testování je dále nutné vybrat GPIB adresy testovacích přístrojů na daném testovacím pracovišti, tj. adresu zdroje napětí, osciloskopu a generátoru. K tomu slouží dialog přístupný z menu konfigurace → přístroje.

4.2.2 Uložení a načtení projektu

Data načtená z LDF souboru spolu se všemi dodatečně zadanými informacemi lze uložit do XML souboru s příponou DLF pomocí tlačítka „Uložit DLF“, nebo „Uložit jako DLF“ z nabídky „Soubor“. Po opětovném spuštění programu se již nemusí znovu otvírat LDF dle předešlé podkapitoly, ale stačí pouze vybrat „Otevřít DLF“ a vybrat příslušný soubor. Tyto funkce jsou rovněž dostupné z lišty nástrojů.

Průběh testování lze rovněž uložit do souboru typu XML pomocí položky „Uložit výsledky“ z nabídky „Soubor“. Lze je také při dalším spuštění načíst, ovšem až po načtení LDF nebo DLF souboru. Do souboru jsou uloženy krom výsledků také aktivní logy již proběhlých testů. V konfiguraci lze navíc nastavit automatické ukládání výsledků, které při doběhnutí každého testu uloží výsledky do aktuálně načteného / uloženého výsledku. Výstupy testů lze exportovat do souboru formátu XLSX. K tomu je však nutné mít na daném počítači nainstalován program Microsoft Excel 2010.

4.2.3 Spuštění testů

Po načtení LDF souboru se na panelu testů zobrazí všechny dostupné testy. V případě, že se test skládá z více kroků, lze tyto kroky zobrazit jednotlivě po stisku „+“ u příslušného testu. V seznamu jsou zobrazovány pouze testy a kroky testů, které mají pro danou konfiguraci nebo testovanou jednotku smysl, pokud je to známo dopředu. Po vybrání testu nebo kroku testu se aktualizuje panel s výsledky testů a panel aktivních logů dle zvoleného testu.

Každý test i podtest se nachází vždy v jednom ze stavů signalizovaných ikonou. Ten se po spuštění testu nebo po načtení výsledků testů aktualizuje. Význam jednotlivých stavů ukazuje tabulka níže.

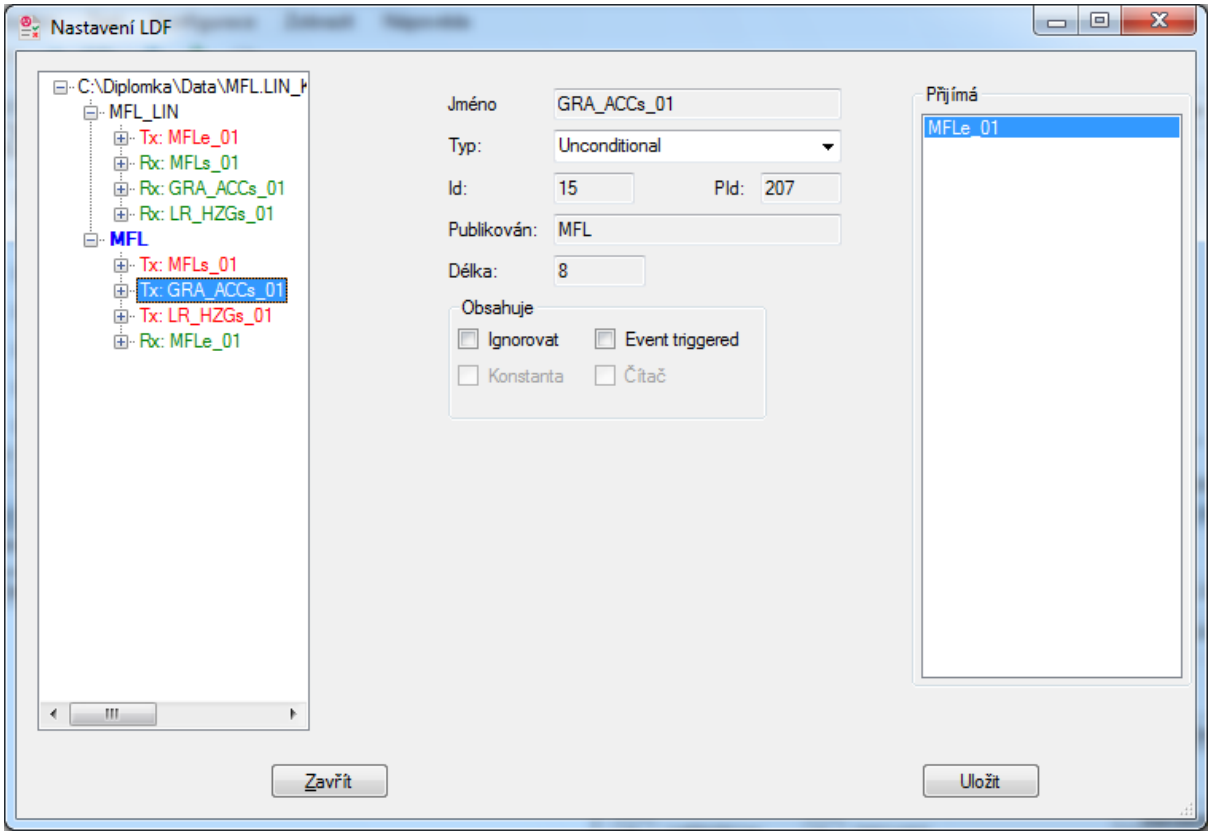
	Test nebo krok testu právě běží
	Test nebo krok testu není nakonfigurován
	Některé kroky testu obsahují chyby
	Všechny kroky testu nebo krok testu obsahuje chyby
	Test obsahuje některé validní a některé dosud nespuštěné kroky
	Testy nebo kroky testu dosud nebyly spuštěny
	Test není pro danou konfiguraci relevantní
	Test úspěšně doběhl

Tab. 4.1 Ikony stavu testů. Zdroj: vlastní zpracování

Pro spuštění testů je nutné testy, případně konkrétní kroky testů označit a poté vybrat v menu testy „Spustit vybrané testy“ či rovnou „Spustit všechny testy“. Výběr a spouštění testů usnadňují ikony na liště nástrojů panelu testů: „Označit vše“, „Odznačit vše“ „Vybrat nespuštěné testy“, „Vybrat nenakonfigurované testy“. Z tohoto panelu jde i vybrané testy spustit nebo přerušit právě probíhající testování.

4.2.4 Vizualizace LDF

Informace o aktuálně načteném LDF souboru, které popisuje podkapitola 3.3 LIN Description File, jsou přístupné z menu vybráním Konfigurace a LDF. V levé části tohoto okna se zobrazí strom objektů, uprostřed okna jsou zobrazeny informace o právě vybrané položce. Strom obsahuje čtyři úrovně, v každé jiný typ objektu. Do další úrovně se lze dostat po stisku „+“ i příslušného elementu. V nejvyšší úrovni se zobrazí informace o celém testu jako takovém. V druhé úrovni je seznam nodů obsažených v clusteru. V další úrovni se zobrazuje seznam všech rámců, které tento příslušný uzel vysílá nebo přijímá. Vysílané rámce jsou zvýrazněny červenou barvou, přijímané zelenou. V poslední úrovni je zobrazen seznam všech signálů, které příslušný rámec využívá.



Obr. 4.2 Vizualizace parametrů LDF. Zdroj: vlastní zpracování

Především tyto informace:

- Definiční soubor jednotky – stisknutím „…“ se vylistuje soubor typu TBD a stiskem import se soubor načte. TBD je definiční soubor s dalšími informacemi nezbytnými pro testování, které nejsou v LDF. Týká se zejména diagnostických protokolů.
- Signál zapalování – aktivační signál jednotek v clusteru
- Schedule table – při testování slave jednotky se využívá toto vybrané schéma.

4.2.5 Ovládání přístrojů

Připojené přístroje lze ovládat přímo z uživatelského rozhraní pomocí panelu „Ovládání přístrojů“.

Ovládání přístrojů umožňuje:

- zapínat a vypínat zdroj napětí
- zadávat napětí zdroje v rozsahu 0 - 25 V
- nastavit omezení proudu v rozsahu 0 – 7 A
- odečítat aktuální hodnoty proudu
- nastavení osciloskopu do režimu monitorování standardních LIN zpráv.

4.2.6 Manuální konfigurace vnější jednotky

U vnější jednotky popsané v podkapitole 3.1.1 Vnější jednotka lze ručně zadat, které relé má sepnout či rozepnout. Jednu právě vybranou „konfiguraci“ sepnutých relé lze uložit tlačítkem „Zapamatuj“ a opět vyvolat tlačítkem „Obnov“. Konfigurace se ukládá do souboru a v případě znovu otevření aplikace ji lze vyvolat zpět.

4.3 Popis implementace

4.3.1 Rozvržení zdrojového kódu

Zdrojové kódy projektu jsou tvořeny dvěma vzájemně propojenými projekty. Prvním z nich je projekt pojmenovaný „LINLib“. Jedná se o klasickou dynamickou knihovnu typu win32, která obsahuje všechny zdrojové kódy psané v C+ +. Kromě lehce modifikovaných, původních zdrojových kódů z předešlého projektu CANTest obsahuje i nově vytvořené třídy pro volání z řízené části aplikace, tzv. wrappery uložené ve zvláštní složce. Dále jsou zde nové třídy pro práci s LDF formátem „LDFFileC“ a nastavení parametrů testovacího prostředí ve třídě „LINTest“. Přibylo i mnoho nových funkcí pro testování LIN obsažených ve složce Test Case. Všechny zdrojové kódy testů v této složce byly implementovány Ing. Tomášem Grusem.

Druhý projekt, pojmenovaný LINTest, obsahuje především implementaci samotných formulářů v jazyce C#. Využívá k tomu rozhraní Winforms, popsané v podkapitole 2.4 Winforms a datové vazby, a wrappery popsané v další kapitole.

4.3.2 Implementace wrapperů

Pro implementaci interoperability popsané v podkapitole 2.5 Volání nativního kódu z .NET je využito rozšíření CLI jazyka C++. Wrappery byly vytvořeny pro každý objekt, který je třeba ovládat nebo zobrazit z/v GUI.

Wrapper v principu nedělá nic jiného, než že si ukládá ukazatele na původní neřízený objekt a definuje vlastnosti a metody, které zpřístupňují a volají metody a vlastnosti tohoto objektu. V určitých případech jsou ovšem nutné určité konverze mezi datovými typy, které nemají implicitní konverzi. K tomu pomáhá funkce marshal_as definovaná v knihovně „marshal_cppstd.h“.

Od ostatních zdrojových souborů se soubor wrapperů odlišuje především nastavením kompilace. Jelikož se z něj vytváří řízená třída, je nutné nastavit před kompilací přepínač /clr. To lze provést jak v rámci celého řešení (solution), tak v rámci jednotlivých souborů. S ohledem na to, že cílem práce je maximálně zjednodušit kompilaci a ladění (debugging) nativních části kódu, bylo rozhodnuto neoddělovat v řešení projekty na řízené a neřízené, ale přidat wrappery do existujícího projektu a nastavit příznak /clr pouze na tyto soubory. Pokud by tímto způsobem byla spuštěna kompilace, zjistilo by se, že řada funkcí není v případě kompilace řízeného kódu podporovaná a proces kompilace skončí s chybou. Z chyb, které hlásí kompilátor, by se postupně došlo k seznamu několika dalších parametrů, které CLR prostředí nepodporuje. Přehledný soupis těchto parametrů, které je nutné změnit, je zachycen v tabulce níže.

Název parametru	Umístění na kartě	Nunto nastavit na	Zkratka
Common language runtime support	General	Common Language RunTime Support	/clr
Debug Information format	General	Program Database for Edit And Continue	/ZI
Enable minimal rebuild	Code generation	No	/GM-
Enable C++ exceptions	Code generation	Yes with SEH Exceptions	/EHa
Basic runtime checks	Code generation	Default	
Precompiled Headers	Precompiled headers	Not using precompiled headers	
Enable Run-time type information	Language	Yes	/GR

Tab. 4.2 Změněné parametry souboru kompilovaného jako CLR. Zdroj: vlastní zpracování.

Nutno podotknout, že během práce s wrappery jsem narazil na chybu vývojového studia. Pokud se na vytvořeném .cpp souboru změní přepínač /clr dodatečně, Visual Studio si tuto informaci ukládá do zdrojového souboru celého projektu. V daném případě je to soubor LINLib.vcxproj, jehož formát je čitelný, jelikož se jedná o klasické XML. K danému souboru si vývojové studio poznačí „<CompileAsManaged“ s podmínkou, která tento atribut omezí na konkrétní konfiguraci a platformu. V daném případě ale vývojové prostředí chybně volilo první možnou platformu místo aktuální editované.[27] Chybný řádek ilustruje příklad níže. Pro vyhnutí se tomuto problému je buďto nutné volbu /clr zadat na všech konfiguracích zároveň, nebo zdrojový soubor upravit ručně.

```
<ClCompile Include="Wizard.cpp">
<CompileAsManaged
Condition="'$(Configuration)|$(Platform)'=='Debug64|Win32'">true</CompileAsManaged>
</ClCompile>
```

Výpis zdrojového kódu všech vytvořených wrapperů by byl příliš obsáhlý. Příklad funkčnosti wrapperu je uveden v podkapitole 2.6 Využití jazyka C++ CLI jako wrapperu. Níže je pouze uveden výpis všech wrapperů z aplikace ve formě tabulky.

Název wrapperu	Náze zapouzdřené třídy
CANMessageCWrapper	CANMessageC
CANSignalCWrapper	CANSignalC
CANUnitCWrapper	CANUnitC
DbcFileWrapper	DbcFileC
LdfFileWrapper	LdfFileC
LINMessageCWrapper	LINMessageC
LINScheduleTableCWrapper	LINScheduleTableC
LINSignalCWrapper	LINSignalC
LinTestWrapper	LinTest
LINUnitCWrapper	LINUnitC
LogEntryWrapper	CLogEntry
LogWrapper	CLog
PciCanCWrapper	PciCanC

[27] Is it possible to run unmanaged C++ normally from a managed C++/CLI project?. In: *Stackoverflow* [online]. 2013 [cit. 2014-12-29]. Dostupné z: http://stackoverflow.com/questions/15598445/is-it-possible-to-run-unmanaged-c-normally-from-a-managed-c-cli-project

TestCaseWrapper	TestCase
TestCaseResultWrapper	TestCaseResult
TestStepResultWrapper	TestStepResult
TestStepWrapper	TestStep

Tab. 4.3 Soupis všech tříd wrapperu v aplikaci. Zdroj: vlastní zpracování

Za zvláštní zmínku stojí jeden wrapper s názvem LogEntryWrapper. Jako jediný nedostává jako vstupní parametr ukazatel na třídu, nýbrž ukazatel standardní knihovny C++ - shared_ptr<CLogEntry>. V řízené třídě nelze použít jako proměnnou třídy instanci nativní třídy jako takové, pouze ukazatel na něj. Nelze tedy využít shared_ptr jako proměnnou třídy. Pokud bychom navíc vzali adresu, kam shared_ptr odkazuje, např. metodou get, mohlo by se stát následující: Po nějaké době metoda, která předávala shared_ptr, objekt zničí a tím se dostane počet referencí shared_ptr na 0 a zavolá destruktor objektu, na který ukazuje. Tento problém řeší managed_ptr, jež zajišťuje vytvoření nového shared_ptr tak, aby se počet referencí nesnižoval. Wrapper zmíněný na začátku nepracuje s CLogEntry*, ale s managed_ptr<CLogEntry>. Kód třídy managet_ptr je ukázán níže.[28]

```
template <class T>
ref class managed_ptr
{
	shared_ptr<T>* ptr;
public:
	managed_ptr(){
		ptr = NULL;
	}
	managed_ptr(T* data){
		ptr = new shared_ptr<T>(data);
	}
	managed_ptr(shared_ptr<T> _ptr) : ptr(new shared_ptr<T>(_ptr)) {
	}
	~managed_ptr(){
		int used1 = ptr->use_count();
		delete ptr;
	}
	operator shared_ptr<T>() {
```

[28] C++/CLI Wrapping a Function that Returns a std::shared_ptr. In: *C++/CLI Wrapping a Function that Returns a std::shared_ptr* [online]. 2011 [cit. 2014-12-29]. Dostupné z: http://stackoverflow.com/questions/6172361/c-cli-wrapping-a-function-that-returns-a-stdshared-ptr

```
		return *ptr;
	}
	managed_ptr<T>% operator=(T* _ptr) {
		delete ptr;
		ptr = new std::shared_ptr<T>(_ptr);
		return *this;
	}
	managed_ptr<T>% operator=(shared_ptr<T> _ptr) {
		delete ptr;
		ptr = shared_ptr<T>(_ptr);
		return *this;
	}
	T* operator->(){
		return (*ptr).get();
	}
	T* get(){
		return (*ptr).get();
	}
	void reset(){
		ptr->reset();
	}
	shared_ptr<T> GetShared(){
		return shared_ptr<T>(*ptr);
	}
};
```

4.3.3 Třídy popisující testy

Hlavním účelem aplikace je provádění testů. Tyto testy jsou ve většině případů děleny do více kroků – subtestů. Třídy implementující tyto testy se liší dle konkrétního testovacího případu. Všechny mají ale společného předka. V případě testů je to třída TestCase, v případě kroků testu je to třída TestStep. V některých případech je celý test složen z jednoho kroku a lze ho spustit samostatně tak, jako by se jednalo o krok testu. Proto i třída TestCase je potomkem třídy TestStep.

Obě třídy, TestCase i TestStep, byly převzaty z původního projektu, nicméně byly modifikovány pro účely testeru LIN. Dynamicky vytvořený seznam všech TestCase je možné získat metodou CreateTestCases, třídy LINTest. Následně lze z každé třídy TestCase získat seznam jeho TestStepů metodou GetTestSteps. Při změně konfigurace se ale některé testy nebo kroky testů mohou stát pro danou testovanou jednotku irelevantní a měly by být ze seznamu odstraněny. K tomu je nutné volat metodu UpdateTestCases.

Třída TestStep zpřístupňuje všechny informace o příslušném testovacím kroku, jako je název, identifikátor, zda je krok nakonfigurován a použitelný. Hlavně však obsahuje metodu Run, která spustí příslušný test. Tato metoda je přetížena v každém potomku kroku testu dle logiky příslušného testu.

4.3.4 Import a struktura LDF

Dalším úkolem bylo načíst informace ze souboru LDF. O významu jednotlivých elementů souboru LDF pojednává podkapitola 3.2 Popis standardu LIN. Třída, která obsahuje jak parsování souboru LDF, tak také všechny odkazy na celou strukturu objektů, je nazvaná LDFFileC. Jelikož bylo nutné tyto elementy nějak reprezentovat a aplikace vyžaduje práci jak se soubory LDF, tak se soubory CBD a oba tyto standardy definují jiné názvosloví, s vedoucím práce jsme se dohodli, že přizpůsobíme názvosloví objektů podle původního modelu převzatého z CANu. Při parsování se proto uzel uloží do třídy LINUnitC, Frame do LINMessageC a signál do LINSignalC. Dále jsou ještě během parsování naplněny třídy LINScheduleTableC a LINSignalEncodingTypeC, které ale nemají v CANu obdobný protějšek. Přehledný popis objektů v LINu je v tabulce níže.

Název Třídy	Originální název prvku v LINu a číslo kapitoly	
LdfFileC	3.3.1	LIN description file
LINUnitC	3.3.2	Node + 3.3.3 Node attribute
LINMessageC	3.3.6	Frame
LINSignalC	3.3.4	Signal
LINSignalEncodingTypeC	3.3.8	Signal Encoding Type
LINSignalValueC	3.3.8	Signal representation
LINScheduleTableC	3.3.7	Schedule table
LINUnitCMaster	3.3.2	Node + 3.3.3 Node attribute

Tab. 4.4 Třídy wrapperů souboru LDF. Zdroj: vlastní zpracování

Implementaci všech těchto tříd lze nalézt v souboru LdfFileC. Během parsování LDF jsou tvořeny instance výše zmíněných tříd, které jsou pak uloženy za užití standardní C++ knihovny do asociativního pole - mapy. Jako klíč je zvolen název daného objektu, jež je podle definic LINu unikátní identifikátor. Během parsování jsou tyto třídy dále propojeny pomocí ukazatelů, které vždy ukazují na některou instanci z těchto map. Proto i destrukce

všech těchto instancí je řízena z destruktoru třídy LdfFileC. Deklarace celé třídy LdfFileC včetně těchto je následující:

```
class LdfFileC
{

public:
	LdfFileC();
	virtual ~LdfFileC();

	typedef map<wstring, LINMessageC*>::iterator tit_messages;
	typedef map<wstring, LINSignalC*>::iterator tit_signals;
	typedef map<wstring, LINUnitC*>::iterator tit_units;

	wstring m_LdfName;
	wstring testedUnitName;
	LINMasterUnitC *master;
	map<wstring, LINUnitC*> slaves;
	map<wstring, LINSignalC*> signals;
	map<wstring, LINMessageC*> messages;
	map<wstring, LINScheduleTableC*> scheduleTable;
	map<wstring, LINSignalEncodingTypeC*> signalEncodingType;

	void Init();
	void Clear();
	wstring GetPath();
	wstring GetOriginalLDFPath();
	int NewFromFile(wstring path);
	int LoadFile(wstring path);
	int SaveFile(wstring path);

	LINUnitC *FindSender(LINMessageC *msg);

	wstring LIN_protocol_version;
	wstring LIN_language_version;
	wstring TBDFilePath;
	float LIN_speed;

	LINSignalC *findSig(wstring name);

private:
	wstring m_path;
	wstring lastLine;
	size_t lastIndex;
	wifstream ldfFile;
```

```
        long lineNumber = 0;
        wstring TrimUntil(wstring ignoreCase, wstring notIgnoreCase);
        wstring Trim(wstring ignoreCase);
        void FindWord(wstring key);
        void NextLine();
        void UpdateTrRcvMaps();
};
```

Samotné parsování souborů je implementované v metodě NewFromFile. Ta využívá ke své funkci několik proměnných.

4.3.5 Implementace serializace XML

Jedním z bodů zadání je i serializace informací do souboru typu XML. Způsob implementace, který se ihned nabízí, je pomocí wrapperu předat veškerá data k serializaci do GUI s využitím celkem komfortního systému serializace v jazyka C# popsané v podkapitole 2.3.7 Serializace objektů do XML. Tento postup má ale jednu vadu: jakékoliv změny v struktuře serializovaných tříd by se nutně musely implementovat i v příslušném wrapperu a v jazyce C#. To je bohužel v rozporu s požadavkem 9) v seznamu požadavků v podkapitole 4.1 Požadavky na aplikaci. Proto jsme byli nuceni rozdělit serializaci dat do XML na dvě nezávislé části: Serializaci dat LDF souboru tvořenou čistě v C++ a Serializaci nastavení GUI.

Serializace LDF

Serializace dat LDF souboru je díky výše zmíněnému problému realizována čistě v C++ pomocí knihovny BOOST[29] verze 1.56. Pro kompilaci zdrojového kódu programu je nutné tuto knihovnu stáhnout a uložit do složky „C:\boost\boost_1_56_0“. Na implementaci této serializace se značně podílel Ing. Tomáš Grus.

Pro účely implementace serializace vznikly v příslušných třídách pomocné proměnné s prefixem „xs_“. Ty v sobě ukládají identifikátory příslušných objektů tam, kde třída obsahuje ukazatel na tyto objekty. Veškeré třídy, které se serializují, musejí mít přetíženou metodu serialize. Pomocí makra BOOST_SERIALIZATION_INST_MEMBER_NVP jsou v této metodě proměnné určené k serializaci přidávány do výstupního streamu. Níže uvádíme pro ilustraci jednu z implementací metod serialize, pro třídu LINSignalC:

[29] *Boost c++ libraries* [online]. 2014 [cit. 2015-01-04]. Dostupné z: http://www.boost.org/

```
template<class Archive>
void serialize(Archive& ar, LINSignalC& r, const unsigned int version)
{
        ar & BOOST_SERIALIZATION_INST_MEMBER_NVP(xs_signalEncoding, r)
        & BOOST_SERIALIZATION_INST_MEMBER_NVP(type, r)
        & BOOST_SERIALIZATION_INST_MEMBER_NVP(signalName, r)
        & BOOST_SERIALIZATION_INST_MEMBER_NVP(signal_size, r)
        & BOOST_SERIALIZATION_INST_MEMBER_NVP(signal_init_value, r)
        & BOOST_SERIALIZATION_INST_MEMBER_NVP(published_by, r)
        & BOOST_SERIALIZATION_INST_MEMBER_NVP(subscribers, r)
        & BOOST_SERIALIZATION_INST_MEMBER_NVP(isDiagnostic, r);
        & BOOST_SERIALIZATION_INST_MEMBER_NVP(init_value, r);
  }
```

Výňatek kódu níže z metody SaveFile ukazuje volání serializace včetně všech tříd. Dále obsahuje ukládání souboru pomocí operátoru „<<” do streamu.

```
ostringstream oss;
boost::archive::xml_oarchive oa(oss);

try
{
    oa  << BOOST_SERIALIZATION_NVP(testedUnitName)
        << BOOST_SERIALIZATION_NVP(m_LdfName)
        << BOOST_SERIALIZATION_NVP(signalEncodingType)
        << BOOST_SERIALIZATION_NVP(signals)
        << BOOST_SERIALIZATION_NVP(slaves)
        << BOOST_SERIALIZATION_NVP(messages)
        << BOOST_SERIALIZATION_NVP(scheduleTable)
        << BOOST_SERIALIZATION_NVP(master)
        << BOOST_SERIALIZATION_NVP(LIN_protocol_version)
        << BOOST_SERIALIZATION_NVP(LIN_language_version)
        << BOOST_SERIALIZATION_NVP(TBDFilePath)
        << BOOST_SERIALIZATION_NVP(LIN_speed);
}
catch (...)
{
        return 1;
}

ofstream outf;
outf.open(path, ofstream::out);
outf << oss.str();
```

Serializace nastavení GUI

Nastavení aplikace je uložené ve dvou souborech, oba se nacházejí ve složce Windows lokálních aplikačních dat, v podsložce LinTest. Její umístění je proměnná prostředí Windows, standardně se nachází ve složce aktuálního uživatele, např. “C:\Users\<User Name>\AppData\Local\LinTest”. Z toho je zřejmé, že nastavení aplikace se vytváří pro každého uživatele zvlášť.

První soubor obsahuje nastavení všech panelů na formuláři. Využívá k tomu metody dockpanelu suite, dodané tvůrcem balíčku. O tomto prvku pojednává více podkapitola 4.4.1 Plovoucí panely.

Druhý soubor, obsahující ostatní nastavení aplikace, již serializuje proměnné třídy LocalSettings pomocí jazyku C#. Princip této serializace je popsán v podkapitole 2.3.7 Serializace objektů do XML. Zde pouze ukážeme část kódu, která definuje vlastnosti třídy, jež se do souboru ukládají:

```
public int StartUpDelay { get; set; }
public int SleepAfter{ get; set; }
public int TransmitDelay { get; set; }
public int PauseAfterTest { get; set; }
public bool AutoSave { get; set; }
public bool ConnectAnalyzer { get; set; }
public bool DebugLog { get; set; }

public sbyte AddressPowerSource { get; set; }
public sbyte AddressOsciloscop { get; set; }
public sbyte AddressGenerator { get; set; }
public bool ShowDeviceControl { get; set; }
public bool ShowTest { get; set; }
public bool ShowRelay { get; set; }
public bool ShowResults{ get; set; }
public bool ShowLogs { get; set; }
public List<ChbNameCheck> presetRelay { get; set; }
```

4.3.1 Logování souboru

Jedním z dodatečných funkčních požadavků, které jsou zmíněny na začátku kapitoly, je umožnit zobrazení logů testů pomocí zpětného volání. Z původního projektu je pro tento účel převzata třída CLog, která funkci logování umožňuje, a to i s možností zpětného volání běžně využívaného v C++. Jak víme z podkapitoly 2.3.5 Události a delegáti, C#

tento mechanismus implementuje odlišně, pomocí událostí a delegátů. Aby informace např. o novém záznamu v logu doputovala až na GUI, je nutné, aby wrapper implementoval metodu, která zareaguje na zpětné volání z nativního kódu a poté vygeneruje událost pro GUI. Příslušná implementace části wrapperu reagující na volání AddEntry je následující:

```
delegate void CLogEntryCallBack(void*, const CLog*, int);
delegate void LogEntryAddedDelegate(LogEntryWrapper^);
event LogEntryAddedDelegate^ LogEntryAdded;

//Register consuption CLog->AddEntry and generate event on each AddEntry call
void RegisterAddEntryEvent(){
//Get callback ptr to local member function
        ptrAddEntryDelegate = gcnew CLogEntryCallBack(this,
        &LogWrapper::OnAddEntryEventCallBack);
        GCHandle gch = GCHandle::Alloc(ptrAddEntryDelegate);
        IntPtr ip =
            Marshal::GetFunctionPointerForDelegate(ptrAddEntryDelegate);
        CLog::LogEntryCallback theCallback =
            static_cast<CLog::LogEntryCallback>(ip.ToPointer());
        wrapped->SetOnAddEntry(theCallback, NULL);
        GC::Collect();
        gch.Free();
        wrapped->EnableCallbacks(true);
}

  //On callback generate C# event
  void OnAddEntryEventCallBack(void* UserData, const CLog* Log, int Index){
            if (Log != NULL){
                    LogEntryAdded(gcnew  LogEntryWrapper((*Log)[Index]));
            }
  }
```

Prvně se vytvoří delegát, CLogEntryCallBack, jehož typy parametrů a návratová hodnota musí odpovídat signatuře funkce zpětného volání:

```
typedef void(__stdcall *LogEntryCallback)(void* UserData, const CLog* Log, int Index);
```

Druhý delegát, LogEntryAddedDelegate spolu s událostí LogEntryAdded a metodou OnAddEntryEventCallBack slouží k vygenerování události tak, jak je popsáno v podkapitole 2.3.5 Události a delegáti. Pro vygenerování delegátu je nutné nejprve

vytvořit instanci delegátu CLogEntryCallBack pomocí konstruktoru, který má jako parametr název ukazatele na metodu (v tomto případě je nutno uvést i název třídy) a konkrétní instanci, na níž má ukazovat. Nutno podotknout, že instance tohoto delegátu musí existovat po celou dobu, kdy je možné, že bude volána. Dále je pomocí metody GetFuncitonPointerForDelegate vytvořen z delegátu ukazatel na metodu, ovšem už nativního typu. Tento ukazatel už může být po přetypování předán jako parametr funkce SetOnAddEntry, která si ho uloží a s jeho pomocí generuje zpětné volání.

4.4 Formuláře uživatelského rozhraní

Pro tvorbu uživatelského rozhraní v jazyce C# byl vybrán standardní prostředek pro tvorbu formulářových aplikací, Winforms, viz podkapitola 2.4 Winforms a datové vazby. Tato knihovna je přímo určena k tvorbě formulářových aplikací.

K Winforms na platformě .NET existuje i jiná alternativa, technologie Windows Presentation Foundation (WPF). Ta kromě jiného pozměňuje způsob práce při návrhu formuláře a umožňuje lépe pracovat s multimediálním obsahem. Nicméně žádná z těchto funkcí by nebyla nikde v programu využita. Některé formulářové prvky využívané v aplikaci jsou ve WPF více limitované. Jedním z požadavků na náš program navíc je, aby byl podobný původní verzi programu. Pro tento účel se více hodí WinForms.[30]

4.4.1 Plovoucí panely

Při návrhu formulářů se vycházelo z původního uživatelského rozhraní. První věc, která byla na tomto rozhraní zajímavá, byly „plovoucí panely“ ve stylu Visual studia. Winforms bohužel podporu takovéto funkcionality nemají. Bylo proto potřebné vyhledat a využít knihovnu třetích stran.

Jako první byla zkoušena open knihovna Crom.Control[31]. Tato knihovna sice umožňuje vytvořit tyto panely, bohužel jsme při práci s ní narazili na několik problémů. Neřešila dostatečně do hloubky „zahnizďování“ plovoucích panelů do sebe, např. zakotvení panelu na spodní okraj již zakotveného panelu. S dodanými funkcemi se

[30] Bachaalany,Advantages and disadvantages of WPF over winforms. *Code project* [online]. 2014 [cit. 2015-01-01]. Dostupné z: http://www.codeproject.com/Questions/66468/Advantages-and-disadvantages-of-WPF-over-winforms

[31] MAZARINE, Cristinel. *Visual Studio IDE like Dock Container - Second Version* [online]. 2009 [cit. 2014-12-29]. Dostupné z: http://www.codeproject.com/Articles/42800/Visual-Studio-IDE-like-Dock-Container-Second-Versi

nepodařilo docílit požadovaného uspořádání programově po startu programu tak, aby bylo vyhověno požadavku číslo 6 ze začátku této kapitoly. Druhý problém byl v nefunkčním skrývání ukotvených panelů. Navíc při některých událostech aplikace padala. Proto jsme se rozhodli toto řešení opustit, hledat dál a zkusit jinou implementaci plovoucích panelů.

Jako druhé open řešení jsme nalezli DockPanelSuite[32]. Toto řešení je dostupné i jako „NuGet“ balíček přímo z vývojového studia. Tento balík rozšiřuje pomocí dědění standardní formuláře o funkcionalitu plovoucích panelů v dostatečném rozsahu. S jeho pomocí se podařilo vytvořit všechny potřebné panely a umožnit komfortní přizpůsobení vzhledu aplikace. Navíc tato knihovna podporuje funkci ukládání pozic všech panelů do souboru, kterou lze lehce využít pro požadavek 6). Toto řešení bylo vloženo do aplikace jako NuGet balíček.

4.4.2 Hlavní formulář

Při spuštění aplikace se zobrazí hlavní formulář, ukázka je vidět na obr 4.1 Hlavní okno aplikace. V kódu ho lze nalézt v souboru FrmLINTest.cs. Kromě nabídek menu je jeho jediným „viditelným“ prvkem kontejner plovoucích panelů typu dockpanel. Tento formulář zpřístupňuje veškeré ostatní panely a formuláře v aplikaci. Zdrojový kód konstruktoru třídy je následující:

```
public FrmLINTest()
{
    InitializeComponent();
    MainTest = new LINCLI.LinTestWrapper();
    MainTest.Initialize(LocalSettings.Settings.AddressPowerSource, 0,
        LocalSettings.Settings.AddressOsciloscop, LocalSettings.Settings.AddressGenerator);

    MainTest.CreateTestCases();

    //Initialze all dock panel
    dckTest = new DckTest(MainTest);
    dckResult = new DckResult();
    dckLog = new DckLog();
    dckRelay = new DckRelay();
    dckDeviceControl = new DckDeviceControl();

    //Load panel settings and local settings
    LocalSettings.LoadPanel(mainDockPanel, GetContentFromPersistString);
```

[32] *DockPanel Suite* [online]. 2013 [cit. 2014-12-29]. Dostupné z: http://dockpanelsuite.com/

```
        dckLog.DebugEnable = LocalSettings.Settings.DebugLog;
        TestStepWrapper.LINalyzer = LocalSettings.Settings.ConnectAnalyzer;

        //events between dock panels
        dckTest.TestStepComplete += frmTest_TestStepComplete;
        dckTest.SelectedNodeChanged += dckResult.SelectedNodeChanged;
        dckTest.TestRunned += dckRelay.OtherTestRunned;
        dckTest.TestStoped += dckRelay.OtherTestStoped;
        dckTest.TestRunned += dckDeviceControl.OtherTestRunned;
        dckTest.TestStoped += dckDeviceControl.OtherTestStoped;
        dckTest.SelectedNodeChanged += dckLog.SelectedNodeChanged;

        //Call show if enabled in event checked
        tsmShowDeviceControl.Checked = LocalSettings.Settings.ShowDeviceControl;
        tsmShowLogs.Checked = LocalSettings.Settings.ShowLogs;
        tsmShowRelay.Checked = LocalSettings.Settings.ShowRelay;
        tsmShowResults.Checked = LocalSettings.Settings.ShowResults;
        tsmShowTest.Checked = LocalSettings.Settings.ShowTest;
    }
```

Tento konstruktor vytváří jedinou instanci LinTestWrapperu v celé aplikaci. Poté inicializuje přístroje pomocí adres zařízení z nastavení. Dále volá metodu CreateTestCases, která inicializuje seznam všech TestCase. Konstruktor inicializuje všechny panely, čímž se deserializuje jejich nastavení ze souboru pomocí metody LoadPanel, viz serializace GUI v podkapitole 4.3.5 Implementace serializace XML. Na konec nastaví potřebné komunikační události mezi těmito panely.

4.4.3 Panel Testů

Panel testů složí k zobrazení a ovládání všech dostupných testů a kroků testů, které mají pro danou konfiguraci smysl. V podkapitole 4.2.3 Spuštění testů je popsáno ovládání tohoto panelu, společně s popisem stavů, do kterých se test může dostat. V kódu tyto stavy reprezentuje tento výčtový typ:

```
public enum TestStepState
{
    //Ordered by display priority
    Running = 0,
    NotConfigured = 1,
    SomeError = 2,
    Error = 3,
    UnknownAndValid = 4,
```

```
    Unknown = 5,
    NotApplicable = 6,
    Completed = 7
}
```

Logika, která přiřazuje jednotlivým TestStepům stav, je popsána v metodě GetNodeState:

```
private TestStepState GetNodeState(TestStepWrapper step)
{
        if (step == CurrentRunningTest) return TestStepState.Running;
        if (!step.IsApplicable())
                return TestStepState.NotApplicable;
        if (!step.IsConfigured()) return TestStepState.NotConfigured;
        if (step.GetResult().IsValid()) { return TestStepState.Completed; }
        //Has stop time, but isnt valid means that were error.
        if (step.GetResult().GetStopTime() > DateTime.MinValue)
                return TestStepState.Error;
        return TestStepState.Unknown;
}
```

U TestCase je tato logika poněkud komplikovanější. Některé TestCase lze spustit samostatně, některé pouze obsahují TestStepy a TestCase je vykonán spuštěním těchto testů. Některé TestCase mohou být tvořeny více TestStepy, ale nejsou v nich žádné zobrazeny, jelikož vykonání těchto TestStepů není možné díky aktuální konfiguraci. K rozlišení těchto dvou případů proto byla implementována metoda IsTestStep, která vrátí true, pokud se jedná o samostatně spustitelný test. V tomto případě je jeho stav přiřazen stejně, jako by se jednalo o TestStep. Jinak je jeho stav dán stavem s nejnižším pořadovým číslem ze všech jeho podtestů. Pouze stavy SomeError a UnknowAndValid tvoří výjimku a jsou nastaveny dodatečně podle počtu stavů TestStep uzlů. Tato logika je implementována v metodě SetNodeInfo.

Samotné spouštění testů se neprovádí ve vlákně formuláře, ale pomocí background workeru, jehož příklad najdete v podkapitole 2.3.8 Background Worker. Kód metody volající testy je následující:

```
private void bgwRunTest_DoWork(object sender, DoWorkEventArgs e)
{
    TestStepWrapper.Terminate(false);
    foreach (TreeNode node in trvTest.Nodes)
    {
        foreach (TreeNode subNode in node.Nodes)
        {
            if (subNode.Checked)
            {
                CurrentRunningTest = (TestStepWrapper)subNode.Tag;
                bgwRunTest.ReportProgress(0, subNode);
                if (CurrentRunningTest.IsConfigured())
                {
                    CurrentRunningTest.Run();
                    Thread.Sleep(LocalSettings.Settings.PauseAfterTest * 1000);
                }
                CurrentRunningTest = null;
                bgwRunTest.ReportProgress(100, subNode);
                if (bgwRunTest.CancellationPending) return;
            }
        }
        if (node.Checked){
            CurrentRunningTest = (TestStepWrapper)node.Tag;
            bgwRunTest.ReportProgress(0, node);
            if(((TestCaseWrapper)node.Tag).IsRunableTestCase())
            {
                if (CurrentRunningTest.IsConfigured())
                {
                    CurrentRunningTest.Run();
                    Thread.Sleep(LocalSettings.Settings.PauseAfterTest * 1000);
                }
            }
            CurrentRunningTest = null;
            bgwRunTest.ReportProgress(100, node);
            if (bgwRunTest.CancellationPending) return;
        }
    }
}
```

Background worker v daném příkladu běží v cyklu, prohledává strom TestCase a TestStep, vše zaškrtnuté zkontroluje, a pokud jsou správně nakonfigurované, spustí metodu Run. Hlavní vlákno je informováno o spuštění a dokončení testu metodou ReportProgress. Ta je v hlavním vlákně vyvolána metodou

bgwRunTest_ProgressChanged, která odznačí provedený test a zajistí aktualizaci stavu TestStepu. Ukazatel na uzel asociovaný s TestCase je poslán parametrem UserState. Parametr ProgressPercentage nabývá hodnoty 0, pokud se test teprve spouští, a hodnoty 100, pokud byl běh testu dokončen. Zdrojový kód této metody je následující:

```
private void bgwRunTest_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
        // 100 means Run is complete
        if (e.ProgressPercentage == 100)
        {
                //Unregister event to prevent cancelation while test is running
                trvTest.BeforeCheck -= trvTest_BeforeCheck;
                ((TreeNode)e.UserState).Checked = false;
                trvTest.BeforeCheck += trvTest_BeforeCheck;
                OnTestStepComplete();
        }
        SetNodeInfo((TreeNode)e.UserState);
}
```

4.5 Ladění ve smíšeném módu

Dneska by si už asi žádný programátor nedokázal představit vývoj programu bez použití nějakého ladícího nástroje. Tyto nástroje jsou zpravidla integrovány přímo do vývojového prostředí. Pokud se ale program skládá z částí, které nejsou ve stejném programovacím jazyku a nepoužívají stejné běhové prostředí, může se proces ladění zkomplikovat. Naštěstí ale vývojový nástroj Visual Studio umožňuje ladění kódů ve smíšeném módu (mixed mode), které umožňuje při ladění plynule přecházet z neřízených částí programu do řízených.

Jak již víme z podkapitoly 4.3.1 Rozvržení zdrojového kódu, řešení (solution) aplikace obsahuje dva projekty. LINLib, který je psan v jazyce C++ a jeho větší část využívá neřízený kód, a LINTest, který je v jazyce C# a jedná se kompletně o řízený kód. Pro využití smíšeného módu ladění je potřeba nastavit následující:

- Na záložce vlastnosti projektu LINTest, na kartě „Debug“ zaškrtnout „Enable native code debugging“
- Na záložce vlastností projektu LINTest, na kartě „Build“ zaškrtnout „Allow unsafe code“

- V hlavní nabídce „Tools“ → „Option“, na záložce „Debug“ → „General“ zaškrtnout volbu „Use Managed compatibility Mode“
- Na záložce vlastností projektu LINLib, na kartě „Configuration Properties“ → „Debugging“, vybrat u položky „Debuger Type“ volbu „Mixed“.

5 Výsledky testů

Testování grafického rozhraní se bohužel špatně kvantifikuje. Většina interakcí s GUI je provedena v řádech mikrosekund, které lidské oko sotva postřehne, a nemá proto moc smysl je měřit. I parsování větších LDF souborů probíhá s lineární složitostí, v řádu sekund není proto třeba ho měřit. Omezíme se proto pouze na testování funkční, tj. program dělá to co má, podle požadavků na něj kladených, testování integrační, kdy program pracuje správně se zařízeními k tomu potřebnými, a zákaznické, kde je popsáno první kolo zákaznického testování, které již proběhlo.

5.1 Funkční testy

Aplikace byla v průběhu implementace opakovaně testována přímo na vstupních LDF souborech společnosti Škoda Auto a. s. Tyto soubory představují know-how Škoda Auto a proto je není možné k práci přiložit. Testování probíhalo následovně:

V první fázi byl proveden import LDF souboru a poté byla manuálně ověřena správnost všech importovaných položek v příslušných datových strukturách. Následně byly importovány různé LDF soubory a bylo vyhodnocováno, zda automaticky generované seznamy testů odpovídají konfiguraci specifikované LDF souborem. Současně bylo kontrolováno, zda jsou testy správně nakonfigurovány, zda tedy byly správně importovány také parametry nezbytné pro spuštění testů.

5.2 Integrační testy

Aplikace byla opakovaně testována na testovacím pracovišti zřízeném na katedře měření ČVUT v místnosti číslo B3-458. Pro testování byly použity přístroje tak, jak jsou uvedeny v podkapitole 3.1 Sestava testovacího pracoviště.

Testovanou jednotkou byla jednotka pro „ovládání rádia na volantu“. Na této jednotce byly opakovaně provedeny kompletní testy, byly testovány interakce GUI s uživatelem v průběhu automatického testování, byly testovány reakce GUI na chybové stavy testovací knihovny. Byla také testována možnost interaktivního manuálního ovládání připojených přístrojů a vnější propojovací jednotky prostřednictvím k tomu určených ovládacích panelů. Na testování se podílelo několik osob, aby byl eliminován vliv uživatelských stereotypů.

5.3 Zákaznické testování

Aplikace byla 8. 12. 2014 předána zákazníkovi k prvnímu zákaznickému testování, verze pouze pro testování jednotek slave. To znamená, že zatím ještě nepodporovala možnost načíst a upravovat data ze souboru CDB a ani nebyl dodán žádný TestCase, který by testoval komunikaci chování uzlu typu master pomocí sběrnice CAN. Uživatelé ze společnosti Škoda Auto měli k aplikaci několik následujících připomínek, resp. požadavků na změnu:

- V okně „Nastavení LDF“ se jako "Výchozí LDF soubor" ukazuje upravené DLF. Zobrazujte správný název výchozího souboru LDF.
- Pokud je testována slave jednotka, aplikace se chová jako master. Umožněte proto vybírání časovací tabulky (schedule table) v těchto případech uživatelem.
- V okně „Nastavení LDF“ je potřeba propagovat změnu vlastnosti signálu Čítač do vlastnosti čítač její nadřazené zprávy.
- Přidejte možnost zadání "výchozí hodnoty signálu" uživatelem.
- V okně „Nastavení LDF“ umožněte editovat položky „P2 min“ a „ST min“.
- Volby „Aktivace jednotky“ jsou relevantní pouze pro master jednotku. Odstraňte tyto volby pro slave jednotky.
- Pokud se provádí TestStep, je u něj zobrazena ikona, že právě běží. Zobrazte tuto ikonu i pro nadřazený TestCase.
- Po vybrání TestCase, který je tvořen více TestStepy, zobrazujte v okně výsledků testů graficky oddělené výsledky všech těchto TestStepů.

Všechny tyto změny jsou již zapracovány do aplikace. Dále byla po tomto testování aplikace rozšířena i o možnost načítání souborů DBC (při vytváření nového projektu) a CDB (při načítání/uložení projektu), tj. o podporu rozhraní CAN pro uzly typu Master.

6 Závěr

Tato diplomové práce se vyznačuje především faktem, že produkt této práce, program pro testování řídicích jednotek, se bude využívat v komerční sféře. Musí proto bezproblémově fungovat tak, jak si přeje zákazník. Z reakce uživatele, firmy Škoda Auto a. s., je po prvním testování zřejmé, že program splnil jejich očekávání a kromě několika drobných požadavků k němu neměli žádné výhrady. Grafické zpracování programu oživuje jeho vzhled, ale nijak nevybočuje z původního konceptu. Program je v přiložené podobě možné předat zákazníkovi k užívání v běžném provozu.

Ačkoli bylo třeba vyřešit problém interoperability jazyka C# a C++, volba jazyka C# pro tvorbu grafického rozhraní s sebou díky platformě .Net přinesla i vítaný bonus, možnost rozšiřovat aplikaci i za použití jiných programovacích jazyků. Integrace původního kódu, testeru CAN, se vydařila bez větších problémů.

Do budoucna se v programu počítá s tím, že se budou přidávat další a další testovací scénáře. Vhodným rozvrstvení aplikace a strukturou tříd je ale možné toto rozšiřování provádět jako doposud v jazyce C++, bez změny uživatelského rozhraní a tedy i bez znalosti příslušného kódu C#. Bohužel toto rozvržení aplikace znemožnilo využití serializace XML souboru pomocí jazyka C# a musely k tomu být využity jiné komplikovanější postupy s nasazením knihoven třetí strany. Na druhou stranu se návrh a úprava formulářů díky intuitivnímu ovládání vývojového nástroje Visual Studio zjednodušila.

7 Zdroje

1. NAGEL, CH. et al.*C# 2008 Programujeme profesionálně*. Brno: C Press, 2009. ISBN: 978-80-251-2401-7.
2. RICHTER J. .*NET Framework programování aplikací.* Praha: Grada publishing a. s., 2003.ISBN: 80-247-0450-1
3. NOVÁK J. *CAN Test příručka programátora v. 2. 1*.2014. Zdroj: osobní archiv Jiřího Nováka.
4. LIN Consortium: LIN Specification Package Revision 2.1 November 24, 2006
5. Using the DllImport Attribute. MICROSOFT. *Microsoft Development Network* [online]. 2014 [cit. 2014-12-28]. Dostupné z: http://msdn.microsoft.com/en-us/library/aa984739%28v=vs.71%29.aspx
6. Name mangling. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2014-12-28]. Dostupné z: http://en.wikipedia.org/wiki/Name_mangling
7. Passing Structures MICROSOFT. *Microsoft Development Network* [online]. 2014 [cit. 2014-12-28]. Dostupné z: http://msdn.microsoft.com/en-us/library/awbckfbz.aspx
8. SUTORÝ, Tomáš. LIN (Local Interconnect Network). Elektrorevue [online]. 2004, roč. 2004, č. 12 [cit. 2014-12-19]. Dostupné z: http://www.elektrorevue.cz/clanky/04012/index.html
9. Is it possible to run unmanaged C++ normally from a managed C++/CLI project?. In: *Stackoverflow* [online]. 2013 [cit. 2014-12-29]. Dostupné z: http://stackoverflow.com/questions/15598445/is-it-possible-to-run-unmanaged-c-normally-from-a-managed-c-cli-project
10. MAZARINE, Cristinel. *Visual Studio IDE like Dock Container - Second Version* [online]. 2009 [cit. 2014-12-29]. Dostupné z: http://www.codeproject.com/Articles/42800/Visual-Studio-IDE-like-Dock-Container-Second-Versi
11. *DockPanel Suite* [online]. 2013 [cit. 2014-12-29]. Dostupné z: http://dockpanelsuite.com/
12. *Boost c++ libraries* [online]. 2014 [cit. 2015-01-04]. Dostupné z: http://www.boost.org/

13. C++/CLI Wrapping a Function that Returns a std::shared_ptr. In: *C++/CLI Wrapping a Function that Returns a std::shared_ptr* [online]. 2011 [cit. 2014-12-29]. Dostupné z: http://stackoverflow.com/questions/6172361/c-cli-wrapping-a-function-that-returns-a-stdshared-ptr
14. Bachaalany, Advantages and disadvantages of WPF over winforms. *Code project* [online]. 2014 [cit. 2015-01-01]. Dostupné z: http://www.codeproject.com/Questions/66468/Advantages-and-disadvantages-of-WPF-over-winforms

8 Přílohy

8.1 Ukázka definice souboru LDF[33]

```
LIN_description_file;
LIN_protocol_version = “2.1”;
LIN_language_version = “2.1”;
LIN_speed = 19.2 kbps;
Channel_name = “DB”;

Nodes {
	Master: CEM, 5 ms, 0.1 ms;
	Slaves: LSM, RSM;
}

Signals {
	InternalLightsRequest: 2, 0, CEM, LSM, RSM;
	RightIntLightsSwitch: 8, 0, RSM, CEM;
	LeftIntLightsSwitch: 8, 0, LSM, CEM;
	LSMerror, 1, 0, LSM, CEM;
	RSMerror, 1, 0, LSM, CEM;
	IntTest, 2, 0, LSM, CEM;
}
Frames {
	CEM_Frm1: 0x01, CEM, 1 {
		InternalLightsRequest, 0;
	}
	LSM_Frm1: 0x02, LSM, 2 {
		LeftIntLightsSwitch, 0;
	}
	LSM_Frm2: 0x03, LSM, 1 {
		LSMerror, 0; IntError, 1;
	}
	RSM_Frm1: 0x04, RSM, 2 {
		RightIntLightsSwitch, 0;
	}
	RSM_Frm2: 0x05, RSM, 1 {
		RSMerror, 0;
	}
}
Event_triggered_frames {
Node_Status_Event : Collision_resolver, 0x06, RSM_Frm1, LSM_Frm1;
```

[33] LIN Consortium: LIN Specification Package Revision 2.1

```
}
Node_attributes {
	LSM {
		LIN_protocol = “2.0”;
		configured_NAD = 0x20;
		initial_NAD = 0x01;
		product_id = 0x4A4F, 0x4841;
		response_error = LSMerror;
		fault_state_signals = IntTest;
		P2_min = 150 ms;
		ST_min = 50 ms;
		configurable_frames {
			CEM_Frm1 = 0x0001; LSM_Frm1 = 0x0002; LSM_Frm2 = 0x0003;
		}
	}
	RSM {
		LIN_protocol = “2.0”;
		configured_NAD = 0x20;
		product_id = 0x4E4E, 0x4553,
		response_error = RSMerror;
		P2_min = 150 ms;
		ST_min = 50 ms;
		configurable_frames {
			CEM_Frm1 = 0x0001; LSM_Frm1 = 0x0002; LSM_Frm2 = 0x0003;
		}
	}
}
Schedule_tables {
	Configuration_Schedule {
		CEM_Frm1 delay 15 ms;
		LSM_Frm2 delay 15 ms;
		RSM_Frm2 delay 15 ms;
	}
	Normal_Schedule {
		CEM_Frm1 delay 15 ms;
		LSM_Frm2 delay 15 ms;
		RSM_Frm2 delay 15 ms;
		Node_Status_Event delay 10 ms;
	}
	MRF_schedule {
		CEM_Frm1 delay 15 ms;
	}
	SRF_schedule {
		CEM_Frm1 delay 15 ms;
	}
```

```
        Collision_resolver {
                CEM_Frm1 delay 15 ms;
                LSM_Frm2 delay 15 ms;
                RSM_Frm2 delay 15 ms;
                RSM_Frm1 delay 10 ms;
                LSM_Frm2 delay 15 ms;
                RSM_Frm2 delay 15 ms;
                LSM_Frm1 delay 10 ms;
        }
}
Signal_encoding_types {
        Dig2Bit {
                logical_value, 0, "off";
                logical_value, 1, "on";
                logical_value, 2, "error";
                logical_value, 3, "void";
        }
        ErrorEncoding {
                logical_value, 0, “OK”;
                logical_value, 1, “error”;
        }
        FaultStateEncoding {
                logical_value, 0, “No test result”;
                logical_value, 1, “failed”;
                logical_value, 2, “passed”; logical_value, 3, “not used”;
        }
        LightEncoding {
                logical_value, 0, “Off”;
                physcial_value, 1, 254, 1, 100, “lux”;
                logical_value, 255, “error”;
        }
}
Signal_representation {
        Dig2Bit: InternalLightsRequest;
        ErrorEncoding: RSMerror, LSMerror;
        FaultStateEncoding: IntTest;
        LightEncoding: RightIntLightsSwitch, LeftIntLightsSwitch;
}
```