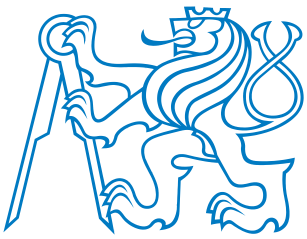
diplomová práce

Framework DLMS/COSEM pro sběr dat v AMM systémech

Bc. Matěj Jirka



Květen 2016

doc. Ing. Jiří Novák Ph.D.

České vysoké učení technické v Praze
Fakulta elektrotechnická, Katedra řídící techniky

Poděkování

Současné poděkování patří Ing. Zdeňkovi Štěpkovi, který mi umožnil vytvořit tuto práci a v jejím průběhu mi věnoval svůj drahocenný čas a poskytoval velmi přínosné rady. Automaticky patří mé poděkování také doc. Ing. Jiřímu Novákovi Ph.D. Tímto bych chtěl současně poděkovat své rodině za podporu a trpělivost. Nakonec poděkování patří i firmě TECHSYS HW a SW, a.s. za poskytnutí tohoto tématu diplomové práce.

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: .....

.....
Podpis

Abstrakt

Tato práce se zabývá návrhem a implementací průmyslového komunikačního protokolu (DLMS/COSEM) definovaného souborem norem ČSN EN 62056. Další potřebné standardy a normy, které protokol využívá, a jejich návrhy a implementace jsou v této práci úplně nebo alespoň z části popsány. Výsledkem práce je framework v jazyce C++, který je přenositelný na úrovni zdrojového kódu mezi operačními systémy (Windows, Linux a RTX) s přihlédnutím na omezené kapacity operační paměti na vestavných zařízeních. Práce zahrnuje potřebnou část pro sběr dat ze zařízení v AMM sítích.

Jako vedlejší produkt práce vznikla knihovna implementující profil protokolu HDLC, který úzce souvisí s protokolem DLMS, a jehož funkcí DLMS protokol využívá.

V rámci frameworku DLMS/COSEM vznikla také skupina tříd a funkcí implementující potřebnou část standardu ASN.1 (BER a AXDR), který je použit pro popis syntaxe protokolu DLMS. Tento soubor tříd je možné dále rozvíjet a použít v dalších implementacích protokolů, které využívají ASN.1 pro zápis vlastní syntaxe.

Klíčová slova

DLMS; COSEM; AMM; HDLC; OBIS; ARM

Abstrakt

This thesis deals with design and implementation of industrial communication protocol (DLMS/COSEM) which is defined in set of norms ČSN EN 62056. Next necessary standards and norms used by this protocol and their design and implementations are completely or partially described. Result of this work is C++ library which is transferrable on source code level between operating systems (Windows, Linux abd RTX) and includes necessary part to collect data from devices in AMM networks.

As side product was created a library which implements profile of HDLC protocol which interconnects with protocol DLMS and which is used by functions of DLMS protocol.

Within framework DLMS/COSEM was created group of classes which implements necessary part of ASN.1 standard which is used to describe syntax of DLMS protocol. It is possible to extend and develop this library to use it in another protocol implementations which uses ASN.1 to describe own syntax.

Keywords

DLMS; COSEM; AMM; HDLC; OBIS; ARM

Obsah

1 Úvod **1**

2 Cíle práce **2**

3 Logický model komunikačních protokolů **3**
3.1 OSI/ISO Model . . . 3
3.2 Protokol HDLC . . . 3
3.2.1 Služby LLC vrstvy . . . 4
Linkové spojení . . . 4
Linkové rozpojení . . . 5
Data . . . 5
3.2.2 Funkce LLC vrstvy . . . 6
Struktura LPDU . . . 6
3.2.3 Služby MAC vrstvy . . . 7
MAC spojení . . . 7
MAC rozpojení . . . 7
MAC data . . . 8
3.2.4 Funkce MAC vrstvy . . . 8
3.2.5 Struktura MAC PDU . . . 9
Frame format . . . 9
Zdrojová a cílová adresa . . . 10
Kontrolní součty HCS a FCS . . . 10
Pole *Control* . . . 10
3.3 Protokol DLMS/COSEM . . . 11
3.3.1 Aplikační vrstva DLMS/COSEM . . . 11
3.3.2 ASN.1 jako notace použitá pro kódování posílaných dat . . . 13
3.3.3 Specifikace použitých služeb . . . 13
AARQ . . . 14
AARE . . . 15
RLRQ a RLRE . . . 16
GET . . . 16
SET . . . 17
ACTION . . . 17
Read . . . 18
Write . . . 18
UnconfirmedWrite . . . 18
3.3.4 OBIS . . . 18
3.3.5 COSEM třídy rozhraní . . . 19
3.4 Druhy COSEM tříd rozhraní . . . 20
3.4.1 Parametry a data měření . . . 20
Data . . . 20
Register . . . 20
Extended register . . . 20
Demand register . . . 20
Register activation . . . 21
Profile generic . . . 21

3.4.2 Řízení přístupu . . . 21
Association SN . . . 21
Association LN . . . 21
3.4.3 Čas a řízení hranic událostí . . . 22
Clock . . . 22
Script table . . . 22
Schedule a Special days table . . . 23
Register monitor . . . 23
Disconnect control . . . 23
Limiter . . . 23

4 Analýza a návrh řešení 24
4.1 Analýza problémů a možností . . . 24
4.1.1 Alternativní řešení - knihovna Gurux.DLMS.cpp . . . 24
4.1.2 Paměť na vestavěných systémech . . . 25
Klady a zápory dynamické alokace paměti . . . 26
Vyhodnocení dynamické alokace a dealokace . . . 26
4.1.3 Generování kódu z ASN.1 notace . . . 27
4.2 Návrh HDLC . . . 27
4.2.1 Logický model knihovny . . . 27
4.2.2 Komunikace s nižšími vrstvami . . . 28
4.2.3 Uživatelský přístup ke knihovně . . . 28
4.2.4 Vnitřní struktura knihovny . . . 29
4.2.5 Chování knihovny při 30
Příjímání dat . . . 30
Připojení . . . 31
Odpojení . . . 31
Přenos informačních paketů . . . 32
4.3 Návrh DLMS/COSEM . . . 33
4.3.1 Uživatelský přístup k frameworku . . . 34
4.3.2 Komunikace s nižšími vrstvami . . . 35
4.3.3 Logický model frameworku . . . 35
4.3.4 Vnitřní struktura frameworku . . . 35
4.3.5 Návrh použití frameworku . . . 36
Asociace . . . 36
Získání asociačního objektu a spárování jmen . . . 36
Čtení . . . 36
Zápis . . . 36

5 Implementace 39
5.1 HDLC . . . 39
5.2 DLMS/COSEM . . . 42
5.2.1 DLMS . . . 43
Připojení . . . 43
Získání asociačního objektu . . . 44
Čtení . . . 45
5.2.2 COSEM . . . 45
5.2.3 ASN.1 . . . 45

6 Testy **47**
6.1 HDLC . . . 47
6.1.1 Testování proti vlastnímu SW . . . 47
6.1.2 Testování s reálným zařízením . . . 51
6.1.3 Paměťové nároky . . . 52
6.2 DLMS . . . 52
6.2.1 Kompilace . . . 52
6.2.2 Běh aplikace . . . 53
6.2.3 Paměťové nároky . . . 53
Velikost binárního programu . . . 53
Velikost dynamicky alokované paměti . . . 54
Zátěž Stacku . . . 54

7 Vyhodnocení a Závěr **56**

Literatura **57**

A Obsah přiloženého CD **58**

Seznam obrázků

1 Referenční model ISO/OSI . . . 3
2 LLC - Linkové spojení . . . 4
3 LLC - Linkové rozpojení . . . 5
4 LLC - Linková data . . . 6
5 Struktura LPDU . . . 6
6 MAC spojení . . . 7
7 MAC rozpojení . . . 8
8 MAC data . . . 9
9 MAC Packet . . . 9
10 Frame format . . . 9
11 MAC adresa . . . 10
12 MAC kontrolní součty . . . 10
13 MAC kontrolní pole . . . 10
14 Hierarchie vrstev DLMS/COSEM . . . 12
15 Servisní primitivy DLMS/COSEM . . . 13
16 Ilustrace významu tříd a objektů COSEM . . . 19

17 Fragmentace dynamické paměti . . . 26
18 Vnitřní logická struktura HDLC . . . 28
19 Stavový automat chování knihovny . . . 30
20 HDLC Průběh připojení . . . 31
21 HDLC Průběh odpojení . . . 32
22 HDLC - Odesílání informačních paketů . . . 33
23 Návrh kooperace částí DLMS/COSEM a HDLC . . . 34
24 DLMS/COSEM - Průběh při zavolání asociace . . . 37
25 DLMS/COSEM - Průběh při získávání asociačního objektu . . . 38

26 Základní třídy HDLC knihovny . . . 40
27 Podpůrná hierarchie HDLC frameworku . . . 41
28 Třídy rozhraní COSEM . . . 46

Seznam tabulek

1 HDLC – paměťové nároky 52
2 Velikost binárního programu v paměti 54
3 Velikost alokované dynamické paměti 54
4 Statická analýza zátěže stacku 55

Zkratky

Zkratky použité v textu diplomové práce.

AA	Application Association
AARQ	A-Associate Request
AARE	A-Associate Response
ACSE	Association Control Service Element
APDU	Aplication Protocol Data Unit
ASE	Application Service Element
A-XDR	Advanced eXternal Data Representation
b	bit
B	Byte
BE	Big Endian
BER	Basic Encoding Rules
COM	COMmunication port
COSEM	COmpanion Specification for Energy Metering
DISC	Disconnect
DL	Data Link
DLMS	Device Language Message Specification
DM	Disconnected Mode
DNP	Distributed Network Protocol
DSW	Dynamic Stack Window
DM	Disconnected Mode
FCS	inFormation Check Sum
FIFO	First In First Out
FW	Firmware
GPLv2	General Public License version 2
HCS	Header Check Sum
HDLC	High-Level Data Link Control
HW	Hardware
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
LE	Little Endian
LIFO	Last In First Out
LLC	Logic Link Control
LN	Long Name
LPDU	LLC Protocol Data Unit
LSAP	Link Service Access Point
MAC	Medium Access Control (Sub-layer)
MITM	Man In The Middle
MSDU	MAC Service Data Unit
NRM	Normal Response Mode
OBIS	OBject Identification System
OS	Operační systém
PDU	Protocola Data Unit
PLC	Programmable Logic Controller
RAA	Release Application Association
RRR	expected Received number of information
S-FSK	Spread Frequency Shift Keying

SN	Short Name
SNRM	Set Normal Response Message
SSS	Send Sequence number of information
STL	Standard Template Library
SW	Software
UA	Unnumbered Acknowledgement
UNC	Unbalanced operation Normal response mode Class
xDLMS	extended DLMS

1 Úvod

Komunikační protokoly jsou nezbytnou součástí počítačové komunikace. Bez nich bychom jen těžko získávali data z jiných zařízení. Komunikace by jinak byla nepřesná, ztrátová a zajisté také neúčinná a redundantní. Proto vznikají komunikační protokoly, které tento tok dat usměrňují a mají snahu ho maximálně zefektivnit. Toto řešení se snaží brát ohled i na způsob využití protokolu.

Existuje mnoho protokolů, které vzhledem ke svému zařazení do komunikačního modelu, plní jak funkce „univerzální“ – zajišťují přenos dat bez znalosti jejich obsahu, tak mohou zastávat i funkce „vyšší“, které zajišťují přenos dat specifických pro cílové nebo zdrojové zařízení. Takovými protokoly jsou například průmyslové protokoly, které se zaměřují na sběr a zasílání naměřených hodnot nebo nastavení, ovládání cílového zařízení a případnou další správu různorodých průmyslových zařízení. Jako příklad mohu uvést protokoly Modbus, DNP 3.0 nebo ISO IEC 61850. Jedním z nich je také protokol DLMS, který využívá systém tříd rozhraní COSEM. Tento protokol byl vytvořen pro pokročilá měření a kontrolní funkce pro různé druhy energie tak, aby vyhověl všem požadavkům trhu a byl schopný komunikovat skrz veškerá komunikační média[1].

Obsah této práce se zabývá implementací protokolu DLMS/COSEM a také jeho podpůrné vrstvy HDLC. Zadavatelem této práce je společnost TECHSYS – HW a SW, a.s., která se zabývá dodávkami monitorovacích, řídicích a informačních systémů pro energetiku, průmysl a dopravu [2]. Na základě průzkumu trhu a požadavků potenciálního zákazníka byl tento protokol vybrán pro implementaci do stávajících zařízení firmy TECHSYS. Cílem tohoto rozhodnutí je rozšířit stávající portfolio nabízených služeb a tím pokrýt větší část trhu.

V kapitole 2 jsou specifikovány hlavní cíle této práce. Ve stejné kapitole jsou také určena omezení, s kterými se musí řešení vypořádat. Kapitola 3 slouží jako úvod do protokolů DLMS/COSEM a HDLC, kterými se práce zabývá. Následně v kapitole 4 jsou analyzovány problémy spojené s implementací protokolů. Tato kapitola se také zabývá návrhem výsledného řešení. Vytvořené řešení je popsáno v kapitole 5. Postup testování obou implementací je popsán v kapitole 6, vyhodnocení a závěr je uveden v kapitole 7.

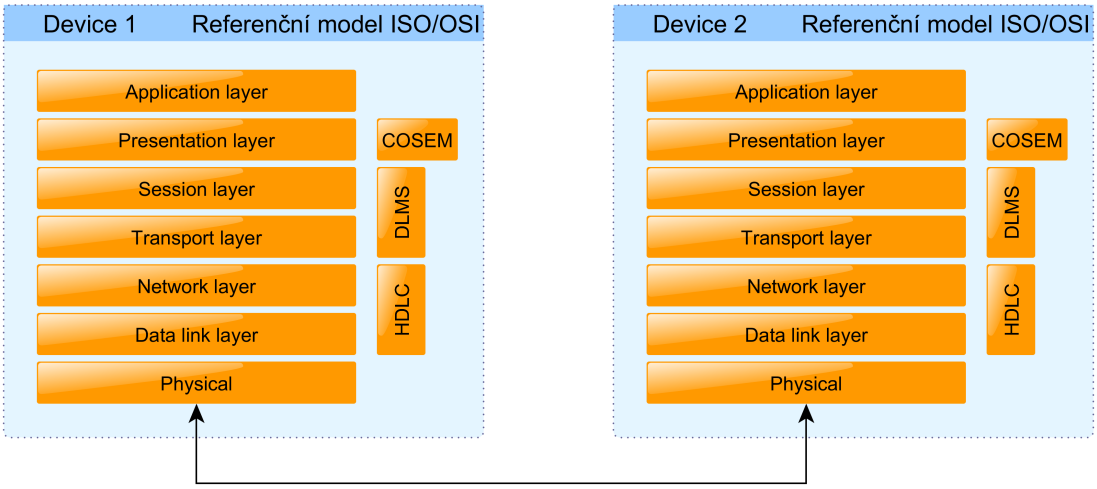
2 Cíle práce

Cílem této práce je seznámit se s protokolem DLMS a objektovým modelem COSEM. S využitím aktuálního hardwarového a softwarového vybavení firmy TECHSYS navrhnout a implementovat framework, který bude přenositelný mezi platformami na úrovni zdrojového kódu. Cílem je, aby výsledné řešení podporovalo komunikaci po sériovém rozhraní a rozhraní Ethernet. Implementace celého objektového modelu COSEM je nad rámec této práce, proto je cílem vybrat jen nejdůležitější podmnožinu objektů a funkcí, které budou využitelné v praxi a budou demonstrovat možnosti celého frameworku. Výsledná implementace musí být připravena na rozšíření o další objekty a funkce modelu COSEM. Koncové řešení musí být použitelné v softwarovém vybavení firmy TECHSYS, tudíž je cílem framework v jazyce C++. S cílem přenositelnosti výsledné implementace i na real-time operační systémy (hlavně RTX) je nutné dbát na omezené kapacity operační paměti.

Dalším cílem této práce je vytvořit podpůrný protokol HDLC. Tento protokol bude nižší vrstvou vzhledem k protokolu DLMS/COSEM. Výsledné řešení musím být použitelné v rámci softwaru a hardwaru firmy TECHSYS. Cílem je dbát na omezené kapacity operační paměti dostupných na stávajícím hardwarovém vybavení.

Hardwarovým vybavením je především myšleno vestavěné zařízení Storm-01, které běží na procesoru STM32F40x s frekvencí 168Mhz. Storm-01 má k dispozici 2MB externí SRAM paměti a 1MB flash paměti, která je ale již strukturována pro další funkce zařízení.

3 Logický model komunikačních protokolů



Obr. 1 – Referenční model ISO/OSI

3.1 OSI/ISO Model

Referenční model OSI/ISO je abstraktní popis komunikace mezi zařízeními z hlediska přenosu dat. Skládá se ze sedmi vrstev, kde každá vrstva plní svoji funkci v rámci dialogu mezi zařízeními.

Vztah protokolů DLMS, HDLC a objektového modelu COSEM vzhledem k referenčnímu modelu ISO/OSI je ilustrován na obrázku 1. Protokol HDLC má na starosti jak uspořádávání a řazení rámců, tak směrování v síti, proto náleží vrstvám „Data Link“ layer a „Network layer“. Protokol DLMS se oproti tomu zabývá dialogem mezi oběma zařízeními a výměně dat mezi nimi. Náleží tím pádem vrstvě „Session layer“ a „Transport layer“. Objektový model COSEM je zodpovědný za formátování přenášených dat a náleží proto vrstvě „Presentation layer“.

3.2 Protokol HDLC

HDLC je komunikační protokol pro sériovou komunikaci odpovídající linkové vrstvě v OSI/ISO referenčním síťovém modelu. HDLC je definován v mezinárodním standardu ISO/IEC 13239, který nahrazuje předchozí standardy ISO/IEC 3309:1993, ISO/IEC 4335:1993, ISO/IEC 7809:1993 and ISO/IEC 8885:1993 [3]. Tato práce popisuje jen nejdůležitější části a zabývá se pouze spojově orientovaným asynchronním komunikačním profilem, který je založený na HDLC a který je uzpůsobený pro měření elektrické energie - výměnu dat pro odečet elektroměru, řízení tarifu a regulaci zátěže.

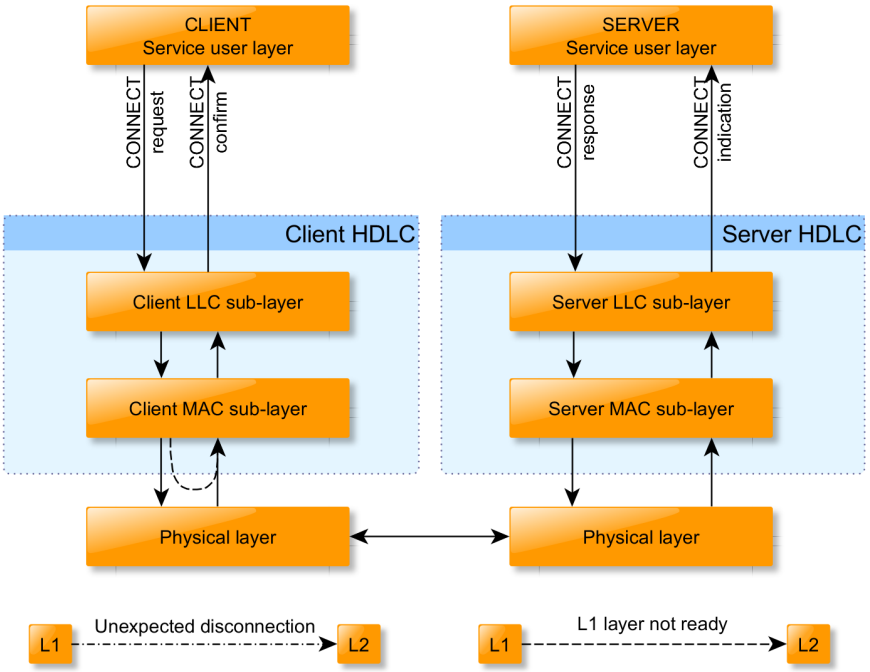
Protokol HDLC obsahuje dvě podvrstvy MAC a LLC. Vrstva MAC zastává v tomto

profilu hlavní roli, zatímco vrstva LLC zde slouží jen jako transparentní zprostředkovatel služeb linkového spojení, rozpojení a zasílání dat pro vyšší vrstvy síťového modelu využívající služeb vrstvy MAC.

Výměna dat v rámcích HDLC probíhá až po navázání spojení, které vyvolává klient připojující se k serveru. Po navázání spojení a výměně komunikačních parametrů spojení, může začít probíhat přenos dat. Jakmile klient vyvolá požadavek na rozpojení spojení a server mu odpoví, je spojení ukončeno. V případě rozpadu spojení, které je indikováno fyzickou vrstvou, je spojení ukončeno také.

3.2.1 Služby LLC vrstvy

Služby LLC vrstvy jsou zajišteny pomocí tzv. „service primitives“. Sekvenci těchto „service primitiv“ iniciuje vždy klient a v případě rozpadu spojení nižší vrstva MAC, která informaci o rozpadu přijala od fyzické vrstvy. Tyto servisní primitivy bývají zpravidla čtyři a ve spojení probíhají v pořadí *request*, *indication*, *response* a *confirm*.



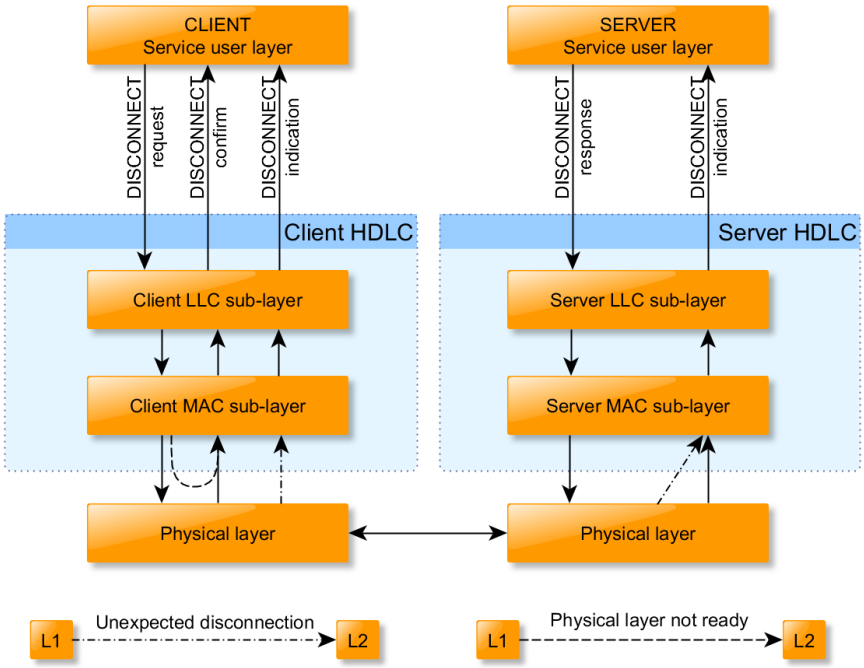
Obr. 2 – LLC - Linkové spojení

Linkové spojení

K navázání linkového spojení používá vrstva LLC čtyř tzv. „service primitives“ (viz obrázek 2):

- **CONNECT request**
  Klientská „service primitive“, kterou vyvolává klientská servisní vrstva.
- **CONNECT indication**
  Serverová „service primitive“, kterou oznamuje vrstva LLC serverové servisní vrstvě, že byl přijat požadavek na spojení.

- **CONNECT response**
  Serverová „service primitive“, pomocí které odpovídá serverová servisní vrstva na požadavek na spojení.
- **CONNECT confirm**
  Klientská „service primitive“, kterou odpovídá vrstva LLC klientské servisní vrstvě na požadavek na spojení.



Obr. 3 – LLC - Linkové rozpojení

Linkové rozpojení

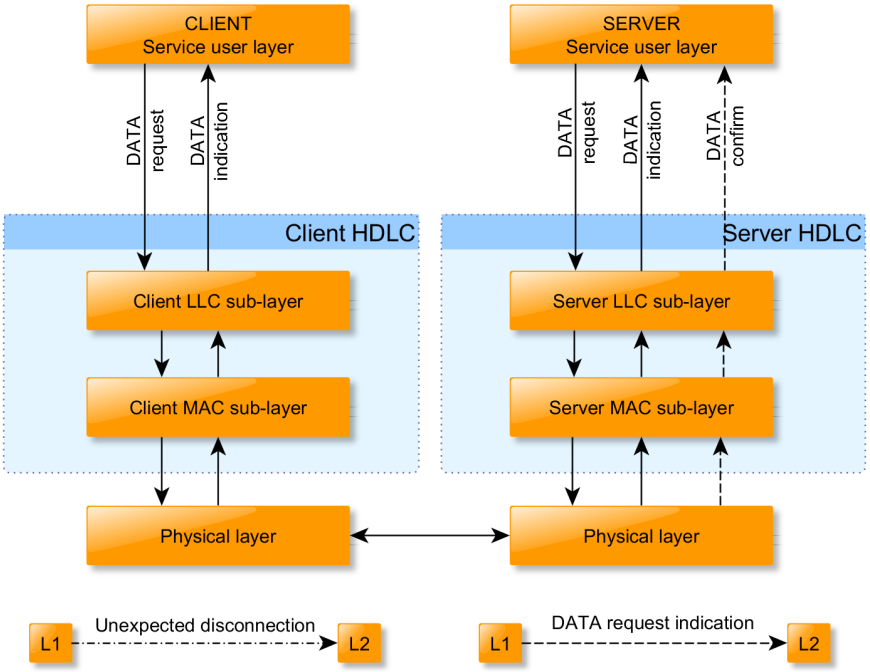
Viz obrázek 3.

- **DISCONNECT request**
  Klientská „service primitive“, kterou vyvolává klientská servisní vrstva.
- **DISCONNECT indication**
  Serverová i klientská „service primitive“, kterou LLC vrstva oznamuje serverové servisní vrstvě, že byl přijat požadavek na rozpojení (server), nebo že se spojení rozpadlo.
- **DISCONNECT response**
  Serverová „service primitive“, pomocí které odpovídá serverová servisní vrstva na požadavek na rozpojení.
- **DISCONNECT confirm**
  Klientská „service primitive“, kterou odpovídá vrstva LLC klientské servisní vrstvě na požadavek na rozpojení.

Data

Viz obrázek 4.

- **DATA request**
  Klientská „service primitive“, kterou vyvolává klientská servisní vrstva.

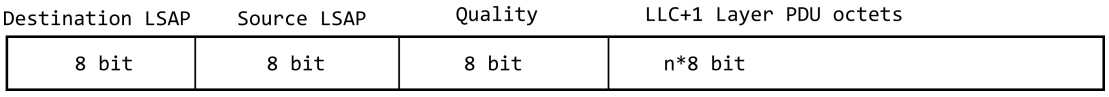


Obr. 4 – LLC - Linková data

- **DATA indication**
  Serverová i klientská „service primitive“, kterou předává vrstva LLC servisní vrstvě přijatá data.
- **DATA confirm**
  Serverová „service primitive“, kterou oznamuje LLC vrstva serverové servisní vrstvě o přijetí „DATA request“.

3.2.2 Funkce LLC vrstvy

Pro „service primitives“ *CONNECT* a *DISCONNECT* zprostředkovává LLC vrstva pouze předání nižší příp. vyšší vrstvě, ale při „service primitive“ *DATA* připojuje k datům vlastní hlavičku. Protilehlá strana pak tuto hlavičku kontroluje a při předání vyšší vrstvě ze zprávy odstraňuje.



Obr. 5 – Struktura LPDU

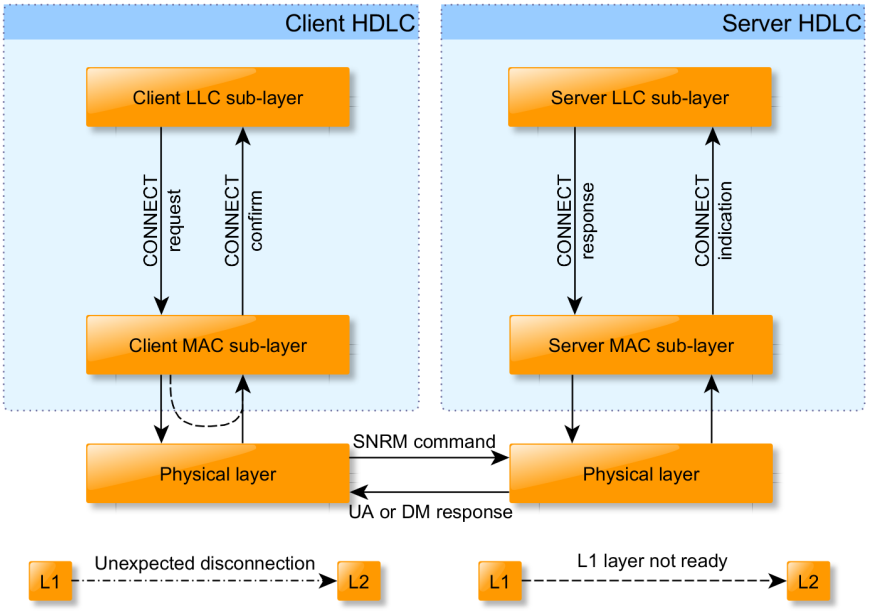
Struktura LPDU

Pro potřeby DLMS/COSEM jsou hodnoty hlavičky dány následně (viz obr. 5):

- Destination LSAP: **0xE6**
- Source LSAP: **0xE6** jako „command“ nebo **0xE7** „response“
- Quality: **0x00**

3.2.3 Služby MAC vrstvy

I přes to, že komplexnost MAC vrstvy je větší než u LLC, poskytuje stejné služby jako vrstva LLC. Rozdíl je jen ve vrstvách, s kterými komunikuje. MAC je začleněná mezi vrstvu fyzickou a LLC. Stejně jako u podvrstvy LLC, tak i u podvrstvy MAC jsou služby zajištěny pomocí tzv. „service primitives“. Sekvenci těchto „service primitive“ iniciuje vždy klient a v případě rozpadu spojení fyzická vrstva, která rozpad detekovala. Tyto „service primitives“ bývají zpravidla čtyři a ve spojení probíhají v pořadí *request*, *indication*, *response* a *confirm*.



Obr. 6 – MAC spojení

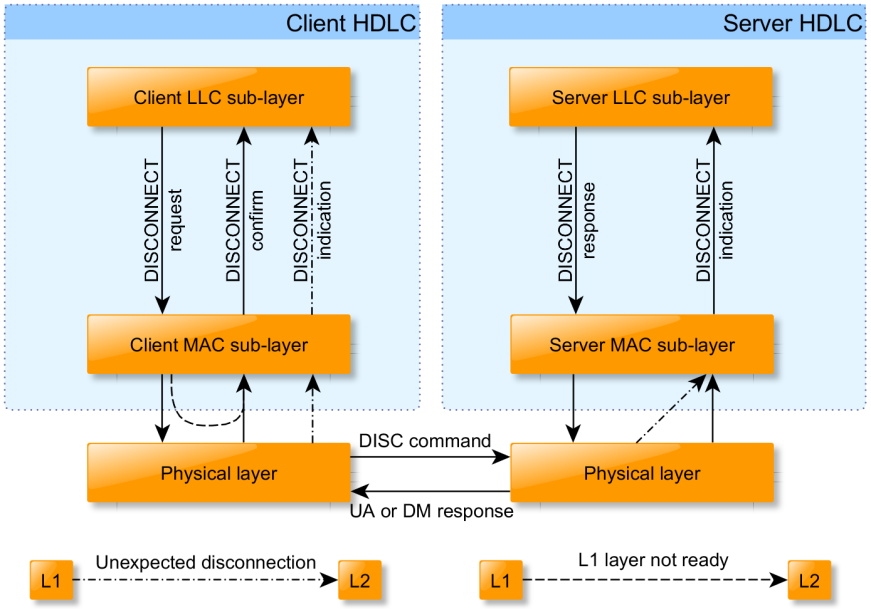
MAC spojení

Viz obrázek 6

- **CONNECT request**
  Klientská „service primitive“, která je vyvolána (přeposlána) LLC vrstvou. MAC vrstva formuje a posílá SNRM packet.
- **CONNECT indication**
  Serverová „service primitive“, kterou oznamuje MAC vrstva vrstvě LLC o příjmu požadavku na spojení.
- **CONNECT response**
  Serverová „service primitive“, pomocí které odpovídá LLC na požadavek na spojení.
- **CONNECT confirm**
  Klientská „service primitive“, kterou oznamuje MAC vrstva vrstvě LLC, že přijala odpověď na dotaz na spojení. Serverová strana odpovídá buď rámcem UA při pozitivní odpovědi nebo DM při negativní odpovědi.

MAC rozpojení

Viz obrázek 7.



Obr. 7 – MAC rozpojení

- **DISCONNECT request**
  Klientská „service primitive“, kterou žádá LLC vrstva o rozpojení. MAC vrstva formuje a odesílá DISC rámec.
- **DISCONNECT indication**
  Klientská i serverová „service primitive“, kterou vrstva MAC oznamuje vyšším vrstvám o žádosti o rozpojení (server), nebo informuje o rozpadu spojení (server i klient).
- **DISCONNECT response**
  Serverová „service primitive“, kterou odpovídá LLC vrstva na dotaz na rozpojení.
- **DISCONNECT confirm**
  Klientská „service primitive“, kterou oznamuje MAC vrstva o výsledku rozpojení.

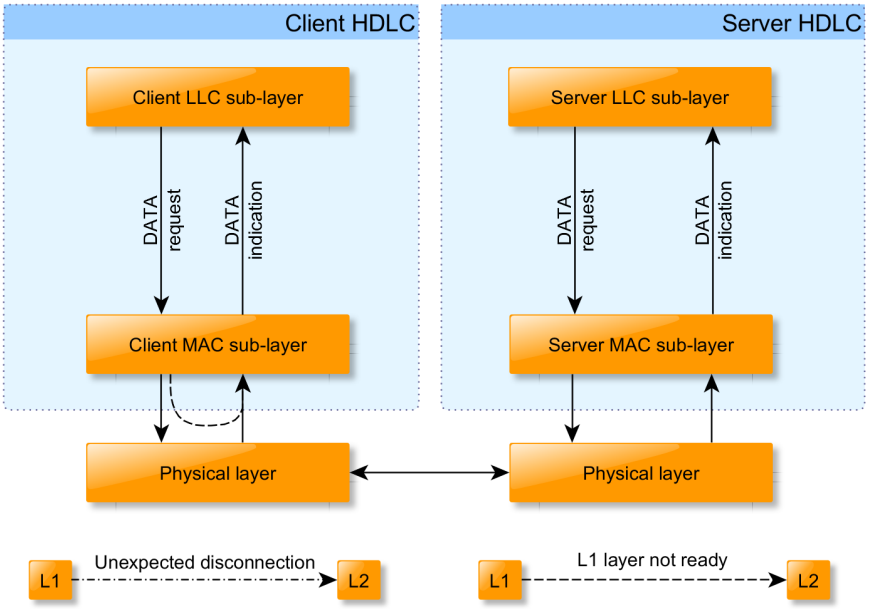
MAC data

Viz obrázek 8.

- **DATA request**
  Klientská „service primitive“, kterou z pohledu MAC vrstvy vyvolává LLC vrstva.
- **DATA indication**
  Serverová i klientská „service primitive“, kterou předává vrstva MAC vyšší vrstvě LLC přijatá data.
- **DATA confirm**
  Serverová „service primitive“, kterou oznamuje MAC vrstva serverové LLC vrstvě informaci o přijetí „DATA request“.

3.2.4 Funkce MAC vrstvy

MAC vrstva zajišťuje všechny funkce a vlastnosti, které vrstva LLC v rámci protokolu HDLC nezajistí. Prvotní komunikace začíná rámcem SNRM, na který odpovídá serverová strana buď rámcem UA při kladné odpovědi, nebo rámcem DM při záporné



Obr. 8 – MAC data

odpovědi. V navázání spojení také probíhá výměna a akceptace hodnot a velikostí, důležitých pro následnou komunikaci. Po navázaní spojení je možné posílat rámce typu I, které nesou informaci důležitou pro vyšší vrstvy OSI/ISO modelu. Zrušení spojení je zajištěno pomocí rámce DISC, na který může být opět odpověď kladná (UA) nebo záporná (DM).

3.2.5 Struktura MAC PDU

1B	2B	1-4B	1-4B	1B	2B	0-nB	0/2B	1B
Flag	Frame format	Dest. address	Src. address	Control	HCS	Information	FCS	Flag

Obr. 9 – MAC Packet

Struktura rámce z obrázku 9 odpovídá HDLC rámci typu 3. Rámec je ohraničen „flagy“, které mají hondotu **0x7E** a ukončovací „flag“ může být zároveň i počátečním „flagem“ následujícího rámce.

Frame format

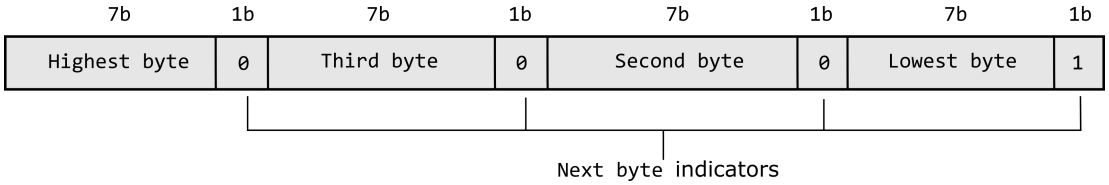
4b	1b	11b
Frame Type (0xA0)	S	Packet Length

Obr. 10 – Frame format

Formát rámce z obr. 10 je 16-ti bitové číslo, které je rozděleno na tři části. První, 4-bitová, část odpovídá rámci typu 3 a jeho hodnota je 1010 (binárně). Následuje 1 bit, který určuje segmentaci. Posledních a zbylých 11b určuje celkovou délku rámce bez

prvního a posledního „flagu“. Z délky rámce se také nedá vyčíst délka cílové a zdrojové adresy, viz dále.

Zdrojová a cílová adresa

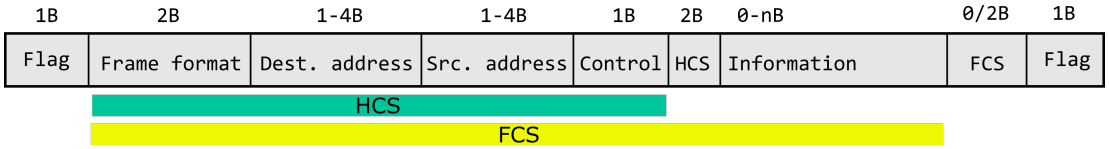


Obr. 11 – MAC adresa

Počet bytů adresy může být proměnný. Můžeme tedy dostat 1B, 2B a 4B adresu. Z každého bytu adresy je horních 7b určeno jako část hodnoty adresy a nejnižší bit značí, jestli má adresa další byte.

Adresu klienta, která může být jak zdrojová, tak cílová, tvoří pouze jeden byte. Avšak adresu serveru tvoří zpravidla 4 byty, které jsou ještě rozděleny na horní a dolní dvojici. Horní dvojice tvoří adresu logického zařízení a dolní dvojice tvoří adresu fyzického zařízení.

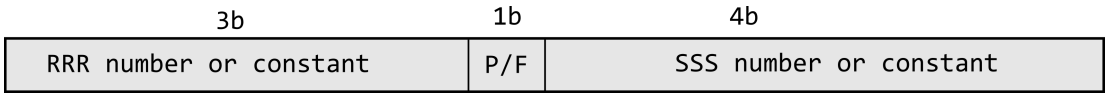
Kontrolní součty HCS a FCS



Obr. 12 – MAC kontrolní součty

V rámci na obrázku 12 jsou obsažena dvě pole, která kontrolují, jestli byl rámec přijat tak, jak byl vyslán. **HCS** je v rámci pokaždé a určuje kontrolní součet polí *Frame format*, *Dest. address*, *Src. address* a *Control*. Jestliže je součástí rámce informační část s daty, je na závěr přidáno i pole **FCS**, které obsahuje hodnotu s kontrolním součtem celého rámce vyjma *Flag* polí.

Pole *Control*



Obr. 13 – MAC kontrolní pole

Pole *Control* z obrázku 13 určuje typ a funkci, kterou HDLC rámec plní.

- I – (Information) – Informační rámec
- RR – (Receive Ready) – Příjem připraven
- RNR – (Receive Not Ready) – Příjem není připraven

- SNRM – (Set Normal Response Mode) – Nastav NRM
- DISC – (Disconnect) – Odpojit
- UA – (Unnumbered Acknowledge) – Nečíslované potvrzení
- DM – (Disconnected Mode) – Stanice odpojena
- FRMR – (Frame Rejected) – Rámec odmítnut
- UI – (Unnumbered Information) – Nečíslovaná informace

Obecné schéma pole *Control* z obrázku 13 je rozděleno na tři části, přičemž u každého typu paketu plní jinou roli. Ze složení kontrolního pole je vždy jasné, o jaký rámec se jedná.

3.3 Protokol DLMS/COSEM

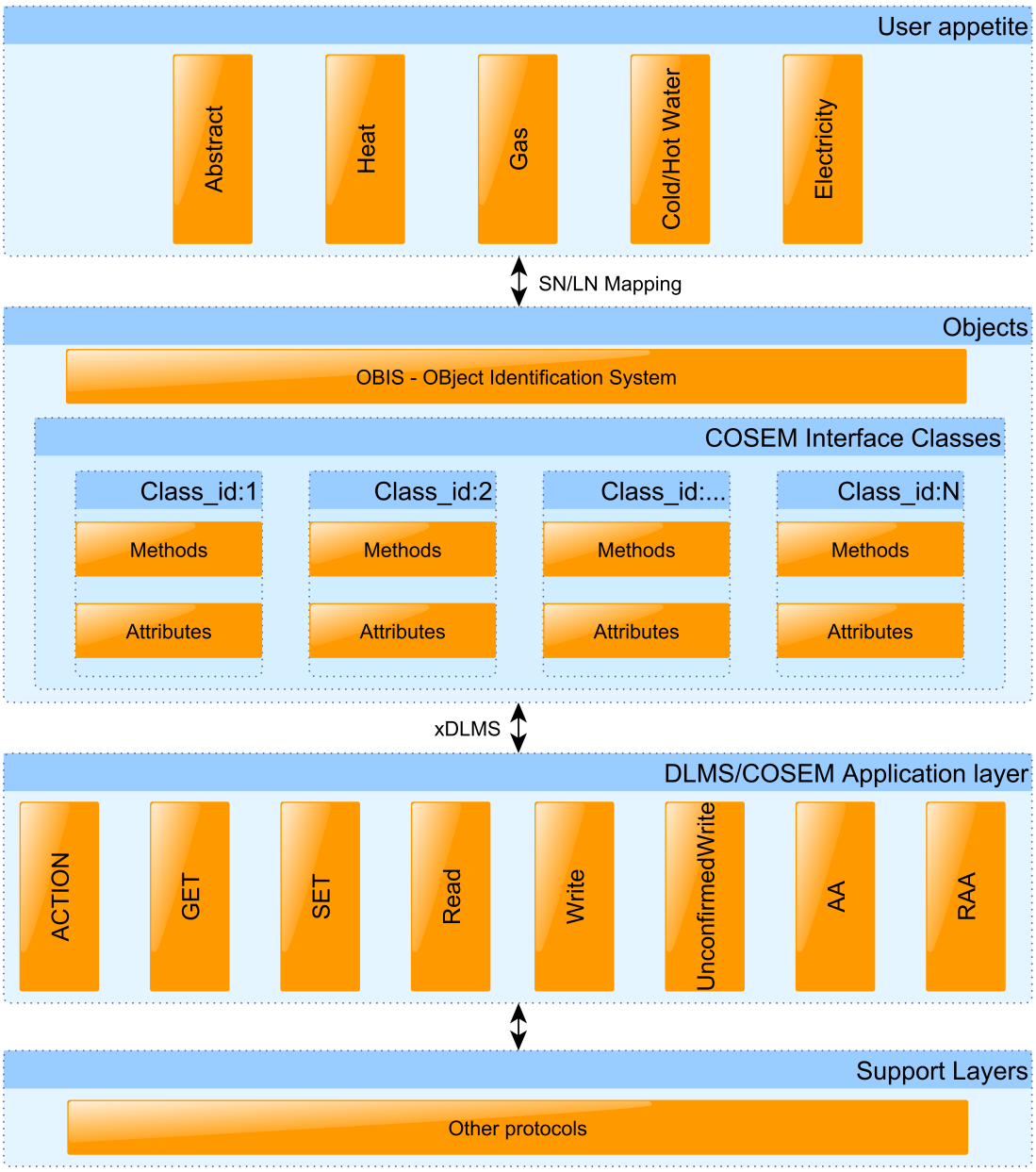
Tato část diplomové práce se bude zabývat průmyslovým komunikační protokolem DLMS/COSEM, který plní funkci transportní až prezentační vrstvy v komunikačním modelu OSI/ISO. Bude popsán logický model a struktura posílaných rámců protokolu DLMS/COSEM. Dále pak budou popsány třídy rozhraní COSEM, jejich případné použití, přístup k objektům pomocí LN (Long Name) nebo SN (Short Name) a identifikace pomocí kódů OBIS.

Na obrázku 14 je názorně ilustrováno hierarchické uspořádání vrstev v komunikačním profilu DLMS/COSEM. Z obrázku je patrné, že přístup k objektům je pro uživatele zjednodušen na přístup pomocí LN (Long Name) nebo SN (Short Name). Tento parametr je předán do nižší vrstvy COSEM, která požadavek dle příslušné třídy rozhraní zpracuje a předá formátovaný výstup aplikační vrstvě. Aplikační vrstva vloží výstup do příslušného PDU a celý dotaz předá nižším vrstvám ke zpracování.

3.3.1 Aplikační vrstva DLMS/COSEM

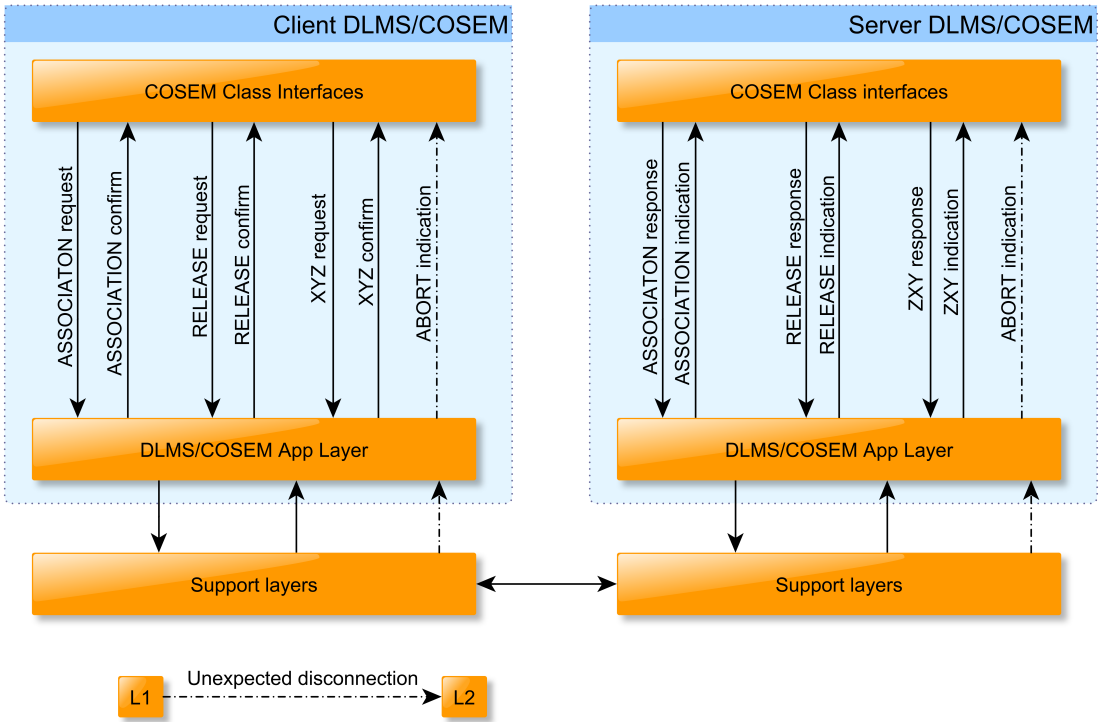
Aplikační vrstva DLMS/COSEM, stejně jako např. protokol HDLC, má vlastní *service primitives*, které slouží k navázání spojení, zrušení spojení a další komunikaci se vzdáleným zařízením. *Service primitives* jsou znázorněny na obrázku 15.

- **ASSOCIATION request**
  Klientská „service primitive“, která žádá cílovou stanici o navázání spojení. DLMS/COSEM vrstva tento požadavek zpracuje a odešle ve formě AARQ paketu.
- **ASSOCIATION indication**
  Serverová „service primitive“, která oznamuje COSEM vrstvě přijetí žádosti na asociaci.
- **ASSOCIATION response**
  Serverová „service primitive“, kterou odesílá server jako odpověď na žádost na asociaci. Odpověď může být jak pozitivní, tak negativní.
- **RELEASE request**
  Klientská „service primitive“, kterou klient žádá o rozpojení asociace.
- **RELEASE indication**
  Serverová „service primitive“, která oznamuje COSEM vrstvě o přijetí žádosti o uvolnění spojení / asociace.
- **RELEASE response**
  Serverová „service primitive“, která je odpovědí na RELEASE indication.
- **RELEASE confirm**
  Klientská „service primitive“ oznamuje vyšší vrstvě o výsledku rozpojení spojení.



Obr. 14 – Hierarchie vrstev DLMS/COSEM

- **XYZ request**
  Klientská „service primitive“, která žádá o datový přenos.
- **XYZ confirm**
  Klientská „service primitive“, která slouží jako odpověď na žádost o datový přenos.
- **ZXY indication**
  Serverová „service primitive“, která značí přijetí žádosti o datový přenos.
- **ZXY response**
  Serverová „service primitive“, která je odpovědí na žádost o datový přenos.
- **ABORT indication**
  Klientská i serverová „service primitive“, která indikuje náhlé rozpojení spojení.



Obr. 15 – Servisní primitivy DLMS/COSEM

Celý průběh komunikace může být popsán následně: Jakmile je ustáleno spojení na nižších vrstvách (např. HDLC), přichází na řadu aplikační vrstva DLMS/COSEM, která také vyžaduje navázání spojení před tím, než se bude moci dotazovat na data v zařízení. Tomuto navázání spojení se v aplikační vrstvě říká asociace.

V rámci asociace se zařízení domluví na vzájemných komunikační parametrech, které bude server i klient v průběhu komunikace dodržovat. Po navázání asociace je možné ihned dotazovat data na zařízení. Před dotazováním na data je však žádoucí dotázat tzv. asociační objekt, který nám pomůže při mapování mezi LN a SN a jejich třídami rozhraní, viz dále. Jakmile je klient hotov s dotazováním na objekty, měl by vyvolat řádné zrušení asociace.

3.3.2 ASN.1 jako notace použitá pro kódování posílaných dat

Protokol DLMS/COSEM používá k zápisu syntaxe standard s názvem Abstract Syntaxe Notation One. ASN.1 je formální jazyk pro specifikaci logických datových struktur, která si vyměňují dvě koncová zařízení. Tento standard je uzpůsoben tak, aby byl nezávislý na HW, OS a programovacím jazyku [4]. ASN.1 používá několik druhů kódovacích pravidel, která mohou jednu strukturu / datový typ vyjádřit s různými odlišnostmi. Protokol využívá pro kódování a dekódování asociačních služeb (ACSE) sadu pravidel BER (Basic Encoding Rules) [5], avšak služby pro datové přenosy (xDLMS ASE) jsou kódovány typem A-XDR (Advanced eXternal Data Representation), který je zjednodušením typu BER [6] .

3.3.3 Specifikace použitých služeb

Tato sekce se zaměří na PDU (Protocol Data Unit/y) služeb a na služby, které jsou použity k navázání asociace, získání atributů objektu, nastavení atributů objektu a uvol-

nění asociace. Tyto služby lze rozdělit na dvě části. První částí jsou služby pro asociační spojení, rozpojení a druhou částí jsou služby pro datový přenos.

- Asociační služby – ACSE
  - **AA spojení** – K zajištění dotazu této funkcionality se používá PDU s názvem AARQ jako dotaz na cílové zařízení a AARE jako odpověď.
  - **AA uvolnění spojení** – K zajištění dotazu této funkcionality se používá PDU s názvem RLRQ jako dotaz na cílové zařízení a RLRE jako odpověď.
- Služby pro datové přenosy – xDLMS ASE
  - **LN služby**[7]
    - **GET** – Služba, která je použita pro získání jednoho nebo více atributů z COSEM tříd rozhraní. K zajištění této služby se používá PDU s názvem *get-request* jako dotaz na cílové zařízení a *get-response* jako odpověď.
    - **SET** – Služba, která je použita pro nastavení jednoho nebo více atributů do COSEM tříd rozhraní. K zajištění této služby se používá PDU s názvem *set-request* jako dotaz na cílové zařízení a *set-response*.
    - **ACTION** – Služba, která je použita pro vyvolání jedné nebo více metod z COSEM tříd rozhraní. K zajištění této služby se používá PDU s názvem *action-request* jako dotaz na cílové zařízení a *action-response*.
  - **SN služby**[7]
    - **Read** – Služba, která se používá ke čtení jednoho nebo více atributů nebo k vyvolání jedné nebo více metod, u které se předpokládá návratový parametr. K zajištění této služby se používá PDU s názvem *readRequest* jako dotaz na cílové zařízení a *readResponse*.
    - **Write** – Služba, která se používá k zápisu jednoho nebo více atributů nebo k vyvolání jedné nebo více metod, u které se nepředpokládá návratový parametr. K zajištění této služby se používá PDU s názvem *writeRequest* jako dotaz na cílové zařízení a *writeResponse*.
    - **UnconfirmedWrite** – Služba, která se používá k zápisu jednoho nebo více atributů, nebo k vyvolání jedné nebo více metod. K zajištění této služby se používá PDU s názvem *unconfirmedWriteRequest* jako dotaz na cílové zařízení a *unconfirmedWriteResponse* jako odpověď.

Zápis čtyř asociačních APDU v syntaxi ASN.1 vypadá dle normy ČSN EN 62056-5-3 [7] následně:

```
xDLMS-APDU::=
{
   ...

   -- the four ACSE APDUs
   aarq AARQ-apdu,
   aare AARE-apdu,
   rlrq RLRQ-apdu,-- OPTIONAL
   rlre RLRE-apdu,-- OPTIONAL

   ...
}
```

AARQ

Je asociační datová struktura, která musí být pro navázání asociace poslána jako první. Obsahuje jak pole povinná, tak pole nepovinná. Důležitá je část *application-context-*

name, ve které se specifikuje referencování COSEM tříd rozhraní. V této části se také specifikuje, je-li použito šifrování nebo ne. Možnosti můžou být celkem 4:

1. LN without ciphering – 2.16.756.5.8.1.1
2. SN without ciphering – 2.16.756.5.8.1.2
3. LN with ciphering – 2.16.756.5.8.1.3
4. SN with ciphering – 2.16.756.5.8.1.4

Další důležitý parametr datové struktury AARQ je pole *user-information*, které obsahuje xDLMS APDU typu *InitiateRequest*. V této části je možnost specifikovat velikost přenášeného PDU, verzi protokolu, referencování COSEM tříd rozhraní, ale také obsahuje část *Conformance*, ve které je možno žádat o příslušnou službu nebo funkcionalitu.

```
AARQ-apdu ::= [APPLICATION 0] IMPLICIT SEQUENCE
{
    -- [APPLICATION 0] == [ 60H ] = [ 96 ]
    protocol-version           [0] IMPLICIT BIT STRING {version1 (0)} DEFAULT {version1},
    application-context-name   [1] Application-context-name,
    called-AP-title            [2] AP-title         OPTIONAL,
    called-AE-qualifier        [3] AE-qualifier       OPTIONAL,
    called-AP-invocation-id    [4] AP-invocation-identifier OPTIONAL,
    called-AE-invocation-id    [5] AE-invocation-identifier OPTIONAL,
    calling-AP-title           [6] AP-title         OPTIONAL,
    calling-AE-qualifier       [7] AE-qualifier       OPTIONAL,
    calling-AP-invocation-id   [8] AP-invocation-identifier OPTIONAL,
    calling-AE-invocation-id   [9] AE-invocation-identifier OPTIONAL,

    sender-acse-requirements   [10] IMPLICIT ACSE-requirements OPTIONAL,
    mechanism-name             [11] IMPLICIT Mechanism-name   OPTIONAL,


    calling-authentication-value [12] EXPLICIT Authentication-value   OPTIONAL,
    implementation-information   [29] IMPLICIT Implementation-data   OPTIONAL,
    user-information             [30] EXPLICIT Association-information  OPTIONAL
}
```

AARE

Touto datovou strukturou odpovídá vzdálené zařízení na dotaz na asociaci. V této odpovědi v části *result*, *result-source-diagnostic* je zakódován výsledek asociace a případně i diagnostika při negativní odpovědi. AARE datová struktura je velmi podobná AARQ.

```
AARE-apdu ::= [APPLICATION 1] IMPLICIT SEQUENCE
{
-- [APPLICATION 1] == [ 61H ] = [97]

protocol-version          [0] IMPLICIT BIT STRING {version1 (0)} DEFAULT {version1},
application-context-name  [1] Application-context-name,
result                    [2] Association-result,
result-source-diagnostic  [3] Associate-source-diagnostic,
responding-AP-title       [4] AP-title OPTIONAL,
responding-AE-qualifier   [5] AE-qualifier OPTIONAL,
responding-AP-invocation-id [6] AP-invocation-identifier OPTIONAL,
responding-AE-invocation-id [7] AE-invocation-identifier OPTIONAL,

-- The following field shall not be  present if only the kernel is used.
responder-acse-requirements [8] IMPLICIT ACSE-requirements OPTIONAL,

-- The following field shall only be present
if the authentication functional unit is
-- selected.
mechanism-name               [9] IMPLICIT Mechanism-name OPTIONAL,

-- The following field shall only be present if the authentication
 functional unit is selected.
responding-authentication-value [10] EXPLICIT Authentication-value OPTIONAL,
implementation-information       [29] IMPLICIT Implementation-data OPTIONAL,
```

```
user-information              [30] EXPLICIT Association-information OPTIONAL
}
```

RLRQ a RLRE

RLRQ a RLRE je dvojice datových struktur, které jsou použity pro servisní primitivu RELEASE. Existují ale zařízení, která nemají tuto funkcionalitu implementovanou. V takových případech je uvolnění asociace založeno na rozpojení podpůrných vrstev [6]. Struktura RLRQ a RLRE v notaci ASN.1 vypadá následně:

```
RLRQ-apdu ::= [APPLICATION 2] IMPLICIT SEQUENCE
{
-- [APPLICATION 2] == [ 62H ] = [ 98 ]
reason            [0] IMPLICIT Release-request-reason OPTIONAL,
user-information  [30] EXPLICIT Association-information OPTIONAL
}

RLRE-apdu ::= [APPLICATION 3] IMPLICIT SEQUENCE
{
-- [APPLICATION 3] == [ 63H ] = [ 99 ]
reason            [0] IMPLICIT Release-response-reason OPTIONAL,
user-information  [30] EXPLICIT Association-information OPTIONAL
}
```

GET

Pokud je v asociaci dohodnuto dotazování atributů a metod pomocí LN, je tato služba použita právě pro čtení atributů z připojeného zařízení. Datová struktura použitá v této službě je opět v ASN.1 notaci, ale používá se kódování A-XDR.

```
Get-Request ::= CHOICE
{
get-request-normal    [1] IMPLICIT Get-Request-Normal,
get-request-next      [2] IMPLICIT Get-Request-Next,
get-request-with-list [3] IMPLICIT Get-Request-With-List
}
```

V *Get-request* datové struktuře je povinností pro správné získání atributu definovat strukturu *Cosem-Attribute-Descriptor*, ve které je nutno uvést trojici hodnot *class-id*, *instance-id* a *attribute-id*. Tato trojice přesně určuje, který atribut a ze které třídy je klientem žádán. *Class-id* je typ COSEM třídy rozhraní, *instance-id* je název její instance a *attribute-id* je index atributu, který chceme. Výsledný formát pak může vypadat následně [7]:

```
Surová data:
  C001C1
  00010000800000FF0200

V XML:
  <GetRequest>
    <GetRequestNormal>
      <InvokedIdAndPriority Value="C1" />
      <AttributeDesriptor>
        <ClassID Value="0001" />
        <InstanceId Value="0000800000FF" />
        <AttributeId Value="02" />
      </AttributeDesriptor>
    <GetRequestNormal>
  <GetRequest>
```

ASN.1 syntaxe datových struktur *Get-Request* a *Get-Response*:

```
Get-Request ::= CHOICE
{
get-request-normal [1] IMPLICIT Get-Request-Normal,
get-request-next [2] IMPLICIT Get-Request-Next,
get-request-with-list [3] IMPLICIT Get-Request-With-List
}
```

```
Get-Response ::= CHOICE
{
get-response-normal          [1] IMPLICIT Get-Response-Normal,
get-response-with-datablock  [2] IMPLICIT Get-Response-With-Datablock,
get-response-with-list       [3] IMPLICIT Get-Response-With-List
}
```

Data v odpovědi jsou uložena v následující struktuře:

```
Get-Data-Result ::= CHOICE
{
data [0] Data,
data-access-result [1] IMPLICIT Data-Access-Result
}
```

Data jsou definována jako výběr z následujících datových typů:

```
Data ::= CHOICE
{
null-data        [0] IMPLICIT NULL,
array            [1] IMPLICIT SEQUENCE OF Data,
structure        [2] IMPLICIT SEQUENCE OF Data,
boolean          [3] IMPLICIT BOOLEAN,
bit-string       [4] IMPLICIT BIT STRING,
double-long      [5] IMPLICIT Integer32,
double-long-unsigned [6] IMPLICIT Unsigned32,
--floating-point [7] is not used in DLMS/COSEM. It uses [23] and [24]
octet-string      [9] IMPLICIT OCTET STRING,
visible-string    [10] IMPLICIT VisibleString,
-- time [11] is not used in DLMS/COSEM. It uses [27]
utf8-string      [12] IMPLICIT OCTET STRING,
-- An ordered sequence of characters encoded as UTF-8.
bcd              [13] IMPLICIT Integer8,
integer          [15] IMPLICIT Integer8,
long             [16] IMPLICIT Integer16,
unsigned         [17] IMPLICIT Unsigned8,
long-unsigned    [18] IMPLICIT Unsigned16,
compact-array    [19] IMPLICIT SEQUENCE
{
contents-description [0] TypeDescription,
array-contents   [1] IMPLICIT OCTET STRING
},
long64   [20] IMPLICIT Integer64,
-- long64-unsigned [21] IMPLICIT Unsigned64,
enum              [22] IMPLICIT Unsigned8,
float32           [23] IMPLICIT OCTET STRING (SIZE(4)),
float64           [24] IMPLICIT OCTET STRING (SIZE(8)),
date-time         [25] IMPLICIT OCTET STRING (SIZE(12)),
date              [26] IMPLICIT OCTET STRING (SIZE(5)),
time              [27] IMPLICIT OCTET STRING (SIZE(4)),
dont-care         [255] IMPLICIT NULL
}
```

SET

SET je služba pro zápis jednoho nebo více atributů na připojené zařízení s referencováním typu LN. K zápisu se využívá datové struktury *Set-Request*. Tato struktura má stejnou strukturu jako *Get-Request* s tím rozdílem, že se k „requestu“ přidává navíc hodnota, kterou chceme zapsat. Odpovědí na *Set-Request* je *Set-Response* s výsledkem operace zápisu.

ACTION

ACTION je služba, která na cílovém zařízení vyvolává metody COSEM tříd rozhraní a to jak s očekávaným výsledkem operace, tak bez něj. K zajištění této služby slouží

datová struktura *Action-Request*. Odpovědí na tuto službu je pak datová struktura *Action-Response*.

Read

Služba Read má stejný význam jako služba GET a navíc může vyvolat metodu, u které se předpokládá návratový parametr. Rozdílem mezi těmito dvěma službami je, že služba Read referencuje cílový atribut jménem typu SN. V tomto případě není potřeba trojice, jako v případě LN, ale SN adresuje přímo metodu nebo atribut, který chceme ze zařízení vyčíst.

Write

Služba Write má stejný význam jako služba SET a navíc může vyvolat metodu bez návratového parametru. Rozdílem mezi těmito dvěma službami je, že služba Write referencuje cílový atribut jménem typu SN. V tomto případě není potřeba trojice, jako v případě LN, ale SN adresuje přímo metodu nebo atribut, který chceme na zařízení zapsat nebo vyvolat.

UnconfirmedWrite

Je-li žádoucí na zařízení zapisovat nebo spouštět metody, aniž by byla služba potvrzena, je použita služba s názvem *UnconfirmedWrite*.

3.3.4 OBIS

OBIS poskytuje unikátní identifikátor pro všechna data v měřících zařízení. Lze ho použít nejen k identifikaci měřených hodnot, ale i k nastavení a získání informací o chování daného zařízení[8].

V rámci protokolu DLMS/COSEM je tento unikátní identifikátor používán jako LN a slouží tedy jako adresa instance objektu COSEM tříd rozhraní.

OBIS je šestice čísel **A** až **F**, kde každé číslo plní svůj význam a ve spojení s ostatními čísly šestice vytvoří unikátní identifikátor se specifickým významem.

- **Skupina A** udává typ měřené energie nebo média. V případě hodnoty 0, je identifikátor přiřazen k abstraktním objektům. Druhy energií, které skupina **A** může identifikovat jsou:
  - 0 – abstraktní
  - 1 – elektřina
  - 4 – akumulátor topných nákladů
  - 5 – chlazení
  - 6 – topení
  - 7 – plyn
  - 8 – studená voda
  - 9 – teplá voda
- **Skupina B** udává, z kterého kanálu pochází data měření.
- **Skupina C** přesněji specifikuje, co se měří. Například můžeme měřit proud, napětí a výkon.
- **Skupina D** identifikuje typ nebo výsledek zpracování fyzikálních veličin, specifikovaných skupinami A až C. Například minimum, maximum, průměrnou hodnotu a další...

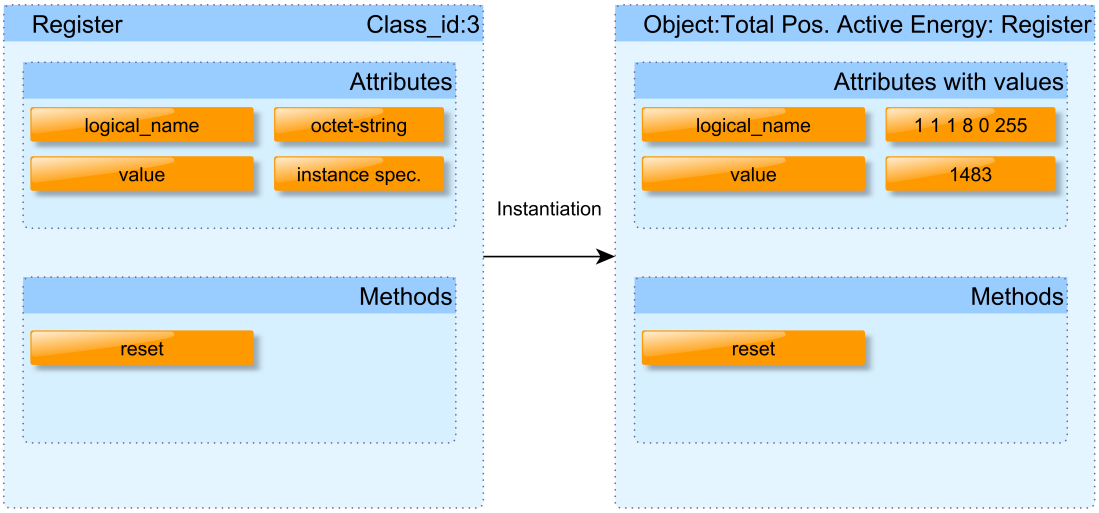
- **Skupina E** blíže určuje hodnoty měření, specifikované skupinami A až D. Může určovat hodnoty tarifů, harmonické a úhly fází...
- **Skupina F** je použita k identifikaci historických dat. Pokud tato identifikace není zapotřebí, může určovat další bližší klasifikaci předchozí pětice čísel.

3.3.5 COSEM třídy rozhraní

Tato sekce se bude zabývat třídami rozhraní COSEM, které jsou v profilu DLMS/COSEM použity jako objekty, které slouží pro přístup a práci s daty na cílovém zařízení a případně i na zdrojovém zařízení.

Objekt je kolekce atributů a metod. Atributy reprezentují charakteristiky objektu. Hodnota atributu může měnit chování celého objektu. Hodnota prvního atributu je vždy LN. Metody objektu mohou poskytovat nebo měnit hodnoty atributů [9].

Třídy rozhraní jsou definicemi kolekcí atributů a metod. Každá třída má vlastní id číslo, kterému přísluší typ třídy. Instanci-ováním těchto tříd dostaneme objekty, které v daném systému již plní specifickou roli. Tato role souvisí s přiřazeným LN. Pro ilustraci viz obrázek 16.



Obr. 16 – Ilustrace významu tříd a objektů COSEM

Přístup k objektům je zajištěn pomocí referencování LN (Long Name) nebo SN (Short Name).

- **LN reference**: Referencování pomocí LN udává přístup k celému objektu. Chceme-li přistupovat k určité metodě či atributu, musíme do žádosti přidat také *class_id* a *method/attribute_index*.
- **SN reference**: Referencování pomocí SN je jednoduší v tom, že nemusíme uvádět další příslušné parametry pro přístup k atributu nebo metodě. Pod SN si můžeme představit adresu, která odkazuje na všechny atributy a metody všech objektů na zařízení. Problémem však může být, že každé zařízení může mít vlastní přiřazení SN jiným LN. V dokumentaci a normách se ale uvádí pouze hodnoty LN.

V normě ČSN EN 62056-6-2 [9], jsou pro vyrovnání se s neznalostí SN definovány tzv. „base_names“, které pomáhají získat asociační objekt. Pomocí něho jsou pak získány SN a zařízení může na základě této znalosti přistupovat k příslušným atributům a metodám [9]. „Base_name“ pro asociační objekt je vždy **0xFA00**. Offset pro seznam objektů je pak **0x08**.

3.4 Druhy COSEM tříd rozhraní

COSEM třídy rozhraní jsou rozděleny do několika celků. Každý z těchto celků reprezentuje určitou skupinu dat nebo nastavení, které lze z měřícího zařízení získat nebo na něm nastavit.

- Parametry a data měření
- Řízení přístupu
- Čas a řízení prahů událostí
- Nastavení výměny dat skrz lokální porty a modemy
- Nastavení výměna dat skrz M-Bus
- Nastavení výměny dat přes Internet
- Nastavení výměny dat používající S-FSK PLC
- Nastavení vrstvy LLC pro ISO/IEC 8802-2

Tato část práce se omezí pouze na první tři skupiny a jejich třídy, které se v praxi využívají nejvíce.

3.4.1 Parametry a data měření

Data

Třída s id číslem 1 umožňuje vytvořit objekt pro všechny základní datové typy COSEM. Má pouze dva atributy, z toho jeden je LN a druhý je uchovávaná hodnota. Tato třída poskytuje konfigurační data a parametry.

Register

Třída *Register* je rozšířením třídy *Data* o měřítko a jednotku, ve které jsou data v registru uchována. Jednotky této třídy jsou různé a obsahují jak čas, stupně, jednotky objemu, tak i jednotky pro elektrické veličiny, tlak, výkon, a mnoho dalších. Měřítkem je exponent pro základ 10. Na rozdíl od třídy *Data* je *Register* zbaven datových typů, které jsou pro měření nepoužitelné jako např. boolean, enum a další.

Register také poskytuje metodu *reset*. Tato metoda nastaví výchozí stav hodnoty, která je specifická pro daný objekt.

Extended register

Třída *Extended register* je rozšířením třídy *Register*. Narozdíl od ní obsahuje navíc atribut *status* a *capture_time*. Atribut *status* blíže určuje uchovávanou hodnotu a je nutné hledat v manuálu výrobce zařízení, co nese tato informace. Atribut *capture_time* uchovává datum a čas, kdy byl atribut *value* získán.

Demand register

Třída *Demand register* je rozšířením třídy *Extended register* nejen o další atributy ale také o funkčnost. *Demand register* měří a periodicky vypočítává aktuální průměrnou hodnotu (*current_average_value*) a uchovává poslední průměrnou hodnotu (*last_average_value*). Tyto hodnoty jsou vypočítávány z posledních n period (*number_of_period*). Metodou *next_period* je možné spustit další periodu výpočtů. Aktuální perioda je ukončena a současná průměrná hodnota je uložena do atributu *last_average_value*.

Tato třída se využívá pro měření průměrné hodnoty za určité období. Například průměrná hodnota odebíraného proudu za den a další.

Register activation

Tato třída umožňuje přepínat mezi různými tarify. Každému *Register activation* náleží skupina složená z *Register*, *Extended Register* nebo *Demand Register*. Je možné přepínat například mezi nočním a denním tarifem. Atribut *active_mask* určuje, které objekty jsou aktivní v aktuální tarifu. Všechny registry, které nejsou uvedeny v atributu *register_assignment* jsou aktivní stále.

Metodami *add_register*, *add_mask* a *delete_mask* lze upravovat objekt *Register activation* a měnit jeho přidružené a aktivní registry.

Profile generic

Profile generic je třída pro uchování, řazení a přístupu ke skupinám datových objektů. Tato třída má atribut *buffer*, kde uchovává hodnoty všech objektů specifikované atributem *capture_objects*. Všechna data uchovaná v atributu *buffer*, lze libovolně řadit dle jednoho z objektů v poli *capture_objects*. Objekt tohoto typu může být poměrně velký a jeho velikost je dána atributy *entries_in_use* a *profile_entries*, který určuje maximální počet vzorků v bufferu. Poslední vstup do bufferu bude při zaplnění smazán.

Objekty této třídy slouží například pro uchování alarmů, chyb v komunikaci zařízení, maximálních a minimálních hodnot nebo uchování naměřených dat z určitého zúčtovacího období.

Řadící metody lze specifikovat atributy *sort_method* (lifo, fifo, od největšího, od nejmenšího, ...) a *sort_object*.

3.4.2 Řízení přístupu

Association SN

Association SN je třída, která pomáhá připojenému zařízení k získání všech SN daných objektů. V rámci cílového zařízení existuje pouze jeden objekt tohoto typu. Tato třída má předem definováno svoje SN (0xFA00) pro přístup k ní a jejím atributům.

Jedním z atributů je *object_list*, který obsahuje pole struktur, které uchovávají hodnoty:

- *base_name* – hodnota SN
- *class_id* – číslo třídy rozhraní COSEM
- *version* – verze třídy rozhraní COSEM
- *logical_name* – název LN

Tento atribut je nutné stáhnout při neznalosti přiřazení LN k SN.

Dalším atributem je *access_rights_list*. Tento atribut obsahuje pole struktur, které definují přístupová práva ke každému atributu a metodě.

Tato třída obsahuje také metody, pomocí kterých lze nastavovat nízkou či vysokou bezpečnost (*change_secret*), nebo získávat atributy a id tříd jiných objektů na základě jejich LN (*read_by_logicalname*).

Association LN

Association LN je třída pro asociaci navázanou pro referencování pomocí LN. Stejně jako *association SN* nabízí atribut *object_list*, kterým lze vyčíst všechny objekty na zařízení plus jejich SN, *class_id*, *version* a LN. Navíc nabízí tento atribut přístupová práva ke každému objektu v poli a nemusí být práva vyčítána zvlášť. To ale znamená vyšší paměťové nároky.

Navíc od *association SN* má atributy pro vyčtení *application_context_name*

a *xDLMS_context_type*, které byly nastaveny při asociaci. Z tohoto objektu lze také vyčíst *authentication_mechanism_name* a *secret* s aktuálním nastavením zabezpečení.

3.4.3 Čas a řízení hranic událostí

Clock

Clock je třída rozhraní COSEM, která uchovává veškeré informace, které souvisí s datem – časovou zónu, začátek a konec letního času, odchylku letního času, lokální čas a další...

Její hlavní atribut *time* obsahuje strukturu *date_time* (*OCTET_STRING(12)*), která uchovává:

- rok
- měsíc
- den v měsíci
- den v týdnu
- hodinu
- minutu
- vteřinu
- milisekundu
- odchylku
- status

Dalšími atributy jsou:

- *time_zone* – obsahuje odchylku lokálního času vzhledem k UTC.
- *status* – je stejný jako *status* získaný z atributu *date_time*
- *daylight_savings_begin*, *daylight_savings_end*, *daylight_savings_deviation* a *daylight_savings_enabled* – všechny tyto atributy se starají o nastavení letního času.
- *clock_base* je zajímavý z pohledu přesnosti času a obsahuje zdroj hodnoty času.
  - nedefinován
  - vnitřní krystal
  - síťová frekvence 50Hz
  - síťová frekvence 60Hz
  - GPS (Global Positioning System)
  - rádiově řízený

V této třídě také existují metody, kterými lze různým způsobem upravovat čas. Zarovnání času k určitě pevné hodnotě je prováděno pomocí metod začínající *adjust_to_*. Zbylé dvě metody *preset_adjusting_time* a *shift_data* nastavují a posunují čas dle požadavků uživatele.

Script table

Script table umožňuje spouštět předem definované operace v atributu *scripts*, kde jsou definovány i objekty a atributy/metody nad kterými se skript spouští. Předem definované operace se vztahují jen k operaci zápis atributu a spuštění metody.

Schedule a Special days table

Tyto dvě třídy umožňují řízení aktivit na základě data a času. Třída *schedule* obsahuje atribut *entries*, který obsahuje pole struktur se jménem skriptu a definovaným časem spuštění. Dále obsahuje metody (*insert*,*delete*,*enable/disable*) pro administraci atributu *entries*.

Přidružená třída *Special days table* definuje výjimky z normálních dní a tím umožňuje měnit standardní chování systému.

Podobnou funkci jako *Schedule* má i *Activity calendar* s tím rozdílem, že *Schedule* má vyšší prioritu při volání operací. *Activity calendar* nabízí volání různých operací na základě aktuálního tarifu (období).

Register monitor

Tato třída umožňuje monitorovat skupinu objektů tříd *Data*, *Register*, *Demand Register* a *Extended Register*. Poskytuje také možnost nastavení různých prahů, hodnot k monitorování a také spouštět definované skripty při překročení prahu.

Tato třída nemá žádné metody a vše je zajištěno pomocí atributů:

- *thresholds* – práh stejného datového typu jako je monitorovaný atribut.
- *monitored_value* – upřesňuje objektový atribut, který je monitorován
- *actions* – tento atribut definuje skript, který se bude spouštět při překročení prahu směrem nahoru a skript při překročení prahu směrem dolu.

Disconnect control

Objekty typu *Disconnect control* řídí připojování vnitřních nebo vnějších připojených zařízení. Pro zjištění stavu připojeného zařízení slouží atributy *output_state*, *control_state* a *control_mode*. Pro ovládání příslušného připojeného zařízení slouží metody *remote_disconnect* a *remote_connect*

Limiter

Limiter má podobnou funkci jako *Register monitor*. Navíc u této třídy je definován interval po jehož naplnění je spuštěna příslušná operace (skript). Hodnota prahu má zde dvě úrovně *normal* a *emergency*. Při přechodu do úrovně *emergency* je definován nouzový režim atributem *emergency_profile*.

4 Analýza a návrh řešení

Jedním z klíčových produktů firmy TECHSYS je vestavěné zařízení Storm-01 s FW Storm. Storm-01 je malý terminál, který je již hojně využíván pro telemetrii. FW Storm z toho důvodu již obsahuje další funkce a má definovanou vlastní strukturu FW. Celý FW je napsaný pomocí programovacího jazyka C++, běží na operačním systému RTX a je programován vývojovými nástroji Keil. Stávající rozložení paměti nám dovoluje aby měl binární program pouhých 383kB. Aktuálně je velikost binárního programu 360 208B. Z toho plyne, že celá implementace může mít maximálně 23kB, aniž by se musel binární program členit dle funkcí, které budou poskytovány.

Každý modul komunikačního protokolu v rámci FW Storm má vlastní task. Každý takový task má definovanou maximální zátěž stacku 4096B. To bude dalším omezením při analýze a implementaci knihovny HDLC a frameworku DLMS/COSEM.

Zařízení Storm-01 disponuje 2MB externí RAM paměti s odděleným prostorem 1,9 MB pro heap.

Implementace celého protokolu DLMS/COSEM je nad rámec této práce. Tudíž pro využití frameworku byly na základě konzultací vybrány veškeré funkce protokolu HDLC a pro protokol DLMS/COSEM byly vybrány funkce pro navázání asociace, funkce Read a z rozhraní tříd COSEM byly vybrány třídy Registr a Clock. Vybranou funkčnost je následně třeba demonstrovat na zařízení Storm-01, které je považováno za „úzké hrdlo“ a které ověří minimální nároky implementace. Důraz je ale také kladen na snadnou rozšiřitelnost celé implementace o další třídy rozhraní COSEM a další funkce pro čtení/zápis objektů ze zařízení.

4.1 Analýza problémů a možností

4.1.1 Alternativní řešení - knihovna Gurux.DLMS.cpp

Opensource řešení nabízí knihovna Gurux.DLMS.cpp, která nahrazuje knihovnu GuruxDLMSLib. Tato open source knihovna je vydávána pod licencí GPLv2 a tím pádem je nutné zveřejnit zdrojový kód programu, který tuto knihovnu využívá. Na základě této podmínky bylo rozhodnuto o nezveřejnění zdrojových kódů firmy TECHSYS a tím není možné použití knihovny Gurux.DLMS.cpp. To ale nebrání bližšímu prozkoumání a analýze zdrojových kódů knihovny nebo případnému vyzkoušení.

Již při prvním pohledu je vidět použití kontejnerů. Vzhledem k rozhodnutí o zamezení dynamické dealokace je to další důvod, proč je knihovna na vestavěných systémech nepoužitelná. Ve vztahu ke kontejnerům se nejedná jen o využívání dynamické paměti, ale zároveň i zvětšení velikosti binárního programu. Cílem práce je omezit využití paměťových zdrojů.

Knihovna Gurux.DLMS.cpp je složením jak protokolu HDLC, tak protokolu DLMS/-COSEM. Protokol HDLC je ale zakomponován do knihovny tak, že není poznat rozdíl mezi DLMS/COSEM a HDLC. To by z hlediska funkčnosti nevadilo, ale problém nastává při případném ladění, kdy není možné jasně oddělit tyto dvě vrstvy.

Poslední negativum, které knihovna měla, bylo „hard-coding“. Zakódování AARQ paketu vypadá totiž takto:

```
//User information code is special case...
int CodeData(std::vector<unsigned char>& buff)
{
  buff.push_back(0xBE); //Tag
  buff.push_back(0x10); //Length for AARQ user field
  buff.push_back(0x04); //Coding the choice for user-
     information (Octet STRING, universal)
  buff.push_back(0x0E); //Length
  buff.push_back(InitialRequest); // Tag for xDLMS-Initiate
      request
  buff.push_back(0x00); // Usage field for dedicated-key
     component not used
  buff.push_back(0x00); // Usage field for the response
     allowed component not used
  buff.push_back(0x00); // Usage field of the proposed-
     quality-of-service component not used
  buff.push_back(DLMSVersioNumber);
  // Tag for conformance block
  buff.push_back(0x5F);
  buff.push_back(0x1F);
  buff.push_back(0x04);// length of the conformance block
  buff.push_back(0x00);// encoding the number of unused
     bits in the bit string
  GXHelpers::AddRange(buff, &m_ConformanceBlock, 3);
  buff.push_back(((MaxReceivePDUSize >> 8) & 0xFF));
  buff.push_back(MaxReceivePDUSize & 0xFF);
  return ERROR_CODES_OK;
}
```

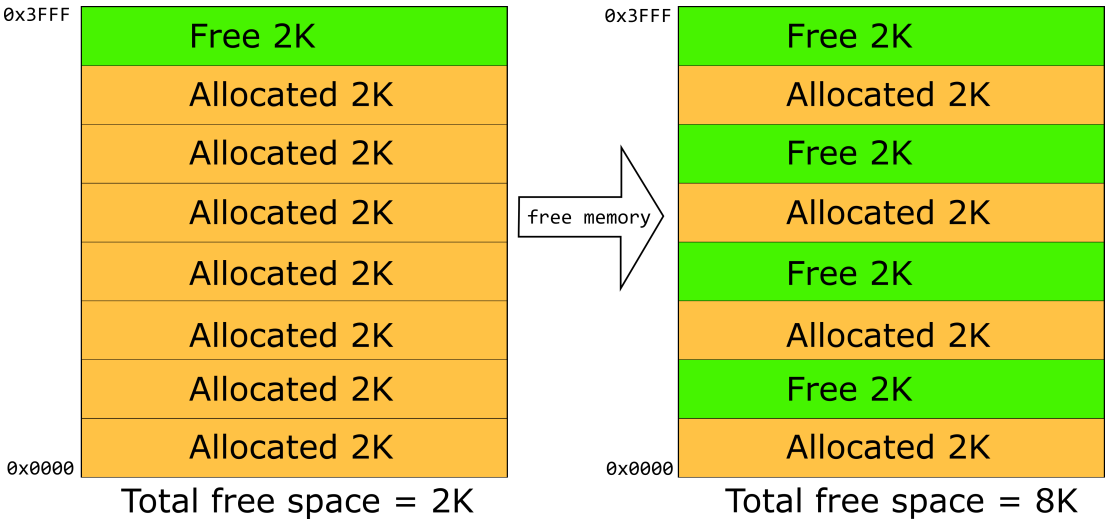
Zdrojový kód je sice efektivní, ale rozhodně z něj není poznat, co dané byty znamenají. Při požadované změně není možné tento kód jakkoliv jednoduše upravovat bez znalosti ASN.1.

I přes všechny nedostatky zdrojového kódu je knihovna funkční a s dostupnými zařízeními komunikujícími protokolem DLMS/COSEM je použitelná. Knihovna je také z programátorského hlediska velmi dobře komentovaná.

4.1.2 Paměť na vestavěných systémech

Aby byla zajištěna přenositelnost i na zařízení, jejichž paměť je značně omezena, musí být dodrženy podmínky, které zabraňují využívání většiny pohodlných prostředků jazyka C++. Zároveň je kladen důraz na nižší nároky na výpočetní výkon. Ve vestavěných zařízeních existuje mnoho argumentů, proč opětovně nepoužívat dynamickou alokaci a dealokaci paměti, která se ve většině běžných aplikací hojně využívá. To klade velký důraz na programátora/vývojáře, aby si dopředu rozmyslel, kolik paměti a v jakém okamžiku bude aplikace potřebovat, analyzoval její průměrnou a maximální spotřebu paměti, a to jak na *stacku*, tak na *heapu*.

Problémy ale netvoří jen paměť dat, ale i místo v paměti, kde je uložen binární program. Čím rozsáhlejší funkčnost implementujeme, nebo přidáváme rozsáhlé knihovny, tím rapidně roste i jeho velikost.



Obr. 17 – Fragmentace dynamické paměti

Klady a zápory dynamické alokace paměti

+ Výhodou dynamické alokace paměti může být šetření místem, které je potřeba jen v určitou dobu chodu programu. Následně může být uvolněno a použito jinou nebo stejnou částí programu.

+ Při neznalosti velikosti potřebného místa v okamžiku vývoje aplikace je na místě dynamická alokace a s ní spojené ušetření nadbytečného (rezervního) místa, které je většinou uloženo na *stacku*.

– Dynamická alokace paměti není deterministická operace, čili nemůžeme na 100 % určit, že požadovanou paměť dostaneme a zároveň, pokud požadovaná paměť není k dispozici, nemůžeme určit, kdy k dispozici bude. Pokud by tento problém nastal na „hard real-time“ systémech či na podobných zařízeních, jejichž funkce nesmí být přerušena výpadkem, mělo by to vážné důsledky a snižovalo spolehlivost a bezpečnost celého systému.

– Fragmentace paměti je další příčinou negativního výsledku dynamické alokace paměti. Jak je znázorněno na obrázku 17, i přes to, že je k dispozici až 8 kB paměti, budeme-li chtít blok větší než 2 kB, nedostaneme ho. Tento problém může nastat i na zařízeních, jejichž paměť je ve srovnání s vestavěnými zařízeními až několika set násobně větší. Avšak pravděpodobnost nedostatku dynamické paměti je velmi malá a následky mají malou závažnost.

– Dynamická alokace je pomalým procesem a snižuje důležitý výkon, kterého je už takto málo.

Vyhodnocení dynamické alokace a dealokace

Na základě předešlé analýzy se postupuje i na cílovém vestavěném zařízení, do kterého se bude tato implementace zakomponovávat. Bez dynamické alokace bychom přišli o některé výhody jako například dědičnosti, která se vyhodnocuje za běhu pomocí virtuální

tabulky metod. Proto bude použita dynamická alokace ale jen při inicializaci a nebude možnost dynamicky dealokovat a znovu alokovat. Tím je zabráněno fragmentaci paměti, zátěži paměti stacku a při běhu programu nedochází k výpadkům na základě negativního výsledku dynamické alokace.

4.1.3 Generování kódu z ASN.1 notace

Norma protokolu DLMS/COSEM [7] obsahuje kapitolu se specifikací posílaných paketů v ASN.1 notaci. Bylo zamýšleno, že tato specifikace bude předložena překladači **asn1c**, který vygeneruje zdrojové kódy, které dokáží s příslušným objektem pracovat dle jeho kódování. Problémem ale byla opět práce s dynamickou pamětí a navíc **asn1c** nedokáže pracovat s kódováním typu A-XDR. Na základě toho je nutné implementovat práci s objekty v ASN.1 notaci vlastními silami.

4.2 Návrh HDLC

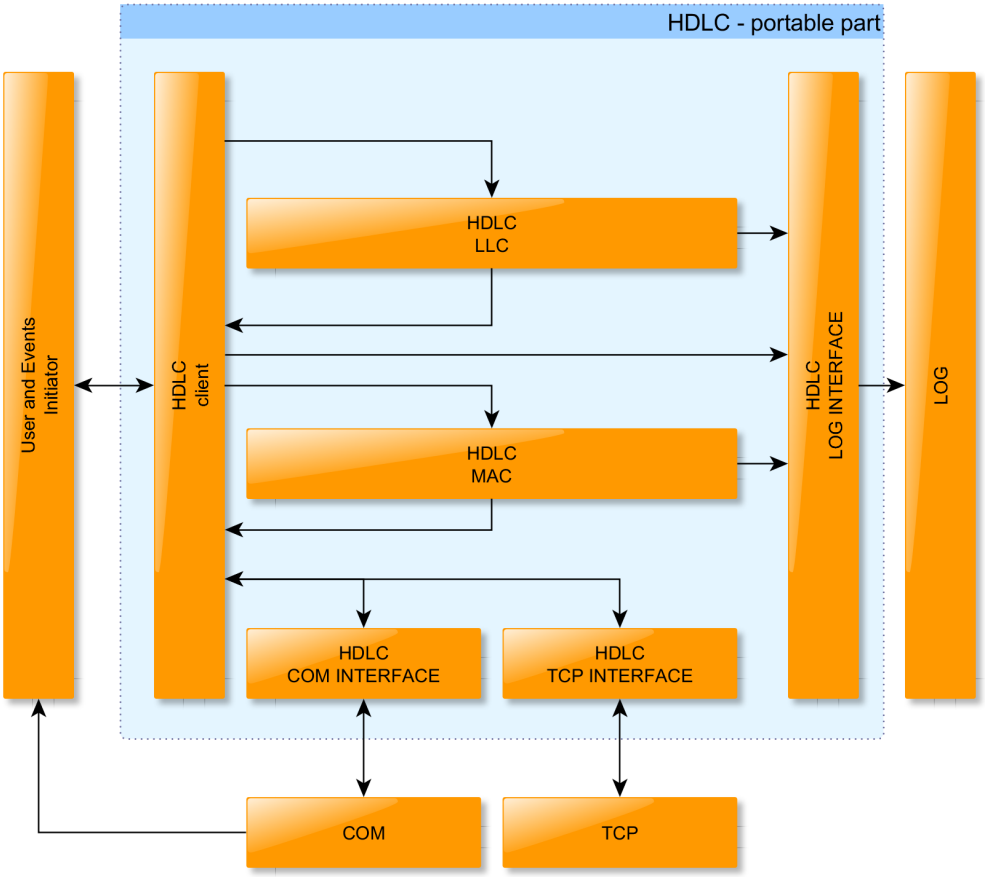
Vycházíme z předpokladu, že celá knihovna bude používána mezi vrstvami OSI/ISO modelu a tím pádem musí mít definované rozhraní, které bude používat pro komunikaci s vyššími a nižšími vrstvami. Dále musí být definováno rozhraní, které umožní správu a ovládání této knihovny. Aby byla zajištěna přenositelnost, musí být z knihovny odděleny části, které jsou závislé na platformě. Tyto části pak musí být vytvořeny dle dané platformy. Jedná se zejména o části pro komunikaci po TCP/UDP a COM. Další částí, kterou je potřeba implementovat až dle platformy, je „logování“, které nemá vliv na funkci knihovny, ale je důležité pro nalezení chyb, ladění a následnou správu aplikace/knihovny. Celý návrh knihovny je tedy možné rozdělit na několik částí, které je potřeba prozkoumat a navrhnout. Tyto části jsou:

- Logický model knihovny.
- Komunikace s nižšími vrstvami.
- Vnitřní struktura knihovny.
- Uživatelský přístup ke knihovně.

4.2.1 Logický model knihovny

Obrázek 18 ilustruje hlavní bloky, které jsou nezbytné pro správnou funkci HDLC knihovny.

- HDLC client – Hlavní část knihovny, která má na starosti komunikaci s uživatelskou vrstvou a zároveň zajišťuje kooperaci vnitřních částí HDLC knihovny.
- HDLC LLC – Tato část je nezbytná k zajištění správné funkce LLC vrstvy protokolu HDLC.
- HDLC MAC – Tato část je nezbytná k zajištění správné funkce MAC vrstvy protokolu HDLC. Tuto vrstvu můžeme pokládat za jádro této knihovny, jelikož na ní leží většinová logika celého protokolu HDLC.
- HDLC TCP INTERFACE – Je část, která předepisuje rozhraní pro komunikaci s nižší vrstvou zajišťující komunikaci po TCP/UDP.
- HDLC COM INTERFACE – Je část, která předepisuje rozhraní pro komunikaci s nižší vrstvou zajišťující komunikaci po sériovém rozhraní.



Obr. 18 – Vnitřní logická struktura HDLC

4.2.2 Komunikace s nižšími vrstvami

Protokol HDLC využívá pro komunikaci s ostatními zařízeními další nižší standardizované protokoly. Implementace a začlenění těchto protokolů do SW případně do FW byla již v minulosti firmou TECHSYS vyvinuta a není proto třeba tyto protokoly vyvíjet. Protože se jedná o linkový protokol, tak nižšími vrstvami, které HDLC využívá, jsou hlavně TCP/UDP a sériové rozhraní. Implementace a použití těchto knihoven pro MS Windows a Linux jsou poměrně známé, avšak knihovna, která je implementována na „real-time“ operačním systému RTX má výrazně jiné použití z hlediska indikace spojení, rozpojení a toku dat. Indikace je na tomto systému navržena pomocí zasílání událostí do příslušného tasku, který v nekonečné smyčce spouští jejich obsluhu. Musí proto existovat příslušná část, která plní funkci MITM a která získává informace o událostech a na jejich základě volá příslušné funkce knihovny HDLC. Tato část může být do jisté míry na operačních systémech MS Windows a Linux vynechána a funkce např. pro příjem a odesílání dat mohou být volány ihned a čekání na data bude zajištěno ve vlastních funkcích.

4.2.3 Uživatelský přístup ke knihovně

Z hlediska uživatele (vyšší vrstvy) musí být definovány metody, které mohou parametrizovat a ovládat danou vrstvu protokolu HDLC. Hlavní metody pro parametrizování a ovládání jsou:

- Vytvoření instance
- Nastavení cílové a zdrojové adresy
- Přiřazení uživatelských tříd pro komunikaci s nižšími vrstvami
- Spojení s cílovým zařízením – Stávající zařízení vyšle dotaz na spojení. S pozitivním výsledkem je pak umožněno zasílat a přijímat data z cílového zařízení.
- Odesílání dat na cílové zařízení
- Příjem dat
- Rozpojení spojení
- Nastavení „logování“

4.2.4 Vnitřní struktura knihovny

Hlavní třída **CTs_HDLC_client** bude obsahovat třídy podvrstev a zároveň bude sloužit jako nástroj pro ovládání tohoto protokolu a zajišťovat všechny funkce z podsekce 4.2.3. Třídy podvrstev třídy **CTs_HDLC_client** jsou logicky **CTs_HDLC_LLC** a **CTs_HDLC_MAC**. Třída MAC bude mít na starosti většinu funkcí protokolu. Bude komunikovat s rozhraními pro komunikaci s TCP/UDP a COM. Tato rozhraní pak bude používat pro příjem a vysílaní dat. Pro příjem dat bude sloužit třída s názvem **CTs_MAC_FSM**. Tato třída bude na základě přijatého bytu obsluhovat stavový automat, který se stará o správné přijetí HDLC rámce. Po úspěšném přijetí je tato událost předána zpět třídě **CTs_HDLC_MAC**, která dále rámec zpracovává dle vlastní logiky. Pokud rámec obsahuje data pro vyšší vrstvy, je výsledkem zpracování tohoto rámce předání dat vyšším vrstvám.

Knihovna bude také disponovat menšími datovými strukturami, které usnadní zpracování rámců, zpřehlední kód a tím zjednoduší práci při vývoji. Těmito třídami a strukturami budou:

- **CTs_HDLC_MAC_Addr** – Tato třída se bude starat o uchování adresy a poskytovat ji nejen ve formě, která je protokolem HDLC žádána, ale také ve formě, která je uživatelem čitelná, čili její nekódovanou formu. Protože je velikost adresy proměnná, je potřeba se k adrese chovat dle její velikosti a také ji tak zpracovávat.

- **S2B** a **S4B** – Struktury, které budou sloužit pro práci s endianitou a přechody mezi BE a LE. Tato struktura bude z větší části použita zároveň s výčtovým typem *enum*, který obsahuje paralelně i hodnotu primitivního nebo složeného typu, ale o stejné velikosti.

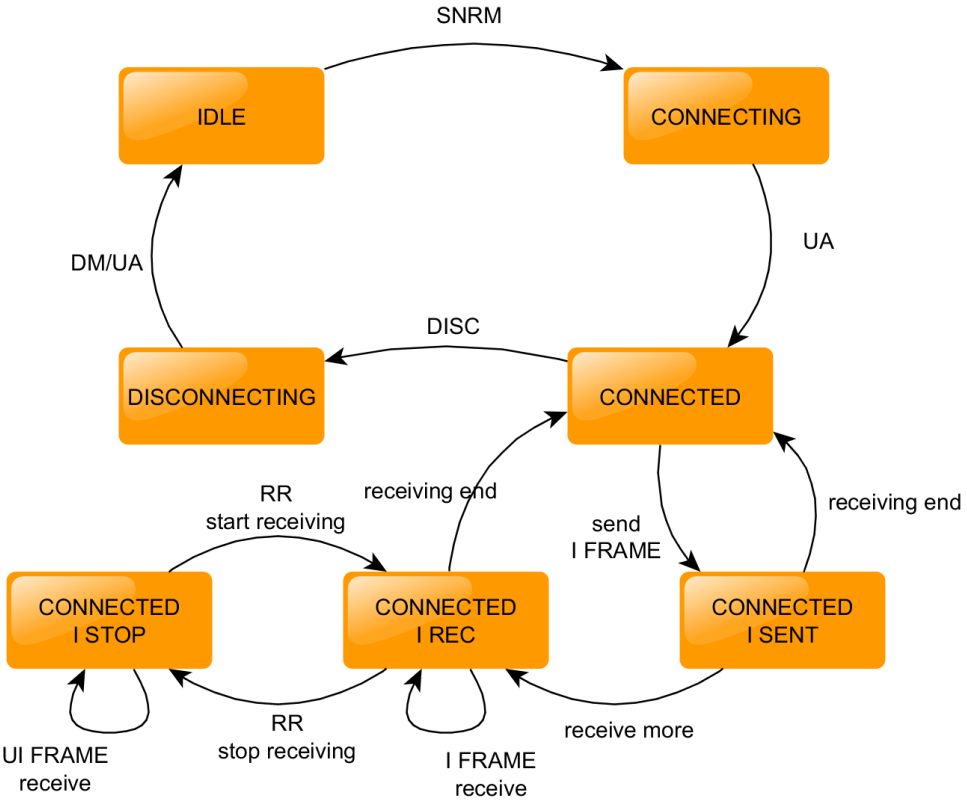
- **UHDLC_Frame_Format** – Union, který bude umožňovat práci a bitový přístup k poli „frame Format“ v HDLC paketu.

- **UHDLC_Control_Format** – Union, který bude umožňovat práci a bitový přístup k poli „Control“ v HDLC paketu.

- **UHDLC_Header_Addr** – Union, který bude specifikovat vlastní strukturu adresy a umožňovat bitový přístup k ní. Tento datový typ se bude používat v rámci třídy **CTs_HDLC_MAC_Addr**.

- **SHDLC_Header** – Struktura, která bude obsahovat celou hlavičku HDLC a zároveň poskytovat data v obrácené endianitě.

- **SHDLC_Header_WI** – Struktura, která bude obsahovat jak hlavičku HDLC, tak i její informační část a stejně jako **SHDLC_Header** bude poskytovat data v obrácené endianitě.



Obr. 19 – Stavový automat chování knihovny

4.2.5 Chování knihovny při ...

Aby se knihovna chovala deterministicky, musí mít definovaný stavový automat, na základě kterého bude knihovna provádět příslušné reakce. Na obrázku 19 je takový automat znázorněn. V automatu pro přehlednost chybí přechody při hrubém rozpojení nižší vrstvou nebo uživatelem. Pokud se tak stane, je vždy následující stav IDLE. Dále se v obrázku nenachází reakce na špatný rámec. V takovém případě se knihovna chová stejně, jakoby se žádný rámec nepřijal.

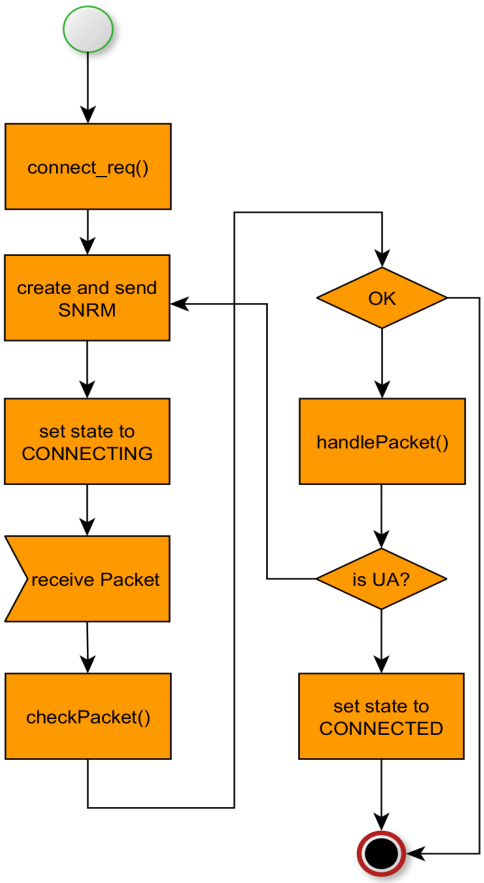
Příjímání dat

Příjem dat bude probíhat po jednotlivých bytech a paralelně s příjmem bytů paketu bude částečně vyhodnocována správnost částí paketu. Přijaté byty budou formovány do struktury **SHDLC_Header** nebo případně do struktury **SHDLC_Header_WI**. Jakmile bude celý paket přijat, dojde k dalšímu zpracování a kontrole správnosti dat.

1. V první fázi budou vyhodnoceny kontrolní součty HCS a pokud to bude nutné, tak i FCS. Při kolizi dat a kontrolních součtů bude paket vyhodnocen jako špatný a nebude brán v potaz.

2. V druhé fázi dojde k vyhodnocení cílové a zdrojové adresy. Pokud nejsou adresy v pořádku, bude paket ignorován.

3. Ve třetí a čtvrté fázi bude knihovna vyhodnocovat pole „Frame format“ a „Control“. Na základě těchto dvou polí a aktuálním stavu bude knihovna podnikat další kroky.



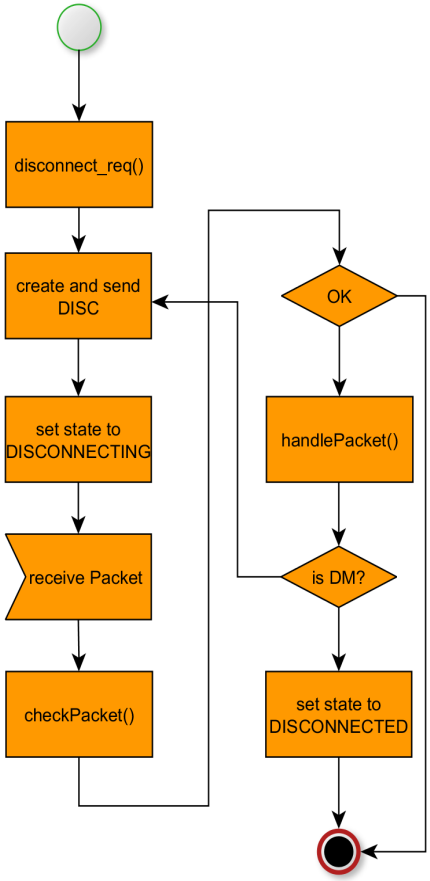
Obr. 20 – HDLC Průběh připojení

Připojení

Připojení začíná uživatelským voláním požadavku na spojení (*connect_req*). Jakmile bude zpracován, vytvořen a odeslán SNRM paket, dojde k přepnutí knihovny do stavu CONNECTING. Poté dojde k předání řízení zpět do rukou uživatele. Jakmile budou k dispozici data, měl by uživatel předat řízení zpět knihovně, která bude přijímat data z nižších vrstev tak dlouho, dokud budou data k dispozici, nebo dokud nenačte celý paket. Po přijetí dat dojde ke kontrole paketu a vyhodnocení, zda-li se jedná o UA paket. Pokud je vše v pořádku, přejde knihovna do stavu CONNECTED a bude připravena pro příjem a zasílání informačních paketů.

Odpojení

Odpojení probíhá analogicky k připojení. Celý proces bude začínat uživatelským voláním požadavku na odpojení. Jakmile bude zpracován, vytvořen a odeslán DISC paket,



Obr. 21 – HDLC Průběh odpojení

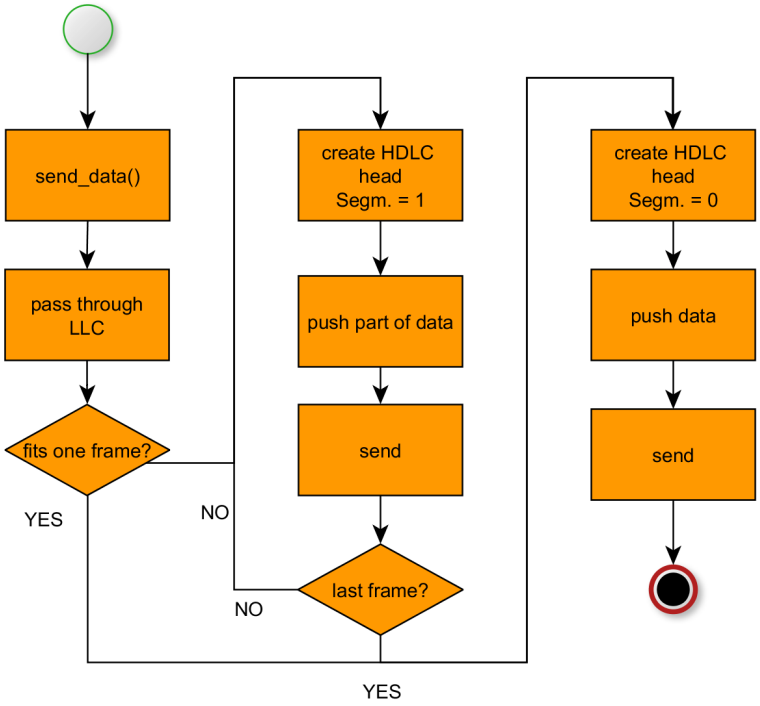
dojede k přepnutí stavu knihovny do stavu DISCONNECTING a knihovna bude v dalším kole příjmu očekávat paket UA, který značí správné rozpojení. Pokud se tak nestane, je knihovna nucena opakovat požadavek na rozpojení až do definovaného počtu pokusů.

Přenos informačních paketů

Příjem a vysílání informačních paketů je z hlediska knihovny nejdůležitější funkce. Každá MSDU jednotka protokolu HDLC musí mít algoritmus, který zaručuje přenos dat, která se nevejdou do jednoho rámce v informačních částech paketu. Toto indikuje a umožňuje „segmentation bit“ v poli *Control*.

Knihovna HDLC bude mít tento algoritmus také. Bude aplikován jak pro vysílání větších paketů, tak pro příjem paketů.

Průběh odesílání informačních paketů je znázorněn na obrázku 22.



Obr. 22 – HDLC - Odesílání informačních paketů

4.3 Návrh DLMS/COSEM

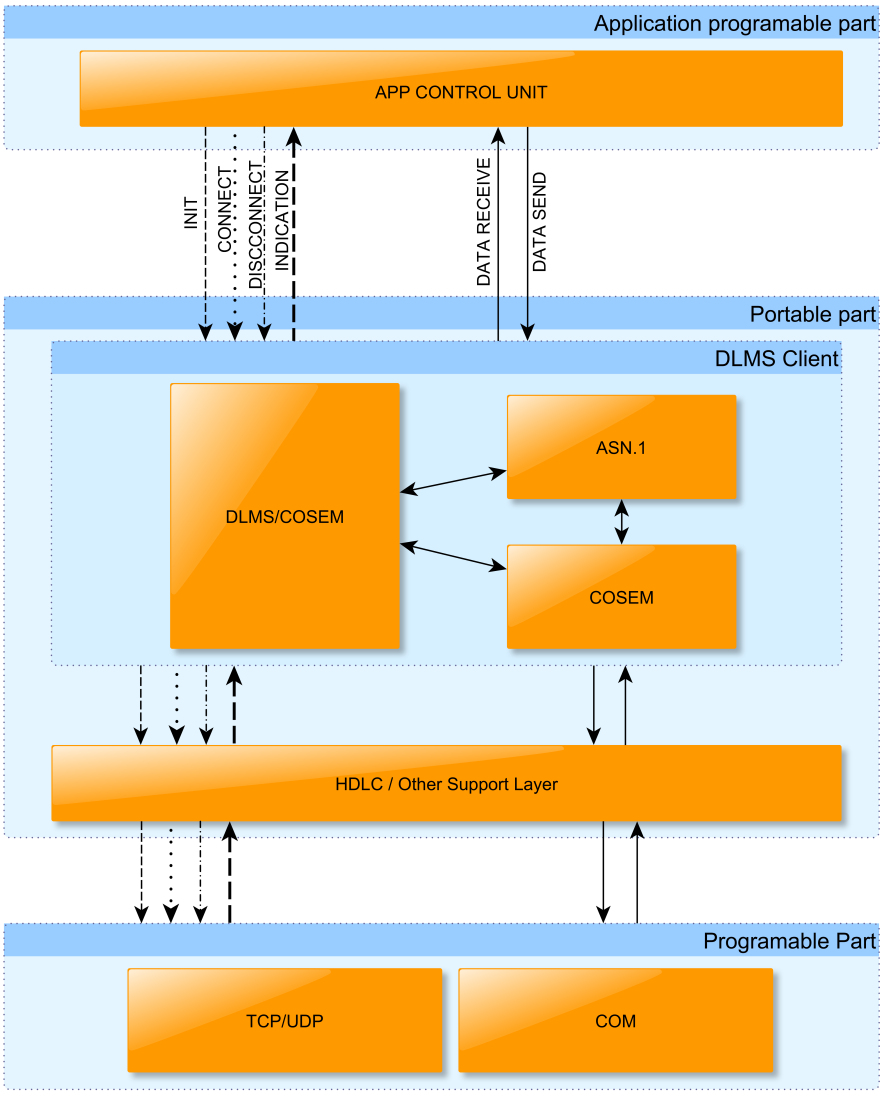
Stejně jako při analýze knihovny HDLC je potřeba přistupovat k analýze frameworku DLMS/COSEM. Předpokládáme, že framework DLMS/COSEM náleží 5 až 7 vrstvě OSI/ISO modelu. Z toho vyplývá, že je nutné přihlížet na návrh frameworku nejen z pohledu uživatele, ale také z pohledu nižších vrstev, logické struktury a vnitřní struktury frameworku.

V tomto případě bude uživatelem nejspíše aplikace, která bude žádat přenos dat na a z cílového zařízení. Po zpracování příkazu od uživatele bude muset framework využít dosavadní rozhraní pro komunikaci s nižší vrstvou. Pro zjednodušení budeme předpokládat, že touto vrstvou je protokol HDLC, který v našem případě podporuje jak přenos po TCP/UDP, tak po COM.

Za druhé je potřeba přihlížet na návrh frameworku z pohledu nižších vrstev. Stejnou funkcionalitu, jakou bude požadovat uživatel po frameworku, bude požadovat i framework po nižší vrstvě. Těmito požadavky je myšlena funkcionalita pro připojení, rozpojení, zasílání a získávání dat a případná indikace změny stavu.

Třetím bodem analýzy je logická struktura frameworku. Nejen začlenění rozhraní pro komunikaci s nižší i vyšší vrstvou, ale také vnitřních spojení logických celků, které zaručují příslušnou funkcionalitu protokolu DLMS/COSEM.

A poslední, čtvrtý, pohled přináší analýza na návrh vnitřní struktury frameworku. Je potřeba zohlednit správné rozdělení logických funkcí mezi třídy, jejich spolupráci, další rozšiřitelnost zdrojového kódu, minimální paměťové nároky a zároveň využitelnou funkcionalitu.



Obr. 23 – Návrh kooperace částí DLMS/COSEM a HDLC

4.3.1 Uživatelský přístup k frameworku

Je nutné specifikovat nejen přístup v rámci komunikace mezi vrstvami, ale musíme brát v potaz i vytvoření instance objektu frameworku DLMS/COSEM. Dále je nutné mít možnost celý objekt parametrizovat, ovládat a případně i debugovat. Výčet příslušných metod, které jsou potřeba pro všechny popsané funkce, je následující:

- Vytvoření instance a přidělení paměti, kterou bude objekt při komunikaci používat.
- Nastavit cílovou a zdrojovou adresu, které budou použity nižšími vrstvami pro navázání spojení a následnou komunikaci s cílovým zařízením.
- Vytvoření a zrušení asociace s ostatními zařízeními
- Požadavek na asociační objekt a správné spojení LN, SN a *class_id*
- Metody pro zápis, čtení a akci nad cílovým objektem
- A na konec nastavení logování pro případné ladění frameworku

4.3.2 Komunikace s nižšími vrstvami

V tomto případě je komunikací s nižší vrstvou myšlena vrstva HDLC. V profilu TCP-UDP/IP je používán TCP wrapper. Praxe je však taková, že tomu tak není a pokaždé je použit protokol HDLC jak pro komunikaci přes sériové rozhraní, tak pro TCP-UDP/IP. Pakety mohou být tím pádem zbaveny hlavičky TCP/UDP a přímo poslány dalšímu zařízení na linii, které je připojeno sériovým rozhraním.

Rozhraní komunikace s touto vrstvou bude ale poměrně univerzální a v dalším rozvoji aplikace může být lehce nahrazeno jiným protokolem. Nižší vrstvě je potřeba před jejím použitím předat velikost bufferu, který bude používat, třídu implementující předdefinované rozhraní TCP/UDP nebo COM a nakonec adresu cílového a zdrojového zařízení. Po úspěšném připojení nižších vrstev může vrstva protokolu DLMS/COSEM vyslat dotaz na asociaci. Z hlediska nižší vrstvy je ale jedno, jaká data mu framework předá. Zajišťuje pouze jejich přenos. Tím pádem budou využity jen metody *send_data()* a *receive_data()*. Při ukončení asociace bude zavolána funkce *disconnect_req()*, která zajistí rozpojení spojení.

4.3.3 Logický model frameworku

Na obrázku 23 je možno vidět začlenění frameworku DLMS/COSEM. Framework je celá skupina logických bloků nazvaných DLMS Client. Logické bloky pak plní funkci, která vyplývá z definice protokolu DLMS/COSEM, COSEM rozhraní tříd nebo dalších jako např. ASN.1. [3][7]

4.3.4 Vnitřní struktura frameworku

Hlavní třídou bude **CTs_DLMS_Client**, která bude obsahovat třídy nižších vrstev. V tomto případě **CTs_HDLC_Client**. **CTs_DLMS_Client** se bude starat o připojení nižších vrstev, navázání asociace, odesílání dat, přijímání dat a rozpojení asociace. Pomocnou třídou bude **CTs_HDLC_Core**, která bude zodpovědná za kódování a dekódování přijatých a odesílaných dat. Bude také spolupracovat se skupinami tříd **CTs_BER** a **CTs_AXDR**, které zajistí správné formování a dekódování dat dle ASN.1.

Úlohou **CTs_BER** bude práce s PDU asociačních služeb tj. zakódování paketu AARQ a RLRQ a dekódování paketu AARE a RLRE. Třída **CTs_BER** bude rodičovskou třídou, od které budou dědit nejdříve primitivní typy a od nich následně typy definované protokolem DLMS/COSEM.

Stejnou úlohu, jakou plní skupina tříd **CTs_BER** pro práci s daty, bude plnit i skupina tříd, které dědí od **CTs_AXDR** s tím rozdílem, že tato skupina bude mít na starosti služby pro datové přenosy tj. GET, SET, Read a Write.

Poslední hierarchií tříd, která bude použita, jsou třídy rozhraní COSEM. Tento systém tříd bude obsahovat rodičovskou třídu **CTs_COSEM_ObjectBase**. Od této třídy pak budou dědit všechny implementované třídy rozhraní COSEM. Třídy poděděné od **CTs_COSEM_ObjectBase** budou sloužit jak pro uchování hodnoty, tak pro dekódování datových struktur, které obsahují. Vzhledem k požadavku na přenositelnost i na vestavěná zařízení zde nastává problém. Některé tyto třídy totiž mohou obsahovat složité struktury, v nich velké množství dat a na vestavěných systémech tak velké množství paměti není obvyklé. Pokud budeme uvažovat, že dané systémy plní i další funkce a působí například jako datové koncentrátory, je potřeba používanou paměť snížit na minimum. Tímto redukováním je možné omezit tyto systémy jen do té míry, aby plnily svoji požadovanou funkci a neubíraly na vlastní využitelnosti víc, než

je třeba. Tím je myšlen už výběr objektů, které bude uživatel chtít ze zařízení získávat. Dále je nutné alokovat paměť až v momentě, kdy je opravdu nutná. Vzhledem k použití tohoto frameworku hlavně pro měření elektrických veličin budou v první fázi vybrány jen takové třídy rozhraní COSEM, u kterých se předpokládá, že se velikost přijatých dat v další iteraci žádostí nebude měnit. Tím je myšleno, že pokud přijdou první data o velikosti x daného atributu dané třídy, může se předpokládat, že další příjem nebude větší. Pokud by tato situace nastala, budou muset být data ignorována. Takové třídy, u kterých se předpokládá stejná velikost, jsou například *Register*, *Extended Register*, *Clock* a další.

Třídy, které budou v rámci této práce implementovány jsou *Register* a *Clock*. Na základě těchto dvou tříd a jednoduše rozšiřitelné výsledné implementace bude moci být framework v případě potřeby rozšířen o další třídy.

4.3.5 Návrh použití frameworku

Asociace

Aby mohl uživatel využívat služeb frameworku, bude nutné po vytvoření třídy zavolat metodu asociace. Tato metoda se bude starat o spojení všech nižších vrstev a následně zařídí vytvoření a odeslání AARQ paketu. Po odeslání vyčká na příjem dat z cílového zařízení a přijatá data zpracuje, vyhodnotí zda-li se jedná o AARE a pokud ano, zpracuje jeho obsah. Výsledkem operace asociace je buď spojeno nebo nespojeno.

Získání asociačního objektu a spárování jmen

Jakmile bude uzavřena asociace s cílovým zařízením, může být získán asociační objekt. Získání tohoto objektu není povinné, ale v případě neznalosti SN a tříd rozhraní COSEM je to nutné. Parametry této metody budou LN názvy, pro které chceme získat informace a prázdný spojový seznam **CTs_COSEM_ObjectBase**. Klient sestaví dotaz na asociační objekt a ten odešle na cílové zařízení. Po obdržení odpovědi páruje přišlá LN s požadovanými LN a při shodě vytvoří (dynamicky) a vloží objekt se správným id do spojového seznamu. Takto vytvořený spojový seznam pak vrátí jako výstupní parametr a uživatel má možnost dotazovat se na atributy objektů v seznamu, kde budou zároveň uchovány. Velikost tohoto seznamu je předem odvoditelná z počtu tříd, které uživatel požaduje a z velikosti jednotlivých objektů, atributů a metod. I přes to, že je asociační objekt třídou rozhraní COSEM, nebude v paměti uchován.

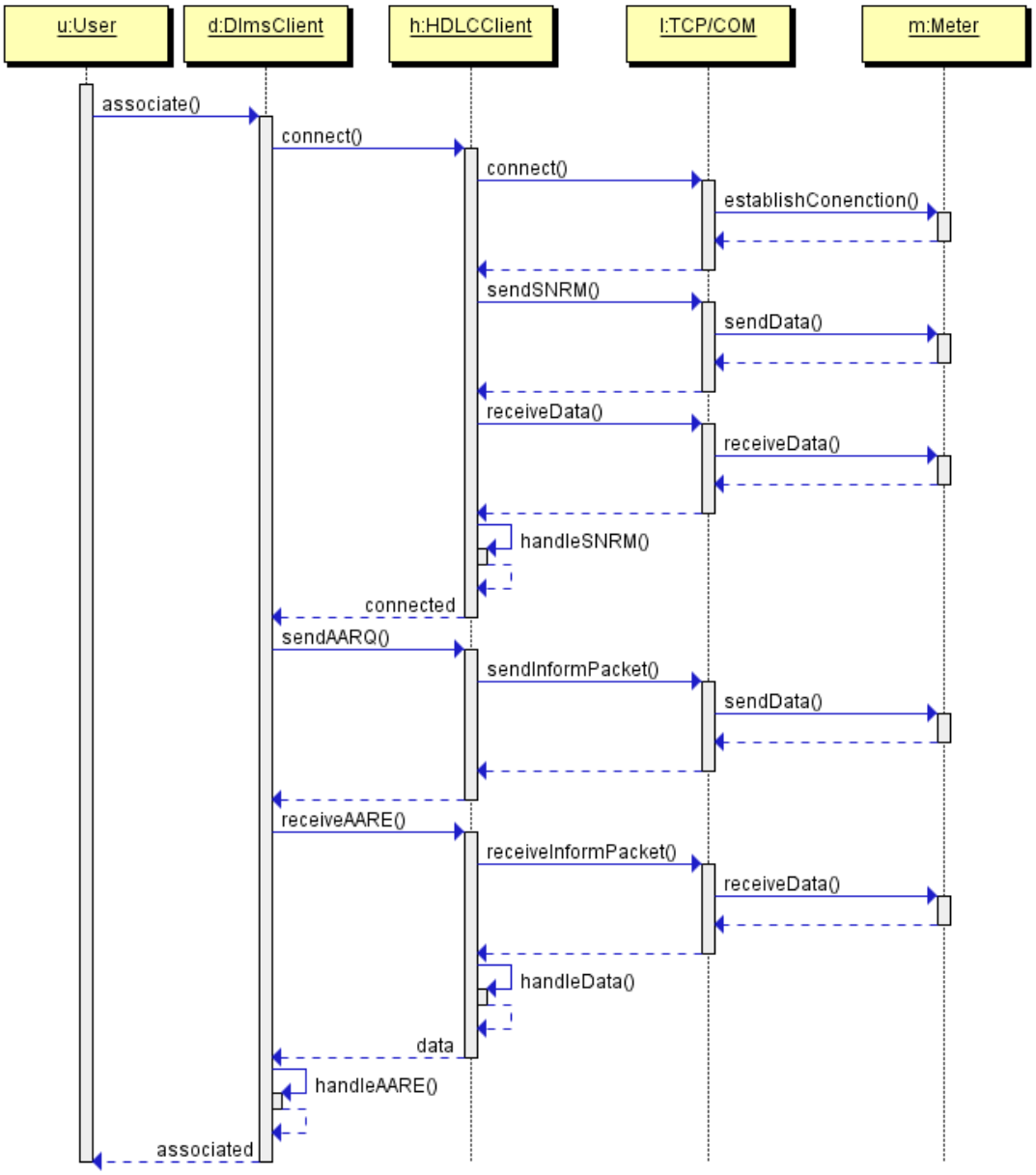
Čtení

Čtení ze zařízení bude probíhat podobně jako získání asociačního objektu. Vstupem do metody budou LN nebo SN názvy s indexem atributu, který chceme přečíst a spojový seznam objektů, kam se budou přišlá data ukládat.

Pro čtení ze zařízení klient sestaví požadavek Read nebo GET dle pole s názvy a příslušným indexem atributu. Takto sestavený požadavek odešle na cílové zařízení a čeká na odpověď. Po obdržení odpovědi data dekóduje a vloží hodnoty do správného objektu ke správném atributu.

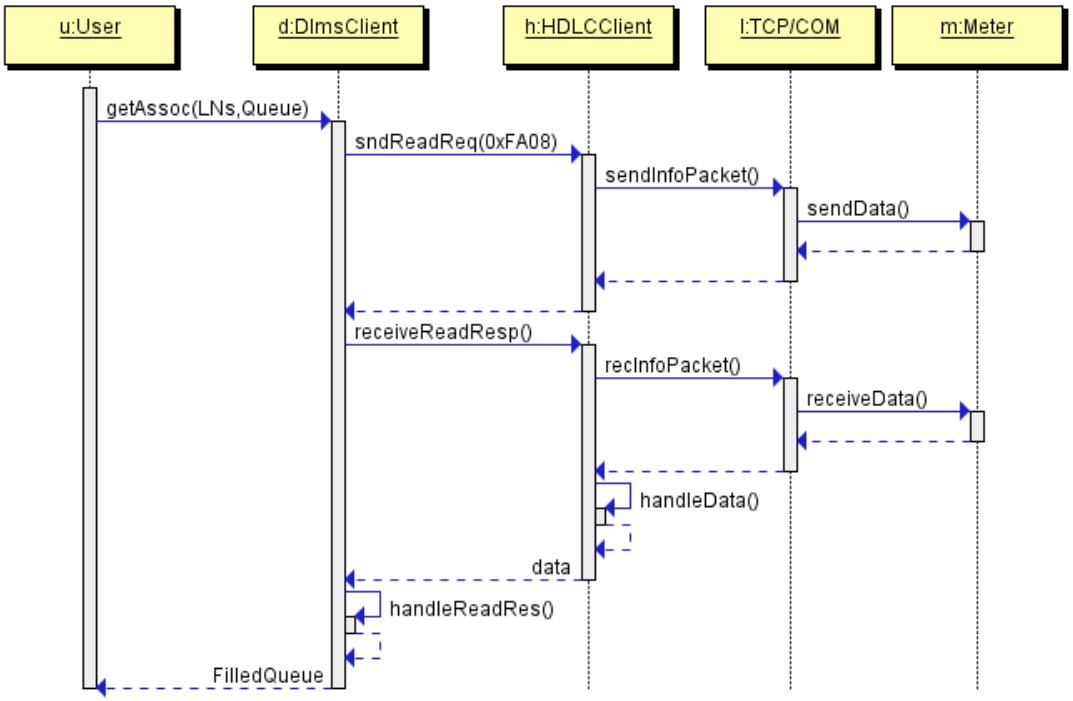
Zápis

Zápis je analogií ke čtení. Vstupem bude opět pole s názvy objektů a spojový seznam objektů, kde již budou uložena data k zápisu na cílové zařízení. Klient sestaví požadavek



Obr. 24 – DLMS/COSEM - Průběh při zavolání asociace

Write nebo SET. Do požadavku vloží data z objektů dle názvů a seznamu objektů. Tento požadavek následně odešle a čeká na potvrzení, které vyhodnotí a vrátí výsledek uživateli.



Obr. 25 – DLMS/COSEM - Průběh při získávání asociačního objektu

5 Implementace

5.1 HDLC

Výsledná hierarchie tříd je vidět na obrázku 26, kde byly pro přehlednost smazány metody a ponechány jen nejdůležitější atributy. Pomocná hierarchie, která slouží pro zpracování a přenos paketů je znázorněna na obrázku 27.

Jako první vznikla třída **CTs_HDLC_client**, která „obaluje“ vrstvy MAC a LLC protokolu HDLC. Tato třída se stará o volání příslušných metod z podtříd **CTs_HDLC_MAC** a **CTs_HDLC_LLC**. Hlavní metody třídy **CTs_HDLC_client** jsou:

- **CTs_HDLC_client(CTs_HDLC_TCP *_tcp, uint32_t _recBuffSize);**
  – Konstruktor třídy, která bude využívat TCP/UDP a vytvoří si vlastní buffer o velikosti *_recBuffSize*

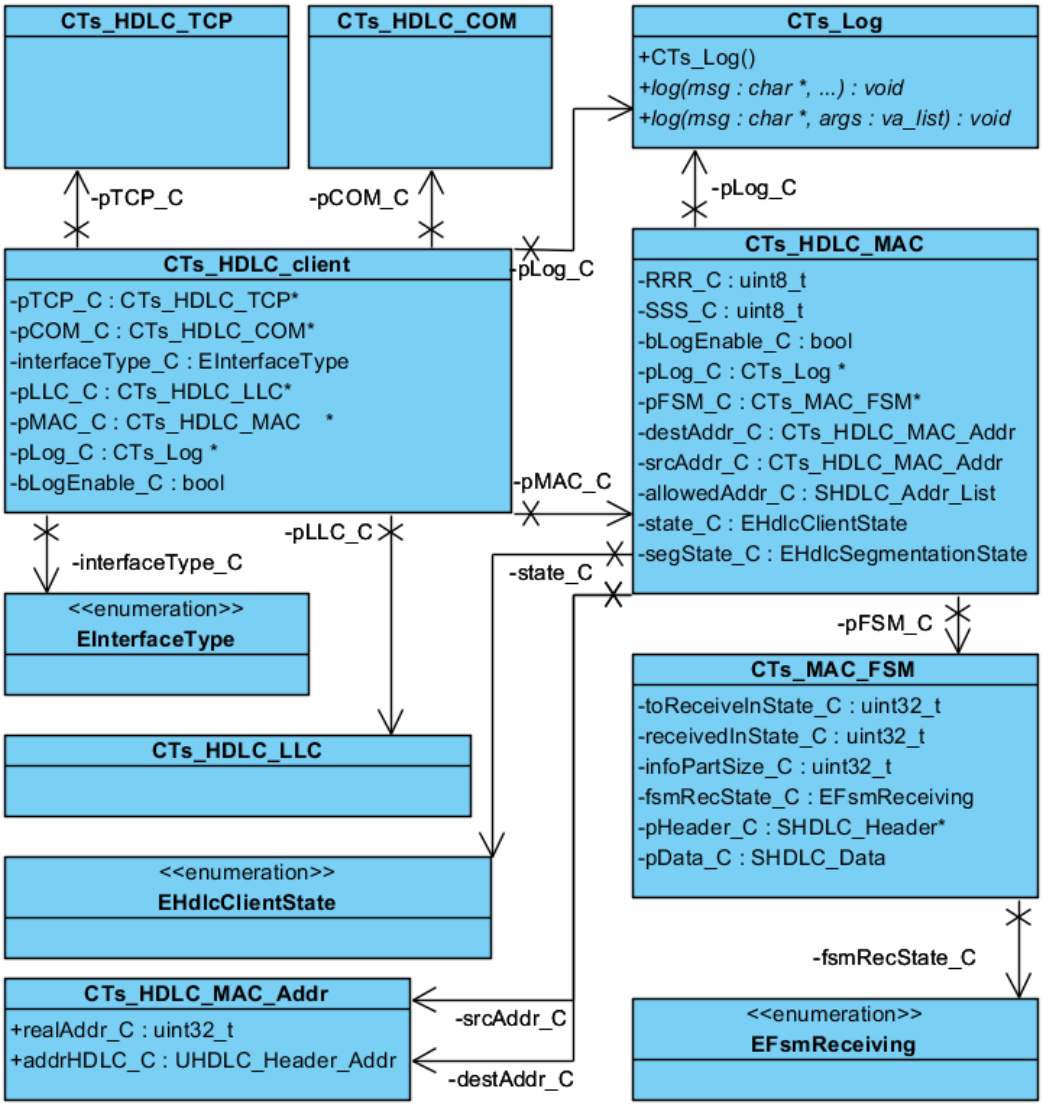
- **CTs_HDLC_client(CTs_HDLC_COM *_tcp, uint32_t _recBuffSize);**
  – Konstruktor třídy, která bude využívat COM a vytvoří si vlastní buffer o velikosti *_recBuffSize*.

- **bool connect_req(uint8_t _srcAddr, uint32_t _destAddr);**
  – Je metoda, která nepřipojenou nižší vrstvu (TCP/UDP, COM) připojí a která využívá třídy **CTs_HDLC_MAC** pro formování SNRM paketu. Po odeslání SNRM paketu čeká tato metoda na přijetí dat a zpracování paketu, ke kterému opět využívá podtřídu **CTs_HDLC_MAC** a která paket zkontroluje a zpracuje. Návratová hodnota metody je pak použita jako výsledek operace připojení. Metoda **connect_req()** využívá svoje parametry k adresaci cílového a zdrojového zařízení. Pokud je použit model s logickou a fyzickou adresou serveru, musí být tato adresa zformována a předávána jako vstupní parametr.

- **bool disconnect_req();**
  – Metoda analogická k metodě **connect_req()**. Pro formování **DISC** paketu využívá třídy **CTs_HDLC_MAC** a po odeslání čeká na přijetí paketu, který je opět zpracován podtřídou **CTs_HDLC_MAC**.

- **bool send_data(uint8_t *_data, uint32_t _dataLen);**
  – Slouží pro odesílání dat skrz vrstvu HDLC. Tato metoda využívá jak vrstvu LLC, jejíž funkce implementuje třída **CTs_HDLC_LLC**, tak vrstvu MAC. Třída **CTs_HDLC_LLC** připojí vlastní hlavičku a dále jsou data pomocí segmentačního algoritmu předávána třídě **CTs_HDLC_MAC**, která vytváří informační pakety, které se následně mohou odeslat.

- **EHDLC_Error_Codes receive_data(uint8_t *_data, uint32_t _dataLen, uint8_t **_upData, uint32_t *_upLen);**
  – Tato metoda je určena pro příjem dat z připojeného zařízení. Průběh příjmu probíhá dotazováním jednoho bytu z nižší vrstvy a ten je předkládán třídě **CTs_HDLC_MAC** pro zpracování. Jakmile třída oznámí úspěšné přijetí paketu, je paket zkontrolován a zpracován. Při chybě zpracování dat je metoda ukončena neúspěchem pomocí výčtu *EHDLC_Error_Codes*.

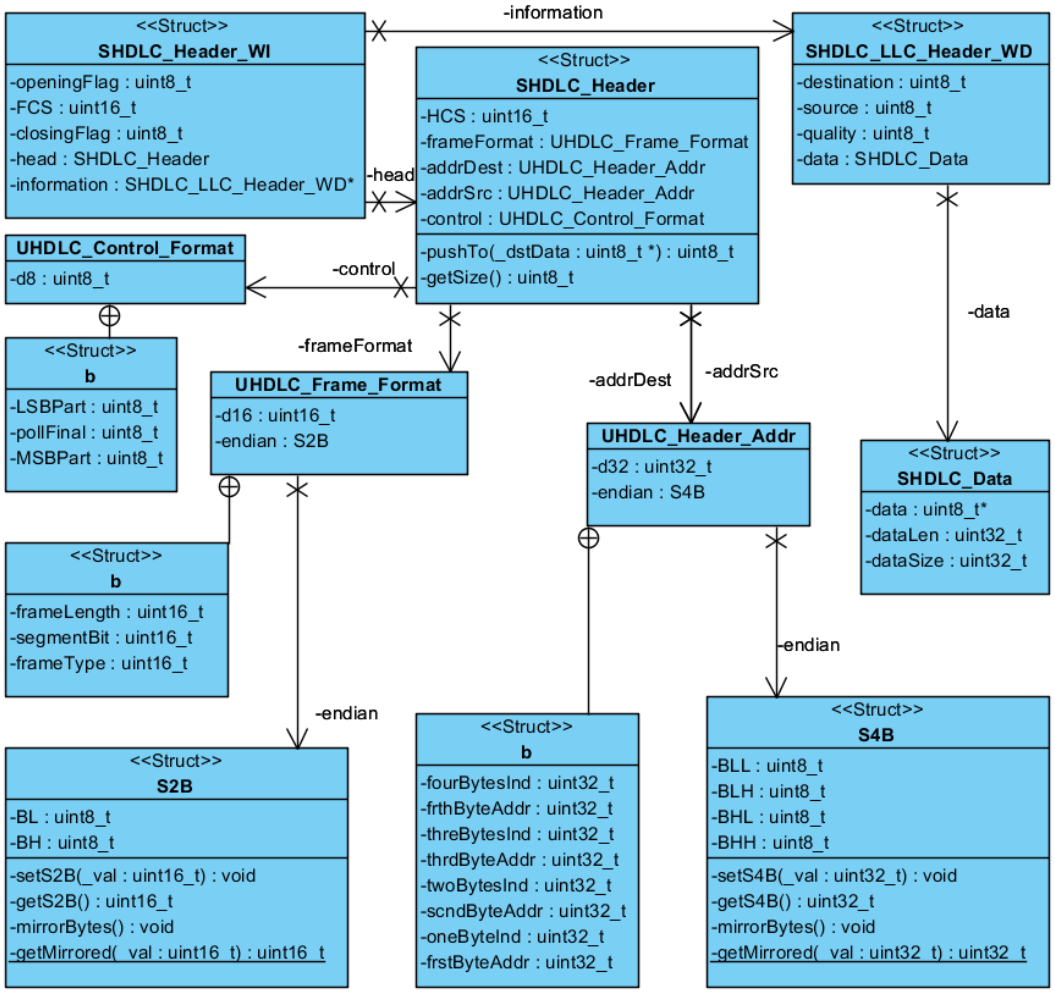


Obr. 26 – Základní třídy HDLC knihovny

Zároveň s třídou **CTs_HDLC_client** vznikala i třída **CTs_HDLC_MAC**. Tato třída se stará hlavně o formování rámců, a to jak odchozích, tak příchozích. Kontroluje, jestli jsou přijaté rámce správně formovány, jestli je kontrolní součet rámce správný a jestli všechny části rámce mají očekávanou hodnotu. Podobnou úlohu plní také u odesílaných rámců, kde daný rámec skládá, příslušným polím přiřazuje žádoucí hodnoty

a nakonec počítá kontrolní součty HCS a případně i FCS.

Hierarchii zobrazenou na obrázku 27 využívá hlavně tato třída. Pomocí ní může třída přistupovat nejen k daným polím rámce, ale i příslušným bitům aniž by příliš používala bitové operace *OR*, *AND*, *XOR* a případně i bitové posuny. Pomáhá jí také měnit endianitu při konečném odesílání paketu.



Obr. 27 – Podpůrná hierarchie HDLC frameworku

Důležité metody třídy **CTs_HDLC_MAC** jsou:

- **bool createSNRM(SHDLC_Header_WOI_Compl *_head);**
  – Metoda, která vytvoří SNRM rámec pro spojení s cílovým zařízením a nastaví stav HDLC_CONNECTING.

- **bool createDISC(SHDLC_Header_WOI_Compl *_head);**
  – Metoda, která vytvoří DISC rámec pro rozpojení spojení s cílovým zařízením a nastaví stav HDLC_DISCONNECTING.

- **bool createI(SHDLC_Header_WI *_packet, const SHDLC_LLC_Header_WD & _information, bool _PFbit, bool _segmentation);**
  – Metoda, která vytvoří I rámec pro odeslání části dat na cílové zařízení.

- **EHDLC_Error_Codes receiveData(uint8_t _byte);**
  – Metoda, která byte předaný v argumentu předá podtřídě **CTs_MAC_FSM** a ta ho s prvotní kontrolou přiřadí na správnou pozici do struktury **SHDLC_Header_WI**.

- **EHDLC_Error_Codes handlePacket(bool *_sendRR);**
  – Metoda, která zkontroluje a zpracuje připravený paket. Do vstupně výstupního parametru *_sendRR* uloží, zda-li se má, či nemá poslat odpověď.

- **EHdlcClientState getState();**
  – Metoda vrací aktuální stav připojení.

Dále třída obsahuje privátní metody, které používá ke kontrole a formování rámců.

- Metody pro zjištění typu rámce:
  - **bool isIControl();**
  - **bool isRRControl();**
  - **bool isRNRControl();**
  - **bool isSNRMControl();**
  - **bool isDISCControl();**
  - **bool isUAControl();**
  - **bool isDMControl();**
  - **bool isFRMRControl();**
  - **bool isUIControl();**
- Metody pro nastavení polí v rámci
  - **bool setFrameFormat();**
  - **bool setAddr();**
  - **bool setControl();**
  - **bool setInformation();**
  - **bool setCheckSequence();**
- Metody pro kontrolu polí v rámci
  - **bool checkCheckSequence();**
  - **bool checkDestAddr();**
  - **bool checkSrcAddr();**
  - **bool checkInformationFCS();**
  - **EHDLC_Error_Codes checkControl();**
  - **EHDLC_Error_Codes checkFrameFormat();**

5.2 DLMS/COSEM

Výsledná hierarichie tříd je rozdělena do tří podmnožin. První podmnožinou jsou třídy **CTs_DLMS_Client** a **CTs_DLMS_Core**, které se starají o dialog se vzdáleným zařízením. Druhá podmnožina jsou třídy rozhraní COSEM, kde předkem tříd je **CTs_COSEM_ObjectBase** a všechny třídy rozhraní dědí právě od této třídy. Tato

podmnožina je zodpovědná za formátování přenášených dat a k jejich uchování v paměti, k čemuž slouží třída **CTs_COSEM_VarVal**. Třetí podmnožinou jsou třídy **CTs_AXDR** a **CTs_BER**, které se starají o kódování a dekódování dat jako takových.

5.2.1 DLMS

Třída **CTs_DLMS_Client** disponuje dvěma konstruktory, které dle zvoleného rozhraní (sériové či Ethernet) vytvoří potřebnou instanci HDLC vrstvy. Dalšími dvěma důležitými parametry konstruktorů jsou velikosti bufferů, které se vytvoří jak pro HDLC, tak pro DLMS. Tyto buffery slouží pro odesílání a přijímání dat a je na uživateli, jak velké je zvolí. Doporučením je volit HDLC buffer minimálně 128B, tato velikost vychází z normy [3], a velikost DLMS bufferu dle počtu objektů na cílovém zařízení. Jeden objekt odpovídá 18B. Zařízení, které bylo použito, mělo 323 objektů a přišlá data zabírala 5821B. Dále je v konstruktoru vytvořen objekt **CTs_DLMS_Core**, který slouží pro zpracování APDU a který je hojně využíván.

Klientská třída poskytuje uživateli následující metody, kterými lze třídu ovládat.

- **bool associate();**
  – Metoda, která je zodpovědná za připojení instancí tříd nižších vrstev a za navázání asociace.
- **bool release();**
  – Metoda, která je zodpovědná za ukončení spojení s připojeným zařízením.
- **bool getAssociation(const SDLMS_Names& names, CTs_COSEM_ObjQueue *queue);**
  – Metoda, která je zodpovědná za získání asociačního objektu a naplnění spojového seznamu validními třídami rozhraní COSEM.
- **bool readObject(const SDLMS_Names& names, CTs_COSEM_ObjQueue *queue);**
  – Metoda, která z připojeného zařízení přečte požadované objekty a jejich hodnoty uloží do spojového seznamu.
- **bool writeObject(const SDLMS_Names& names, CTs_COSEM_ObjQueue *queue);**
  – Metoda, která je zodpovědná za zápis dat na cílové zařízení.

Připojení

Průběh připojení je znázorněn na obrázku 24. Před zavoláním metody pro připojení, je nutné nastavit HDLC adresy cílového a zdrojového zařízení zavoláním metody *CTs_DLMS_Client::setHDLCAddress()*. Této metodě se parametry předává klientská adresa, fyzická adresa serveru a logická adresa serveru. Samotné připojení začíná zavoláním metody *CTs_DLMS_Client::associate()*. Tato metoda zjišťuje stav připojení nižších vrstev a pokud nejsou připojeny, vyvolá jejich připojení. Po připojení nižších vrstev se zavolá metoda *CTs_DLMS_Core::assembleAARQ()* pomocné třídy **CTs_DLMS_Core**. Tělo této metody za pomoci skupiny tříd **CTs_BER** vytvoří AARQ request. Při úspěšném vytvoření je požadavek uložen ve společném bufferu pro odesílání a přijímání a buffer je dále předán nižším vrstvám, které se postarají o jeho odeslání. Po přijetí dat, je volána metoda *CTs_DLMS_Core::handleData()*, která data zpracuje a vyhodnotí, jestli je AARE odpověď v pořádku. Pokud je v pořádku, přepne

aplikaci do stavu DLMS_ASSOCIATED. Pokud není v pořádku, zůstává stále ve stavu DLMS_IDLE a vrátí neúspěch připojení.

Získání asociačního objektu

Získání asociačního objektu je časově i paměťově náročná operace. Data asociačního objektu se na sériovém rozhraní přijímají ze serveru řádově desítky vteřin a na Ethernetu až 5 vteřin. Velikost asociačního objektu je zařízení od zařízení různá a nutno počítat s 18B na 1 objekt v zařízení. Tento asociační objekt je potřeba pouze jednou, proto se v paměti neuchovává a jako dočasné úložiště je využíván přijímací buffer vrstvy DLMS.

Parametry funkce *CTs_DLMS_Client::getAssociation()*, která zajišťuje získání asociačního objektu, jsou struktura **SDLMS_Names** obsahující pole s LN a objekt třídy **CTs_COSEM_ObjQueue**, kam se po nalezení shody budou vkládat dynamicky vytvořené objekty tříd rozhraní COSEM.

Při volání této metody je nejprve zkontrolováno spojení nižších vrstev, poté je zavolána metoda *CTs_DLMS_Core::assembleASSOC()*, která za pomocí skupiny třídy **CTs_BER** vytvoří Read dotaz na asociační objekt a vloží do něj všechna LN ze struktury **SDLMS_Names**. Po úspěšném sestavení Read dotazu jsou data předána nižším vrstvám, které se postarají o odeslání na připojené zařízení. Po obdržení dat, je odpověď předána metodě *CTs_DLMS_Core::handleData()*, která naplní spojový seznam třídami rozhraní COSEM na základě požadovaných a přijatých LN.

```
/// Naming structure
struct SDLMS_Names{
    uint8_t     **  lnArr;  ///< Long Name array
    uint16_t    *   snArr;  ///< Short Name array
    uint8_t     *   index;  ///< Array of indexes of
       attributes
    uint16_t        num;    ///< Number of names/index in
       array
    ESnLn           ref;    ///< Use short names or long
       names
};

/// Queue
class CTs_COSEM_ObjQueue {
public:
    CTs_COSEM_ObjQueue();
    virtual ~CTs_COSEM_ObjQueue();

    /// Push _obj to the end of queue
    void     pushObject( CTs_COSEM_ObjectBase *_obj );
    /// get number of object in queue
    uint16_t getNumObjs(){ return numObjs_C; };
    /// ... other methods
private:
    struct STs_COSEM_Node{
        CTs_COSEM_ObjectBase    *obj;
        STs_COSEM_Node          *next;
    };
```

```
    /// pointer to first object
    STs_COSEM_Node *root;
    /// number of objects in queue
    uint16_t    numObjs_C;
};
```

Čtení

Čtení ze zařízení je de facto zobecněné získání asociačního objektu. Probíhá zavoláním metody *CTs_DLMS_Client::readObject()*, jejíž parametrem je opět struktura **SDLMS_Names** a seznam objektů **CTs_COSEM_ObjQueue**. Průběh se ale liší. Zavoláním metody *CTs_DLMS_Core::assembleReadReq()* se vytvoří požadavek Read tak, že se z pole se jmény vyberou ta jména, která mají i svoji třídu v seznamu **CTs_COSEM_ObjQueue** a vloží se do požadavku. Data požadavku se předají nižším vrstvám a ty zajistí jejich přenos na cílové zařízení. Přijatou odpověď zpracuje metoda *CTs_DLMS_Core::handleReadResponse()*, která vyhodnotí přijatá data a přiřadí hodnoty atributům v objektech.

5.2.2 COSEM

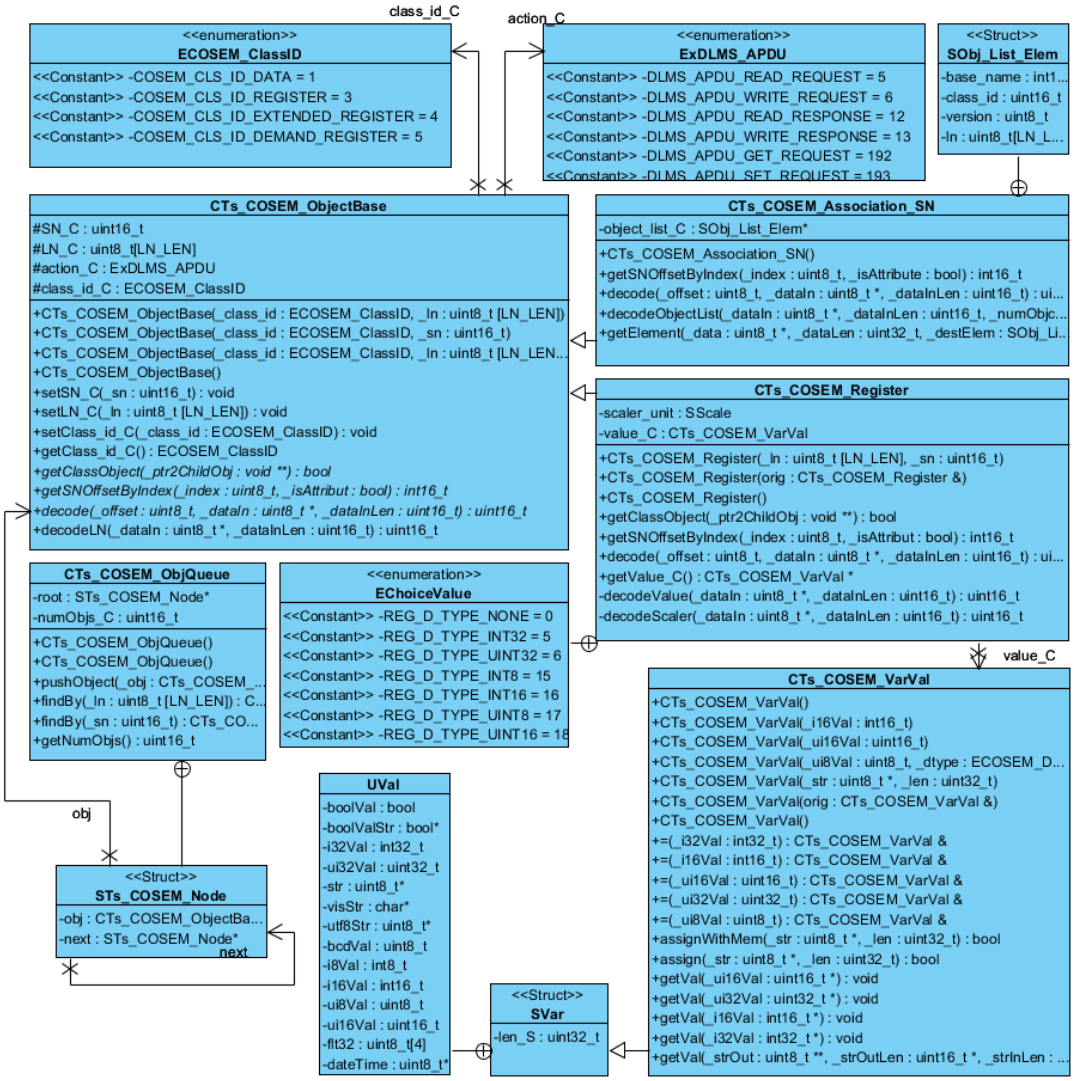
Na obrázku 28 je znázorněna hierarchie tříd rozhraní COSEM. Základní třídou je **CTs_COSEM_ObjectBase**, od které dědí všechny ostatní třídy rozhraní. Pro přehlednost byly do obrázku vybrány jen některé metody a atributy. Demonstrací využití je třída **CTs_COSEM_Register**, která dědí od třídy **CTs_COSEM_ObjectBase**, implementuje čistě virtuální metody i metody specifické pro její atributy. Jelikož třída **CTs_DLMS_Client** vyžaduje spojový seznam **CTs_COSEM_ObjQueue**, pak pro přístup ke specifickým metodám každé třídy rozhraní je nutné implementovat čistě virtuální metodu *getClassObject((void**)_ptr2ChildObj);*, která je zodpovědná za dynamický downcasting. Jednoduše řečeno je to metoda, která vrátí do výstupního parametru *_ptr2ChildObj* ukazatel na sebe.

V této skupině tříd lze nalézt i třídu **CTs_COSEM_VarVal**, která slouží pro univerzální uchování hodnot. Tato třída je potomkem struktury **SVar**, která obsahuje *union* se všemi datovými typy použitými v třídách rozhraní COSEM. Dále obsahuje velikost a specifikaci, o který datový typ se jedná. Potomek (**CTs_COSEM_VarVal**) této struktury implementuje přetížené konstruktory, přetížené operátory a přetížené funkce pro vytváření a uchování různých datových typů ze struktury **SVar**. Třída **CTs_COSEM_VarVal** je libovolně použitelná jako datový typ atributu v třídách rozhraní COSEM a není nutné pro každou třídu rozhraní vytvářet podobnou datovou strukturu. To je výhodné obzvlášť, pokud není dopředu jasné, jaký datový typ třída rozhraní COSEM uchovává.

5.2.3 ASN.1

Skupina tříd ASN.1 slouží pro kódování a dekódovaní datových struktur DLMS/COSEM. Tato skupina je rozdělena na dvě podskupiny.

Třídy **CTs_BER** jsou použity pro kódování a dekódování APDU. Byly vytvořeny třídy **CTs_ASN_AARQ**, **CTs_ASN_AARE**, **CTs_ASN_RLRQ** a **CTs_ASN_RLRE**. Z názvu je patrné, o které APDU se jedná. Každá z těchto tříd obsahuje atributy dle jejich definice v normě [7]. Každý z těchto implementovaných atributů je třída, která dědí od základního datového typu ASN.1 (BER).



Obr. 28 – Třídy rozhraní COSEM

Třídy **CTs_AXDR** jsou použity pro kódování a dekódování xDLMS ASE. Skupina obsahuje třídy, jako jsou např. **CTs_AXDR_ReadRequest** nebo **CTs_AXDR_ReadResponse**.

6 Testy

Jako první byla testována knihovna HDLC a následně framework DLMS/COSEM. Pro testování bylo k dispozici měřící zařízení LandisGyr+ E650.

6.1 HDLC

Pro testování HDLC byla vytvořena jednoduchá demo aplikace jako serverová protistrana, která simuluje HDLC komunikaci. Knihovna byla testována jak na operačním systému MS Windows 64b i 32b ve verzi 7 a 10, tak na operačním systému Linux, a to konkrétně na Ubuntu 64b i 32b 14.0.4. Jelikož byla aplikace vyvíjena na operačním systému Windows, byly očekávány problémy při přechodu na operační systém Linux. Při kompilaci se však žádné problémy neobjevily a aplikace byla zkompilována bez dalších chyb a varování. Pomocná debugovací aplikace Valgrind ale odhalila neinicializované proměnné a tzv. „stack smashing“, který byl ihned opraven.

Následující testy byly provedeny jak na OS Windows, tak na OS Linux.

6.1.1 Testování proti vlastnímu SW

První testy probíhaly proti demo aplikaci tsDLMSsim, která se chová jako localhost server. Aplikace má jednoduché rozpoznávání přijetí paketu a na základě přijatých dat, odešle příslušnou odpověď. Jakmile byly odladěny chyby na této aplikaci, bylo možné přejít na testování s reálným zařízením. Výstup z testování vlastním SW je následující:

```
Starting HDLC (DLMS) simulator
0 TCP Listening
accepted connection
Received 12B
Packet Length:12

----------------------
----------------------
Packet Header Info
Flag:0x7E
Frame Format:0xA0 0x0A
d.Adr:0x258E3
s.Adr:0x0021
C:0x93
HCS:0x4C 0x4B
F:0x7E
----------------------
DETAILS: S:0 P/F:1 len:10 RRR:4 SSS/0:3
----------------------
SNRM received -> UA sending
----------------------
```

```
----------------------
Received 14B
Received 34B
Packet Length:48

----------------------
----------------------
Packet Header Info
Flag:0x7E
Frame Format:0xA0 0x2E
d.Adr:0x258E3
s.Adr:0x0021
C:0x10
HCS:0x2A 0xF1

Information: 0xE6 0xE6 0x00 0x60 0x1D 0xA1
0x09 0x06 0x07 0x60 0x85 0x74 0x05 0x08
0x01 0x02 0xBE 0x10 0x04 0x0E 0x01 0x00
0x00 0x00 0x06 0x5F 0x1F 0x04 0x00 0x1C
0x03 0x20 0xFF 0xFF
FCS:0x80 0xAB
Flag:0x7E
----------------------
DETAILS: S:0 P/F:1 len:46 RRR:0 SSS/0:0
----------------------
AARQ received -> AARE sending
----------------------
----------------------
Received 14B
Received 61B
Received 3B
Packet Length:78

----------------------
----------------------
Packet Header Info
Flag:0x7E
Frame Format:0xA8 0x4C
d.Adr:0x258E3
s.Adr:0x0021
C:0x22
HCS:0x39 0x5D

Information: 0xE6 0xE6 0x00 0x00 0x01 0x02
0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A
0x0B 0x0C 0x0D 0x0E 0x0F 0x10 0x11 0x12
0x13 0x14 0x15 0x16 0x17 0x18 0x19 0x1A
0x1B 0x1C 0x1D 0x1E 0x1F 0x20 0x21 0x22
0x23 0x24 0x25 0x26 0x27 0x28 0x29 0x2A
0x2B 0x2C 0x2D 0x2E 0x2F 0x30 0x31 0x32
```

```
0x33 0x34 0x35 0x36 0x37 0x38 0x39 0x3A
0x3B 0x3C
FCS:0x57 0x3E
Flag:0x7E
----------------------
DETAILS: S:1 P/F:0 len:76 RRR:1 SSS/0:2
----------------------
Nothing processed
----------------------
----------------------
Received 78B
Packet Length:78

----------------------
----------------------
Packet Header Info
Flag:0x7E
Frame Format:0xA8 0x4C
d.Adr:0x258E3
s.Adr:0x0021
C:0x22
HCS:0x39 0x5D

Information: 0x3D 0x3E 0x3F 0x40 0x41 0x42
0x43 0x44 0x45 0x46 0x47 0x48 0x49 0x4A
0x4B 0x4C 0x4D 0x4E 0x4F 0x50 0x51 0x52
0x53 0x54 0x55 0x56 0x57 0x58 0x59 0x5A
0x5B 0x5C 0x5D 0x5E 0x5F 0x60 0x61 0x62
0x63 0x64 0x65 0x66 0x67 0x68 0x69 0x6A
0x6B 0x6C 0x6D 0x6E 0x6F 0x70 0x71 0x72
0x73 0x74 0x75 0x76 0x77 0x78 0x79 0x7A
0x7B 0x7C
FCS:0xB3 0xE7
Flag:0x7E
----------------------
DETAILS: S:1 P/F:0 len:76 RRR:1 SSS/0:2
----------------------
Nothing processed
----------------------
----------------------
Received 78B
Packet Length:78

----------------------
----------------------
Packet Header Info
Flag:0x7E
Frame Format:0xA8 0x4C
d.Adr:0x258E3
s.Adr:0x0021
```

```
C:0x22
HCS:0x39 0x5D

Information: 0x7D 0xFF 0x7F 0x80 0x81 0x82
0x83 0x84 0x85 0x86 0x87 0x88 0x89 0x8A
0x8B 0x8C 0x8D 0x8E 0x8F 0x90 0x91 0x92
0x93 0x94 0x95 0x96 0x97 0x98 0x99 0x9A
0x9B 0x9C 0x9D 0x9E 0x9F 0xA0 0xA1 0xA2
0xA3 0xA4 0xA5 0xA6 0xA7 0xA8 0xA9 0xAA
0xAB 0xAC 0xAD 0xAE 0xAF 0xB0 0xB1 0xB2
0xB3 0xB4 0xB5 0xB6 0xB7 0xB8 0xB9 0xBA
0xBB 0xBC
FCS:0x45 0x9D
Flag:0x7E
----------------------
DETAILS: S:1 P/F:0 len:76 RRR:1 SSS/0:2
----------------------
Nothing processed
----------------------
----------------------
Received 78B
Packet Length:78

----------------------
----------------------
Packet Header Info
Flag:0x7E
Frame Format:0xA8 0x4C
d.Adr:0x258E3
s.Adr:0x0021
C:0x22
HCS:0x39 0x5D

Information: 0xBD 0xBE 0xBF 0xC0 0xC1 0xC2
0xC3 0xC4 0xC5 0xC6 0xC7 0xC8 0xC9 0xCA
0xCB 0xCC 0xCD 0xCE 0xCF 0xD0 0xD1 0xD2
0xD3 0xD4 0xD5 0xD6 0xD7 0xD8 0xD9 0xDA
0xDB 0xDC 0xDD 0xDE 0xDF 0xE0 0xE1 0xE2
0xE3 0xE4 0xE5 0xE6 0xE7 0xE8 0xE9 0xEA
0xEB 0xEC 0xED 0xEE 0xEF 0xF0 0xF1 0xF2
0xF3 0xF4 0xF5 0xF6 0xF7 0xF8 0xF9 0xFA
0xFB 0xFC
FCS:0x7A 0xF5
Flag:0x7E
----------------------
DETAILS: S:1 P/F:0 len:76 RRR:1 SSS/0:2
----------------------
Nothing processed
----------------------
----------------------
```

```
Received 17B
Packet Length:17

----------------------
----------------------
Packet Header Info
Flag:0x7E
Frame Format:0xA0 0x0F
d.Adr:0x258E3
s.Adr:0x0021
C:0x32
HCS:0x6C 0x0F

Information: 0xFD 0xFE 0xFF
FCS:0x5C 0x55
Flag:0x7E
----------------------
DETAILS: S:0 P/F:1 len:15 RRR:1 SSS/0:2
----------------------
Nothing processed
----------------------
----------------------
Received 12B
Packet Length:12

----------------------
----------------------
Packet Header Info
Flag:0x7E
Frame Format:0xA0 0x0A
d.Adr:0x258E3
s.Adr:0x0021
C:0x53
HCS:0x40 0x8D
F:0x7E
----------------------
DETAILS: S:0 P/F:1 len:10 RRR:2 SSS/0:3
----------------------
DISC received -> UA sending
----------------------
----------------------
```

6.1.2 Testování s reálným zařízením

Testování s reálným zařízením vzhledem k chybějící vyšší vrstvě (DLMS/COSEM) probíhalo krátce a odhalilo pouze chyby v číslování a posílání informačních paketů protokolu HDLC. Vzhledem k nedostatečnému popsání v příslušné normě se to dalo očekávat. Na základě vzorových přenosů byly tyto chyby opraveny a další odesílání a příjem již nevykazoval chyby.

6.1.3 Paměťové nároky

Po otestování funkčnosti knihovny HDLC proběhla i analýza paměťových nároků.

OS	Demo App [B]	Demo App bez HDLC [B]	Dyn. pam. [B]
Linux	171337	170737	496
Windows	255147	254617	496

Tabulka 1 – HDLC – paměťové nároky

Z tabulky je vidět, že rozdíl ve velikosti binárního programu mezi aplikací používající implementovanou knihovnu HDLC a stejnou aplikací, která knihovnu HDLC nepoužívá je pouhých 600B na Linuxu a 530B na MS Windows. Zátěž na dynamickou paměť je pak 496B a z toho je 128B pole, které je použito pro příjem informační části paketu. V průběhu chodu aplikace se její dynamická paměť nemění a tím pádem můžeme prohlásit 496B za koncových a tudíž maximálních.

6.2 DLMS

Framework DLMS/COSEM byl vyvíjen na operačním systému MS Windows s překladačem Cygwin. Kompilace pod OS Linux proto nepřinesla žádné další chyby. Cílem této práce bylo také použít framework DLMS/COSEM ve stávajícím SW a HW vybavení firmy TECHSYS. Proto byl framework zakomponován a testován ve FW vestavěného zařízení Storm-01. Zdrojové kódy této aplikace nejsou na základě firemní politiky zveřejnitelné, proto zde uvádím pouze výsledky daného zakomponování.

Výsledkem tohoto zakomponování je funkční a parametrizovatelný komunikační modul.

6.2.1 Kompilace

Po přípravě byla knihovna HDLC i DLMS/COSEM začleněna do projektu Storm. Vzhledem k tomu, že kompilátorem není Cygwin ani gcc ale Keil C51, byla nalezena různá další upozornění a chyby.

- Kompilátor nedokázal vyhodnotit kompilační operátor *sizeof()*, který se snažil zjistit velikost nonstatic proměnné ve struktuře. Proto místo proměnné byl jako parametr vložen přímo její datový typ.

```
/// Frame header with information
struct SHDLC_Header_WI{
    uint8_t               openingFlag;   ///< 0x7E
    SHDLC_Header          head;  ///< HDLC only header
    SHDLC_LLC_Header_WD *information;   ///< LLC + Data
    uint16_t              FCS; ///< Frame check sequence
    uint8_t               closingFlag;   ///< 0x7E
};

void foo(){
  // Toto kompilator nedokazal vyhodnotit
  int fVel = sizeof( SHDLC_Header_WI::FCS );
```

```
  // Takto to bylo zmeneno
  int fVel = sizeof( uint16_t ) // FCS
}
```

- Další varování, která kompilátor Keil C51 vypsal, byla o skrytí přetížených virtuálních metodách. Tato varování byla jen drobností a tam, kde bylo zapomenuto na klíčové slovo *virtual*, bylo doplněno.

6.2.2 Běh aplikace

Běh aplikace byl testován proti zařízení LandisGyr+ E650, které bylo připojeno po Ethernetu ve stejné síti, jako zařízení Storm-01. FW Storm se dokázal připojit po TCP, poté se připojila i vrstva HDLC. Asociace na úrovni DLMS se ale nezdařila. Při pokusu o asociaci s cílovým zařízením nebyl asociační požadavek přijat. Při bližším zkoumání bylo zjištěno, že následující řádek kódu není normou C++ definován a každý z kompilátorů ho vyhodnotil jinak.

```
//Nedefinovane chovani
*pOutArr = pInArr[indexIn++]*40 + pInArr[indexIn++];
// Pokud pInArr[0]=2, pInArr[1]=16 a~indexIn = 0
// pak vysledek muze byt 0x82 i~0x60

//Definovane chovani dle ocekavani
*pOutArr = pInArr[indexIn] * 40;
indexIn++;
*pOutArr += pInArr[indexIn];
indexIn++;
```

Po odstranění této a několika dalších drobných chyb aplikace úspěšně cyklicky prováděla následující operace:

1. Spojení TCP
2. Spojení HDLC
3. Asociace DLMS
4. Získání asociačního objektu
5. Vyčtení několika atributů
6. Odpojení HDLC (i DLMS)
7. Rozpojení TCP

6.2.3 Paměťové nároky

Paměťové nároky byly zjišťovány jen na vestavěném zařízení Storm-01, kde byly očekávány problémy s paměťovými zdroji.

Velikost binárního programu

Velikost binárního programu před a po začlenění DLMS/COSEM, HDLC a potřebných tříd a struktur do projektu vypadá následně:

Optim.	s DLMS [B]	bez DLMS [B]	Rozdíl [B]	K dispozici [B]
O2	380 668	360 496	20 172	392 704
O1	409 240	381 152	28 088	392 704
O0	463 656	432 412	31 244	392 704

Tabulka 2 – Velikost binárního programu v paměti

Velikost dynamicky alokované paměti

Celková velikost dynamicky alokované paměti při inicializaci je znázorněna v tabulce 3. Celková dynamická paměť potřebná uvnitř modulu je 23 872B. Minimální velikost TCP bufferu je definována v třídě zajišťující komunikaci po TCP a není možné jí měnit. Výstupní TCP buffer je standardně určen na 2048B. DLMS vstupní/výstupní buffer je buffer, do kterého se ukládají přijatá data, která se následně zpracovávají. Na zařízení LandisGyr+ E650 byla zjištěna nedefinovaná maximální velikost PDU, tím pádem předpokládáme, že v tuto chvíli je největším PDU asociační objekt, který má na tomto zařízení velikost 5821B. Framework DLMS/COSEM je v vytvářen s volitelným parametrem pro velikost vstupně výstupního DLMS bufferu. Tato velikost je s rezervou určena na 8192B. Velikost HDLC bufferu je také parametrem konstruktoru frameworku DLMS/COSEM a je určena na 128B a nemusí být nijak upravována. Buffer pro logování snižuje zátěž stacku a předává logovací zprávy společné třídě pro logování ve FW Storm. Velikost dynamicky alokované paměti jen třídou DLMS/COSEM a HDLC je po odečtení 8868B. To odpovídá pouhým 0,4% z celkové dynamické paměti, která je k dispozici.

Velikost [B]	Zdroj alokace
23 872	Celkem
4 380	vstupní TCP buffer (min)
2048	výstupní TCP buffer
8 192	DLMS vstupní/výstupní buffer
128	HDLC buffer
256	buffer pro logování
8 868	Třídy DLMS/COSEM a HDLC

Tabulka 3 – Velikost alokované dynamické paměti

Zátěž Stacku

Zátěž na Stack je také důležitým kritériem. Ve FW Storm má každý běžící task nastavenou velikost stacku 4096B. Jakmile tuto hodnotu přesáhne, nastávají problémy a zařízení přechází do režimu BOOT. Statistika zátěže na stack knihovny HDLC a DLMS/-COSEM je znázorněna v tabulce 4. Ostatní metody a funkce nejsou důležité vzhledem k tomu, že jsou všechny volány z těchto nebo z dalších zanořených metod.

Statistika zátěže stacku je funkce kompilátoru Keil C51 a je proto prováděna staticky při kompilaci. V tabulce je u každé velikosti uvedeno „+ Unknown Stack Size“. To je velikost všech nedohledatelných zatížení stacku - všech cyklů a rekurzí, nevypátratelných ukazatelů na funkce a také například i přerušení.

Funkce	**Max Stack Depth [B]**
CTs_DLMS_Client::associate()	952 + Unknown Stack Size
CTs_DLMS_Client::release()	896 + Unknown Stack Size
CTs_DLMS_Client::getAssociation()	944 + Unknown Stack Size
CTs_DLMS_Client::readObject()	936 + Unknown Stack Size

Tabulka 4 – Statická analýza zátěže stacku

7 Vyhodnocení a Závěr

Cílem této práce bylo navrhnout a implementovat framework DLMS/COSEM pro sběr dat v AMM systémech. Dalším cílem této práce bylo navrhnout a implementovat knihovnu HDLC, která bude sloužit vyšší vrstvě DLMS/COSEM pro komunikaci s cílovými zařízeními. Na framework i knihovnu byl kladen důraz na minimální paměťové nároky a přenositelnost mezi operačními systémy.

Výsledkem této práce je framework DLMS/COSEM a podpůrná knihovna HDLC. Obě tyto implementace jsou napsány programovacím jazykem C++, jsou přenositelné mezi operačními systémy a splňují minimální nároky, které lze využít i na vestavěných systémech. Velikost binárního programu výsledné aplikace je **20 172B** pro vestavná zařízení a maximální nároky na dynamickou paměť jsou **8868B** pro oba protokoly s volitelnou velikostí přijímacích a odesílacích bufferů. Statická analýza stacku ukázal maximální zátěž frameworkem DLMS/COSEM **952B**.

Framework DLMS/COSEM splnil implementační požadavky, ale vzhledem k rozsahu celého protokolu není kompletní. Byla také splněna podmínka rozšiřitelnosti frameworku o další třídy rozhraní COSEM a funkce DLMS.

Obě implementace byly přeloženy a otestovány na operačních systémech MS Windows, Linux a real-time systému RTX. Celek DLMS/COSEM a HDLC byl začleněn do FW vestavného zařízení Storm-01 a otestován proti zařízení LandisGyr+ E650.

Jedním z možných rozšíření frameworku DLMS/COSEM můžou být další funkce protokolu DLMS a třídy rozhraní COSEM. Zároveň je také možné rozšířit tuto knihovnu o opačnou stranu komunikace. Funkčnost serverové strany by v kombinaci s klientskou vytvořili komplet použitelný nejen pro sběr dat, ale také pro jejich poskytování.

Literatura

[1] DLMS User Association. *We speak the same language...* Dostupné z WWW: `http://dlms.com/downloads/we-speak-the-same-language.pdf`. online. [cit. 2016-5-07].

[2] TECHSYS HW a SW, a. s. *Monitorovací, řídící a informační systémy.* Dostupné z WWW: `https://www.techsys.cz/`. [cit. 2016-5-08]. 2016.

[3] Ing. Jaroslav Mezera Energoprojekt Praha a.s. *ČSN EN 62056-46, Měření elektrické energie – Výměna dat pro odečet elektroměru, řízení tarifu a regulaci zátěže – Část46: Vrstva datového spoje používající HDLC protokol.* [cit. 2016-5-07]. 2003.

[4] Alessandro Triglia. *ASN.1 for More Effective Network Standards.* Dostupné z WWW: `http://bit.ly/1Tx4ln5`. [cit. 2016-5-07]. 2010.

[5] Burton S. Kaliski Jr. *A Layman's Guide to a Subset of ASN.1, BER, and DER.* Dostupné z WWW: `http://luca.ntop.org/Teaching/Appunti/asn1.html`. [cit. 2016-5-07]. RSA Laboratories, 1993.

[6] Autor neuveden. *Transmisja danych licznika EQM – protokół DLMS.* Dostupné z WWW: `http://docplayer.pl/6155490-Transmisja-danych-licznika-eqm-protokol-dlms.html`. [cit. 2016-5-07]. Zakład Elektronicznych Urządzeń PomiarowychPozyton sp. z o.o., 2004.

[7] Ing. Stanisla Roškota ČEPS a.s. *ČSN EN 62056-5-3, Výměna dat pro měření elektrické energie – Soubor DLMS/COSEM – Část 5-3: Aplikační vrstva DLMS/COSEM.* [cit. 2016-5-07]. 2014.

[8] Ing. Stanisla Roškota ČEPS a.s. *ČSN EN 62056-6-1, Výměna dat pro měření elektrické energie – Soubor DLMS/COSEM – Část 6-1: Systém identifikace objektů (OBIS).* [cit. 2016-5-07]. 2013.

[9] Ing. Stanisla Roškota ČEPS a.s. *ČSN EN 62056-6-2, Výměna dat pro měření elektrické energie – Soubor DLMS/COSEM – Část 6-2: Třídy rozhraná COSEM.* [cit. 2016-5-07]. 2014.

Příloha A

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `tsHDLC` ..... zdrojové kódy knihovny HDLC
    - `tsDLMS` ..... zdrojové kódy frameworku DLMS/COSEM
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
    - `Img` ..... obrázky publikované v práci
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF