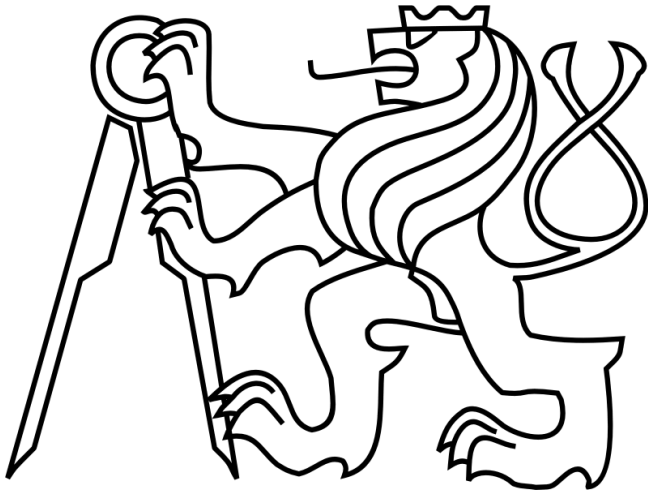
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA ELEKTROTECHNICKÁ



DIPLOMOVÁ PRÁCE

V PRAZE 2012 Bc. Martin Boháček

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a vyznačil všechny citace z pramenů.

V Praze dne 2. ledna 2012 .....

podpis studenta

Poděkování

Děkuji panu Ing. Františkovi Vackovi a panu Ing. Pavlovi Píšovi, PhD. za jejich cenné rady a připomínky při vypracovávání mé diplomové práce. Dále děkuji také mé rodině a Kateřině Vodehnalové za podporu během celého mého studia.

Abstrakt

Tato práce popisuje sadu Qt knihoven a aplikací projektu ulan-admin pro přístup k zařízením a jejich objektovým proměnným na sběrnici µLAN. V šesti kapitolách je popsán návrh a implementace komunikace a rozdělení funkcionality do logických celků. Výsledkem této práce jsou znovupoužitelné komponenty, umožňující Qt aplikacím komunikovat se sběrnicí µLAN.

Klíčová slova: Qt, sběrnice, µLAN

Abstract

This thesis describes a set of Qt-based libraries and applications of project ulan-admin for accessing devices and their object variables on µLAN bus. Six chapters describe the design, implementation and separation of functionality into logical parts. Results of this project are reusable components allowing applications written in Qt framework to communicate with µLAN bus.

Keywords: Qt, bus, µLAN

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Martin B o h á č e k

Studijní program: Kybernetika a robotika (magisterský)

Obor: Robotika

Název tématu: Systém pro správu a monitoring sběrnice uLan

Pokyny pro vypracování:

1. Realizujte bridge + proxy pro přístup k topologii a objektům na uLan Sběrnici pomocí protokolu TCP/IP.
2. Vytvořte program, který bude tuto proxy využívat pro monitoring a konfiguraci sběrnice uLan.
3. Použijte knihovnu QT4.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí diplomové práce: Ing. František Vacek

Platnost zadání: do konce letního semestru 2011/2012

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

L.S.

prof. Ing. Boris Šimák, CSc.
děkan

V Praze dne 9. 2. 2011

Obsah

Seznam použitých zkratek

PDO	Process Data Object
SDO	Service Data Object
CID	Connection ID
OID	Object ID
NCS	Network Control Services
PICO	PDO input CID/OID mapping
POCO	PDO output CID/OID mapping
PEV2C	PDO event to CID mapping
PIOM	PDO input/output mapping meta data
QML	Qt Modeling Language
XML	Extensible Markup Language
JSON	JavaScript Object Notation
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
GPL	General Public License
LGPL	Lesser General Public License

Seznam obrázků

Seznam tabulek

Úvod

Motivací pro tuto práci byla absence univerzálního, multiplatformního a jednoduše rozšiřitelného nástroje pro přístup ke sběrnici µLAN a její správu. Již existující nástroje pro správu sběrnice nebyly navrženy tak, aby se jejich části, starající se o komunikaci se sběrnicí, daly použít v dalších projektech. Neexistovala žádná knihovna, která by poskytovala vyšší vrstvu abstrakce nad sběrnicí. Každý, kdo by chtěl integrovat do své aplikace komunikaci se zařízeními na sběrnici, musel začít stavět na základním C-API. Hlavní myšlenkou tedy bylo navrhnout a implementovat knihovnu pro komunikaci přes sběrnici ve vyšším programovacím jazyce. Zároveň zde byl požadavek, aby bylo možno komunikaci mezi sběrnicí a samotnou aplikací využívající služeb knihovny vést vzdáleně pomocí TCP spojení.

Náročnou částí při sestavování funkční sítě je konfigurace PDO zpráv pro jednotlivá zařízení. Toto nastavení není nijak uloženo ve zdrojových kódech zařízení, je však naprosto podstatné pro funkčnost sítě. Částí této práce proto byla tvorba nástroje pro uložení a opětovné načtení celé PDO konfigurace všech zařízení ze zálohy. Zároveň vznikla myšlenka vytvořit v budoucnu nástroj, který by tvorbu této konfigurace usnadnil.

Paralelně s touto prací vyvíjel kolega Bc. Jan Štefan projekt ulan-genmod pro tvorbu virtuálních zařízení sloužících pro testování návrhu sítě před její fyzickou realizací. Již od začátku bylo počítáno s tím, že části knihovny *libulproxy*, konkrétně nástroje pro konverzi µLAN datových typů do Qt a zpět, budou použity v tomto projektu.

1 Použité technologie

Pro implementaci byl zvolen jazyk C++, konkrétně knihovna Qt. C++ má tu výhodu, že ho lze jednoduše použít s existujícím µLAN C-API. Implementace v čistém C byla též zvažována, ovšem vzhledem k náročnosti a rozsahu požadovaných funkcí a omezeným lidským zdrojům bylo od tohoto upuštěno. Je třeba říci, že od začátku bylo počítáno s nasazením výsledné knihovny nejen do PC, ale i do různých embedded zařízení, například do bezdrátových routerů, které by pak mohly sloužit jako přístupové body ke sběrnici. Tento požadavek nahrával nutnosti využití pouze jazyka C, nicméně při dalším zkoumání bylo zjištěno, že již existují zařízení, která zvládnou provozovat i software napsaný pomocí Qt.

Při výběru technologie pro implementaci bylo kromě knihovny Qt uvažováno ještě o dalších alternativách, konkrétně se nabízela možnost využít C++ knihovnu Boost. Tato volba by jistě nahrávala provozu výsledného softwaru na embedded zařízeních, celá knihovna Boost je v podstatě systém C++ šablon a lze tedy očekávat, že velikost a systémová náročnost výsledné implementace by byla nižší, než za použití Qt. Nicméně pro Qt hovořila skutečnost, že svými možnostmi pokrývá velmi širokou oblast problémů, od grafických komponent až po nízkoúrovňové funkce. V případě volby jiné technologie bychom si patrně nevystačili pouze s jednou knihovnou, což by celou práci zkomplikovalo. Pro úplnost následuje shrnutí základních vlastnosti knihovny Qt:

- Qt je navrženo multiplatformně, mělo by tedy být jednodušší přenášet hotovou knihovnu/aplikaci na jiné operační systémy, než Linux, na kterém byla tato práce vytvořena.

- Knihovna obsahuje množství vestavěných funkcí a tříd, které byly při implementaci využity.

- Architektura signál – slot, sloužící jako náhrada za používání ukazatelů na funkce.

- Přítomnost funkčního návrhu Model-View-Controller architektury.

- Komponenty pro tvorbu grafického uživatelského rozhraní s možností odděleně navrhovat rozložení a funkcionalitu grafických prvků pomocí jazyka QML a JavaScriptu.

- Historická vyzrálost knihovny – Qt existuje již od roku 1999.

- V neposlední řadě byly výhodou zkušenosti vedoucího práce, Ing. Františka Vacka, který Qt dlouhou dobu používá.

Protože bylo počítáno s tím, že knihovna pro přístup ke sběrnici bude umožňovat i komunikaci dvou nebo více jednotek přes TCP protokol, bylo třeba zvolit formát dat vhodný pro přenos. V současnosti se nabízejí dvě hlavní alternativy: XML a JSON. Oba formáty jsou knihovnou Qt poměrně dobře podporovány. Volba padla na JSON, hlavně z těchto důvodů:

- Jednodušší serializace a deserializace datového typu *QVariant* do a z tohoto formátu, jednotlivé datové typy v JSONu jsou přímo mapovány na typy podporované v *QVariant*.

- Stručnější zápis než XML, tím pádem dochází k úspoře dat při přenosu.

- JSON je dobře čitelný pro uživatele.

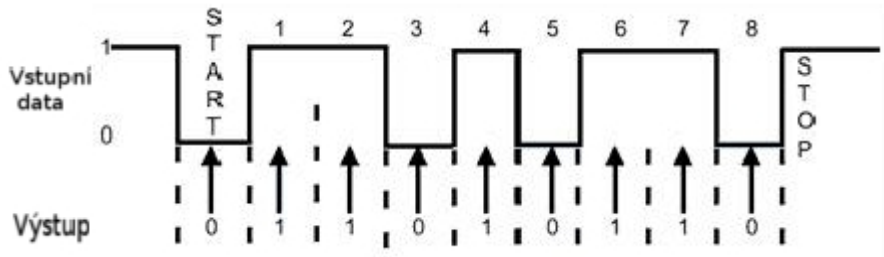
Jako alternativa ke komunikaci pomocí JSON a XML se nabízel protokol D-Bus. Jeho hlavní nevýhoda je, že při jeho použití je nutno navrhovat speciálně celou topologii, včetně samostatného démona, který řídí vlastní komunikaci a konfigurace tohoto systému vyžaduje určité zkušenosti. Navíc D-Bus není dobře podporován na různých platformách, například pro Windows je stále v beta fázi.

2 Sběrnice µLAN

Sběrnice µLAN byla navržena Ing. Pavlem Píšou, PhD. Jedná se o průmyslovou sběrnici, která kombinuje mnohé dobré vlastnosti z ostatních průmyslových sběrnic. Jde především o rozumnou arbitraci sběrnice při zachování rozumné přenosové rychlosti a možnosti propojit zařízení relativně dlouhými vodiči, řádově stovky metrů. Sběrnice µLAN je fyzicky realizována protokolem RS-485. V následujícím popisu sběrnice jsem se inspiroval dokumentem od pana Ing. Pavla Píši, PhD. „*ul_drv - uLan RS-485 Communication Driver*“ [1]. Kapitoly 2.1 až 2.5 a kapitola 2.9 jsou citovány z mé bakalářské práce „*Návrh software pro řízení a monitoring hydroponického systému*“ [8].

2.1 Fyzická vrstva – RS-485

Obrázek 1: Formát dat na RS-485



Zdroj: Root.cz

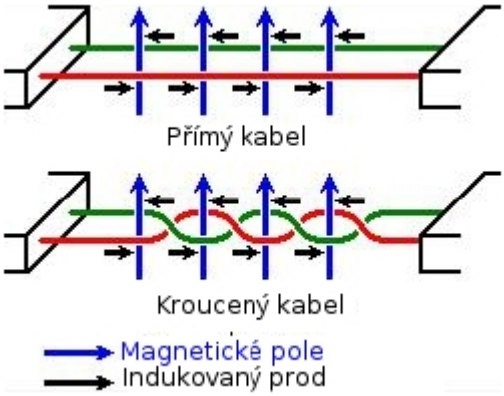
Protokol RS-485 je v průmyslu používán především pro přenos dat na velkou vzdálenost. Data se po něm přenášejí sériově, bez nutnosti použití modulace. Sběrnice nemá vyveden samostatný hodinový signál, který by sloužil pro synchronizaci vysílaných a přijímaných dat. Proto se veškeré zprávy na sběrnici posílají asynchronně.

Pro přenos dat se používají pouze dva signálové vodiče. Lze přidat třetí vodič, který určuje signálovou nulu. Veškeré řízení přenosu dat je prováděno programově, většinou na základě softwarového handshakingu. Přijímač a vysílač musí používat stejnou frekvenci hodinového krystalu, aby nedošlo k chybě přenosu rámce.

2.2 Charakteristika sběrnice

Stav bitu při přenosu po dvoudrátovém vedení je definován rozdílem potenciálu mezi oběma vodiči – jedná se o tzv. diferenciální přenos. Dvoudrátový přenos má výhodu v možnosti použití kroucené dvojlinky, umožňující vysokou přenosovou rychlost bez výraznějšího vyzařování signálu do okolí. U kroucené dvojlinky jsou oba vodiče vlněním ovlivněny stejně, takže přijímač může být velmi citlivý, není potřeba vysokých napětí. Na rozpoznání jedné logické úrovně stačí napětí 200mV. Na obrázku 2 je znázorněn vliv magnetického rušení na přímý a kroucený kabel.

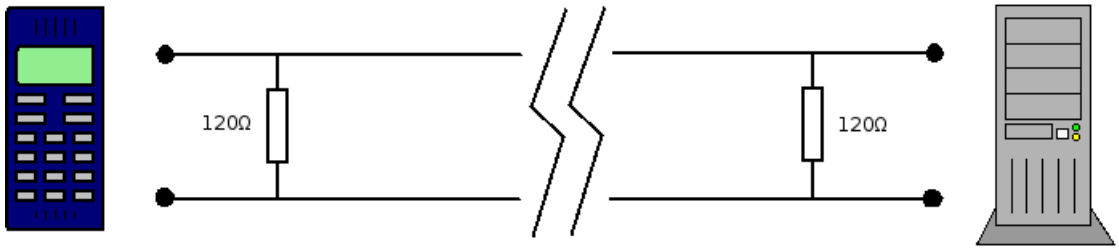
Obrázek 2: Vliv magnetického pole na kabel



Zdroj: Bc. Jan Štefan

Přenos může být uskutečněn až na vzdálenost 1200 metrů. při maximální přenosové rychlosti 100kb/s. Na vzdálenost 15 metrů lze dosáhnout přenosové rychlosti až 15Mb/s. Při realizaci RS-485 vedení je třeba na oba konce sběrnice připojit rezistory

Obrázek 3: Zapojení kroucené dvojlinky



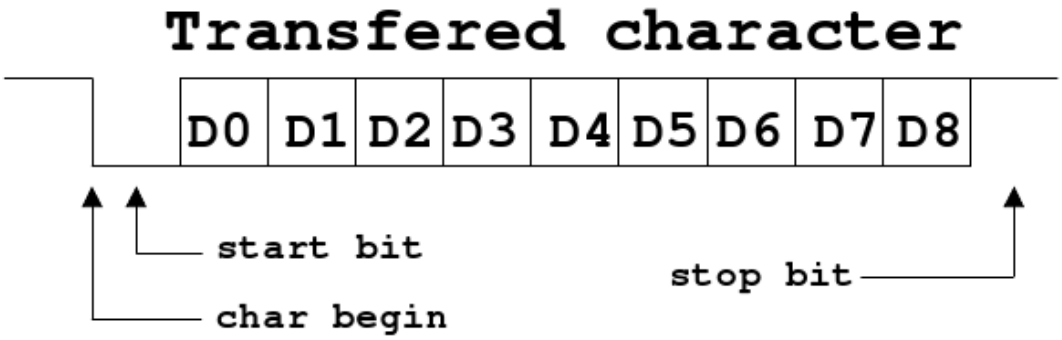
Zdroj: vlastní zpracování

o odporu zhruba 120Ω, což by mělo zhruba odpovídat impedanci kroucené dvojlinky. Typické zapojení je vidět na obrázku 3.

2.3 Formát dat

µLAN je 9-bitový komunikační protokol. Přenos znaku začíná start bitem, následuje sekvence devíti bitů a končí stop bitem. Doba přenosu jednoho znaku je tedy rovna době přenosu 11 bitů. Toto je znázorněno na obrázku 4. Řídící znaky jsou přenášeny s bitem D8 nastaveným na „1“. Tyto speciální znaky se vyskytují pouze na začátku a na konci datového rámce.

Obrázek 4: Přenos znaku přes sběrnici µLAN



Zdroj: Ing. Pavel Píša, PhD.

2.4 Datový rámec

Datový rámec začíná znakem adresy příjemce (DAdr) nebo znakem odpovědi (uL_Beg). První znak je doručen všem zařízením na sběrnici, každé zařízení se podle něj rozhodne, zda bude přijímat i zbytek zprávy. Další dva znaky ve zprávě jsou adresa odesílatele (SAdr) a číslo příkazu nebo typ služby (Com). Tyto dva znaky a všechny další v těle zprávy mají nastaven bit D8 na „0“. V odeslané zprávě není uložena informace o její délce. Ta může být libovolná od nuly až po maximální délku zprávy. Maximální délka závisí na maximální době odezvy a na rychlosti komunikace. Nicméně se doporučuje, aby délka zprávy nepřesáhla 2kB. Konec zprávy je označen

koncovým kontrolním znakem následovaným kontrolním součtem (xor_sum). Kontrolní součet je vypočítán ze všech kontrolních i datových znaků zprávy. Koncový kontrolní znak určuje, co by mělo zařízení, které přijalo zprávu, následně udělat. Koncový znak může nabývat čtyř hodnot:

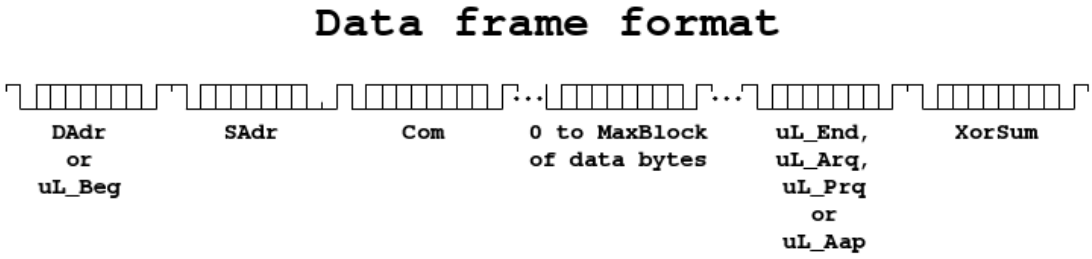
- **uL_End** – konec rámce, rámec je přesunut do vstupní fronty přijímajícího zařízení.

- **ul_Arq** – rámec je přesunut do vstupní fronty přijímajícího zařízení a vysílající zařízení čeká na potvrzení zprávy znakem ul_ACK. V případě chyby kontrolního součtu nebo jiného problému by měl příjemce zaslat negativní potvrzení (ul_NAK). Zpoždění potvrzení o dobu ekvivalentní času přenosu tří znaků je považováno na chybu při přenosu zprávy. Pokud je vstupní fronta přijímacího zařízení plná, může toto zařízení zaslat negativní potvrzení typu „čekej prosím“ (uL_WAK). Tím se předchází zahlcení sběrnice opakovanými pokusy o doručení zprávy.

- **uL_Prq** – značí, že zpráva si žádá okamžité zpracování přijímajícím zařízením. Další chování je definováno příkazem ve zprávě (Com).

- **uL_Aap** – stejný význam, jako ul_Prq, ale přijímací zařízení by mělo zaslat potvrzení ještě před zpracováním zprávy. Negativní potvrzení (uL_NAK) by mělo být zasláno, pokud přijímací zařízení nezná příkaz ve zprávě (Com).

Obrázek 5: Formát datového rámce podle µLAN protokolu



Zdroj: Ing. Pavel Píša, PhD.

2.5 Arbitrace sběrnice

Než dostane zařízení právo vysílat, proběhne tzv. arbitrace. Arbitrace určuje, které zařízení bude mít právo zapisovat na sběrnici. Je dán minimální čas, během kterého se nesmí přístroj připojit. Tento čas je vypočten z adresy předcházejícího přístroje s oprávněním vysílat. Zařízení jsou schopny přijímat znaky, které mají nastaven bit D8. Každé zařízení tedy zná adresu posledního zařízení, které opustilo sběrnici (LAdr). Může si ze své adresy a adresy předcházejícího zařízení spočítat dobu, po kterou bude testovat klid na sběrnici, podle vzorce:

$$t = (LAdr - Adr - 1) \, mod \, 16 + 4$$

Každé zařízení vysílá na sběrnici prázdné intervaly, které jsou po uplynutí periody *t* proložený 000h znaky. Vyslání prázdného znaku znamená odpojení výstupního budiče a po tuto dobu je sběrnice v logické „1“. Pokud zařízení vyšle prázdný znak a sběrnice je v logické „0“ znamená to, že se na sběrnici pokouší připojit jiné zařízení. Zařízení, které vysílalo prázdný znak, považuje sběrnici za obsazenou. Díky době vypočtené podle vztahu uvedeného výše, je kombinace vysílaných signálů nastavena tak, že pouze jedno zařízení najde při poslání prázdného znaku sběrnici v logické „1“. Celý tento proces proběhne třikrát, než zařízení dostane přidělenu sběrnici. Sběrnice takto může být distribuována až mezi 64 přístroji. Aktuální stav sběrnice se sleduje na bitu D7. Zde se definuje příznak obsazenosti *uLF_NB*. Po příjmu jakéhokoli znaku se příznak vynuluje. K testu aktivity sběrnice slouží přístrojům funkce *uL_STROKE*. Ta sleduje, jestli mezi jejím voláním byl přijat alespoň jeden znak. Maximální prodleva během komunikace nesmí překročit dobu, za kterou se přenesou tři znaky.[1]

1 Citováno z: http://ulan.sourceforge.net/pdf/ul_drv.pdf

2.6 Servisní zprávy

Pod pojmem servisní zpráva (SDO – Service Data Object) se obecně rozumí komunikační data, která jsou zaslána jako odpověď na požadavek. Předpokládá se, že ta strana v komunikačním kanálu, která odesílá dotaz, má dostatek informací na to, aby byla schopna dekódovat data v odpovědi. Na sběrnici µLAN jsou procesní zprávy používány například pro tyto účely:

- Správa zařízení na sběrnici (NCS - Network Control Messages). Pomocí těchto zpráv je možno dotazovat se zařízení na jeho identifikaci, nastavovat adresu, atd.

- Vyčítání objektového slovníku (OI), čtení hodnot objektových proměnných a zápis do nich. Vlastní popis objektového rozhraní následuje níže.

2.7 Procesní zprávy

Opakem servisních zpráv jsou zprávy procesní (PDO – Process Data Objects). Při tomto druhu komunikace jsou data zařízením vysílána spontánně, například vždy, když dojde ke změně stavu uvnitř zařízení a je třeba tuto změnu nahlásit do okolí. Data jsou odesílána typicky broadcastem (lze je ale odesílat i na vybrané adresy) a po doručení je na příjemci, co s nimi provede, pokud jim vůbec rozumí.

Tabulka 1: Formát PDO zprávy

Res Lo	Res Hi	Ext len (el)	Ext	CID	Data len (dl)	Data	CID ...
1 byte	1 byte	1 byte	0..el bytes	2 bytes LE	1 (2) byte	dl bytes	

Zdroj: Ing. Pavel Píša, PhD.

Komunikační protokol sběrnice µLAN má standardizován způsob, jak v zařízeních definovat, kdy se budou které servisní zprávy odesílat, co v nich bude obsaženo a o které zprávy se má příjemce vůbec zajímat. Základem PDO komunikace je tzv. kanál. Jde o číselné označení virtuálního spojení mezi více zařízeními (*CID* – channel ID). Pro tento kanál je určeno, kdy má být pomocí něho vyslána informace, co má být

obsahem této informace a jak ji na straně příjemce zpracovat. Všechny tyto definice se provádějí pomocí proměnných v objektovém slovníku. Každé zařízení má k dispozici tyto struktury:

- *PICO* (PDO input CID/OID mapping) – v jednom PICO záznamu je uložena informace o mapování objektové proměnné do kanálu pro příchozí PDO zprávy. Vlastní způsob mapování může být ovlivněn nastavením příznaku *flg*.

- *POCO* (PDO output CID/OID mapping) – POCO záznam obsahuje stejné informace jako PICO, určuje však konstruování odchozích PDO zpráv ze zařízení.

- *PEV2C* (PDO event to CID mapping) – mapování kanálů na události. Pro každou událost lze uvést N mapování na kanály. Zařízení může událost aktivovat zavoláním funkce *uloi_evarr_set_ev()*.

- *PIOM* (PDO input/output mapping metadata) – pomocné pole pro specifikaci metadat používaných při mapování tabulkami PICO/POCO.

2.8 Dynamické přiřazování adres

Služba dynamického přiřazován adres (*ulDy*) umožňuje zařízením připojeným na sběrnici získávat adresu od centrálního arbitra. Každé zařízení může požádat o přidělení adresy na základě svého sériového čísla. Komunikace na této vrstvě probíhá prostřednictvím zpráv typu *NCS – Network Control Messages*. Tyto zprávy mimo jiné umožňují identifikaci jednotlivých zařízení.

2.9 Objektové rozhraní

Objektové rozhraní je vyšší nadstavba nad protokolem µLAN. Pomocí této nadstavby lze na jednom zařízení na sběrnici přímo přistupovat k proměnným jiného zařízení. Objektové rozhraní je postaveno na servisních zprávách.

Pokud chce zařízení poskytnout své proměnné ostatním, musí na své straně definovat tzv. objektový slovník. Proměnná se do objektového slovníku přidá použitím makra *ULOI_GENOBJDES*. Makro má vstupní parametry název proměnné, identifikační číslo proměnné, typ, read-funkce (getter), pointer na proměnnou použitý při čtení, write-funkce (setter) a pointer na proměnnou použitý při zápisu.

Název proměnné určuje, pod jakým jménem bude tato proměnná dostupná ostatním zařízením. Identifikační číslo musí být unikátní v rámci jednoho zařízení. Typ je řetězec, který označuje datový typ proměnné. Do tohoto řetězce je možné vložit další meta informace o proměnné, které by měly být od datového typu odděleny lomítkem a jejich obsah může být libovolný.

Platí pravidlo, že proměnná je v objektovém rozhraní prezentována jako pro čtení, pokud má nastavenu dvojici parametrů read-funkce a pointer na proměnnou použitý pří čtení. Analogicky to platí pro zápis.

Příklad definice proměnné v objektovém slovníku:

```
ULOI GENOBJDES(VARNAME,I_VARNAME,"type/additional_info",
     uloi_rdfnc,&oi_varname,uloi_wrfnc,&oi_varname)
```

V současnosti podporuje objektové rozhraní sběrnice µLAN základní datové typy běžně používané v programovacích jazycích. Navíc jsou podporovány kontejnerové typy pole pevné i proměnlivé délky a typ struktura.

Tabulka 2: Základní datové typy podporované objektovým rozhraním

Popis	Identifikátor	Délka v bajtech
Celé číslo se znaménkem	s1	1
	s2	2
	s4	4
Celé číslo bez znaménka	u1	1
	u2	2
	u4	4
Číslo s pohyblivou řádovou čárkou	f2	2
	f4	4
	f8	8
Řetězec	vs	První bajt obsahuje informaci o délce zbylých dat
Speciální typ „execute“	ee	0

Zdroj: Ing. Pavel Píša, PhD.

Kromě obecných datových typů jsou definovány ještě speciální struktury pro ukládání informací o PDO konfiguraci zařízení. Jedná se o struktury *PICO*, *POCO* a *PEV2C*.

3 Návrh architektury

Cílem práce bylo navrhnout vrstvu vyšší abstrakce, která by umožnila jednoduché zapojení knihovny do Qt aplikací, a která by měla co nejjednodušší interface. Základním požadavkem bylo, aby knihovna umožňovala provádět ty samé operace se sběrnicí, jaké umožňuje C-API µLAN, tj. v prvé řadě zasílat různé typy zpráv a přenášet různé datové typy. Vzhledem k univerzálnosti sběrnice µLAN byl na místě také požadavek, aby navržená knihovna umožňovala jednoduše rozšiřovat funkčnost.

Jakou inspirace při návrhu posloužila UNIXová myšlenka zřetězení vstupů a výstupů malých funkčních celků. Základním prvkem v knihovně je obecný filtr. Ekvivalentem nejjednoduššího filtru je v UNIXU příkaz *cat*. Jednotlivé filtry se dají řetězit a je tak možno vytvářet komplexní funkční celky.

Oproti low-level µLAN C-API došlo i určité datové abstrakci, kdy byla množina datových typů objektového rozhraní µLAN agregována do množiny typů podporovaných třídou *QVariant*. Zároveň byl *QVariant* rozšířen o uživatelské typy odpovídající strukturám PICO, POCO a PEV2C. Využití *QVariant* pro přenos dat odbourává nutnost přenosu popisu typů proměnných, veškeré nutné informace o typech dat lze získat introspekcí proměnné typu *QVariant*.

Navržené agregované typy jsou použity pro komunikaci mezi všemi filtry. Konverze µLAN datových typů na *QVariant* a zpět je vyřešena ve filtru typu ovladač *ULPFDDriver*, který v sobě zároveň zapouzdřuje vše nutné pro samotnou komunikaci se sběrnicí. Navenek probíhá komunikace s tímto filtrem prostřednictvím standardního rozhraní.

Je třeba zdůraznit, že celá komunikační vrstva je navržena asynchronně. Toto pojetí se dobře hodilo k integraci s knihovnou Qt, ve které se ve velké míře využívá událostmi řízeném programování.

Z výše jmenovaných požadavků vyplynuly principy, na kterých je knihovna postavena. Tyto jsou popsány v sekcích 3.1 , 3.2 a 3.3 .

3.1 Systém obecných filtrů

Základním stavebním prvkem knihovny *libulproxy* je obecný filtr. Pod pojmem filtr se rozumí objekt, který stojí mezi samotnou sběrnicí a uživatelskou aplikací. Od filtru se očekává, že umí:

1. přeposílat zprávy přicházejí z nadřazeného filtru

2. propagovat události od podřízeného filtru k nadřazenému

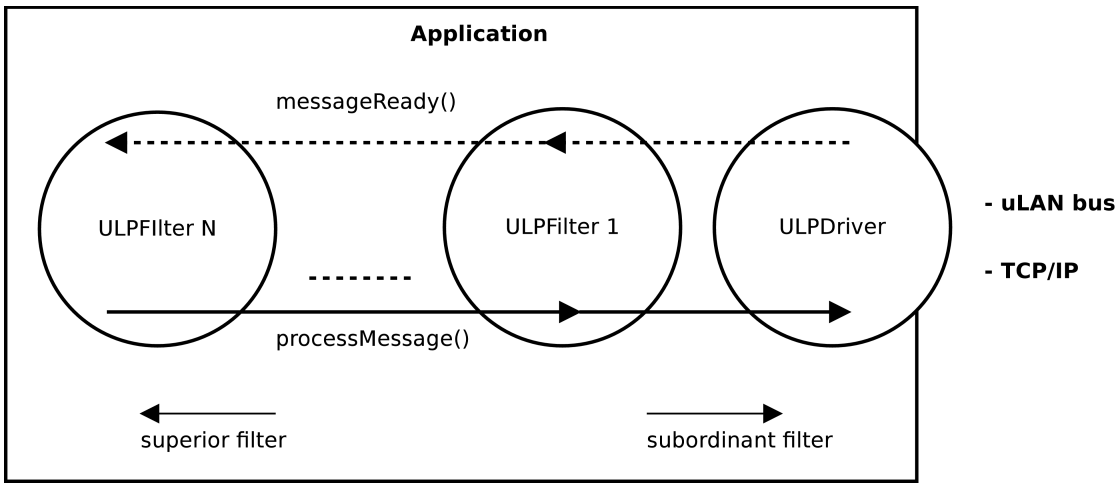
Důležitou vlastností filtrů je to, že se dají řetězit. Aplikace komunikující se sběrnicí obsahuje libovolně dlouhý řetězec filtrů zakončený filtrem typu ovladač. Každý filtr propaguje data od nadřazeného filtru směrem k podřízenému a asynchronní události z podřízeného filtru směrem k filtru nadřazenému. Tento koncept dobře vyhovuje požadavkům na jednoduché rozhraní – ve vlastní implementaci má každý filtr metodu *processMessage()* pro předání dat dál v řetězci a signál *messageReady()* pro signalizaci nově příchozích dat. Navíc je zde možnost rozšiřovat funkcionalitu knihovny novými typy filtrů.

Základem obecného filtru je abstraktní třída *ULPFilter*. Tato třída definuje interface:

- Virtuální metoda *processMessage()*, která slouží pro předávání komunikačních dat dále směrem od aplikace ke sběrnici.

- Signál *messageReady()* pro indikaci nově příchozích dat nadřazenému filtru nebo aplikaci.

Obrázek 6: ULPFilter - komunikační řetězec



Zdroj: vlastní zpracování

K postupnému zapojování filtrů za sebe slouží metoda *setSourceFilter()*, pomocí které lze každému filtru přiřadit jeho následovníka v komunikačním řetězci. Tato metoda automaticky spojí signály *messageReady()* filtru a jeho následovníka. Zvláštním případem filtru je tzv. ovladač (*ULPDriver*). Ovladač nemá definován zdrojový filtr, komunikuje tudíž buď přímo se sběrnicí, nebo předává data dál jiným způsobem, například pomocí TCP socketu.

V rámci práce byly implementovány filtry *ULPProxy*, *ULPSocketDriver* a *ULPFDDriver*. U filtru *ULPFDDriver* se předpokládá, že bude vždy posledním článkem v komunikačním řetězci, protože jeho úkolem je samotná komunikace se sběrnicí.

Konkrétní implementace filtrů navržené v rámci této práce jsou popsány v sekcích 4.2 a 4.3 .

3.2 Komunikační zprávy

µLAN C-API definuje pro komunikaci strukturu *ul_msginfo*. Tento kontejner v sobě obsahuje veškeré nutné meta informace o vlastních komunikačních datech. Její signatura vypadá takto:

```
typedef struct ul_msginfo {
     int  dadr;          /* destignation address */
     int  sadr;          /* source address */
     int  cmd;           /* command/socket number */
     int  flg;           /* message flags */
     int  len;           /* length of frame */
     unsigned stamp;     /* unigue message number */
} ul_msginfo;
```

Pole *dadr*, resp. *sadr* obsahují µLAN adresu příjemce, resp. odesílatele zprávy. Pole *cmd* obsahuje informací o typu zprávy, např. zda-li se jedná o zprávu typu (*Network Control Service*) nebo o zprávu objektového rozhraní. V poli *flg* jsou obsaženy instrukce o tom, jak má být zpráva zpracována v ovladači. Pole *len* udává délku komunikačních dat. V posledním poli *stamp* je uloženo unikátní číslo, které je zprávě přidělováno ovladačem sběrnice.

Tato struktura posloužila jako vzor pro návrh kontejneru pro komunikaci prostřednictvím rozhraní *ULPFilter*. Tento kontejner je implementován ve třídě *ULPMessage* v knihovně *libulproxy*. Třída *ULPMessage* obsahuje všechny atributy uvedené ve struktuře *ul_msginfo*, plus další atributy, které vyplynuly z návrhu komunikace pomocí filtrů. Tyto další atributy jsou podrobněji popsány v kapitole 4.2 . Důležité je, že zpráva obsahuje i vlastní komunikační data. Tato data, spolu s dalšími doplňujícími informacemi, jsou ve zprávě uložena ve struktuře *QVariantMap*, což umožňuje jednoduchou serializaci zprávy do JSON formátu.

Třída *ULPMessage* je navržena jako implicitně sdílená, což značně zefektivňuje předávání jejích instancí architektuře signál – slot. Více o implicitně sdílených objektech je obsaženo v kapitole 4.1 .

3.3 Proxy vrstva

Pro efektivní přístup k zařízením na sběrnici µLAN a k jejich objektovým proměnným je nutné znát velké množství informací. Je třeba udržovat informace o tom, která zařízení jsou přítomna, jaké proměnné a jakých typů mají ve svých objektových slovnících. Dotazovat se na tyto informace jednotlivých zařízení vždy, když je potřeba, by způsobilo nadměrné zatížení sběrnice a prakticky by došlo k jejímu úplnému zahlcení. Z toho důvodu bylo již od počátku jasné, že součástí knihovny *libulproxy* musí být cachovací vrstva pro šetření komunikace na sběrnici a pro minimalizaci objemu servisních zpráv.

3.3.1 Struktura cache

Cachování je v knihovně *libulproxy* implementováno pomocí filtru *ULPProxy*. Díky tomu lze snadno realizovat vícenásobné cachování po cestě v komunikačním řetězci. Tato cachovací vrstva obsahuje jak metadata o zařízeních na sběrnici, tak vlastní hodnoty objektových proměnných. Při inicializace komunikace se sběrnicí dojde nejdříve k vytvoření obrazu sítě. Tyto informace mají charakter stromové struktury. Každý uzel si u sebe drží svoje meta informace a pokud to dává smysl, je u něho uložena i hodnota. Zjednodušeně lze cachovanou strukturu popsat takto:

První úrovní ve stromu jsou tzv. µLAN sítě (*ULPNetwork*). Tato úroveň je zde spíše pro úplnost, v současnosti lze komunikovat pouze s jednou sítí a proto tato úroveň obsahuje vždy jeden uzel.

Uzly na druhé úrovni jsou jednotlivá zařízení (*ULPDevice*). U zařízení má smysl udržovat pouze metadata, jsou jimi adresa a název. Nepočítá se s tím, že by se tyto informace v čase měnily.

Třetí úrovní v hierarchii jsou objektové proměnné v jednotlivých zařízeních (*ULPObjectVariable*). Metadata objektové proměnné jsou její *OID* – unikátní identifikátor v rámci zařízení a její název. U objektových proměnných má jako u jediné úrovně smysl udržovat jejich hodnotu. Počítá se s tím, že tyto hodnoty se často mění a jejich přenos tvoří podstatnou část komunikace na sběrnici.

3.3.2 Adresace pomocí logické cesty

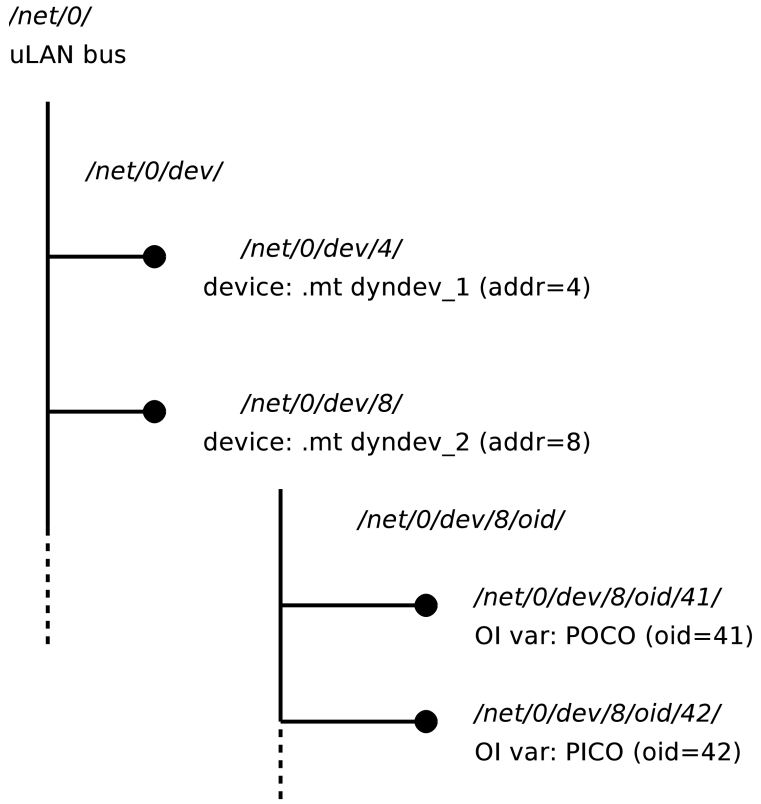
Každý uzel ve stromu u sebe drží informaci o času poslední aktualizace. Při dotazu na něj lze stanovit maximální stáří informace. Pokud je časová značka u uzlu starší než daný limit, dojde k okamžitému vrácení jeho hodnoty a zároveň k odeslání žádosti o aktuální data do podřízeného filtru v komunikačním řetězci. Jakmile nová data dorazí do cachovací vrstvy, dojde k jejich aktualizaci u patřičného uzlu ve stromu a zároveň k odeslání do nadřazeného filtru.

Pro adresaci uzlů ve stromu byla navržena tzv. logická cesta. Díky ní lze adresovat jeden uzel ve stromu, případně se dotázat na výpis všech dětí jednoho uzlu. Jednotlivé úrovně ve stromu, tj. síť, zařízení, objektová proměnná jsou v logické cestě uvozeny jmennými prostory *net*, *dev*, *oid*. Za označením jmenného prostoru v cestě může následovat číselná adresa uzlu. V případě sítě je to vždy adresa 0. Adresou zařízení je jeho µLAN adresa. Pro adresaci objektové proměnné slouží její *OID*. Jednotlivé části cesty jsou spojeny lomítkem. Příklady adresace pomocí fyzické cesty jsou následující:

- */net/0/dev/* - výpis všech zařízení na síti
- */net/0/dev/12/oid/* - výpis objektových proměnných zařízení s µLAN adresou 12
- */net/0/dev/12/oid/30/ - přístup k objektové proměnné s OID 30 v zařízení s µLAN adresou 12.*

Příklad popisu části sběrnice logickými cestami je znázorněn na obrázku 7.

Obrázek 7: Popis objektů na sběrnici pomocí logických cest



Zdroj: vlastní zpracování

Vytvoření stromového obrazu sítě probíhá postupně, jak aplikace přistupující na sběrnici vyžaduje informace. Tímto přístupem lze dosáhnout podstatné úspory komunikace při enumeraci sběrnice, kdy bývá její zatížení zpravidla největší. Aktualizace cachovaných dat v proxy se kromě synchronních dotazů využívá i odposlech asynchronních procesních zpráv (PDO), pokud tyto zprávy obsahují fragmenty cachovaných dat.

Třída *ULPProxy* má oproti ostatním typům filtrů rozšířeno své rozhraní o signály a sloty umožňující její efektivní integraci s Qt model-view architekturou. Předpokládá se, že i při minimální konfiguraci bude aplikace obsahovat alespoň jeden filtr typu *ULPProxy* a tato cachovací vrstva bude přítomna jako první filtr v komunikačním řetězci.

Podrobněji je implementace třídy *ULPProxy* popsána v kapitole 4.2 .

4 Zvolené komponenty a implementace

Tato kapitola podrobně popisuje konkrétní implementaci komponent a aplikací zpracovaných v rámci diplomové práce. Jedná se o tyto celky:

- *libulproxy* – dynamicky linkovaná knihovna obsahující celou abstrakci komunikace se sběrnicí včetně konkrétních implementací filtrů komunikačního řetězce: *ULPProxy* a *ULPSocketDriver* Je zde přítomna i implementace komunikačního kontejneru *ULPMessage*.

- *libulpulan* – dynamicky linkovaná knihovna, obsahující implementaci ovladače *ULPFDDriver*. Tato část byla vyňata z knihovny libulproxy kvůli oddělení částí závislých na µLAN C-API.

- *libulqconv* a *libulqextras* – dvojice dynamicky linkovaných knihoven obsahující funkce pro dekódování µLAN C-API datových typů do typu *QVariant* a zpět. V knihovně *libulqextras* jsou dodefinovány uživatelské typy *QVariant* pro reprezentaci PICO, POCO a PEV2C struktur a funkce pro práci s nimi. Knihovny jsou dvě, kvůli oddělení částí závislých na µLAN C-API – tyto jsou přítomny pouze v knihovně *libulqextras*.

- *ulanbrowser* – aplikace pro procházení zařízení na sběrnici s možností zobrazování a modifikace jejich objektových proměnných a správu PDO konfigurace.

- *ulpsrvd* – jednoduchá konzolová aplikace umožňující spolu s filtrem *ULPSocketDriver* tunelování komunikace se sběrnicí přes TCP protokol pomocí datového formátu JSON.

Protože cela práce je postavena na knihovně Qt, v první části jsou popsány její důležité vlastnosti, které byly využity. Při popisu těchto vlastností bylo čerpáno z online Qt dokumentace.

4.1 Architektura knihovny Qt[2]

Obrázek 8: Logo Qt

Zdroj: Qt Reference Documentation

Následující sekce popisuje vybrané části knihovny Qt, které byly využitu při implementaci projektu ulan-admin.

4.1.1 QVariant

Třída *QVariant* se chová jako *union* pro většinu datových typů Qt. Protože C++ zakazuje vkládání typů bez implicitního konstruktoru nebo destruktoru do *union* struktur, většinu Qt typů s nimi nelze použít.

QVariant objekt může obsahovat jednu hodnotu daného typu *T*. Typ aktuálně obsažené hodnoty lze zjistit metodou *type()*. Konverzi do jiného typu lze provést metodou *convert()*, vrátit hodnotu v daném typu voláním *toT()* a ověřit, zda lze provést konverzi do jiného typu metodou *canConvert()*.

Metody *toT()* (například *toInt()*, *toString()*, …) jsou konstantní, vracejí tedy kopii objektu obsaženého v *QVariant*. Pokud lze *QVariant* konvertovat do požadovaného typu, je tato konverze provedena, je navrácen objekt požadovaného typu a původní data zůstanou nezměněna. Pokud konverzi nelze provést, závisí výsledek na požadovaném typu.

Obsahem *QVariant* mohou být další objekty obsahující *QVariant*, například *QList<QVariant>* nebo *QMap<QString, QVariant>*, čímž lze do objektu *QVariant* uložit libovolně složité datové struktury. Tato vlastnost je v praxi velmi užitečná a umožňuje univerzální použití *QVariant*, může však být méně efektivní, co se týče využití paměti a rychlosti zpracování, než práce s proměnnými konkrétních typů.

2 Obsah této kapitoly vznikl volným překladem z http://doc.qt.nokia.com/ [4].

Seznam datových typů podporovaných *QVariant* je obsažen ve výčtu *QVariant::Type*. Kromě těchto typů lze definovat i vlastní uživatelské typy, které lze pak vkládat do *QVariant*, viz dokumentace ke třídě *QMetaType*..

4.1.2 Implicitní sdílení

Mnoho tříd, které jsou součástí Qt, používá implicitní sdílení dat pro maximalizaci využití zdrojů a omezení kopírování dat. Implicitně sdílené třídy zaručují bezpečné a efektivní používání svých instancí jako argumenty při volání funkcí, protože dochází pouze k předávání ukazatele na data a tato data jsou kopírována teprve tehdy, když jsou změněna. Jde tedy o tzv *copy-on-write* přístup.

Sdílená třída se skládá z ukazatele na sdílený blok dat, jenž má u sebe počítadlo referencí, a z dat.

Při vytvoření sdíleného objektu je počítadlo referencí nastaveno na 1. Toto počítadlo je inkrementováno, resp. dekrementováno při referencování, resp. dereferencování sdílených dat. Pokud počítadlo klesne na hodnotu 0, sdílená data jsou smazána.

Při práci se sdílenými objekty nastávají dva typy jejich kopírování. Jde o tzv. mělké a hluboké kopie objektů. Hluboká kopie znamená duplikování objektu. Při mělkém kopírování se kopíruje pouze reference. Hluboké kopírování může být velmi náročné na paměť a procesorový čas. Oproti tomu mělké kopírování je velmi rychlé, jde pouze o nastavení ukazatele a inkrementování počítadla.

Operátor přiřazení (metoda *operator=()*) je u implicitně sdílených objektů implementován jako mělká kopie.

Výhodou implicitního sdílení je, že aplikace nemusí nezbytně kopírovat data, děje se tak až když je toho třeba. Tím dochází k úsporám zdrojů. Sdílené objekty mohou být efektivně přiřazovány, předávány jako argumenty funkcí a používány jako návratová hodnota.

Implicitní sdílení se odehrává na pozadí, programátor se nemusí o nic starat, k inkrementaci a dekrementaci počítadla referencí dochází automaticky. Navíc je tato operace atomická, takže není problémem ani použití ve vícevláknových aplikacích.

Vlastní implicitně sdílené objekty se implementují pomocí tříd *QSharedData* a *QSharedDataPointer*.

4.1.3 Signál-slot architektura

Signály a sloty slouží ke komunikaci mezi objekty. Jde o základní vlastnost knihovny Qt a také o část, kterou se pravděpodobně nejvíce odlišuje od ostatních frameworků.

V událostmi řízeném programování je běžné, že některé objekty produkují události, o kterých jsou jiné objekty nějakým způsobem informovány. Starší knihovny tento problém řeší pomocí tzv. *callbacků*. *Callback* je ukazatel na funkci. Objektu, který generuje událost, předáme ukazatel na funkci a ten potom informuje ostatní objekty voláním této funkce. *Callbacky* mají dvě podstatné nevýhody:

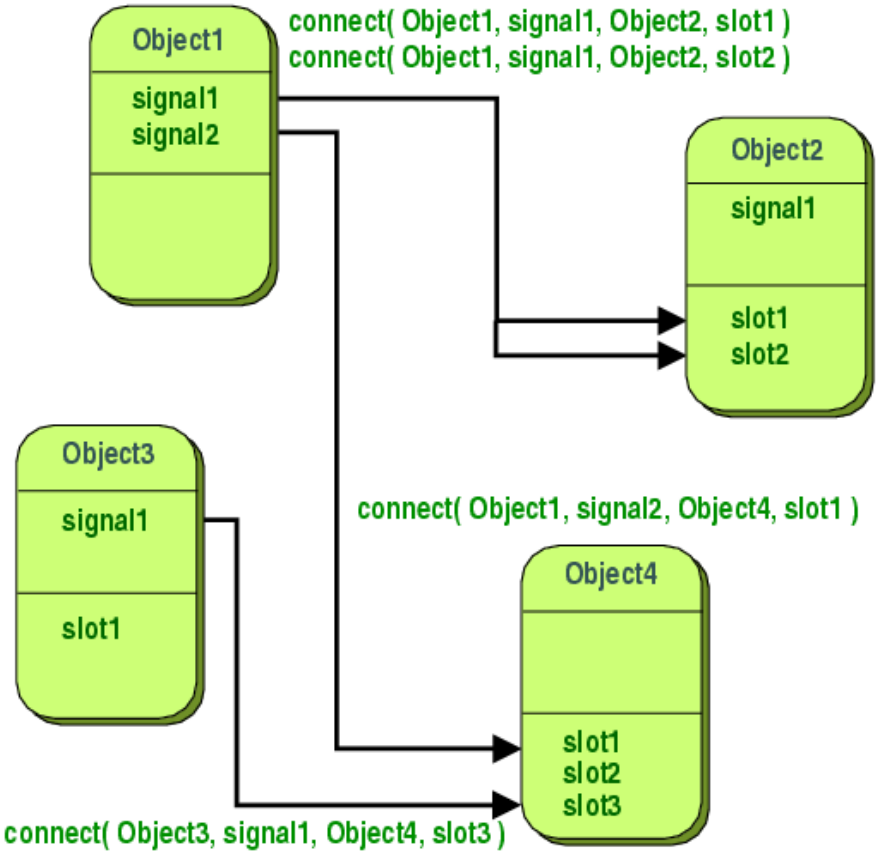
- Nejsou typově bezpečné. Nelze se ujistit, že *callback* bude volán s argumenty správných typů.
- *Callback* je pevně svázán s objektem produkujícím události.
- Pokud je smazán objekt, který obsahoval metodu použitou jako *callback*, druhá strana, která chce *callback* volat, se o tomto nijak nedozví. Hrozí pak, že při volání nastane chyba segmentace, protože ukazatel na *callback* již míří do prázdna (tzv. *dangling pointer*[3]).

Qt nabízí jako alternativu ke *callbackům* architekturu signál-slot. Signál je emitován, pokud nastane určitá událost. Slot je funkce, která je zavolána jako odpověď na signál. Mechanismus signálů a slotů je typově bezpečný, signatura emitovaného signálu musí odpovídat signatuře volaného slotu. Ve skutečnosti může být signatura slotu kratší, chybějící argumenty jsou ignorovány. Signály a sloty spolu nejsou pevně svázány, objekt, který emituje signál nemusí vůbec vědět o slotech, které jsou na signál připojeny.

3 Více informací na http://en.wikipedia.org/wiki/Dangling_pointer.

Všechny třídy, které dědí od *QObject,* mohou obsahovat signály a sloty. Slot může být použit pro přijímání signálů, ale je to zároveň standardní funkce, která může být volána běžným způsobem.

Obrázek 9: Příklad propojení objektů pomocí signálů a slotů



Zdroj: Qt Reference Documentation

K jednomu slotu může být připojeno libovolné množství signálů a jeden slot může být připojen k libovolnému množství slotů. Je dokonce možné připojit signál k jinému signálu. Tím dojde k emitování druhého signálu ihned po prvním.

Spojení signálu a slotu je realizováno voláním funkce *QObject::connect()*. Vstupní argumenty funkce jsou:

- ukazatel na objekt emitující signál
- název signálu včetně signatury

- ukazatel na objekt s přijímajícím slotem
- název slotu včetně signatury
- typ spojení – nepovinný argument

Qt definuje následující typy spojeni mezi signálem a slotem:

- **Qt::AutoConnection (0)** – Výchozí. Pokud je signál emitován z jiného vlákna, než přijímající objekt, signál je zařazen do fronty, stejně jako při *Qt::QueuedConncetion*. Jinak je slot zavolán ihned, stejně jako v případě *Qt::DirectConnection*.
- **Qt::DirectConnection (1)** – Slot je zavolán ihned, jakmile je signál emitován.
- **Qt::QueuedConncetion (2)** – Slot je zavolán, až se CPU přepne do vlákna přijímajícího objektu. Slot je pak vykonán v tomto vlákně.
- **Qt::BlockingQueuedConncetion (4)** – Stejné jako *Qt::QueuedConncetion*, ale vlákno, ze kterého byl signál emitován, je zablokováno, dokud není slot dokončen.
- ***Qt::UniqueConnection (0x80)*** – Stejné jako *Qt::AutoConnection*, ale spojení je vytvořeno pouze tehdy, pokud tím není duplikováno již existující spojení.
- **Qt::AutoCompatConnection (3)** – Výchozí typ spojeni, pokud je povolena podpora Qt 3. Stejné, jako *Qt::AutoConnection*.

4.2 Knihovna libulproxy

Dynamicky linkovaná knihovna *libulproxy* umožňuje aplikacím komunikovat se sběrnicí µLAN pomocí komponenty *ULPMessage* a konkrétních implementací třídy *ULPFilter*. Navíc v sobě obsahuje obraz zařízení na µLAN sběrnici a informace o jejich objektových proměnných.

4.2.1 Struktura cache

Jak již bylo řečeno, na µLAN sběrnici jsou objekty uspořádány hierarchicky. Pro popis této struktury byl navržen systém logických cest. Základem uzlu ve stromu cache je třída abstraktní *ULPNode*. Obsahuje atributy pro uložení obecné hodnoty a metadat jako objekty *QVariant*. Navíc obsahuje odkaz na nadřazený uzel. Od této třídy jsou odděděny další dvě abstraktní podtřídy – jde o třídy reprezentující jeden konkrétní objekt na sběrnici (hodnotu) – *ULPValueNode* a oproti tomu skupinu objektů (adresář) – *ULPDirNode*. Úroveň adresářů byla do struktury vložena proto, aby se bylo možno odkázat na celou skupinu hodnot, například požadavek na výpis všech zařízení lze provést dotazem na logickou cestu *„/net/0/dev/“*.

Ve stromu cache se úrovně uzlů typů hodnota a adresář pravidelně střídají. Každý uzel typu adresář určitých objektů obsahuje vždy děti reprezentující tyto objekty.

Pro konkrétní účely byly implementovány třídy, které dědí z *ULPDirNode* a *ULPValueNode*. Jejich uspořádání v hierarchii cache je následující:

- *ULPNetworkDir* – seznam sítí µLAN. Tato třída je zde spíše pro úplnost, v současnosti není možné pracovat s více než jednou sítí a proto tento adresář obsahuje vždy jedno potomka. V logické cestě je tato úroveň označena řetězcem „*net*“.

- *ULPNetwork* – reprezentace µLAN sítě. Ve stromu je vždy pouze jedna, neboť, jak již bylo řečeno, v současnosti je podporováno připojení pouze k jedné síti. V logické cestě je označena číselným ID „*0*“.

- *ULPDeviceDir* – seznam zařízení na síti. Tato úroveň umožňuje odkázat se na všechna zařízení. V logické cestě je tato úroveň označena řetězcem „*dev*“.

- *ULPDevice* – reprezentace jednoho zařízení. U uzlu typu zařízení jsou držena metadata jméno a adresa µLAN. V logické cestě je tato úroveň označena fyzickou adresou µLAN.

- *ULPObjectDir* – seznam objektových proměnných v jednom zařízení. Obdobně jako *ULPDeviceDir*, tato úroveň umožňuje odkázat se na všechny objektové proměnné zařízení. V logické cestě je označena řetězcem „*oid*“.

- *ULPObjectVariable* – poslední úroveň ve stromu odpovídá objektové proměnné v zařízení. Kromě metadat OID a jméno nese i svoji hodnotu jako objekt *QVariant*. Tato úroveň je ve fyzické adrese označena svým OID.

Kromě logické cesty je pro adresování uzlů ve stromu definováno také určení uzlu pomocí seznamu indexů – tzv fyzická cesta. Každé číslo v tomto seznamu určuje index adresovaného objektu mezi dětmi svého rodiče. Tento způsob adresování byl zaveden z důvodu jednoduché integrace stromu cache s modelem *QAbstractItemModel*, sloužícím pro popis dat v knihovně Qt.

4.2.2 ULPMessage

Komunikační kontejner *ULPMessage* je navržen jako nadstavba nad µLAN C-API strukturou *ul_msginfo*. Objekt *ULPMessage* je předáván pomocí metody *processMessage()* a signálu *messageReady()* při komunikaci mezi filtry. Spolu s těmito metodami definuje celou abstrakci komunikace v knihovně *libulproxy*. *ULPMessage* je implementována jako implicitně sdílený objekt, což umožňuje její předávání hodnotou aniž by docházelo ke kopírování jí obsažených dat. Vždy, když je třeba změnit data ve zprávě, je vytvořena její kopie, nad kterou probíhají úpravy (*copy-on-write*), což je obzvláště výhodné, pokud je signál a slot propojen spojením typu *QueuedConnection*. Tento typ spojení se používá při signál-slot komunikaci mezi thready.

Kromě metadat převzatých ze struktury *ul_msginfo* obsahuje další atributy, které mohou obsahovat další informace využívané při komunikaci mezi filtry. Jde například o informaci o logické cestě, ke které se zpráva vztahuje. Vedle této cesty je to struktura popisující operaci, která se má na logické cestě provést. Vzorem pro tuto strukturu byl formát používaný v JSON RPC v2.0. Informace jsou uloženy v objektu *QVariantMap*, což umožňuje přímou serializaci do JSON formátu.

RPC data pro dotaz na hodnotu objektové proměnné mohou vypadat takto:

```
{"jsonrpc": "2.0", "method": "get", "params": {
    {"path": "/net/0/dev/12/oid/40/"}
}}
```

Odpověď na předchozí dotaz může vypadat takto:

```
{"jsonrpc": "2.0", "method": "get", "result": {
    {"path": "/net/0/dev/12/oid/40/",
    "__value__": [0,1,2,3,4,5,6,7,8,9]}
}}
```

Třída *ULPMessage* obsahuje funkce pro automatizovanou konstrukci těchto struktur. Pro sestavení zprávy pro dotaz na hodnotu, resp. pro zápis hodnoty jsou definovány statické metody *constructGetDataRPC()*, resp. *constructSetDataRPC()*. Obě berou jako argument logickou cestu, druhá jmenovaná pak ještě objekt *QVariant* s hodnotou, která se má zapsat.

Pro konverzi celé zprávy do JSON formátu a následný převod zpět do binární podoby jsou ve třídě *ULPMessage* definovány statické metody *encodeJsonMessage()* a *decodeJsonMessage()*. Díky masivnímu využití *QVariant* objektů pro uložení informací uvnitř zprávy je tato konverze značně zjednodušena.

4.2.3 ULPFilter

Abstraktní třída *ULPFilter* definuje komunikační interface filtru. Obsahuje virtuální metodu *processMessage()*, která slouží pro předávání objektu *ULPMessage* směrem od nadřazeného filtru k podřízenému, a dále signál *messageReady()*, který předává zprávu opačným směrem. Dále třída definuje virtuální metodu *setSourceFilter()*, jejíž volání slouží k přiřazení podřízeného filtru v komunikačním řetězci tím, že spojí signály *messageReady()* dvou filtrů. Není nutné, aby byla v podtřídách tato metoda přetížena.

4.2.4 ULPDriver

Kromě konkrétních implementací filtrů odvozených od třídy *ULPFilter* obsahuje knihovna ještě abstraktní podtřídu *ULPDriver*. Tato by měla být základní třídou při implementaci tzv. ovladačů. Ovladač je filtr, který je v aplikaci na konci komunikačního řetězce a zprávy, které přijímá, posílá dále jinou formou než voláním metody *processMessage()* svého podřízeného filtru. Může například data přímo zapisovat na sběrnici µLAN nebo přeposílat přes TCP socket dále.

4.2.5 Filtr ULPProxy

Cachovací vrstva je implementována jako filtr *ULPProxy*. Z důvodu šetření komunikace se sběrnicí by tento filtr měl být v aplikaci vždy alespoň jednou použit a měl by být první v komunikačním řetězci. Cache struktura ve filtru sestává z hierarchie instancí podtříd *ULPNode*. Její budování probíhá postupně podle toho, která data procházejí přes filtr.

Vlastní dotazování na uzly probíhá pomocí zprávy *ULPMessage* obsahující logickou cestu a typ dotazu (o objektových proměnných je třeba rozlišit, zda-li jde o čtení nebo zápis). Tato zpráva je předána funkci *processMessage()*

Uzly na všech úrovních obsahují časovou značku své poslední aktualizace. Při dotazu na daný uzel lze jako nepovinný argument specifikovat maximální stáří informace. Pokud uložená informace není starší než je požadováno, je okamžitě emitována zpráva

s odpovědí. Pokud je tomu naopak, je taktéž odeslána zpět zpráva se stávajícími daty, ale navíc je k podřízenému filtru odeslán dotaz na aktuální data.

Při příchozí zprávě signálem *messageReady()* od podřízeného filtru je aktualizován obsah cache a tato práva je předána nadřazenému filtru.

Kromě standardního API poskytuje filtr *ULPProxy* navíc metody pro integraci s modelem *QAbstractItemModel*. Jedná se o funkci *data()* a dva signály *dataAboutToBeChanged()* a *dataChanged()*. Metoda *data()* umožňuje modelu přímý přístup k datům v cache bez nutnosti konstruovat zprávy *ULPMessage*. Oba dva signály pak slouží pro signalizaci změn v cache struktuře pro model. Protože při aktualizaci dat v *QAbstractItemModel* je třeba signalizovat i povahu změny, tj. jedná-li se o vložení nového záznamu, aktualizaci nebo smazání, signály nesou i tuto informaci. Všechny tyto metody používají adresaci pomocí fyzické cesty namísto logické.

4.2.6 Filtr ULPSocketDriver

Třída *ULPSocketDriver* je implementace ovladače *ULPDriver*. Slouží jako klientská část pro přenos komunikace s µLAN sběrnicí přes TCP socket. Jako serverová protistrana slouží aplikace *ulpsrvd*, která je popsána v kapitole 4.6 . Při vytváření instance filtru je třeba předat konstruktoru IP adresu nebo případně doménové jméno a číslo TCP portu, na kterém je spuštěna serverová část. Metoda *processMessage()* je implementována, pomocí JSON RPC tak, že předanou zprávu serializuje do formátu JSON a následně zapíše do otevřeného socketu. Při příchozích datech ze socketu jsou tato deserializována do objektu *ULPMessage* a ten je následně signálem *messageReady()* předán nadřazenému filtru.

4.3 Knihovna libulpulan a filtr ULPFDDriver

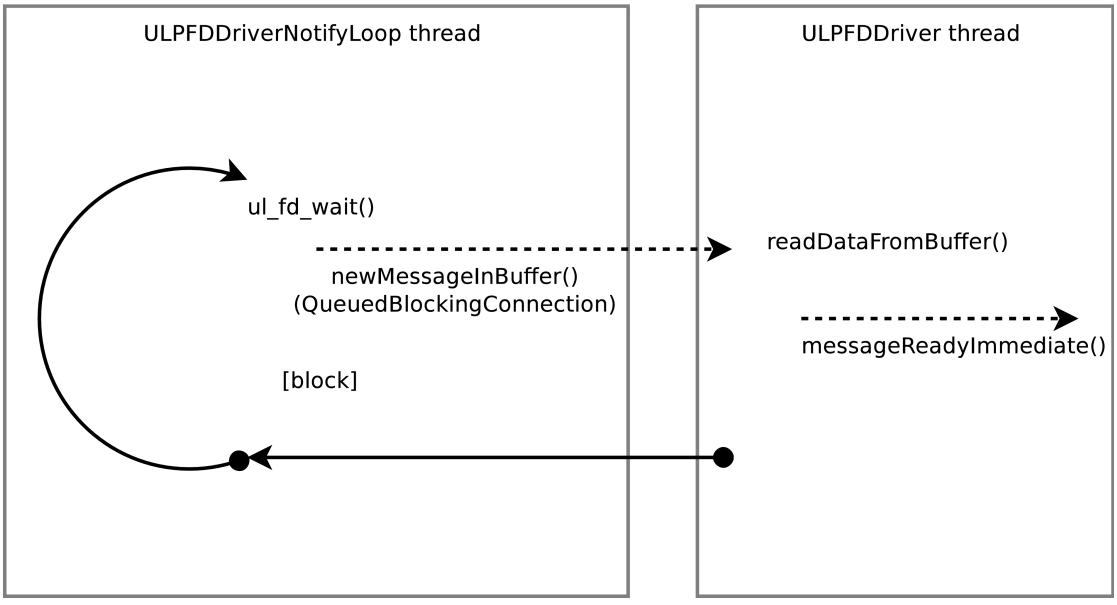
Dynamicky linkovaná knihovna *libulpulan* obsahuje implementaci ovladače *ULPFDDriver*. Důvodem, proč je tento ovladač v samostatné knihovně odděleně od

ostatních filtrů, byla snaha o odstranění závislosti knihovny *libulproxy* na nízkoúrovňovém µLAN C-API.

4.3.1 Komunikace se sběrnicí

Třída *ULPFDDriver* do sebe zapouzdřuje přístup ke sběrnici prostřednictvím otevřeného zařízení */dev/ulan*. Nutnost číst a zapisovat do otevřeného popisovače zároveň vedla k vícevláknové implementaci ovladače. Z technických omezení µLAN C-API však musí operace čtení z otevřeného deskriptoru probíhat ve stejném vlákně, ve kterém byl otevřen a ve kterém se do něj zapisuje. Tento problém se řeší pomocným vláknem *ULPFDDriverNotifyLoop*, které se spustí při inicializace ovladače. Tato smyčka čeká zavoláním funkce *ul_fd_wait*, dokud nejsou připravena data pro přečtení ze sběrnice. Jakmile se tak stane, emituje signál *newMessageInBuffer()*. Tento signál je spojen se slotem *readDataFromBuffer()*. Ten se již nachází ve třídě *ULPFDDriver*. Jelikož je spojení realizováno jako *QueuedBlockingConnection*, dojde při emitování signálu k přepnutí do kontextu hlavního vlákna aplikace a vlákno *ULPFDDriverNotifyLoop* je zablokováno, dokud neproběhne návrat ze slotu. Nyní dojde k vlastnímu přečtení dat ze sběrnice pomocí funkce *ul_read*, data jsou uložena do bufferu nové *ULPMessage* zprávy a tato zpráva je předána signálem *messageReadyImmediate()* dále ke zpracování v ovladači. Touto akcí také končí volání slotu *readDataFromBuffer()*.

Obrázek 10: ULPFDDriver - čtení dat ze sběrnice



Zdroj: vlastní zpracování

Při zpracování proběhne nejdříve podle příznaku *cmd* zjištění, o jaký typ zprávy se jedná. V současnosti *ULPFDDriver* podporuje zpracování těchto zpráv:

- **Network Control Message** (*UL_CMD_NCS*) – servisní zprávy pro správu zařízení připojených ke sběrnici. Ovladač tento typ zpráv používá ke zjišťování, která zařízení jsou přítomna.

- **PDO Message** (*UL_CMD_PDO*) – procesní zprávy.

- U ostatních zpráv se předpokládá, že jde o zprávy objektového rozhraní, zde probíhá další zjišťování typu zprávy podle OID obsaženého ve zprávě.

Podrobný popis zpracování těchto typů zpráv v ovladači je probrán v sekcích 4.3.3 , 4.3.4 a 4.3.5 .

Odesílání zpráv na sběrnici se provádí voláním metody *processMessage()* nad instancí *ULPFDDriver*. Podle logické cesty obsažené ve zprávě je rozhodnuto, jaký typ zprávy je odeslán na sběrnici:

- Pokud logická cesta odpovídá dotazu na seznam zařízení, resp. dotazu na informace o jednom zařízení, je zkonstruován dotaz typu *UL_CMD_NCS* s příkazem *ULNCS_SID_RQ* obsaženým v prvním bajtu dat. Cílová adresa je buď broadcast, pokud šlo o dotaz na seznam zařízení, nebo konkrétní adresát.

- Pokud logická cesta odpovídá dotazu na seznam objektových proměnných, je zkonstruována dvojice zpráv objektového rozhraní (*UL_CMD_OISV*), první s dotazem na seznam proměnné pro čtení s příkazem *ULOI_QOIO*, druhá s dotazem na proměnné pro zápis *ULOI_QOII*. U obou zpráv je zasláno maximální číslo dotazovaného OID v hodnotě 0xFF.

- Při dotazu na čtení, resp. zápis objektové proměnné, je nutno nejdříve ověřit, je-li pro danou proměnnou znám datový typ. Pokud tomu tak je, může ovladač ihned přikročit ke zpracování zprávy. V případě zápisu do objektové proměnné proběhne konverze objektu *QVariant*. Pokud datový typ není znám, je nutno provést jeho zjištění. Tomuto případu se věnuje následující sekce.

4.3.2 Konverze datových typů

Při odesílání a přijímání zpráv objektového rozhraní, které zapisují nebo čtou hodnotu proměnné v objektovém slovníku zařízení je nutno znát datový typ této proměnné. Objekt *ULPFDDriver* má k tomuto účelu navržené struktury, kde se tyto informace uchovávají. Každé zařízení, se kterým probíhá komunikace, má v ovladači pod klíčem „<id_sítě>_<adresa_zařízení>“ uloženu strukturu *ULPFDDriverDeviceData*. V této struktuře je držena odděleně mapa typů objektových proměnných pro čtení, resp. zápis. Mapa je indexována OID objektové proměnné. Zároveň jsou v této struktuře dvě fronty zpráv, jedna pro odchozí, druhá pro příchozí. Pokudje při zpracování zprávy v ovladači třeba znát datový typ proměnné a tento není k dispozici, je zpráva podle kontextu uložena do jedné z těchto front a ovladač odešle zprávu s dotazem na datový typ pro požadovaný OID.

Jakmile ze sběrnice dorazí zpráva s informací o datovém typu proměnné, ovladač ji zařadí do správné struktury *ULPFDDriverDeviceData* a znovu se pokusí odeslat zprávy, které čekají na informaci o datovém typu pro tuto proměnnou.

Pro samotnou konverzi binárních dat do typu *QVariant* a zpět na základě textového předpisu z objektového slovníku µLAN byla vytvořena knihovna *libulqconv*. Ta je popsána v sekci 4.4 .

4.3.3 Zpracování zpráv typu UL_CMD_NCS

Při přijetí zprávy typu *UL_CMD_NCS* proběhne podle prvního bajtu dat ověření zda jde o zprávu typu *ULNCS_SID_RPLY*. Pokud tomu tak je, ovladač založí pro adresu, ze které zpráva přišla, strukturu *ULPFDDriverDeviceData*, pokud dosud neexistovala, a zresetuje její tzv. *clean_counter* na nulu. Tento čítač slouží pro detekci odpojených zařízení. Ovladač v pravidelném časovém intervalu odesílá dotaz *ULNCS_SID_RQ* broadcastem do celé sítě. V každém tomto cyklu je čítač všech zařízení inkrementován o jednu. Pokud pro některé zařízení dosáhne čítač limitní hodnoty *DEVICE_CLEAN_THRESHOLD*, je zařízení odstraněno a je emitována zpráva informující o této změně.

4.3.4 Zpracování zpráv typu UL_CMD_PDO

Procesní zprávy jsou na sběrnici zařízeními vysílány asynchronně podle pravidel zapsanch v jejich POCO a PEV2C tabulkách. Procesní zpráva může být určena jednomu adresátovi nebo všem zařízením na sběrnici. Zároveň může každá zpráva obsahovat více samostatných bloků, například hodnoty více proměnných. Aby ovladač rozuměl obsahu PDO zprávy, musí znát datové typy objektových proměnných, které jsou ve zprávě obsaženy. Navíc musí mít zařízení, kterému je zpráva určena, ve své PICO konfiguraci předpis, jak s daty ve zprávě naložit. V současnosti ovladač podporuje pro PDO zprávy pouze zpracování typu *ULOI_CIDFLG_META_OID*, tedy data, která jsou přenášena PDO zprávou jsou přímo hodnotou cílové objektové proměnné, která je definována v POCO tabulce.

Blok dat PDO zprávy je zpracováván postupně. Pokud ovladač nezná datový typ cílové proměnné, přikročí rovnou ke zpracování dalšího bloku. V opačném případě ovladač zkonstruuje *ULPMessage* objekt s hodnotou konvertovanou do *QVariant* a emituje ho signálem *messageReady()* a pokračuje zpracováním dalšího bloku.

4.3.5 Zpracování objektových zpráv

Ovladače *ULPFDDriver* podporuje zpracování těchto typů zpráv objektového rozhraní:

- Seznam OID proměnných pro čtení, resp. zápis v zařízení (*ULOI_QOIO + 1*, resp. *ULOI_QOII + 1*). Jakmile ovladač přijme zprávu se seznamem proměnných, zaregistruje si všechny nové OID ve struktuře *ULPFDDriver-DeviceData* a označí si je jako neznámé, pokud dosud nebyly přítomny. Následně pro každý nový OID spustí dotaz na jeho datový typ.

- Popis datového typu proměnné pro čtení, resp. zápis (*ULOI_DOIO + 1*, resp. *ULOI_QOII + 1*). Při přijetí této zprávy ovladač vytvoří na základě textového popisu datového typu objekt *ULQDataTypeFactory*, který je později používán pro konverzi mezi binárními daty a *QVariant*. Instance *ULQDataTypeFactory*

je uložena ve struktuře *ULPFDDriverDeviceData* a indexována pomocí OID objektové proměnné, kterou popisuje.

- Odpověď na požadavek čtení dat z objektové proměnné (*ULOI_RDRQ + 1*). Tato zpráva s sebou nese binární data s hodnotou proměnné. V případě, že ovladač zná její datový typ, jsou data ihned rozparsována, je sestavena nová *ULPMessage* s *QVariant* hodnotou a následně emitována do nadřazeného filtru. Pokud není datový typ znám, zpráva je zařazena do fronty a následuje odeslání požadavku na datový typ tak, jak je popsáno v sekci 4.3.2 .

V bloku dat přijatých ze sběrnice může být zřetězeno několik objektových zpráv za sebou. Při zpracování bloku dat je vždy přečtena první, pokud může být zpracována, ihned se pokračuje na další zprávě, atd.

Poněvadž největší část logiky pro zpracování zpráv v ovladači *ULPFDDriver* se týká příchozích objektových zpráv a odchozích zpráv na sběrnici, jsou tyto procesy znázorněny v diagramech v přílohách 1 a 2.

4.4 Knihovny libulqconv a libulqextras

Dvojice dynamických knihoven *libulqconv* a *libulqextras* řeší problematiku konverze mezi binárními daty ve formátu µLAN, které se používají v objektovém rozhraní, a typem *QVariant*. Funkcionalita je rozdělena do dvou knihoven z důvodu minimalizace závislosti na µLAN C-API. Knihovna *libulqextras* definuje rozšíření typu *QVariant* o struktury pro reprezentaci µLAN typů PICO, POCO a PEV2C. Tato část není závislá na C-API µLAN. Knihovna *libulqconv* pak obsahuje vlastní konverzní funkce a je potřeba ji sestavovat spolu s knihovnami µLAN.

4.4.1 Knihovna libulqconv

Knihovna *libulqconv* definuje třídu *ULQDataTypeFactory*, která obsahuje funkcionalitu pro konverzi dat mezi formátem µLAN a *QVariant*. Při vytváření instance třídy *ULQDataTypeFactory* je třeba předat textový popis datového typu µLAN. Na základě tohoto popisu jsou poté prováděny konverze.

Pro převod binárních dat µLAN do *QVariant* slouží funkce *u2q()*. Jako argument přijímá data ve formátu *QByteArray*. Druhým argumentem je ukazatel na *int*. Funkce uloží na jeho adresu počet bajtů, které byly při převodu z pole *QByteArray* zpracovány. Návratová hodnota je výsledný *QVariant*.

Opačný převod z *QVariant* do dat ve formátu µLAN se provádí funkcí *q2u()*. Vstupní argument typu *QVariant* je převeden do pole *QByteArray* s výslednými daty.

Protože v objektových slovnících lze definovat vícerozměrná pole, fungují obě funkce *u2q()* a *q2u()* rekurzivně. Každé zanoření v rekurzi zpracuje jednu úroveň pole.

Každému datovému typu byl přiřazen protějšek v C++, případně v Qt. Pro µLAN datový typ *execute* - „*e*“ je stanovena výjimka, jelikož tento typ v sobě nenese žádnou hodnotu a proměnné jsou pouze pro zápis. V *QVariant* mu odpovídá hodnota *QVariant::Invalid*. Pro uložení datových typů PICO, POCO a PEV2C jako *QVariant* jsou použity odpovídající uživatelsky definované struktury z knihovny *libulqextras*.

Tabulka 3: Přiřazení C++/Qt a µLAN datových typů

Datový typ µLAN	**Datový typ C++/Qt**
s1	int
s2	
s4	
u1	unsigned int
u2	
u4	
f2	double
f4	
f8	
vs	QString
ee	QVariant::Invalid
PICO	ULQ_PICO
POCO	ULQ_POCO
PEV2C	ULQ_PEV2C

Zdroj: vlastní zpracování

4.4.2 Knihovna libulqextras

Tato knihovna deklaruje struktury pro reprezentaci µLAN typů PICO, POCO a PEV2C. Jsou to tyto:

- **ULQ_PICO**
- **ULQ_POCO**
- **ULQ_PEV2C**

Každá struktura obsahuje atributy odpovídající polím v příslušném datovém typu µLAN. Struktury jsou deklarovány jako uživatelské rozšíření *QVariant* pomocí makra *Q_DECLARE_METATYPE*. Díky tomu s nimi lze přirozeně pracovat ve všech funkcích, kde se předává jako argument typ *QVariant*.

Knihovna dále obsahuje třídu *ULQExtras*, která obsahuje statické metody pro práci s výše uvedenými strukturami. Jde o funkce umožňující zjišťování, zda je obsahem proměnné *QVariant* některá z výše uvedených struktur, a konverzní funkce pro převod těchto struktur do instanci *QVariantMap* a obráceně.

4.5 Aplikace ulanbrowser

Aplikace *ulanbrowser* je jednoduchý nástroj s grafickým uživatelským rozhraním, umožňující procházet zařízení na sběrnici a zobrazovat a modifikovat hodnoty jejich objektových proměnných. Jako zdroj dat pro aplikaci slouží instance filtru *ULPProxy*. Hlavní okno aplikace *ulanbrowser* je zobrazeno v příloze 3.

Základem aplikace je dvojice tříd odvozených od modelu *QAbstractItemModel*:

- *DeviceTreeModel*
- *OIDTableModel*

DeviceTreeModel slouží pro reprezentaci dat stromové struktury zařízení na sběrnici. Pro přístup k těmto datům se používá rozšířené API filtru *ULPProxy*, tj. metoda *data()*

a dvojice signálů *dataAboutToBeChanged()* a *dataChanged()*. Pro zobrazování stromu je určena instance standardního *QTreeView*.

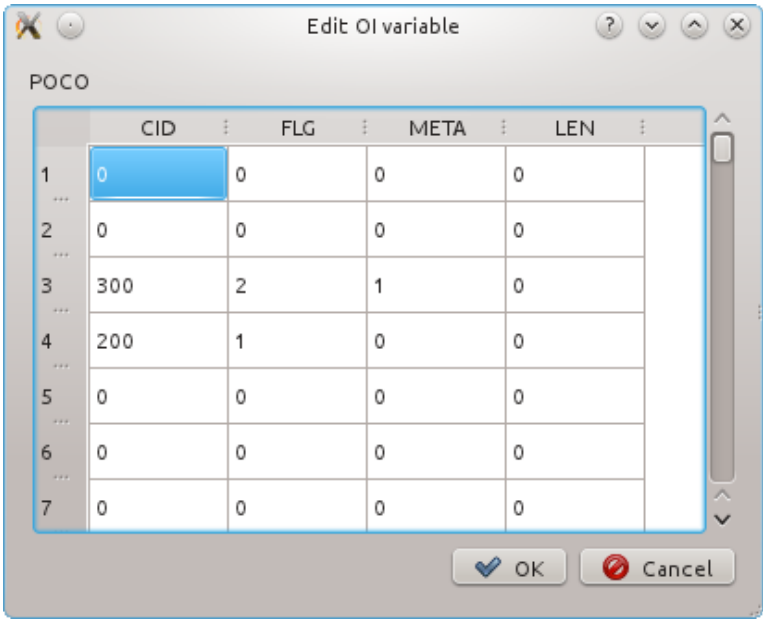
Pro reprezentaci tabulky objektových proměnných slouží druhý jmenovaný model – *OIDTableModel*. Jako zdroj dat mu slouží seznam objektových proměnných jednoho vybraného zařízení v modelu *DeviceTreeModel*. Signalizace změny výběru se provádí signálem *currentChanged()*. Data modelu *OIDTableModel* jsou zobrazována pomocí *OIDTableView*. Tato třída dědí standardní *QTableView* a přetěžuje funkci *edit()* sloužící jako handler pro úpravu dat v tabulce. Tabulka zobrazuje v pěti sloupcích OID objektové proměnné, její název, µLAN popis datového typu, informaci, zda je proměnné pro čtení/zápis a její hodnotu, pokud jde o proměnnou pro čtení. Editace hodnoty se provádí přímo v tabulce dvojklikem na buňku tabulky. Pokud se jedná o proměnnou typu pole, je v buňce s hodnotou zobrazen pouze text „[array]“ a dvojklik na ní vyvolá dialogové okno, ve kterém lze pole upravovat. Přístup k poli v tomto dialogu je opět realizován pomocí model-view komponent. Data jsou reprezentována modelem *OIDEditTableModel* a pro zobrazování slouží standardní *QTableView*. Model počítá kromě pole s primitivními datovými typy také s variantou, kdy jsou prvky struktury PICO, POCO nebo PEV2C.V takovém případě má editační tabulka více sloupců a jejich záhlaví jsou označena názvy příslušných atributů těchto struktur.

Kromě této funkcionality obsahuje *ulanbrowser* ještě nástroj pro hromadné ukládání a obnovování PDO konfigurace zařízení na sběrnici. Jelikož PDO konfigurace je uložena v objektových proměnných s názvy PICO, POCO, PEV2C a PIOM, stačí pro získání aktuální konfigurace projít všechna zařízení a dotázat se každého na čtveřici těchto hodnot. K těmto hodnotám je ve struktuře *QVariantMap* doplněna adresa zařízení ze kterého pocházejí. Získaná data jsou následně serializována do formátu JSON a uložena na disk. Opačný proces je pak deserializace těchto dat a jejich zápis do objektových proměnných patřičných zařízení.

V aplikaci *ulanbrowser* lze použít oba dva implementované typy ovladačů – *ULPFDDriver* a *ULPSocketDriver*. To, který z nich se použije lze určit jako parametr na příkazovém řádku při spuštění. Výchozím typem je ovladač *ULPFDDriver*, který se

připojuje k zařízení „*/dev/ulan*“. Jiný název souboru lze určit parametrem „-d“. Použití filtru *ULPSocketDriver* lze deklarovat parametrem „-s“. V tomto případě je nutno specifikovat hostitele a číslo TCP portu ve formátu „host:port“.

Obrázek 11: Dialog pro editaci objektové proměnné typu „pole“



Zdroj: vlastní zpracování

4.6 Aplikace ulpsrvd

Aplikace *ulpsrvd* je jednoduchý TCP server umožňující komunikovat se sběrnicí µLAN vzdáleně pomocí formátu JSON. Jedná se o protistranu ke ovladači *ULPSocketDriver*. Aplikace umožňuje připojení více TCP klientů zároveň. Jejich zprávy zaslané ve formátu JSON přes TCP socket jsou deserializovány do objektů *ULPMessage* a poté předány prvnímu filtru v komunikačním řetězci v *ulpsrvd*. Naopak při přijetí zprávy z filtru je tato převedena do JSON formátu a zaslána TCP socketem všem připojeným klientům. Celý komunikační řetězec je v aplikaci *ulpsrvd* tvořen filtrem *ULPProxy* a následně ovladačem *ULPFDDriver*, který již komunikuje přímo se sběrnicí. Teoreticky ovšem není problém upravit *ulpsrvd* tak, aby bylo možno jeho instance libovolně řetězit za použití TCP komunikace.

Vlastní server pro obsluhu klientů je implementován ve třídě *ULPSrvd*, která dědí od třídy *QTcpServer*. Spojení s klientem je pak reprezentováno instancí třídy *ULPSrvClient*, odděděné od *QTcpSocket*. Při příchozím spojení je zavolána metoda,

incomingConnection(), které je předáno číslo otevřeného klientského TCP socketu. Na základě tohoto čísla je zkonstruován nový objekt *ULPSrvClient*. Při jeho vytvoření je propojen jeho signál *readyRead(),* indikující příchozí data, se slotem *onReadyRead()*. Ve slotu jsou data z TCP socketu přečtena a následně emitována signálem *messageReady()* do slotu *writeToSourceFilter()* v objektu *ULPSrvd*. Ukazatel na vytvořenou instanci třídy *ULPSrvClient* je uložen jako prvek pole ve třídě *ULPSrvd*. Pří příchozí zprávě z prvního filtru je proveden zápis dat vždy do všech socketů v tomto poli. Využití signálu *readyRead()* a přetížení třídy *QTcpSocket* umožnilo realizovat server v jednom vlákně.

Při spuštění lze z příkazové řádky předat přepínačem „-s“ adresu a TCP port ve formátu „host:port“, na kterém má démon naslouchat. Ve výchozím nastavení se spouští na adrese *0.0.0.0:6789*. Druhým parametrem „-d“ lze specifikovat soubor zařízení µLAN pro ovladač *ULPFDDriver*. Výchozí je „*/dev/ulan*“.

5 Zprovoznění systému

Následující kapitola popisuje nutné kroky ke zprovoznění ovladače µLAN a zprovoznění softwaru vytvořeného v rámci této diplomové práce. Postup byl testován na PC s operačním systémem GNU/Linux Ubuntu 11.10 Oneiric Ocelot 32-bit s verzí jádra *3.0.0-14-generic*. Pro ostatní linuxové distribuce by se neměl příliš lišit, hlavní rozdíly mohou nastat v názvech balíčků, na nichž ulan-admin software závisí.

V poslední části je pak stručně popsán projekt ulan-genmod, který byl vyvíjen kolegou Bc. Janem Štefanem paralelně s knihovnami ulan-admin, a který využívá některé jejich komponenty.

5.1 Získání zdrojových kódů a překlad

Zdrojové kódy jsou uloženy v GIT repozitáři služby *sourceforge.net* na adrese *git://ulan.git.sourceforge.net/gitroot/ulan/ulan-admin*. Celý proces zprovoznění je rozdělen do několika částí:

- instalace nutných balíčků
- stažení, konfigurace a přeložení zdrojových kódů µLAN pro architekturu x86
- zavedení ovladače µLAN do jádra
- stažení zdrojových kódů ulan-admin a jejich překlad

Jelikož se předpokládá, že ulan-admin je používán spolu s projektem ulan-genmod, posledním krokem je stažení a překlad aplikace ulan-genmod.

Pravděpodobně nejproblematičtějším bodem je překlad samotného µLAN ovladače. Provádí se nadvakrát. Po stažení repozitáře projektu µLAN je třeba spustit skript *build-ulan-host*, nacházející se v podadresáři *scripts* v repozitáři. Tím se provede založení adresáře *ulan-build*, ve kterém jsou umístěny výsledky následně spuštěné

kompilace. Dáile je třeba provést konfiguraci tohoto nově vytvořeného prostředí. To se provede založením souboru *config.omk* v adresáři *ulan-build/host* s těmito volbami:

- *CONFIG_OC_UL_DRV_WITH_VIRTUAL=y* – zapíná podporu virtuálních zařízení v ovladači µLAN
- *CONFIG_OC_UL_DRV_WITH_MULTI_DEV=y* – zapíná podporu spuštění více submodulů s vlastními µLAN adresami v rámci jednoho zařízení
- *CONFIG_OC_UL_DRV_WITH_IAC=y* – zapíná podporu okamžité identifikace (*UL_CDM_SID*) ve službě *NCS*
- *CONFIG_ULOI_CON_IO_OPS=y* – zapíná volitelný způsob načítání dat a metadat – využití v PICO a POCO konfiguračních tabulkách
- *CONFIG_ULAN_DY=y* – zapíná podporu služeb dynamické adresace – nutné kvůli službám *NCS*
- *CONFIG_ULOI_LT=y* – zapíná podporu objektového rozhraní
- *-fPIC* – zajistí generování position-independent kódu (PIC) v knihovně funkcí ovladače µLAN[4]

Po provedení této konfigurace je třeba spustit *make* v adresáři *ulan-build/host*. Tímto dostaneme jaderný modul ovladače µLAN s funkcionalitou nutnou pro ulan-admin a ulan-genmod.

Poslední fází je přeložení projektů ulan-admin a ulan-genmod. Projekt ulan-admin používá standardní Qt qmake, proto je třeba v jeho adresáři spustit příkaz *„qmake -r*“ a následně „*make*“. U projektu ulan-genmod je qmake volán sestavovacím systémem OMK, stačí proto spustit pouze příkaz „*make*“.

Pro zjednodušení celého tohoto procesu byl vytvořen skript *ulan_build_script*, který se nachází na přiloženém CD. Ten automatizuje výše zmíněné kroky a navíc provede úpravu konfigurace *udev*, aby bylo možno bez práv uživatele root přistupovat

4 Zdroj: http://www.akkadia.org/drepper/dsohowto.pdf [6]

k souboru zařízení */dev/ulan*. Skript zajistí po přeložení ovladače µLAN jeho zavedení do jádra.

5.2 Zprovoznění verze pro Android

Během vývoje projektu ulan-admin byla zjištěna možnost spouštět Qt aplikace na operačním systému Android. Bylo rozhodnuto, že součástí práce bude i odzkoušení, zda je možno zprovoznit aplikaci *ulanbrowser* v kombinaci s ovladačem *ULPSocketDriver* pod tímto operačním systémem. Kapitola popisuje dodatečné úpravy, které musely být provedeny, aby bylo možno aplikaci *ulanbrowser* spustit.

Systém Android je mobilní operační systém vyvíjený společností Google. Podpora aplikací napsaných za použití knihovny Qt je zajišťována projektem *Necessitas*[5]. Ten se v současnosti nachází v alfa fázi vývoje. Instalace v sobě zahrnuje knihovnu Qt ve verzi 4.8 upravenou pro systém Android, sestavovací nástroj *ant* a upravený nástroj Qt Creator, který umožňuje přímo spouštět vyvíjený software v dodávaném emulátoru nebo na mobilním telefonu připojeném přes USB kabel. Součástí projektu *Necessitas* jsou i sdílené Qt knihovny předkompilované pro systém Android. Tyto je potřeba nainstalovat do telefonu přes instalátor *Ministro*, který lze získat na portále Android Market.

Cílem bylo spustit aplikaci *ulanbrowser* na mobilním telefonu a s využitím ovladače *ULPSocketDriver* se připojit k PC, které mělo přístup na sběrnici µLAN. Dosažení tohoto cíle je zaznamenáno v příloze 4.

5.2.1 Rozdělení sdílených knihoven ulan-admin

Pro portování *ulanbrowseru* na Android bylo nezbytné oddělit knihovny na části závislé na µLAN C-API od zbytku projektu. Kromě aplikace *ulanbrowser* bylo nutno přeložit knihovnu obsahující *libulproxy* filtry *ULPProxy* a *ULPSocketDriver* Z tohoto důvodu z ní musel být vyňat ovladač *ULPFDDriver* a přemístěn do samostatné

5 Domovská stránka projektu: http://sourceforge.net/p/necessitas/home/necessitas/

knihovny *libulpulan*. Při překladu aplikace *ulanbrowser* se neprovádí linkování s knihovnou *libulpulan*. V důsledku tohoto rozdělení se ovladač *ULPFDDriver* stal pluginem, který se při spouštění na PC dynamicky linkuje za běhu pomocí třídy *QLibrary*.

Další částí, na které je *ulanbrowser* závislý, jsou definice Qt struktur *ULQ_PICO*, *ULQ_POCO* a *ULQ_PEV2C* pro reprezentaci PDO konfiguračních tabulek. Tyto definice byly původně v knihovně *libulqqttypes*, která navíc obsahovala i funkcionalitu pro převod hodnot mezi µLAN formátem a *QVariant* a byla tedy závislá na µLAN C-API. Tato knihovna byla tedy rozdělena do dvou nových: *libulqconv* a *libulqextras*. Tím se podařilo z druhé jmenované odstranit závislost na µLAN C-API.

5.2.2 Statické linkování s aplikací ulanbrowser

Při spouštění aplikace *ulanbrowser* na mobilním telefonu bylo zjištěno, že *Necessitas* nesprávně interpretuje závislosti programu na dynamických knihovnách a tyto knihovny poté nejsou součástí dávky, která je při spuštění nahrána do telefonu. Tento problém je popsán v příslušném ticketu na stránkách projektu a měl by být opraven ve verzi beta1[6].

V projektu ulan-admin byl problém vyřešen statickým linkováním knihoven *libqjson*, *libulproxy* a *libulpulan* k aplikaci *ulanbrowser*. Při tomto způsobu kompilace se knihovny stanou součástí výsledného programu. Do konfiguračních .pro souborů celého projektu ulan-admin byla přidána detekce cílové architektury pomocí testu na přítomnost qmake proměnné *ANDROID_TARGET_ARCH*. Tato proměnná je inicializována pouze při překladu pomocí toolkitu *Necessitas*. Díky tomuto bylo možno ukládat výsledky kompilace pro Android do jiného adresáře odděleně od výsledků překladu pro PC:

- Dynamicky linkované verze knihoven vytvářené při překladu pro PC jsou umístěny v adresáři *_build/host/lib*.
- Staticky linkované varianty knihoven pro Android jsou v adresáři *_build/<ANDROID_TARGET_ARCH>/lib*.

6 Hlášený problém s dynamickými knihovnami: http://sourceforge.net/p/necessitas/tickets/106/

Při této úpravě byly do projektu zahrnuty i zdrojové kódy knihovny *libqjson*, která se původně používala z repozitáře operačního systému, což znamenalo problém při jejím využití ve staticky i dynamicky linkované verzi současně. Knihovna je umístěna v adresáři *ulan-admin/3rdparty*.

5.3 Aplikace ulan-genmod

Paralelně s projektem ulan-admin by vyvíjen kolegou, Bc. Janem Štefanem, nástroj pro návrh virtuálních zařízení ulan-genmod. Aplikace ulan-genmod umožňuje připojit ke sběrnici µLAN virtuální zařízení. Popis takovéhoto zařízení je rozdělen do dvou souborů:

- Grafická reprezentace je popsána jazykem QML (Qt Modeling Language).
- Popis µLAN vlastností je uložen jako XML v souboru .xds. Tento popis zahrnuje adresu zařízení, jméno a definici proměnných objektového slovníku.

Aplikace umožňuje uložit celou konfiguraci virtuální sítě. Tuto konfiguraci je poté možné přenést do fyzických zařízení. Navíc je možná interakce mezi reálnými a virtuálními zařízeními.

6 Licencování komponent

Projekt ulan-admin je vyvíjen jako open-source, zdrojové kódy jsou k dispozici na serveru *sourceforge.net*. Všechny součásti projektu včetně znovupoužitelných knihoven *libulproxy*, *libulpulan*, *libulqconv* a *libulqextras* jsou pod LGPL licencí. To umožňuje jejich použití i v komerčních closed-source projektech. Výjimku tvoří knihovna *libqjson*. Ta je vyvíjena pod licencí GPL v2.

Závěr

Cílem práce bylo vytvořit univerzální, multiplatformní a jednoduše rozšiřitelné nástroje pro přístup ke sběrnici µLAN a její správu. Výsledkem je sada knihoven a aplikace *ulanbrowser*, která je využívá.

Univerzálnost a rozšiřitelnost výsledného řešení je zajištěna díky architektuře využívající princip UNIXových filtrů. Dokladem toho jsou kromě samotného filtru *ULPFDDriver*, který je pro komunikaci se sběrnicí esenciální, implementace dalších filtrů *ULPProxy* a *ULPSocketDriver*. Oba tyto filtry posouvají možnosti komunikace se sběrnicí každý zcela jiným směrem, a přitom sdílejí stejné API. Filtr *ULPProxy* dokázal díky cachovací vrstvě výrazně snížit zatížení sběrnice při asynchronním způsobu komunikace. Komunikace se sběrnicí přes TCP protokol je umožněna díky filtru *ULPSocketDriver* v kombinaci s aplikací *ulpsrvd*. Zapouzdření dat v komunikačním kanále mezi filtry je zajištěno třídou *ULPMessage*. Tento kontejner, spolu s návrhem logických cest, který využívá, se ukázal jako velmi univerzální nosič informace mezi filtry.

Díky ovladači *ULPFDDriver* se podařilo uzavřít kód specifický pro práci se sběrnicí µLAN do relativně malého prostoru. Tato skutečnost, společně s univerzálností celé architektury filtrů, dává podnět k využití systému pro přístup i k jiným průmyslovým sběrnicím, než µLAN.

Využitím knihovny Qt při implementaci bylo dosaženo požadované platformní nezávislosti. Důkazem může být relativně snadné zprovoznění nástroje *ulanbrowser* na operačním systému Android.

Část projektu ulan-admin, konkrétně knihovna pro konverzi mezi datovými typy µLAN a *QVariant – libulqconv*, byla ihned použita v paralelně vyvíjeném projektu ulan-genmod. To svědčí o rozumném rozdělení funkcionality do jednotlivých logických celků.

Projekty ulan-admin a ulan-genmod byly jako celek prezentovány na konferenci Real-Time Linux Workshop, která se konala ve dnech 20.-22.10.2011 v budově ČVUT FEL

v Praze. Součástí prezentace bylo připojení aplikace *ulanbrowser* na model inteligentního domu, kde byla kombinována fyzická μLAN zařízení, obsahující poměrně velký počet objektových proměnných, s virtuálními zařízeními z projektu ulan-genmod.

Ačkoliv se podařilo splnit všechny cíle vytyčené na začátku práce, je zde samozřejmě prostor pro další vylepšování celého projektu. Nabízejí se například tyto oblasti dalšího vývoje.

Užitečné by bylo doplnění aplikace *ulanbrowser* o grafický nástroj pro vytváření PDO konfigurace, hlavně pak nastavování pole PIOM. Stejně tak by byla zajímavá možnost připojit se pomocí aplikace *ulanbrowser* na více sběrnic současně. Knihovna *libulq-conv* podporuje zpracování N-rozměrných polí. Tuto podporu je třeba rozšířit i do aplikace *ulanbrowser*. Ten zvládá v současnosti editaci pouze jedno-rozměrných polí. Vylepšení by si zcela jistě zašloužilo uživatelské prostředí aplikace *ulanbrowser* ve verzi pro Android.

Věřím, že projekt ulan-admin najde reálné uplatnění, a že budu mít v budoucnu příležitost dále jej rozvíjet.

Zdroje

[1] Píša Pavel. *ul_drv - uLan RS-485 Communication Driver*. [on-line]. [cit 2010-04-27]. http://ulan.sourceforge.net/pdf/ul_drv.pdf

[2] Root.cz. *Sběrnice RS-422, RS-423 a RS-485*. [on-line]. [cit 2008-12-10]. http://www.root.cz/clanky/sbernice-rs-422-rs-423-a-rs-485/

[3] SourceForge.net. *uLan communication – Open Source implementation of a multi-master protocol*. [on-line]. [cit 2008-12-11]. http://ulan.sourceforge.net/

[4] Qt Reference Documentation. *Qt 4.7*. [on-line]. [cit 2011-12-15]. http://doc.qt.nokia.com/4.7/index.html

[5] Thelin Johan. *Foundations of Qt Development*. Spojené státy americké : Apress, 2007. 528s. ISBN13: 978-1-59059-831-3

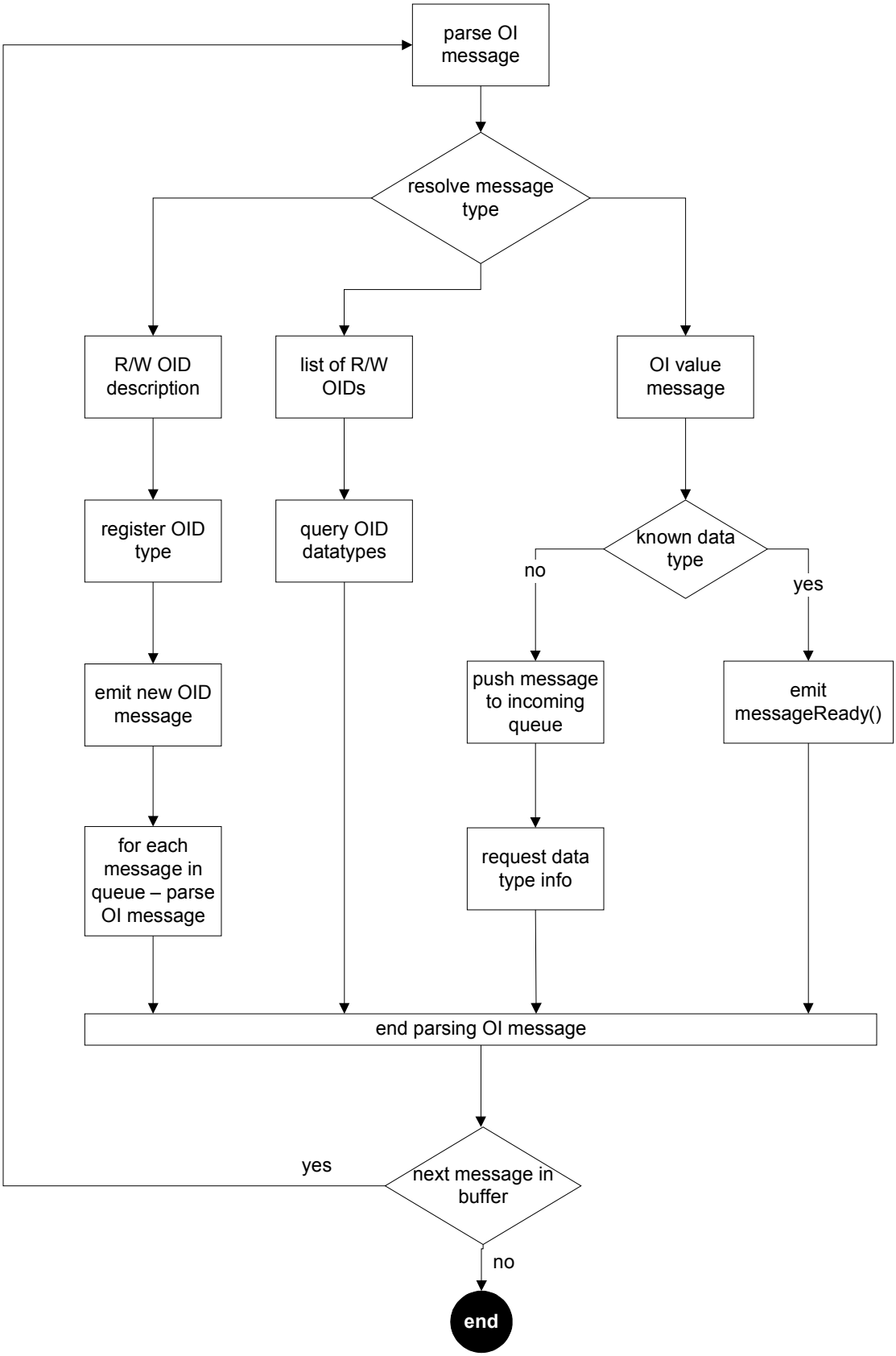
[6] Drepper Ulrich. *How to Write Shared Libraries*. [on-line]. [cit 2011-12-10]. http://www.akkadia.org/drepper/dsohowto.pdf

[7] Píša P., Smolík P., Vacek F., Boháček M., Štefan J., Němeček P. *Process Data Connection Channels in uLan Network for Home Automation and Other Distributed Applications*. [on-line]. [cit 2011-10-28]. https://lwn.net/images/conf/rtlws-2011/proc/Pisa2.pdf

[8] Boháček Martin, *Návrh software pro řízení a monitoring hydroponického systému*. Praha : 2009. 39s.

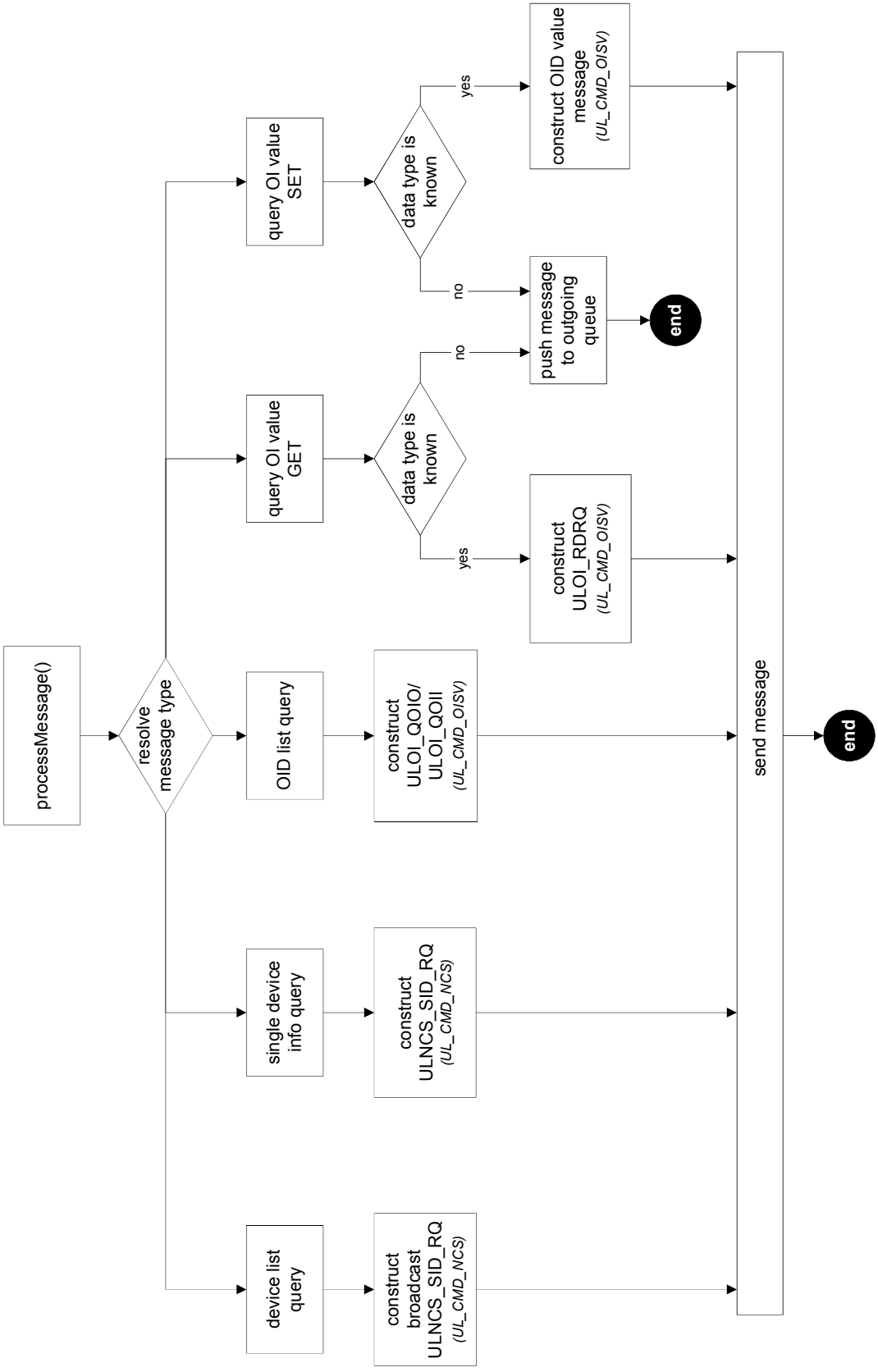
Příloha A – obrázky

Příloha 1: Zpracování příchozí zprávy objektového rozhraní v ovladači ULPFDDriver



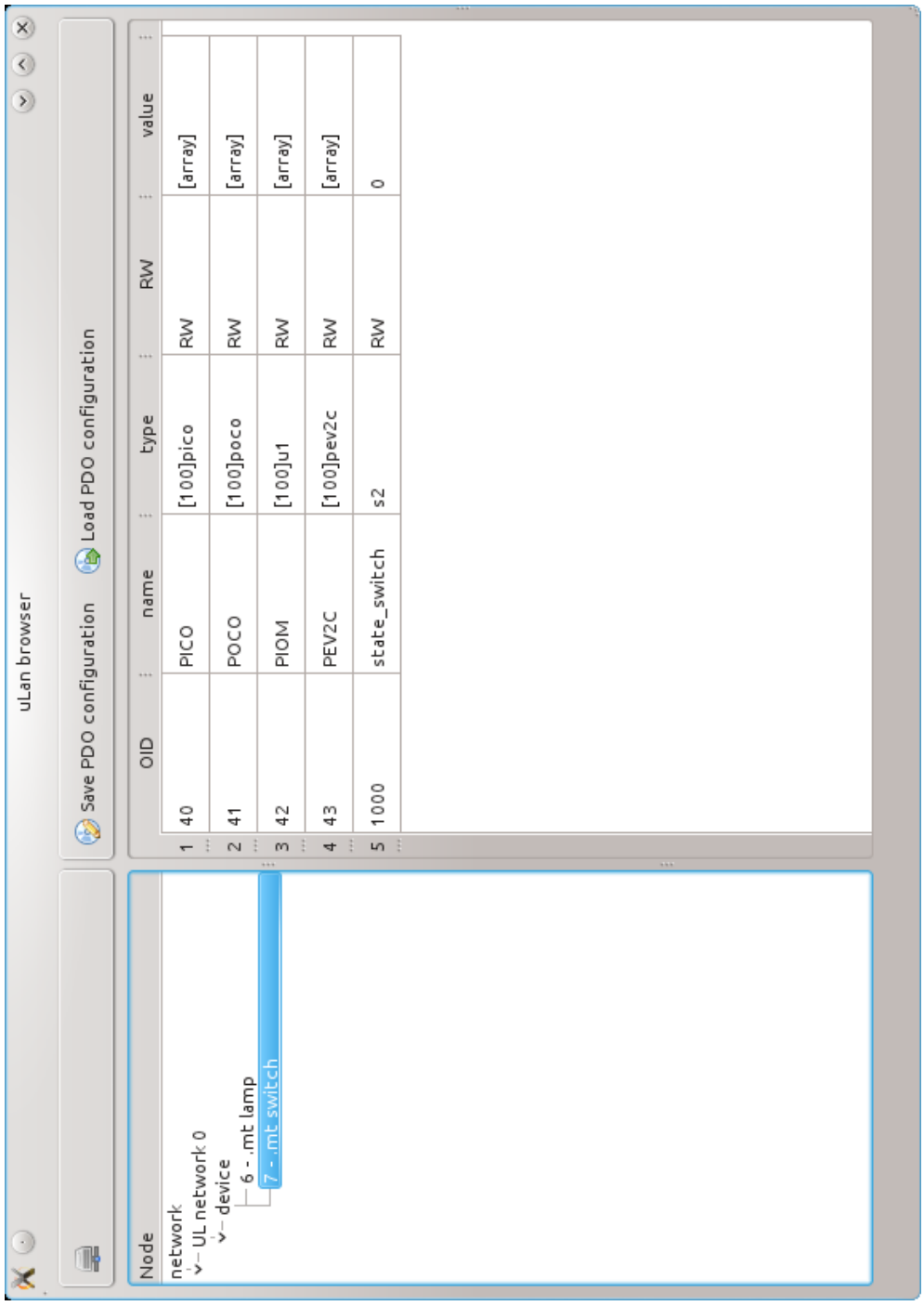
Zdroj: vlastní zpracování

Příloha 2: Zpracování odchozí zprávy v ovladači ULPFDDriver



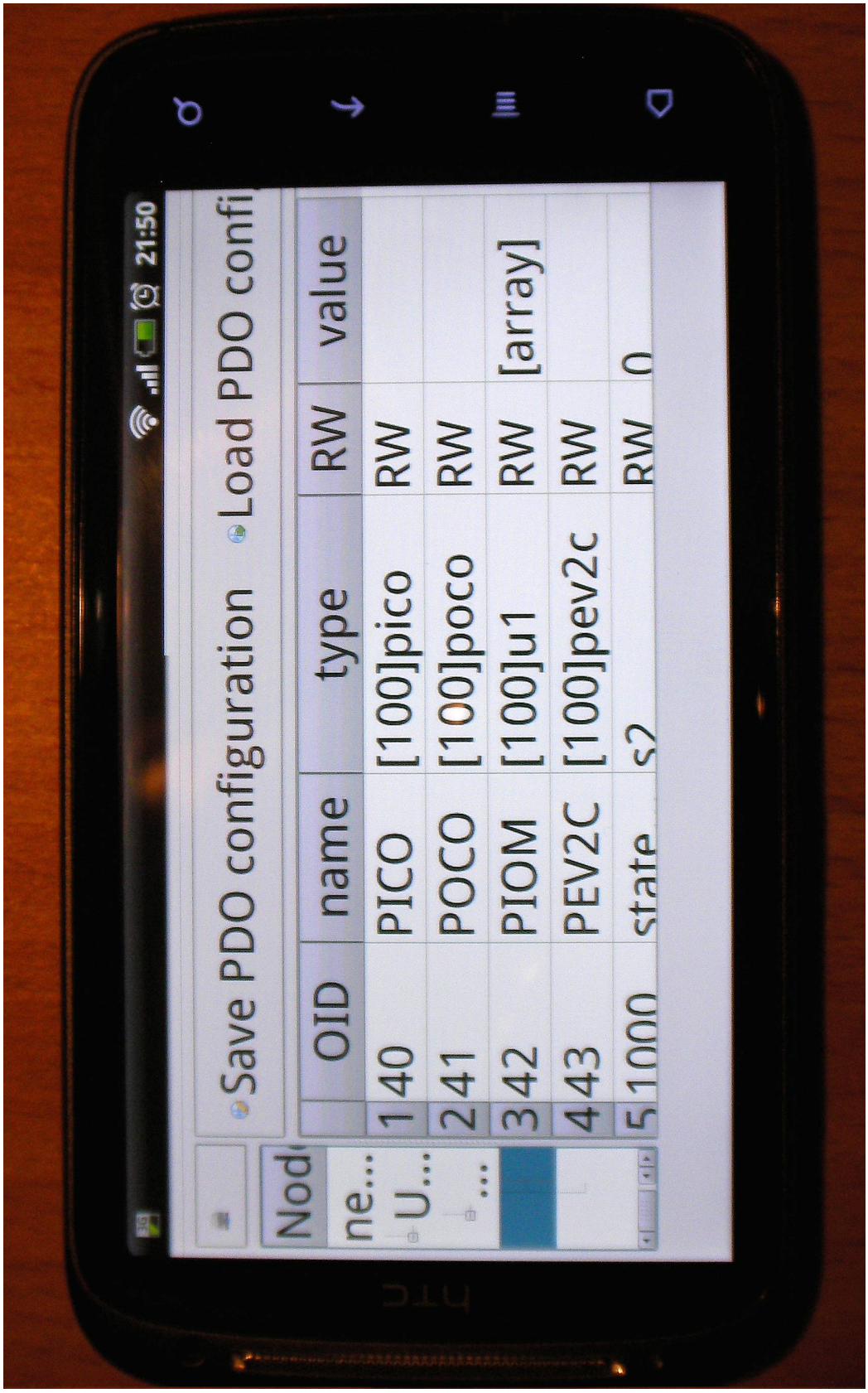
Zdroj: vlastní zpracování

Příloha 3: Hlavní okno aplikace ulanbrowser



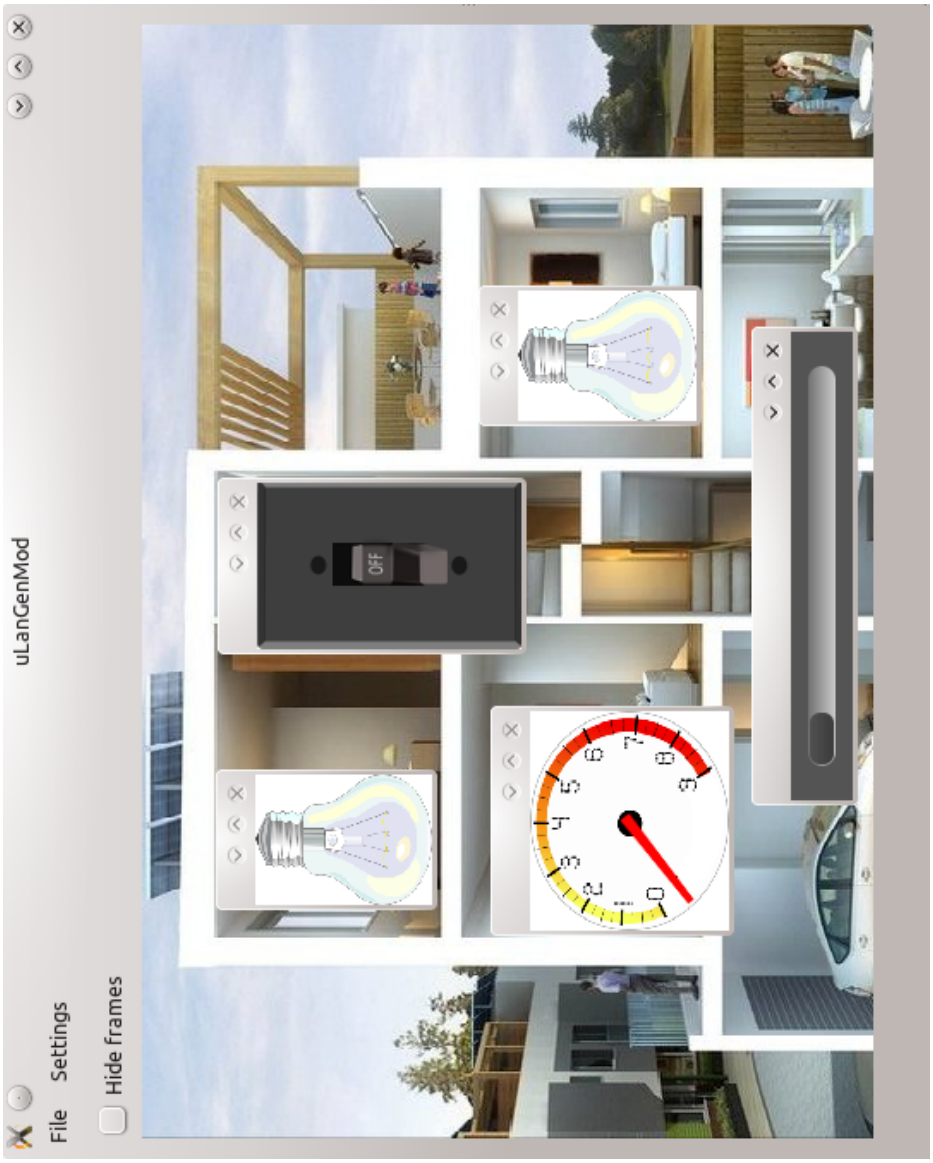
Zdroj: vlastní zpracování

Příloha 4: Aplikace ulanbrowser na mobilním telefonu se systémem Android



Zdroj: vlastní zpracování

Příloha 5: Hlavní okno aplikace ulan-genmod s virtuálními zařízeními



Zdroj: Bc. Jan Štefan

Příloha 6: Prezentace projektů ulan-admin a ulan-genmod na RTLWS 2011

Zdroj: vlastní zpracování

Příloha B – přiložené CD

K této práci je přiloženo CD se skriptem *ulan_build_script* pro automatické stažení a sestavení aplikací ulan-admin a ulan-genmod.