České vysoké učení technické v Praze

Fakulta elektrotechnická

Katedra řídící techniky

Univerzální datový model pro domovní automatizaci

Diplomová práce

Bc. Martin Cicvárek

Praha 2010

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Martin Cicvárek**

Studijní program: Elektrotechnika a informatika (magisterský), strukturovaný
Obor: Kybernetika a měření, blok KM1 - Řídicí technika

Název tématu: **Univerzální datový model pro domovní automatizaci**

Pokyny pro vypracování:

1. Cílem práce je připojit systém WAGO IO SYSTEM do datového modelu systému DAMIC s cílem použití v domovní automatizaci.
2. Seznamte se s možnostmi dynamické detekce vstupně/výstupních modulů v systému WAGO. Uvažujte také systémy DALI a EnOcean, případně i KNX/EIB.
3. Připravte aplikaci pro WAGO, která bude využívat komunikaci Modbus/TCP pro komunikaci s nadřazeným vizualizačním prostředím.
4. Upravte datový model systému DAMIC, aby odpovídal způsobu práce s daty, který používá systém WAGO.
5. Vytvořte vzorovou aplikaci a dobře zdokumentujte všechny postupy a metody.

Seznam odborné literatury:

Dodá vedoucí práce

Vedoucí: Ing. Pavel Burget, Ph.D.

Platnost zadání: do konce letního semestru 2011/2012

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

L.S.

prof. Ing. Boris Šimák, CSc.
děkan

V Praze dne 27. 9. 2010

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil pouze podklady (literaturu, projekty, SW, atd.) uvedené v přiloženém seznamu.

V Praze dne…… ……

Podpis

Poděkování

Děkuji panu Ing. Pavlu Burgetovi Ph.D. za vzorné vedení a trpělivost během zpracování tématu mé diplomové práce.

Rovněž bych rád poděkoval kolegům Ing. Ondřeji Nývltovi, Ing. Ondřeji Fialovi, Ing. Pavlu Krejzovi a Ing. Ivu Kubitovi za konzultace a cenné rady, které mi poskytovali během zpracování této práce a přispěli tak k jejímu vzniku.

ANOTACE

Diplomová práce se zabývá vytvořením aplikace pro automaty WAGO zajišťující univerzální řízení domovní automatizace. Aplikace by měla využít i moderní technologie DALI, EnOcean a KNX/EIB. Dále se práce zabývá úpravou datového modelu vyvíjeného systému DAMIC pro začlenění systému WAGO do tohoto systému a následně vytvořením přidružené aplikace zajišťující propojení automatu s upraveným datovým modelem.

Klíčová slova: Domovní automatizace, systém DAMIC, MODBUS, DALI, EnOcean, KNX/EIB.

ANNOTATION

The thesis deals with the develope of WAGO controller‘s application for universal controlling a home automation system. The application should use modern technologies namely DALI, EnOcean and KNX/EIB. The thesis also deals with the modify of DAMIC model to join WAGO controllers with the DAMIC system. This topic continues with a description of the creation an application for making a connection between the controller and the modified model.

Key words: Home automation, system DAMIC, MODBUS, DALI, EnOcean, KNX/EIB

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod a cíle práce

Automatizační technika se již značně dostává i do domácností v podobě tvz. domovní automatizace. Domovní automatizací se rozumí především ekonomické a komfortní řízení vytápění, ovládání světel, ventilace a žaluzií. Za komfortní řízení je hlavně považováno snadné a rychlé přenastavení ovládání komponent domovní automatizace. Existuje mnoho technologií, které mají rozdílné způsoby ovládání a propojení svých komponent včetně jejich vizualizací. Nevýhodou naprosté většiny z těchto technologií je konfigurace systému pouze pomocí kvalifikovaného pracovníka nebo použitím drahých specializovaných softwarů.

Projekt DAMIC, který je vyvíjen na Katedře řízení Českého vysokého učení technického, má za cíl vytvořit univerzální systém pro vzdálenou vizualizaci, konfiguraci a ovládání domácí automatizace s minimální závislostí na výrobci technologie. Tento projekt by měl umožnit běžnému uživateli snadno měnit vazby a nastavení jednotlivých komponent v domácnosti a tím mu plně dopřát výhod domovní automatizace.

Dnes se v domovní automatizaci běžně používají i centrální řídící jednotky WAGO. Výhodou této technologie jsou jednoduché a tím i levné komponenty. Na druhou stranu hlavní nevýhodou je aplikace naprogramovaná na míru konkrétnímu projektu, což jednak zvyšuje náklady při pořízení a také s sebou nese při budoucím požadavku na změnu vazeb komponent nutnost přeprogramovat aplikaci.

Cílem mé práce je připojit do systému DAMIC řídící jednotku WAGO, která by řídila domovní automatizaci principem systému DAMIC. To znamená vyvinout obecnou aplikaci pro automat WAGO, která by byla uživatelsky plně konfigurovatelná přehráním datového modelu v paměti automatu. Dále upravit datový model systému DAMIC, aby více odpovídal způsobu práce s daty v automatu a vyvinout nadřazenou aplikaci, která by zajišťovala propojení automatu s tímto datovým modelem. Výsledky mé práce budu demonstrovat na dvou modelech domovní automatizace, které má katedra k dispozici.

Kapitola 2

Vize aplikací

2.1 Aplikace pro automat WAGO

Aplikace pro automat WAGO bude postavena na základech aplikace, kterou vytvořil Ing. Nývlt v rámci své diplomové práce v roce 2008 [1]. Program řídí vazby mezi komponentami dle tabulek uložených v paměti a událostech vyhodnocených na vstupech. Na základě fyzických komponent přítomných na jednotlivých modelech domovní automatizace jsem sestavil tabulku komponent s požadovaným chováním, které bude třeba ve vyvíjené aplikaci spravovat.

Digitální vstupy	
Komponenta	**Požadované události**
Tlačítko	jednoduchý klik/dvojklik/držení
Vypínač	sepnuto/rozepnuto.
PIR obvod	přerušen/nepřerušen.
Blokovací kontakt	sepnuto/rozepnuto.
Digitální výstupy	
Komponenta	**Požadované akce**
Světlo	rozsvítit / zhasnout.
Ventilátor	rozběhnout / zastavit.
Ventil	otevřít / zavřít
Kotel	zatopit / zhasnout.
Stmívatelné světlo	rozsvítit / zhasnout / změnit jas
Analogové vstupy	
Komponenta	**Význam hodnoty**
Teplotní čidlo	Teplota v daném prostoru
Zadání teploty	Manuálně požadovaná teplota

Tabulka 2.1 – Základní komponenty

Aplikace by měla umět řídit osvětlení, ventilátory a podobné komponenty ovládané digitálními vstupními signály z vypínačů a tlačítek. Měla by řídit vytápění v domě na základě požadovaných teplot, které lze zadat jak manuálně, tak pomocí uloženého programu. Řízení topení bude probíhat pomocí topných hlavic ovládanými signálem PWM a vytápění se bude moci blokovat okenními kontakty. Rád bych v aplikaci použil i funkci Fantom, časové plány a funkci výpočtu východu a západu slunce. Také by měla být využita technologie DALI, EnOcean a KNX/EIB.

Velikost tabulek bude záviset na zadaných konstantách maximálních počtů připojených karet. Před nahráním aplikace do automatu bude nutností tyto konstanty zadat. Ve své práci jsem se rozhodl použít hodnoty z převzaté aplikace. Maximální počty připojených karet budou zadány dle následující tabulky 2.2.

Počet…	Zadaný počet v aplikaci
vstupních digitálních karet	16
výstupních digitálních karet	10
vstupních analogových karet	6
výstupních analogových karet	6

Tabulka 2.2 – Zadané maximální počty v aplikaci

2.2 Nadřazená aplikace ovladače

Nadřazená aplikace musí samozřejmě zajistit přepsání tabulek uvnitř automatu, které definují vazby mezi komponentami, podle datového modelu DAMIC. Aplikace by ale měla umět i tyto tabulky vyčíst a na základě jejich hodnot sestavit datový model.

Kromě této operace by aplikace měla zajišťovat přenos hodnot a stavů jednotlivých komponent do patřičných atributů v datovém modelu a vzdálenou simulaci nebo ovládání komponent ze systému DAMIC.

Aplikace bude komunikovat s automatem přes ethernetové rozhraní s použitím protokolu MODBUS/TCP. Komunikace bude využívat principu „master-slave“, kde nadřazené aplikaci je přiřazena role „master“.

Nejvýhodnějším řešením aplikace bude její naprogramování jako služby systému Windows.

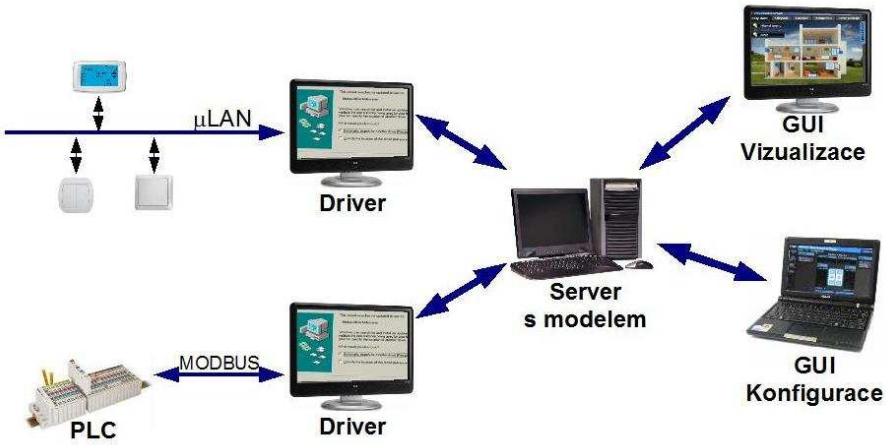
Kapitola 3

Použité technologie

3.1 DAMIC

Jak bylo popsáno výše, cílem projektu DAMIC je vytvořit univerzální systém pro vzdálenou vizualizaci, konfiguraci a ovládání domácí automatice s minimální závislostí na výrobci technologie. Základním pilířem tohoto projektu je vytvoření jednoduchého grafického rozhraní pro koncového uživatele, který by si bez znalosti automatizačního systému, snadno a intuitivně dokázal zobrazit stavy systému (např. světlo rozsvíceno, ventilátor spuštěn, teplota v místnosti,…) a ovládal je. K základnímu uživatelskému ovládání také patří možnost propojení jednotlivých komponent do funkčních vazeb (např. propojení vypínačů se světly).

Aby se tohoto docílilo, byla navržena koncepce systému typu „server a klienti“ dle obrázku 3.1. Server spravuje celkový datový model a poskytuje jej ostatním klientům. Model musí být navržen tak, aby byl dostatečně univerzální vůči použitému fyzickému zařízení. Server sám neumí pracovat s jednotlivými daty, ale poskytuje klientům funkce pro práci s konkrétními daty v modelu. Pokud některý klient změní určitou část modelu, server o tom informuje ostatní klienty. Klientem může být grafické uživatelské rozhraní umožňující zobrazení a konfiguraci systému nebo aplikace typu driveru, která zajišťuje spojení mezi serverem a fyzickými zařízeními.



Obrázek 3.1 – Schéma systému DAMIC

Je samozřejmostí, že klasické místní ovládání domovní automatizace je autonomní a nezávisí na připojení k serveru. Jinými slovy, odpojení od serveru nemá za následek, že by například daný vypínač přestal rozsvěcet dané světlo a podobně.

Každý systém ze začátku potřebuje konfiguraci dle své fyzické technologie, kterou musí provést technik. Ten má za úkol zjistit počet a typy zařízení, identifikovat fyzická zařízení v systému a nastavit jejich počáteční konfiguraci a vazby. Dále by mělo být jeho úkolem nahrání vizualizací celého systému (domu) a kontrola korektní funkčnosti.

3.2 MODBUS

Modbus je otevřený protokol, který slouží pro vzájemnou komunikaci zařízení používaných v oboru automatice, jako jsou například automaty PLC, dotykové displeje, senzory, atd. Protokol definuje strukturu zpráv nezávisle na použitém typu sítě či sběrnice a pracuje na principu „master-slave“. V pozici mastera je řídící zařízení (PC nebo PLC) posílající dotazy a adresované zařízení, slave, na tyto dotazy odpovídá. Dotazy jsou udány číslem funkce, jenž má podřadné zařízení vykonat, a bývají následovány datovou oblastí. Jejím obsahem může být například adresa a počet registrů, které má zařízení přečíst, nebo hodnoty registrů, které má zařízení zapsat. Více informací lze nalézt v [6].

Pro řízení komunikace je nutné znát, které své oblasti a pod jakou adresou používanou v protokolu MODBUS je zařízení zpřístupňuje. Tabulka 3.1 uvádí adresaci oblastí pro řídící jednotky WAGO typu 750-841 a 750-849.

Adresa MODBUS [dek.]	Adresa MODBUS [hex.]	Adresa dle IEC61131	Oblast paměti
0…255	0x0000…0x00FF	%IW0-%IW255 (%QW0-%QW255)	Oblast fyzických vstupů - funkce čtení (Oblast fyzických výstupů -funkce zápisu)
256…511	0x0100…0x01FF	%QW256-%QW511 (%IW256-%IW511)	Oblast pro výstupní PFC proměnné (Oblast pro vstupní PFC proměnné)
512…767	0x0200…0x02FF	%QW0-%QW255 (%QW0-%QW255)	Oblast fyzických výstupů - funkce čtení (Oblast fyzických výstupů -funkce zápisu)
768…1023	0x0300…0x03FF	%IW256-%IW511 (%IW256-%IW511)	Oblast pro vstupní PFC proměnné (Oblast pro vstupní PFC proměnné)
1024…4095	0x0400…0x0FFF	-	Nepovolená oblast
4096…12287	0x1000…0x2FFF	-	Konfigurační registry
12288…24575	0x3000…0x5FFF	%MW0-%MW12287	Adresovatelná oblast paměti
24576…25340	0x6000…0x62FC	%IW512-%IW1275 (%QW512-%QW1275)	Přídavná oblast fyzických vstupů (Přídavná oblast fyzických výstupů)
25341…28671	0x62FD…0x6FFF	-	Nepovolená oblast
28672…29436	0x7000…0x72FC	%QW512-%QW1275 (%QW512-%QW1275)	Přídavná oblast fyzických výstupů (Přídavná oblast fyzických výstupů)
29437…65535	0x72FD…0x7FFF	-	Nepovolená oblast

Tabulka 3.1 – Oblasti paměti adresovatelné protokolem MODBUS

3.3 DALI

DALI (Digital Addressable Lighting Interface) je standardizovaný sběrnicový systém využívaný v domovní automatizaci pro řízení osvětlení. Tento standard, který je stanoven v normě IEC 60929, zahrnuje komunikační protokol a elektrické rozhraní sběrnice. Sběrnice je robustní a odolná proti rušení, ale relativně pomalá. Její rychlost dosahuje pouhých 1200Baudů.

Ke sběrnici je připojen kontrolér a až 64 světel s rozhraním DALI, které obsahují vlastní paměť pro nakonfigurované hodnoty. Kontrolér řídí a spravuje osvětlení pomocí příkazů posílaných po sběrnici. U každého zařízení lze měnit intenzitu jasu a vyvolávat světelné scény. Protokol DALI umožňuje, aby zařízení byla adresována individuálně nebo přiřazením do skupin. Zajištěna je samozřejmě i zpětná vazba od zařízení pro monitoring a detekci poruch.

Tato technologie postupně vytlačuje klasické analogové předřadníky řízené napětím. Více informací o této technologie lze nalézt v [21].

3.4 EnOcean

EnOcean je otevřený standard, který spravuje konsorcium EnOcean Alliance. Jedná se o bezdrátovou, energicky soběstačnou technologii využívanou hlavně v domovní automatizaci. Standard EnOcean je založen na energeticky soběstačných senzorech a spínačích, které vysílají bezdrátové pakety a energii pro toto vysílání získávají efektivní transformací energie z jiného zdroje. K tomuto účelu využívají například piezoelektrické a magnetoelektrické prvky, solární články či termočlánky. Výkon vysílačů vystačí pro dosah 30m v budovách a až 300m na volném prostranství.

Více informací o této technologie lze nalézt v [22] a [23].

3.5 KNX/EIB

KNX je standardizovaný systém pro inteligentní budovy spravovaný konsorciem KNX Association. Tento standard definuje protokol pro síťové propojení zařízení domovní automatizace bez závislosti na výrobci. KNX disponuje širokou škálou možností řízení osvětlení, větrání, topení, klimatizace, měření spotřeby vody, měření elektrické energie, atd. Lze jej aplikovat jak v nových, tak stávajících budovách, ale nevýhodou bývá vysoká pořizovací cena jednotlivých zařízení.

Komunikace na sběrnici KNX probíhá na principu „vysílač-přijímač“. Každé zařízení disponuje od výrobce naprogramovanými komunikačními objekty, které představují rozhraní pro vysílání nebo příjem dat ze sběrnice. Funkční vazby vznikají tak, že se vybraným komunikačním objektům přiřadí stejná skupinová adresa. Musí ovšem platit, že v rámci jedné skupinové adresy existuje jen jeden vysílač. Příjemců může být větší počet. Pro přenos dat mohou být použita média jako kroucená dvoulinka, silové vedení, rádiové vedení, optická vlákna nebo ethernet.

Existují tři kategorie KNX zařízení:

- **Automatic** - zařízení se konfigurují sama a jsou určena pro koncové spotřebitele
- **Easy** - zařízení se konfigurují jednoduchým kontrolérem, mají předprogramované chování a tím omezenou funkčnost
- **System** - zařízení jsou programována pomocí jednotného programátorského prostředí ETS3

Jednotné programátorské prostředí ETS3, které je distribuováno KNX Association, umožňuje úplnou parametrizaci sítě a konfiguraci všech certifikovaných KNX produktů, neboť každý výrobce dodává ke svému zařízení software, který lze do tohoto prostředí importovat.

Více informací o této technologii lze nalézt v [24] a [17].

Kapitola 4

Aplikace pro automat WAGO

4.1 Úvod

Jak jsem uvedl výše, aplikaci pro Wago jsem postavil na základech aplikace, jenž vytvořil Ing. Nývlt v rámci své diplomové práce Automatický návrh řízení pro domovní automatizaci v roce 2008 [1]. Některé algoritmy jsem jen částečně poupravil tak, aby vyhovovali koncepci mé aplikace, a jiné vytvořil zcela od počátku.

Z původní aplikace jsem si přisvojil myšlenku ovládání komponent. Vstupní signály jsou vyhodnocovány původním algoritmem na detekci událostí jednoduchého či dvojitého kliknutí rozšířeným o detekci události dlouhého stisku. Výstupní komponenty (signály) jsou ovládány na základě propojení těchto událostí a akcí. Další události, které mohou ovládat výstupní komponenty, mohou být vyvolané východem či západem slunce, funkcí Fantom, časovým plánem nebo požadavkem uživatele z prostředí DAMIC.

Tato propojení událostí a akcí jsou v paměti automatu uložena pomocí tabulek. Avšak na rozdíl od původní idey, netransformuji při ukládání tyto matice do textových řetězců, ale ukládám je přímo v binární podobě na vnější zálohovanou paměť typu flash. Zvolit tento postup jsem si mohl dovolit hlavně proto, že jsem odstoupil od koncepce konfigurace ovládání systému přes webové vizualizační rozhraní. Ovládání se bude konfigurovat pouze pomocí společného vizualizačního klienta systému DAMIC připojeného ke sdílenému serveru.

Z původní aplikace jsem také převzal algoritmus řízení vytápění. Poupravil jsem v něm ale vztahy mezi jednotlivými komponentami dle zavedené koncepce nadřazeného datového modelu a přidal možnost zadávání požadované teploty z analogového vstupu.

Významnou změnou v aplikaci bylo přidání komunikace přes rozhraní MODBUS/TCP. Přes toto rozhraní přistupuje do paměti automatu nadřazená aplikace (driver), která zapisuje či čte model ovládání v automatu, detekuje změny stavů komponent a simuluje nebo ovládá stavy komponent. Pro všechny tyto úkony musí být v aplikaci v adresovatelné paměti patřičné proměnné.

Dále jsem musel v aplikaci upravit používání sběrnice DALI, aby dobře vyhovovalo koncepci řízení komponent, a rozšířit aplikaci pro použití sběrnice EnOcean. Na závěr jsem se pokusil připojit do aplikace i sběrnici KNX.

4.2 Struktura Aplikace

Automat nám umožňuje programovat aplikace s více paralerními procesy. Toho jsem využil a navrhnul tři základní procesy, ke kterým v průběhu realizace přibyl čtvrtý. Tabulka 4.1 zobrazuje použité procesy s dobou periody jejich volání a prioritou. Čím menší číslo, tím je priorita vyšší.

Název procesu	Perioda volání	Priorita
Hlavni	Volný cyklus	10
SpravaVstupuVystupu	75ms	5
SpravaDALI	500ms	7
AutoDiagnostika	10s	6

Tabulka 4.1 – Paralerní procesy v aplikaci

Hlavní proces běží ve volném cyklu a spouští hlavní program řízení *PLC_PRG*. Tento program zajistí volání odpovídajících podprogramů pro vyhodnocení ovládání všech komponent. Také je zde volán podprogram *MODBUS* pro řízení případného výpisu nebo zápisu modelu do nadřazené aplikace a podprogram *SOUBORY* pro výpis a zápis modelu do vnější paměti typu flash.

Druhý proces s názvem *SpravaVstupuVystupu* běží v cyklickém spouštění s krátkou dobou periody. Je v něm volán program pro správu vstupních signálů a program pro správu výstupních signálů. Program pro správu vstupních signálů musí být vykonáván v rychlém procesu, aby mohly být správně detekovány uživatelovy stisky komponent. A program pro správu výstupních signálů musí být také vykonáván v rychlém procesu, jelikož některé výstupní komponenty se ovládají například rychlým pulsem nebo signálem s PWM modulací. Během realizace jsem do tohoto procesu přidal programy s funkčními bloky přijímačů sběrnic EnOcean a DALI.

Třetí proces s názvem *AutoDiagnostika* také běží v cyklickém spouštění, ovšem s dlouhou dobou periody. V tomto procesu je volán program *KONTROLA_MODULU* pro autodetekci systému. Tento program zjišťuje aktuální hardwarovou konfiguraci automatu. Z principu nemusí být tento program volán často, neboť nepředpokládáme rychlé změny v konfiguraci automatu. Zatěžoval by se tak pouze systém. Rozhodl jsem se pro periodu spouštění 10 sekund.

Čtvrtý proces s názvem *SpravaDALI*, který byl přidán až v průběhu realizace, běží také v cyklickém spouštění a spravuje tok dat na sběrnici DALI. Důvodem pro přidání tohoto procesu byla malá kapacita sběrnice DALI a tím nutná potřeba kontrolovat tok dat na této sběrnici. Iteračně byla doba periody nastavena na 500ms. Více o programu *MONITORING_DALI* volaném v tomto procesu najdete v kapitole 4.12.5.

Programy mohou být v aplikaci volány i systémovými událostmi. Ve své aplikaci jsem využil dvou událostí, které při svém výskytu volají program *POCATECNI_NASTAVENI*. Tyto události nastanou při startu aplikace a po restartu automatu, například po obnově napájení.

4.3 Princip řízení

4.3.1 Vazby komponent

Jak bylo napsáno, propojení událostí a akcí je zajištěno pomocí matic uložených v paměti automatu. Matice může mít podobu tabulky se snadným dekódováním, kde pro danou akci a událost platí, že na výstupní komponentě, jejíž umístění se shoduje s řádkem tabulky, bude vyvolána patřičná akce, pokud se na vstupní komponentě vyskytne patřičná událost, a u které je v patřičném sloupci logická jednička. Pro každou podporovanou dvojici události a akce tak musí být vyčleněna vlastní tabulka.

Šířka tabulek závisí na počtu vstupních signálů. Počítáme-li osm signálů na jednu vstupní kartu, pak se bude šířka matice v bytech rovnat počtu vstupních digitálních karet. Délka tabulky se rovná počtu ovládaných komponent. Tedy pro ovládání digitálních výstupů je délka tabulky rovna právě počtu těchto digitálních výstupů. Vzhledem k rozšíření programu i na použití sběrnice DALI, existují tabulky i pro DALI světla a pro DALI skupiny světel.

Propojení, u kterých platí, že jedna z komponent může být pouze v jediné vazbě, jsou navržena jiným způsobem. Matice se změní ve vektor, kde adresa omezené komponenty odpovídá umístění v poli a hodnota značí adresu druhé komponenty. Tento druh vazby je například použit mezi časovými plány a komponentami digitálního výstupu. Jedna komponenta na digitálním výstupu může být řízena pouze jedním časovým plánem.

Během realizace se objevil drobný problém. Vyhodnocení událostí bylo navrženo pro logiku, kde logická jednička značí aktivitu. PIR obvod na vstupní kartě má ovšem obrácenou logiku, kdy přítomnost osoby je provázena logickou nulou na fyzickém vstupu. To mě přivedlo k myšlence zavedení dvou polí pro vstupní a výstupní komponenty, kdy se po přečtení vstupních signálů nebo před zápisem výstupních signálů, znegují signály,

jenž mají na daném místě v těchto polích logickou jedničku. A tak snadno obsáhneme obě logiky jak na vstupních, tak výstupních signálech.

4.3.2 Konstanty

Při kompilaci potřebuje aplikace znát velikosti alokovaných pamětí pro matice definující propojení komponent. Jak bylo napsáno v předešlé kapitole, velikost těchto matic je závislá na maximálních počtech vstupních a výstupních komponent. Aby nebylo nutné při každé změně maximálního počtu vstupních či výstupních komponent ručně měnit velikosti patřičných tabulek a vektorů, existují v globální deklarační části konstanty nesoucí hodnoty těchto počtů. Tabulka 4.2 zobrazuje tyto konstanty i s popisem jejich významu.

Konstanty počtu karet a plánů mají navíc svůj obraz v adresovatelné části paměti, které musí být shodné s konstantou! Tyto obrazy slouží pro nadřazenou aplikaci driveru.

V aplikaci existuje ještě jeden typ konstant. Ty slouží k označení typu komponenty, která se nachází na dané adrese vstupu či výstupu. Toto označení je důležité pro program řízení, aby v případě vstupní komponenty věděl jak ji vyhodnotit a v případě výstupní komponenty jak ji řídit. Označení komponenty slouží také nadřazené aplikaci driveru, která přistupuje k automatu přes rozhraní MODBUS/TCP. Nadřazená aplikace tak rozpozná typ komponenty a může posléze vygenerovat správný datové typ v nadřazeném datovém modelu DAMIC.

V aplikaci existují čtyři oblasti, které potřebují označení typů komponent. Jsou to samozřejmě oblasti digitálního vstupu, digitálního výstupu, analogového vstupu a analogového výstupu. Pro každou oblast je vytvořen vektor, pro který platí, že hodnota na n-té pozici značí typ komponenty na n-té adrese. Konstanta je typu INT (2 byty), takže každá oblast může využít označení až 65535 typů komponent, což je více než dostatečné množství. Hodnota nuly je rezervována pro nepřítomnou komponentu. Tabulka 4.3 zobrazuje použité označení komponent v mé aplikaci.

Název konstanty	Hodnota použitá v aplikaci	Význam hodnoty
POCET_VSTUPU	128	Konstanta určující maximální počet digitálních vstupů. Musí být rovna POCET_KARET_VSTUPU*8!
POCET_VYSTUPU	80	Konstanta určující maximální počet digitálních výstupů. Musí být rovna POCET_KARET_VYSTUPU*8!
POCET_VSTUPU_ANALOG	24	Konstanta určující maximální počet analogových vstupů. Musí být rovna POCET_KARET_VSTUPU_ANALOG*4!
POCET_VYSTUPU_ANALOG	24	Konstanta určující maximální počet analogových výstupů. Musí být rovna POCET_KARET_VYSTUPU_ANALOG*4!
POCET_KARET_VSTUPU	16	Konstanta určující maximální počet digitálních vstupních karet.
POCET_KARET_VYSTUPU	10	Konstanta určující maximální počet digitálních výstupních karet.
POCET_KARET_VSTUPU_ANALOG	6	Konstanta určující maximální počet analogových vstupních karet.
POCET_BYTU_VSTUPU_ANALOG	3	Konstanta určující počet bytů při přiřazení jednoho bitu ke každému analogovému vstupu.
POCET_KARET_VYSTUPU_ANALOG	6	Konstanta určující maximální počet analogových výstupních karet.
POCET_BYTU_VYSTUPU_ANALOG	3	Konstanta určující počet bytů při přiřazení jednoho bitu ke každému analogovému výstupu.
POCET_PLANU	24	Konstanta určující počet použitých topných i časových plánů v aplikaci.
POCET_BYTU_PLANU	3	Konstanta určující počet bytů při přiřazení jednoho bitu ke každému plánu.

Tabulka 4.2 – Tabulka konstant maximálních počtů

Digitální vstupy		
Označení	**Hodnota**	**Typ komponenty**
TLA_typ	2	Tlačítko, které má dva stavy: stisknuto/nestisknuto.
VYP_typ	3	Vypínač, který má dva stavy: sepnuto/rozepnuto.
PIR_typ	4	PIR obvod, který má dva stavy: sepnuto/rozepnuto.
BLO_typ	5	Okenní blokovací kontakt, který má dva stavy: sepnuto/rozepnuto.
Digitální výstupy		
Označení	**Hodnota**	**Typ komponenty**
OSV_typ	2	Světlo, které je ovládáno: 1-svítí / 0-nesvítí.
VNT_typ	3	Ventilátor, který je ovládán: 1-běží / 0-stojí.
TEP_typ	4	Ventil topení, který je ovládán pomocí PWM.
TEP_KOTEL_typ	5	Kotel, který je ovládán: 1-topí / 0-netopí.
OSI_typ	6	Impulsní světlo s proměnným jasem, které je ovládáno pomocí impulsů.
ZALM_typ	10	Žaluzie pohybové, které jsou ovládány: 1-dolů / 0-nahoru.
ZALS_typ	11	Žaluzie krokové, které jsou ovládány: 1-krok dolů / 0-krok nahoru.
Analogové vstupy		
Označení	**Hodnota**	**Typ komponenty**
TER_typ	2	Teplotní čidlo udávající teplotu ve formátu INT, kde platí 234=23.4°C
TDI_typ	3	Analogový vstup pro zadání požadované teploty, formát INT, kde platí 234=23.4°C

Tabulka 4.3 – Tabulka konstant pro označení typů komponent

4.4 Uložení modelu v aplikaci

4.4.1 Struktura adresovatelné a remanentní paměti

V automatech WAGO máme k dispozici až 256kB paměti pro data. Pro účely aplikace mě nejvíce zajímala velikost paměti, která se dá adresovat a ke které se dá přistupovat pomocí rozhraní MODBUS/TCP. Dále byla důležitá velikost remanentní paměti. V manuálech k použitým jednotkám je uvedeno, že remanentní a adresovatelná paměť se nacházejí ve shodné oblasti, která má maximální velikost 24kB. V manuálech je tato oblast označena jako „Non-volatile memory“ (viz obrázek 4.1).

Technical Data	
Number of I/O modules	64
with bus extension	250
Fieldbus	
Max. input process image	2 Kbytes
Max. output process image	2 Kbytes
Input variables (max.)	512 bytes
Output variables (max.)	512 bytes
Configuration	via PC
Program memory	512 Kbytes
Data memory	256 Kbytes
Non-volatile memory (retain)	24 Kbytes (16 Kbytes retain, 8 Kbytes flag)
Voltage supply	DC 24 V (-25 % ... +30 %)
Max. input current (24 V)	500 mA
Efficiency of the power supply	87 %
Internal current consumption (5 V)	300 mA
Total current for I/O modules (5 V)	1700 mA
Isolation	500 V system/supply
Voltage via power jumper contacts	DC 24 V (-25 % ... +30 %)
Current via power jumper contacts (max.)	DC 10 A

Obrázek 4.1 – Velikost adresovatelné a remanentní paměti

První část paměti lze adresovat, ale její obsah není zálohován. Druhá část paměti je zálohovaná, ovšem nelze ji adresovat. Továrně jsou velikosti oblastí nastaveny na 16kB zálohované paměti a 8kB adresovatelné paměti. Hranici mezi těmito oblastmi však můžeme libovolně posouvat v nastavení programu CoDeSyS, ale platí, že součet velikostí těchto oblastí nemůže být větší než celková velikost oblasti. Podrobný postup nastavení hranice a velikostí oblastí je uveden v příloze B.

4.4.2 Rozmístění modelu v paměti automatu

Pro jednoduché adresování každé tabulky samostatně je rezervovaná oblast paměti příliš malá. Proto jsem musel vymyslet způsob přenášení potřebných dat přes tuto omezenou velikost paměti. Využil jsem skutečnosti, že tabulky, které se váží ke stejným typům komponent, mají stejnou velikost a strukturu. Liší se pouze v interpretaci. A tak jsem umístil tabulky do vnitřní pracovní paměti automatu a do adresovatelné paměti inicializoval pouze jednu sdílenou tabulku stejné struktury. K této sdílené matici jsem přidal bytovou proměnnou, jejíž hodnota určuje, jaký druh tabulky je právě ve sdílené paměti uložen. Takovéto dvojice matic jsem vytvořil nakonec čtyři – pro komponenty na digitálních výstupech, pro DALI komponenty, pro vazby mezi komponentami a časovými plány a pro přenos topných a časových plánů.

Dále jsou v adresovatelné paměti umístěny všechny ostatní matice, ke kterým potřebuje přístup jak aplikace automatu, tak nadřazená aplikace driveru.

Výhodou tohoto postupu je značné ušetření adresovatelné paměti (4 použité sdílené tabulky pro 66 tabulek). Nevýhodou této koncepce je vytvoření programu, který bude mít na starosti řízení toku dat přes tyto sdílené matice. Tento program se jmenuje *MODBUS* a je popsán v kapitole 4.10.1.

Tabulky modelu by se po ztrátě napájení neměly zapomenout. Je třeba zajistit, aby po obnovení napájení automat přešel do stavu, ve kterém byl před výpadkem. Velikost remanentní paměti nám neumožňuje mít všechny tabulky zálohované v této části paměti. Využil jsem skutečnosti, že automaty WAGO mají k dispozici vnější zálohovatelnou paměť umístěnou na kartě flash. Zápis dat na tuto kartu se provede pouze při požadavku načtení modelu z nadřazeného systému do automatu, aby se zde uložil nový model. Také čtení dat z této vnější paměti nastane pouze v jednom případě, a to po obnově napájení. Práce se soubory jsou časově náročné operace, ale v těchto dvou definovaných okamžicích si mohu prodloužení hlavního cyklu automatu dovolit. Proto jsem si dovolil tuto vnější paměť využít.

Remanentní paměť použiji pouze pro proměnné, které ukládají aktuální stav komponent, a tak se mohou během řízení měnit.

4.4.3 Adresace proměnných

Aby mohl nadřazený systém správně komunikovat s automatem přes rozhraní MODBUS/TCP, bylo třeba zajistit, aby oba programy měly pro proměnné nastaveny stejné adresy. Mou snahou bylo vytvořit strukturu adresovatelné paměti tak, aby nadřazený systém mohl komunikovat s automatem, aniž by musel znát rozsahy vstupů a výstupů před svou kompilací.

Příloha C obsahuje všechny matice a proměnné využívané pro komunikaci přes rozhraní MODBUS/TCP. Většina z těchto matic mění svou velikost v závislosti na nastavených konstantách maximálních počtů vstupů, výstupů a plánů. Ale existují i proměnné, které mají svou velikost neměnnou, například informace o chybách automatu, řídící povely, konstanty, atd. Tyto proměnné jsou společné pro všechny modely bez závislosti na zvolených maximálních počtech vstupů, výstupů a plánů.

Rozdělil jsem tedy adresovatelnou paměť na dvě části. V první jsou pevně nastaveny adresy pro proměnné, u nichž se velikost nemění. Tuto oblast jsem rezervoval na velikost 1000 bytů, která je myslím dostatečně veliká i pro další možné rozšíření aplikace. Od adresy MB1000 začnou adresy matic, jejichž velikost závisí na zvoleném maximálním počtu vstupů, výstupů a plánů. Posloupnost matic je pevně dána a je známa od počátku jak v aplikaci automatu, tak v nadřazené aplikaci driveru. Nadřazená aplikace tak bude moci z předem definovaných adres přečíst zvolené konstanty pro maximální počty vstupů, výstupů a plánů a

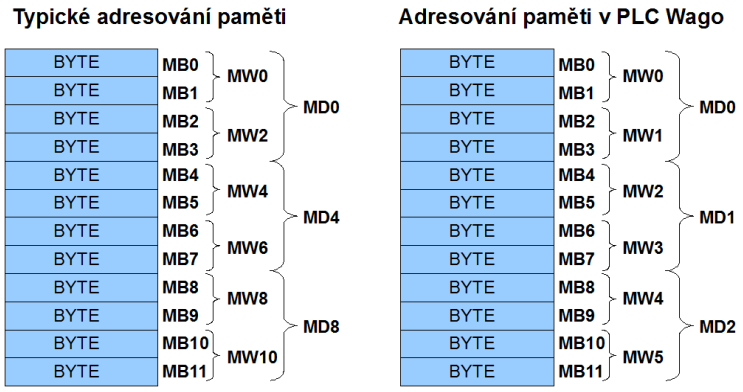
od adresy 1000 si adresy spočítat. Tato koncepce má pro nadřazenou aplikaci výhodu i při případném přidávání matic do posloupnosti. Pouze se změní funkce výpočtu adres a ostatní algoritmy budou pracovat s nově vypočtenými adresami. Pro aplikaci automatu bohužel zůstává nutnost ručně nastavit jednotlivé adresy proměnných.

Příloha C definuje umístění proměnných v pevně definované části paměti a také následnou posloupnost matic s proměnnou velikostí. V tištěné verzi tohoto dokumentu jsou vypsány vypočtené adresy pro mnou odevzdanou aplikaci, ale zdrojový soubor na přiloženém CD (příloha H) obsahuje vzorce pro výpočet adres jak pro automat, tak pro nadřazenou aplikaci ovladače. Lze jej tedy využít při změnách konstant počtů vstupů, výstupů a plánů, nebo kontrolu funkce výpočtu adres v nadřazeném systému.

4.4.4 Zvláštnosti při užití paměti v automatech WAGO

Během realizace programu jsem při adresaci proměnných narazil na dvě zajímavé vlastnosti automatu WAGO.

Na rozdíl od jiných běžných automatů PLC, WAGO čísluje každý datový typ po sobě jdoucími čísly. Jasnější popis poskytuje obrázek 4.2.



Obrázek 4.2 – Způsob adresování paměti

Číslování bytových proměnných je stejné. Pro typ word (2byty) začíná ve většině PLC adresace MW0 následované MW2,MW4,MW6, atd. Tedy číslo wordu se shoduje s adresou spodního bytu. PLC WAGO ale čísluje wordy a byty zvlášť. Tedy typ word začíná MW0, následuje MW1,MW2,MW3, atd. To samé pravidlo je použito i pro ostatní typy proměnných. Při volbě adresy proměnné musíme tedy dávat dobrý pozor, aby se oblast proměnné nepřekrývala s druhou proměnnou jiného typu.

Druhou zvláštností automatů WAGO je jejich ukládání pole s booleovskými proměnnými. Čekal bych, že automat bude toto pole ukládat do paměti po jednotlivých bitech. Ovšem automaty WAGO uloží takovéto pole tak, že každá bitová proměnná zabere celý byte paměti. Tedy například pole s 16 booleovskými hodnotami nezabírá 2 byty, jak jsem původně očekával, ale je uloženo do paměťové oblasti o velikosti 16 bytů!

Vzhledem ke skutečnosti, že většina matic v aplikaci má význam booleovské tabulky, a vzhledem k velikosti paměti automatu, nemohl jsem si dovolit ukládat všechny tabulky jako pole typu bool. Jejich velikost by byla osmkrát větší. Proto jsem většinu tabulek navrhnul jako pole typu byte a pro přístup k jednotlivým bitům použil tečkovou notaci. Některá zbylá pole typu bool mají svůj obraz v adresovatelné části paměti, ovšem v bytové velikosti z důvodu šetření této části paměti.

4.5 Automatická detekce

4.5.1 Možnosti automatické detekce

Konfigurační prostředí CoDeSyS umožňuje programátorovi nahrát do automatu aplikaci s nevyplněnou hardwarovou konfigurací, čímž se ovšem připraví o přímou kontrolu připojených karet. Jediné, co musí programátor zadat, je typ řídící jednotky. Pokud nastavená řídící jednotka neodpovídá typu jednotky připojené, CoDeSyS ukončí spojení. Možnost nevyplnit hardwarovou konfiguraci přispívá k obecnosti aplikace.

Pro mou aplikaci je ale důležité hardwarovou konfiguraci znát. Pro správnou funkčnost je třeba zkontrolovat, zda nejsou v sestavě neznámé nebo nekompatibilní karty, zda je korektní posloupnost karet, zda se v sestavě nachází speciální karty typu DALI nebo EnOcean či zda počet karet nepřekračuje nastavenou mez. Tím, že si aplikace umí tyto podmínky sama zkontrolovat, se zvyšuje komfort použití.

V paměti automatu existuje oblast, kam o sobě připojené karty ukládají informace. Tato oblast je přesně vymezena, ale není programově přístupná. Ani knihovny pro automaty WAGO nenabízejí funkční blok, který by tyto informace o hardwarové konfiguraci programu poskytnul. V praxi se ale běžně tento problém řeší pomocí malého triku.

Použitím některého funkčního bloku pro řízení komunikace přes rozhraní MODBUS/TCP (v mém případě ETHERNET_MODBUSMASTER_UDP), umožníme automatu přístup do paměti podřízeného zařízení. Nastavíme-li IP adresu na 127.0.0.1 („localhost“) a adresu registru na 8240, požádá řídící jednotka samu sebe o data přímo z výše uvedené vymezené oblasti. Pro správné využití je třeba ještě nastavit na hodnotu 64 parametr *wREAD_QUANTITY*, který uvádí požadovaný počet přečtených registrů, na hodnotu 3 parametr *bFUNCTION_CODE* a parametr *ptSEND_DATA* sloužící jako pointer do oblasti, kam se mají přečtená data uložit.

Přečtená data jsou typu word a tvoří spolu pole o délce 64, kde jednotlivá čísla znamenají typ připojené karty, jejíž umístění v sestavě je rovno pozici buňky v poli. Například karta DALI má hodnotu 641. První pozice obsahuje typ řídící jednotky. Jen pro karty digitálních vstupů a výstupů platí výjimka a jejich kód je definován dle tabulky 4.4.

Pozice bitu	Význam
0	Karta digitálních vstupů
1	Karta digitálních výstupů
2-7	Nepoužito
8-14	Počet bitů na kartě
15	Indikace digitální karty

Tabulka 4.4 – Kód identifikující digitální karty

4.5.2 Program KONTROLA_MODULU

Aplikace vykonává vyhodnocení automatické kontroly modulů v programu *KONTROLA_MODULU*, který je volán v cyklickém procesu *AutoDiagnostika*. Protože nepředpokládám časté změny konfigurace, nastavil jsem periodu volání na 10 sekund.

Program vyhodnotí reálné počty jednotlivých karet a celkový počet vstupních a výstupních signálů. Program detekuje chybu, pokud reálné počty převyšují maximální počty zadané v aplikaci konstantami nebo pokud se speciální karty stejného typu vyskytují více než jednou. Chybou je také, pokud se na pozici za kartou DALI objeví vstupní či výstupní analogová karta, neboť karta DALI využívá paměťový prostor určený analogovým kartám a tím by došlo k nesrovnalostem u adres analogových komponent.

Na základě výsledků automatické kontroly hardwarové konfigurace má aplikace možnost blokovat vykonávání některých programů. Při nepřítomnosti speciální karty DALI jsou blokovány programy pro řízení a správu DALI komponent. To samé platí i pro speciální kartu EnOcean, případně KNX. Při výskytu chyby v konfiguraci automatu jsou blokovány všechny řídící programy a chyba je specifikována pomocí chybových příznaků.

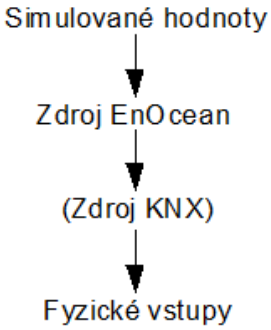
4.6 Zpracování vstupů

4.6.1 Program *ZPRACOVANI_VSTUPU*

V aplikaci jsou zpracovávány digitální i analogové vstupy z několika zdrojů. Mezi tyto zdroje samozřejmě patří fyzické vstupy na kartách automatu. Dále jsou to simulované vstupy z nadřazené aplikace driveru. Při zavádění technologie EnOcean do aplikace, popsané v kapitole 4.13, byl do aplikace přidán další zdroj. Posledním vstupním zdrojem, který byl přidán jen pro aplikaci určenou pro velký model, je sběrnice KNX.

Program *ZPRACOVANI_VSTUPU* spojí tyto zdroje do společného pole, pro digitální signály do pole pojmenovaného *DI_vstupy* a pro analogové hodnoty do pole pojmenovaného *AI_vstupy*. Tím je dosaženo zpracování vstupů bez závislosti na zdroji. Při převodu signálů do pole *DI_vstupy* je provedena i případná negace logických hodnot. Pole *DI_vstupy* slouží dále jako vstupní parametr pro instance funkčních bloků *STISK*, které detekují události na vstupních signálech.

Převod hodnot vstupů ze zdrojů do společných polí se řídí pravidlem poslední změny a priority. Zdroje se vyhodnocují v pořadí uvedeném na obrázku 4.3 a pokud na zdroji došlo ke změně hodnoty, je tento nový stav promítnut do společného pole. Pokud se vyskytne změna hodnoty vstupu na více zdrojích ve stejném okamžiku, má dominantní postavení později vyhodnocený zdroj.



Obrázek 4.3 – Pořadí vyhodnocení zdrojů vstupů

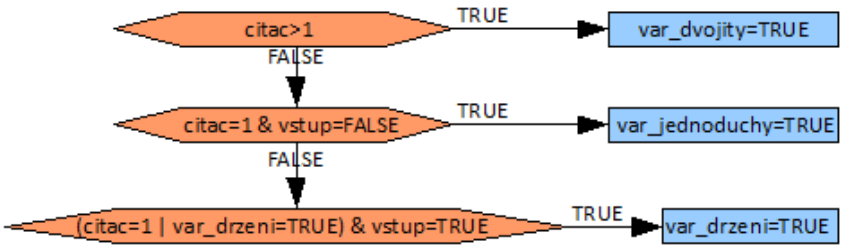
Zvláštní postavení má zdroj simulovaných hodnot. Změny hodnot z ostatních zdrojů se musí promítnout i do patřičného pole určeného pro požadavek simulované hodnoty z nadřazené aplikace. Důvodem je, aby nadřazená aplikace vždy mohla změnit aktuální hodnotu vstupů bez ohledu na její minulou simulovanou hodnotu.

4.6.2 Detekce událostí na digitálních signálech

Na rozdíl od původní aplikace, algoritmus detekce impulsů neběží ve stejném procesu jako algoritmy řízení. Proto je důležité zajistit synchronizaci při předávání dat o vzniklých událostech. Aby algoritmus správně detekoval impulsy na vstupních signálech, je nutné, aby jeho proces běžel v rychlých a častých intervalech. Pokusy bylo zjištěno, že tento interval musí být nejméně 75ms. Proces s algoritmy řízení, který běží ve volné smyčce a je značně náročný na výpočetní čas, je několikrát během svého výpočtu přerušen tímto rychlým procesem.

Obrázek 4.4 – Funkční blok *STISK*

Program *ZPRACOVANI_VSTUPU* obsahuje pro každý vstupní signál funkční blok *STISK*, ve kterém se vyhodnocují události u daného vstupu. Výpočetní algoritmus je jednoduchý. Náběžná hrana na parametru *vstup* spustí časovač, který odměří dobu definovanou v proměnné *Doba_Double* a zároveň je tato náběžná hrana přivedena na vstup čítače. Ten počítá náběžné hrany, které se v době spuštěného časovače objeví na parametru *vstup*. Po uplynutí definované doby se vyhodnotí druh vyskytnuté události dle pravidel na obrázku 4.5. Událost *„drzeni“* je aktivní až do okamžiku spádové hrany na parametru *vstup*.



Obrázek 4.5 – Algoritmus detekce událostí na vstupních signálech

Vyhodnocené události jsou uchovávány v proměnných s předponou „var_“. Výstupní parametry bloku slouží k předávání vzniklých událostí hlavnímu procesu.

4.6.3 Synchronizace při předávání vyhodnocených událostí

Hlavní proces musí na začátku svého cyklu překopírovat vyhodnocené události a uložit si je do svého pracovního pole. Tato data jsou ukládána do pole datového typu *Tlac_Stisk* s délkou dle počtu vstupních signálů. Tento typ obsahuje tři booleovské proměnné nesoucí informaci o vzniku události jednoduchého stisku, dvojitého stisku nebo držení na daném vstupu. S tímto polem pracuje hlavní proces při vyhodnocování řízení výstupních komponent.

Aby nedocházelo ke ztrátám informací o vzniklých událostech nebo naopak opakování řízení při jedné události, je třeba zajistit synchronizaci předávání těchto dat. Je třeba zajistit, aby se po překopírování dat smazaly předané informace v instancích funkčního bloku *STISK*. Dále je třeba zajistit, aby v době předávání dat nedošlo k vyhodnocení události, neboť by mohlo dojít ke zkopírování staré informace a následnému smazání informace nové bez patřičného předání. Jinými slovy, výstupní proměnné funkčního bloku musí být během kopírování zamčené, aby se zkopírovala data z jednoho časového okamžiku. Dále musíme zajistit, že události vyhodnocené v okamžiku kopírování se objeví v dalším cyklu a nebudou smazány při následném potvrzení o přenesení informací.

Synchronizaci zajišťují dvě globální proměnné *Zamek_DI* a *Zkopirovano_DI*. Pokud v hlavním procesu dochází ke kopírování hodnot, je nastavena proměnná *Zamek_DI*. Před jejím resetování je nastavena druhá proměnná *Zkopirovano_DI*, která označuje konec procesu kopírovaní a požadavek na resetování vyhodnocených událostí. Funkční bloky pro detekci impulsů, volané v rychlém procesu, spravují své výstupy pouze, pokud je proměnná *Zámek_DI* v logické nule. Stále však probíhá detekce událostí. Při nastavené proměnné *Zkopirovano_DI*, jsou resetovány zjištěné události. Proměnné s prefixem „*mem_*“ slouží pro případ, kdy je událost vyhodnocena v momentě zamčení výstupních proměnných a kdy by následně došlo ke smazání detekované události bez jejího předání řídícímu programu.

4.7 Zpracování výstupů

O správu fyzických výstupů se stará program *ZPRACOVANI_VYSTUPU*. Protože program obsahuje algoritmus pro generování PWM signálu pro topné ventily a algoritmus pro generování krátkých pulsů, musí být tento program volán z rychlého procesu. Program se dále stará o případnou negaci logických hodnot výstupů a jejich zápis na fyzické výstupy umístěné na kartách automatu. Na závěr program převede pole s funkčními stavy komponent digitálních výstupů do adresovatelné části paměti a do zálohované paměti.

4.8 Algoritmy virtuálních komponent

4.8.1 Topné a časové plány

V původní aplikaci byly topné a časové plány ošetřovány pomocí instancí funkčního bloku *FbScheduleWeekly* knihovny Scheduler_02.lib. Tento funkční blok umožňuje na základě vstupních parametrů sepnout kontakt v nastaveném intervalu konkrétního dne v týdnu. Pro každý senzor bylo inicializováno 10 instancí tohoto bloku.

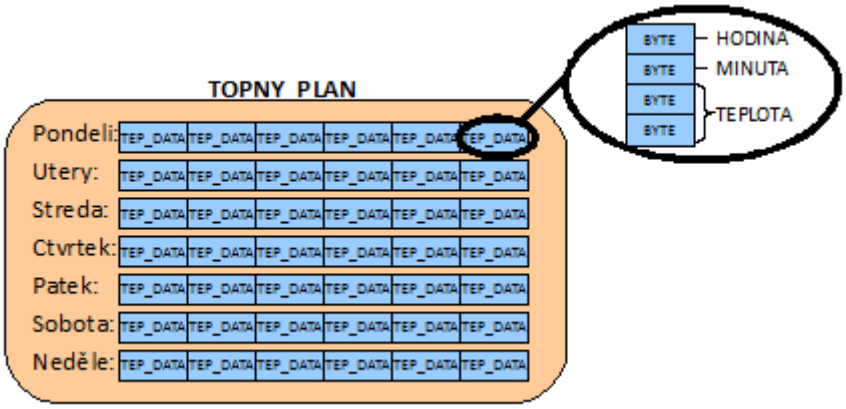
Tento koncept nevyhovoval mým požadavkům na topné a časové plány. Nevýhodou byla velká paměťová náročnost, ukládání teplot do jiného pole a nepřesná datová struktura pro zápis přes rozhraní MODBUS/TCP.

Topný plán

Na počátku mé práce již existovala v nadřazeném datovém modelu DAMIC struktura topného plánu včetně vizualizace. Ponechal jsem tento způsob uložení topného plánu v nadřazeném datovém modelu a na jeho základě vytvořil reprezentaci topného plánu v aplikaci automatu.

V nadřazeném datovém modelu je topný plán uložen jako řetězec číslic oddělenými čárkou, kde se střídají číslice reprezentující jednou čas a podruhé teplotu. Čas byl vyjádřen počtem čtvrthodin a teplota počtem půl stupňů. Na jeden den bylo vyčleněno šest těchto dvojic, na celý týden, respektive plán, vystačilo 42 dvojic. Žádaná hodnota teploty platila do času, který měla ve své dvojici. Po této době platila následující hodnota. I pokud nebyl vyčerpán počet změn v daném dni, stále muselo být k tomuto dni vyhrazeno šest dvojic čísel.

Pro strukturu topného plánu v aplikaci automatu platí stejná pravidla. Vytvořil jsem datový typ pojmenovaný *Topny_plan*, který obsahuje sedm polí pojmenovaných podle dní v týdnu. Prvním dnem je počítáno pondělí. Každé pole obsahuje šest proměnných datového typu *Tep_data*. A datový typ *Tep_data* odpovídá právě jedné dvojici číslic v řetězci v nadřazeném datovém modelu. Typ *Tep_data* obsahuje tři proměnné. První a druhá jsou typu byte, třetí typu word. V první proměnné je uložena hodina změny, ve druhé minuta a ve třetí žádaná teplota s přesností na jedno desetinné místo vynásobená deseti, abych získal celočíselnou hodnotu (např. 24,6°C -> 246). Tedy datový typ *Tep_data* zabírá 4 byty v paměti a jeden celý topný plán potřebuje 42 krát 4 byty, tedy 168 bytů paměti (viz obrázek 4.6).

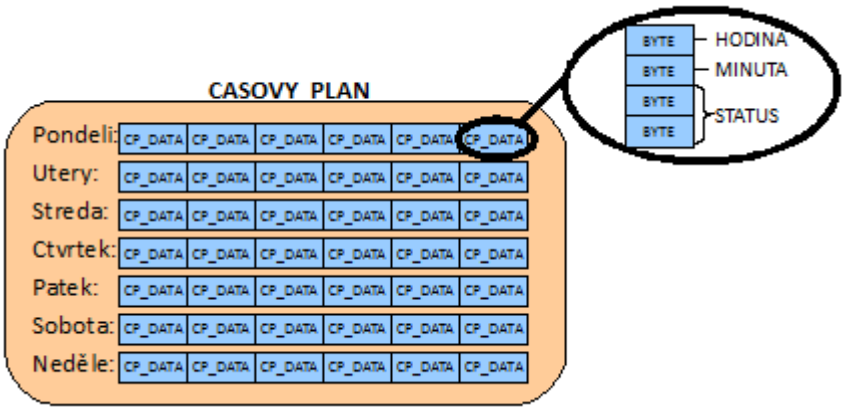


Obrázek 4.6 – Datová struktura topného plánu

Časový plán

Oproti topnému plánu, pro časový plán neexistovala patřičná struktura v nadřazeném datovém modelu DAMIC. Z důvodu podobnosti jsem se ale rozhodl pro téměř identickou reprezentaci. V nadřazeném datovém modelu bude časový plán také reprezentován řadou číslic oddělených čárkou, kde se smysl číslic střídá. Lichá číslice bude opět znamenat čas ve čtvrthodinách a sudé číslo bude mít hodnotu pouze nula pro vypnuto a jedna pro zapnuto. Součet dvojic na jeden plán bude také 42.

I struktura časového plánu v aplikaci automatu je velice podobná struktuře topného plánu. Vytvořil jsem datový typ pojmenovaný *Casovy_plan*, který obsahuje sedm polí pojmenovaných dle dní v týdnu. Prvním dnem je opět počítáno pondělí. Každé pole obsahuje šest proměnných datového typu *CP_data*. A datový typ *CP_data* obsahuje tři proměnné. První a druhá jsou typu byte, třetí je typu word. V první proměnné je uložena hodina změny, ve druhé minuta a ve třetí žádaný stav (zapnutí či vypnutí) vyjádřený pomocí nenulové či nulové hodnoty. Na první pohled se může zdát, že datový typ word není šťastně zvolen pro uložení dvouhodnotové proměnné. Důvodem ale je, že datový typ *CP_data* tak v paměti zabírá 4 byty stejně jako datový typ *Tep_Data* a časový plán má tak shodnou velikost i strukturu jako topný plán (viz obrázek 4.7).



Obrázek 4.7 – Datová struktura časového plánu

Ve své aplikaci jsem nastavil maximální počet plánů na 24, což jen pro topné plány znamená vymezit 4032 bytů potřebné paměti. To je značná část adresovatelné paměti automatu. Pokud bychom použili stejnou velikost paměti i pro časové plány, spotřebovali bychom více než třetinu adresovatelné paměti jen pro funkčnost plánů. A tak se zde projeví výhoda stejné datové struktury obou plánů. Pomocí sdílené oblasti v adresovatelné části paměti, můžeme číst i zapisovat oba typy plánů a nespotřebovat přitom značnou část paměti.

4.8.2 Komponenta Fantom

Funkce Fantom má za úkol simulovat přítomnost obyvatel domu v době jejich nepřítomnosti. Důvodem je snížení pravděpodobnosti vloupání tím, že objekt vypadá z venkovního prostředí obydleně.

V původním programu byla vytvořena pro každou digitální výstupní kartu jedna instance funkčního bloku, která náhodně spínala či vypínala výstupy dané tabulkou. Zachoval jsem způsob generování náhodných signálů i s podobou jednoho funkčního bloku pro jednu výstupní kartu, ale místo přímého ovládání digitálních výstupů jsem zvolil koncepci generování událostí. V hlavním programu řízení funkce Fantom ovládá výstupní komponenty právě pomocí těchto událostí. Tedy při zapnuté funkci Fantom systém vygeneruje náhodné signály u zadaných digitálních výstupů určených patřičnou tabulkou. Při řízení konkrétní výstupní komponenty se při události na vstupu *fantom* vyhodnotí odpovídající akce – zapnutí či vypnutí světla. Tato koncepce je výhodná svou obecností, kdy nezáleží, jakým způsobem se komponenta ovládá. Také mi umožnila rozšířit působnost funkce Fantom na ostatní typy komponent, zejména na komponenty DALI.

Funkci Fantom lze také ovládat patřičnými událostmi podobně jako například světlo. Které události budou funkci Fantoma řídit, záleží opět na patřičných tabulkách. Tedy funkci Fantom lze spínat pomocí tlačítek a vypínačů, časových programů nebo požadavkem z nadřazené aplikace.

4.8.3 Komponenta Východu a západu slunce

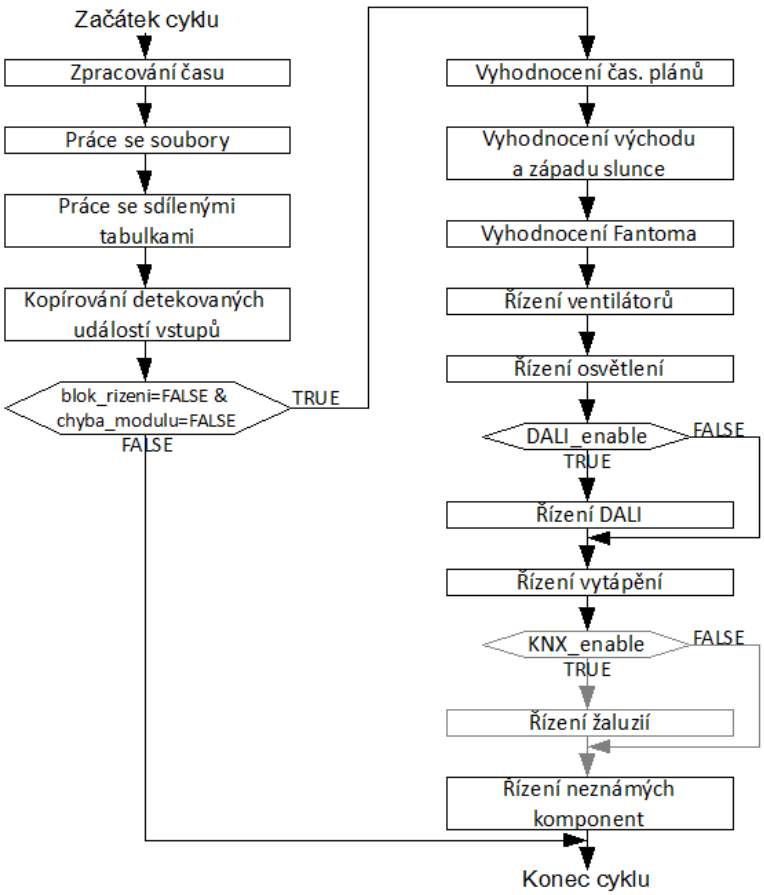
Tento algoritmus byl implementován v původní aplikaci a mě velice zaujal. Rozhodl jsem se udělat z této funkce imaginární komponentu přenesenou i do nadřazeného datového modelu DAMIC. Vlastní výpočetní algoritmus jsem kompletně zachoval. Přidal jsem jen generování událostí od této komponenty. Generovány jsou dvě události – východ a západ slunce. Vnitřní proměnná *Aktualni_Stav* programu *RIZENI_VYCHODU_ZAPADU* indikuje,

zda je den či noc. Logická jednička znamená den. Při změnách této proměnné jsou generovány ony patřičné události. Je třeba dodat, že pro správnou funkčnost této komponenty jsou potřebné nejen dobře nastavené parametry jako zeměpisná šířka, délka a časové pásmo, ale i správně nastavené hodiny automatu.

4.9 Algoritmy řízení

4.9.1 Hlavní program *PLC_PRG*

Hlavní program *PLC_PRG* obsahuje zejména programy pro ovládání komponent. Mimo nich také spravuje programy pro správu času, práci se soubory v paměti flash, práci se sdílenými tabulkami v adresovatelné oblasti paměti a vyhodnocení virtuálních komponent. Obrázek 4.8 znázorňuje pořadí vykonávaných akcí.

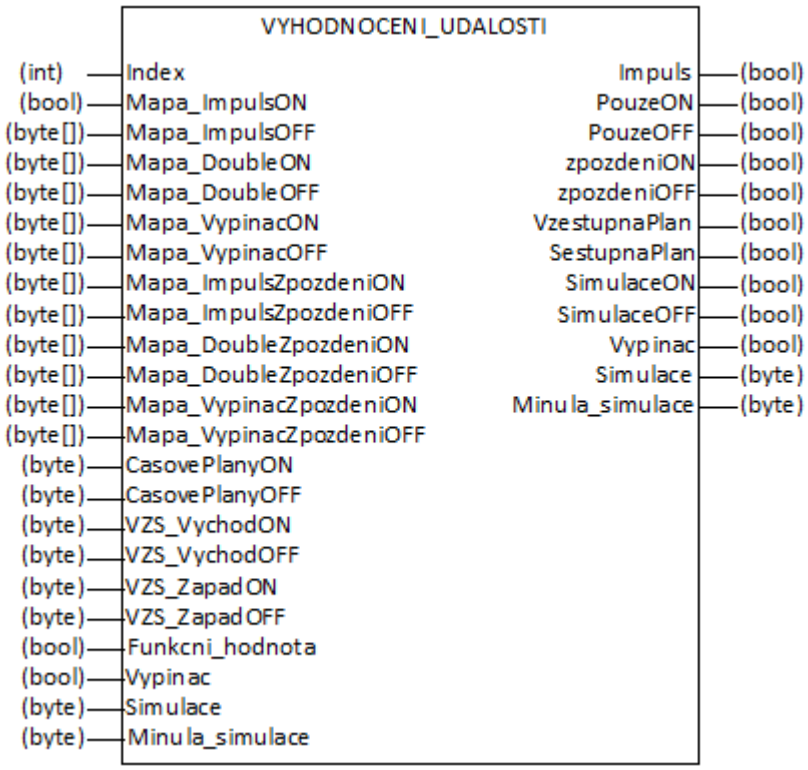


Obrázek 4.8 – Pořadí vykonávaných akcí v programu *PLC_PRG*

V případě detekování chyby na modulu nebo nastaveného blokování řízení se programy pro vyhodnocení virtuálních komponent a ovládání ostatních komponent nevykonávají.

4.9.2 Funkční blok *VYHODNOCENI_UDALOSTI*

Většina výstupních komponent, komponent DALI a funkce Fantom jsou ovládány pomocí shodných událostí a vyhodnocovány dle podobných tabulek. Proto bylo výhodné vytvořit funkci, kde by byly patřičné tabulky zadané jako vstupní parametry a výstupem by byly vyhodnocené signály pro ovládání komponenty. Bohužel podobnou funkci nebylo možné vytvořit, protože podle normy může mít funkce jen jednu výstupní proměnnou. Vytvořil jsem tedy instanci funkčního bloku bez vnitřních proměnných, která je několikrát během cyklu volána. Obrázek 4.9 znázorňuje tento funkčního bloku.



Obrázek 4.9 – Funkční blok *VYHODNOCENI_UDALOSTI*

Z důvodu obecnosti jsou vstupní parametry pro tabulky propojující události na digitálních vstupech s akcemi komponenty pouze jednořádkové a mají pevnou délku dle zadaného maximálního počtu vstupů. Dále vstupní parametry určené pro tabulky propojující časové plány s komponentami mají velikost jednoho bytu k uložení čísla časového plánu, který má komponentu ovládat. Výše zmíněné vytváří potřebu vstupní proměnné před voláním funkčního bloku patřičně vytvořit nakopírováním patřičných částí originálních tabulek.

Další skupinou parametrů jsou vstupně-výstupní proměnné. Proměnná *Vypínač* slouží pro pamatování posledního stavu vypínačů dané komponenty, aby se mohly vyhodnotit změny. Zbylé dvě proměnné slouží pro simulaci stavu komponenty.

Výstupem funkčního bloku jsou vyhodnocené signály pro ovládání dané komponenty. Patří sem signál impulsu, signál pouhého zapnutí či vypnutí, signál zpožděného zapnutí či vypnutí, signál zapnutí či vypnutí od časového plánu, či signál zapnutí či vypnutí od uživatele ze systému DAMIC.

Vlastní algoritmus

Signál impulsu je detekován velice jednoduchým pravidlem. Impuls nastane, pokud se na daném vstupu objeví událost jednoduchého nebo dvojitého stisknutí a zároveň je na daném místě v patřičné dvojici tabulek logická jednička. Patřičnou dvojicí tabulek jsou myšleny dvě tabulky pro stejnou událost generující zapnutí a vypnutí (např. *Mapa_ImpulsON* a *Mapa_ImpulsOFF*). Pokud je logická jednička pouze v jedné z nich, vygeneruje se v závislosti na stavu komponenty a druhu tabulky signál pouhého zapnutí či vypnutí komponenty (výstupy *OnlyON* a *OnlyOFF*).

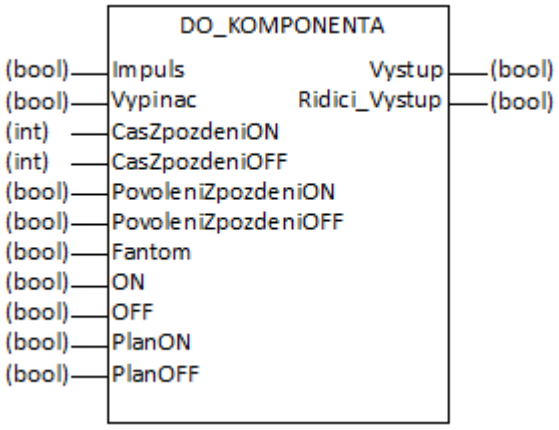
Vyhodnocení vypínačů je závislé jednak na události držení s patřičnými tabulkami, ale i na podtypu vypínače (nastaveno v poli *DI_Prirazeni_Podtypy*). Pokud je v této matici na patřičném místě logická jednička, znamená to, že pro zapnutí komponenty stačí, aby byl sepnut jen jeden takto označený vypínač. Pokud je v matici logická nula, pak vypínače fungují jako klasické křížové přepínače, kdy k zapnutí komponenty je potřeba lichý počet sepnutých spínačů.

Vstupní parametry *CasovePlanyON* a *CasovePlanyOFF* obsahují čísla časových plánů, které zapínají či vypínají komponentu. Vyhodnocení časových plánů probíhá tak, že se z globálního pole, kam systém zapisuje události generované časovými plány nebo funkcí východu a západu slunce, zkopírují aktuální hodnoty. Jinými slovy, pokud časový plán vygeneruje událost, v globálním poli se objeví logická jednička, která se po vykonání funkčního bloku *VYHODNOCENI_UDALOSTI* promítne i na jeho patřičném výstupu.

Simulace je vyhodnocována pomocí dvou vstupně-výstupních proměnných. U proměnné *simulace* je potřebné jednak pomocí proměnné *minula_simulace* vyhodnotit žádost od uživatele z nadřazené aplikace o změnu stavu komponenty, tak ji případně nastavit dle stavu komponenty. To z důvodu, aby například rozsvícené světlo mělo v proměnné pro simulaci logickou jedničku a pro simulaci zhasnutí světla stačilo zapsat logickou nulu.

3.9.3 Program *DO_KOMPONENTA*

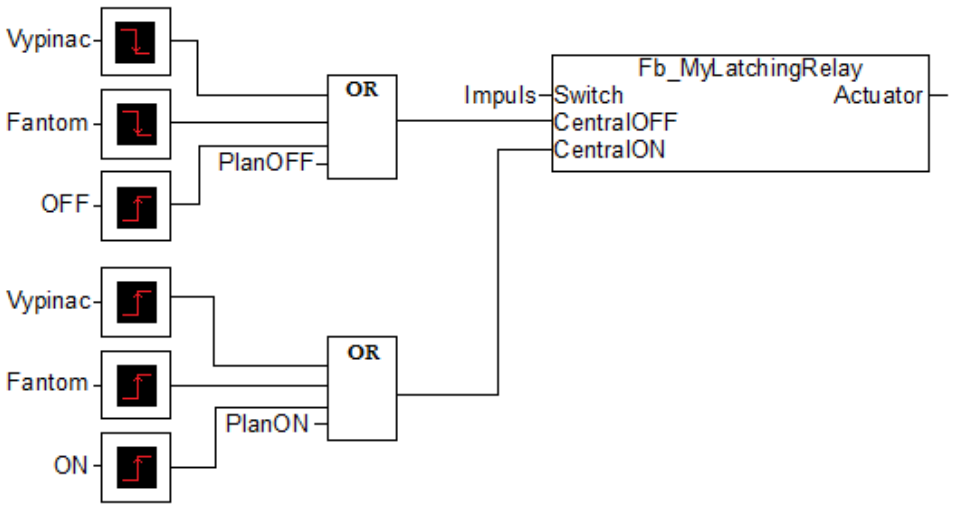
Většina výstupních komponent, jako například světlo, ventilátor, žaluzie, atd., má požadavek na pouhé dvoustavové řízení dle vyhodnocených událostí s přidanou funkčností zpožděného zapnutí či vypnutí. V původní aplikaci byl program s takovým algoritmem pojmenován *SVETLO_VENTILATOR*. Převzal jsem tuto koncepci a rozšířil ji na obecné komponenty digitálního výstupu. Poupravil jsem vnitřní logiku tohoto programu tak, aby lépe vyhovovala koncepci nového modelu. Do logiky jsem přidal řízení pomocí komponenty vypínače, odstranil vyhodnocení typů komponent uvnitř programu a neošetřuji ani centrální zapnutí či vypnutí. Obecná výstupní komponenta je tedy ovládána impulsem, stavem patřičných vypínačů, funkcí Fantom, časovým plánem, simulací z MODBUS/TCP rozhraní nebo nastavením příznaku výchozího stavu při obnovení napájení. Obrázek 4.10 zobrazuje program z vnějšího pohledu.



Obrázek 4.10 – Program *DO_KOMPONENTA*

Protože některé komponenty neřídí svůj stav úrovňovou logikou, jako například následující komponenta impulsního světla, mají komponenty dva výstupní parametry. Parametr *Vystup* označuje aktuální stav komponenty (vypnuto/zapnuto) a parametr *Ridici_Vystup* zajišťuje řídící povely pro fyzickou komponentu. V případě programu *DO_KOMPONENTA* jsou hodnoty obou parametrů identické.

Obrázek 4.11 znázorňuje jednoduchou logiku vyhodnocení vstupních parametrů. Výstup funkčního bloku *Fb_MyLatchingRelay* je dále vyhodnocen pro případ požadovaného opoždění akce, které může být vyvoláno jen ve spojení se vstupy *Impuls* a *Vypinac*.



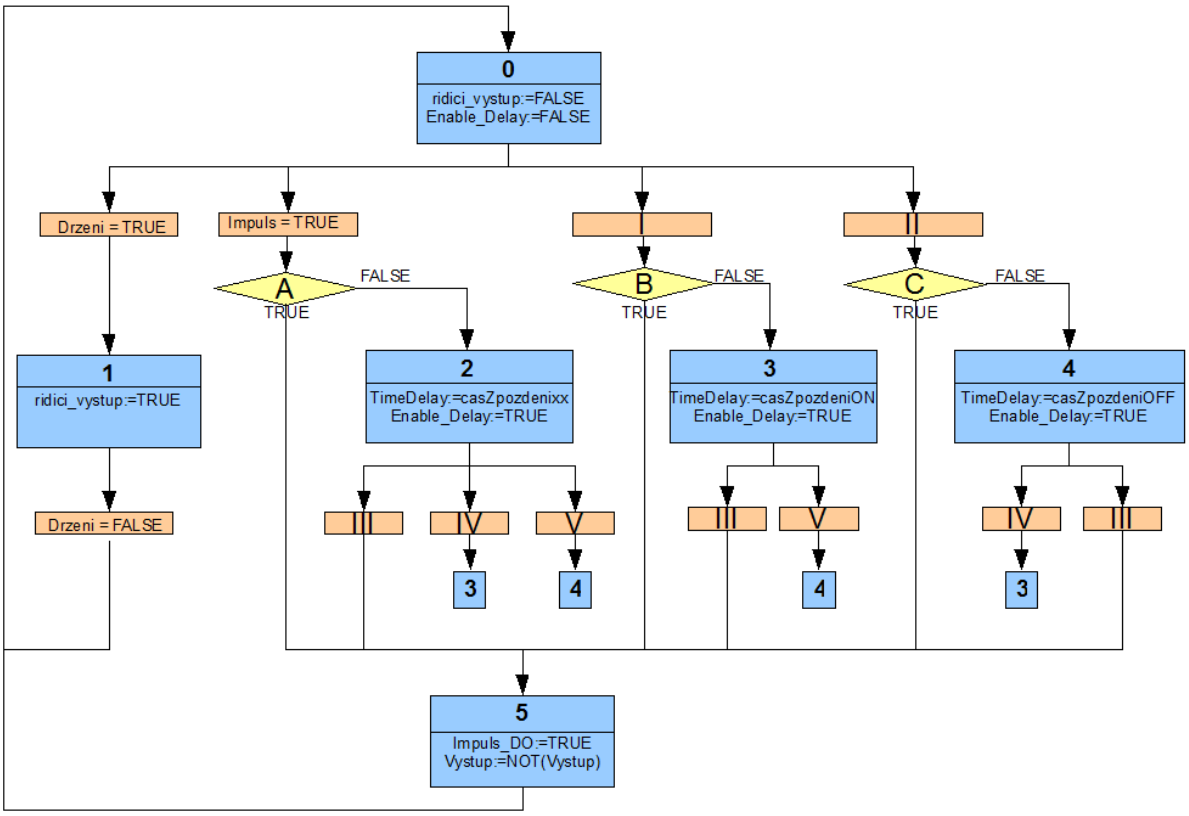
Obrázek 4.11 – Základní logika programu *DO_KOMPONENTA*

Velkou změnou ve vnitřní logice byla výměna funkčního bloku *Fb_LatchingRelay* knihovny Building_common.lib z původní aplikace za vlastní funkční blok pojmenovaný *Fb_MyLatchingRelay*. Problém původního funkčního bloku spočíval ve velké náročnosti na remanentní paměť, kdy jedna instance alokovala několik bytů z remanentní paměti. V novém funkčním bloku jsem převzal funkčnost včetně pojmenování vstupů a výstupů, ale ukládání stavů do remanentní paměti jsem odstranil. Ukládání stavů výstupních komponent do remanentní paměti je řešeno v programu *ZPRACOVANI_VYSTUPU*.

3.9.4 Program *IMPULSNI_SVETLO*

Na reálném velkém modelu je komponenta typu světlo, která má ovšem jiné ovládání. Jedná se o stmívatelné světlo ovládané pomocí impulsů. Krátký puls světlo zapne, případně vypne, a dlouhý puls mění intenzitu jasu od jedné meze k druhé.

Pro tuto komponentu jsem musel sestavit zcela nový algoritmus řízení, program nazvaný *SVETLO_IMPULSNI*. Řídící povely budou stejné jako pro obyčejnou komponentu světla, ale objeví se navíc povel dlouhého pulsu. Obrázek 4.12 vysvětluje použitý algoritmus a tabulka 4.5 doplňuje podmínky přechodů.



Obrázek 4.12 – Princip algoritmu programu *SVETLO_IMPULSNI*

Označení podmínky	Podmínka
A	(*Vystup*=0 & *CasZpozdeniON*=0) | (*Vystup*=1 & *CasZpozdeniOFF*=0)
B	*CasZpozdeniON*=0
C	*CasZpozdeniOFF*=0
I	*Vystup*=0 & (*DO_rtON*[index].Q=1 | *DO_rt_vypinac*[index].Q=1 | *OSV_rtFantom*[index].Q=1)
II	*Vystup*=1 & (*DO_ftON*[index].Q=1 | *DO_ft_vypinac*[index].Q=1 | *OSV_ftFantom*[index].Q=1)
III	*Casovac*.Q=1
IV	*DO_rtON*[index].Q=1 | *DO_rt_vypinac*[index].Q=1 | *OSV_rtFantom*[index].Q=1
V	*DO_ftON*[index].Q=1 | *DO_ft_vypinac*[index].Q=1 | *OSV_ftFantom*[index].Q=1

Tabulka 4.5 – Tabulka podmínek přechodů programu *SVETLO_IMPULSNI*

Program má strukturu jednoduchého krokovače. Základní stav má označení „0“. Do dalších stavů se lze přesunout, pokud obdržíme signál držení, impulsu, zapnutí či vypnutí. Stav „5“ nastavuje požadavek na krátký puls na digitálním výstupu. Jak bylo řečeno, pro ovládání světla je potřeba krátký puls, řádově stovky milisekund. Toto ovládání není možné provádět v hlavním cyklu, proto program pro daný digitální výstup nastaví požadavek na krátký puls (proměnná *Impuls_DO*). V rychlém procesu *SpravaVstupuVystupu* se na základě tohoto požadavku vykoná jeden rychlý puls a fyzické světlo se rozsvítí nebo zhasne.

4.9.5 Program *RIZENI_TOPENI*

Řízení topení bylo opět inspirováno původní aplikací. Ponechal jsem generátor PWM signálu, PID i hysterezní regulátory včetně jejich parametrů. Ale pozměnil jsem vztahy mezi komponentami a přidal nové funkce.

Základní komponentou v původním programu pro řízení vytápění byl digitální výstup typu ventil. K němu se vázaly blokovací kontakty a teplotní čidla. Navíc topné plány neodpovídali struktuře používaných topných plánů v systému DAMIC. Každý plán obsahoval pouze časy a hodnoty deseti změn.

Při programování algoritmu řízení topení jsem se motivoval principem používaným v systému DAMIC. Zde je základní komponentou termostat, ke kterému se připojuje teplotní čidlo, blokovací kontakty, ventily a kotle. Navíc je již zde vymyšlen systém zadávání topných plánů, který se shoduje s běžnou praxí. Nastavování vytápění musí probíhat pouze pomocí nastavení datového modelu a jeho nahráním do automatu. Na základě těchto požadavků, jsem se rozhodl vytvořit základní komponentu z teplotního čidla. A tak pro každý analogový vstup automatu WAGO existuje instance termostatu, která je aktivní, pokud je na daném vstupu jako typ nastaveno teplotní čidlo. Ke každému teplotnímu čidlu (termostatu) může být připojeno několik blokovacích kontaktů a ventilů, ale pouze jeden topný plán, kotel a externí vstup. Zároveň platí, že ventil může být propojen pouze s jedním teplotním čidlem. Kotel, externí vstup a blokovací kontakty mohou mít více vazeb. Z těchto pravidel vznikly následující propojovací tabulky.

- **TEP_prirazeni_senzoru** – Pole o délce zadaného maximálního počtu digitálních výstupů s proměnnými typu INT, kde hodnota proměnné na n-tém místě znamená adresu teplotního čidla spojeného s n-tým ventilem.
- **TEP_Termostat_Kotel** – Pole o délce zadaného maximálního počtu analogových vstupů s proměnnými typu INT, kde hodnota proměnné na n-tém místě znamená pozici kotle spojeného s n-tým teplotním čidlem.
- **TEP_propojeni** - Matice o šířce rovné zadanému maximálnímu počtu vstupních digitálních karet a délce rovné zadanému maximálnímu počtu analogových vstupů. Tato matice značí propojení teplotních čidel a blokovacích kontaktů.
- **TEP_MapaTopnychPlanu** - Pole o délce rovné zadanému maximálnímu počtu analogových vstupů s proměnnými typu BYTE, kde hodnota proměnné na n-tém místě znamená číslo topného plánu připojeného k n-tému teplotnímu čidlu.

- **TEP_ManualTeploty_Vstup** - Pole o délce rovné zadanému maximálnímu počtu analogových vstupů s proměnnými typu INT, kde hodnota proměnné na n-tém místě znamená pozici externího vstupu pro manuální zadání teploty pro n-té teplotní čidlo.

Termostat má navíc proměnnou určující, zda je povoleno řízení teploty na daném termostatu. Tuto hodnotu lze měnit pouze z nadřazené aplikace přes rozhraní MODBUS/TCP. Každý termostat má také proměnnou, do které může nadřazená aplikace zapsat uživatelem aktuálně žádanou teplotu.

Pro každý digitální výstup jsou připraveny instance funkčních bloků pro regulaci teploty a instance funkčního bloku pro řízení výstupu PWM signálem. Pokud je daný výstup označen jako ventil, jsou tyto instance aktivovány.

Každý ventil může být, stejně jako v původní aplikaci, řízen pomocí PID regulátoru nebo pomocí dvoupolohového regulátoru s hysterezí. Typ regulace se nastavuje v poli *TEP_typ_regulace* pomocí číselné hodnoty. Hodnota „1“ je rezervováno pro PID regulátor a hodnota „2“ pro hysterezní. Podobně proměnná *TEP_typ_termoventil* označuje typ termoventilu. Hodnota „1“ je pro ventil, který je bez napětí zavřený a hodnota „2“ pro ventil, který je bez napětí otevřený. Aplikace tak poskytuje snadné rozšíření pro možné jiné způsoby řízení ventilů vytápění.

Vlastní algoritmus

V programu *RIZENI_TOPENI* se nejdříve zkontroluje povolení regulace termostatů a topných plánů. Poté se vyčíslí hodnoty z analogových vstupů jako teploty s desetinou čárkou. Dále se vyhodnotí žádané manuální teploty jak z externího vstupu, tak ty zadané z nadřazené aplikace přes rozhraní MODBUS/TCP. Obě hodnoty se pro řídící algoritmus posléze sloučí do jedné manuálně žádané teploty. Použita je ta, u které došlo k poslední změně.

Pro vyhodnocení aktuálně žádané teploty z topného plánu jsem navrhl funkční blok pojmenovaný *TEPLOTNI_PLAN* (viz obrázek 4.13). Každý termostat má svou instanci, jejímž úkolem je v zadaném topném plánu najít požadovanou teplotu pro aktuální čas. Dalšími parametry tohoto funkčního bloku jsou *PlanON*, který indikuje, zda je povoleno vytápět dle zadaného topného plánu, a *Manualni_Teplota* pro manuálně žádanou hodnotu teploty. Výstup je zaveden do patřičných regulátorů vytápění jako žádaná veličina.

Obrázek 4.13 – Funkční blok *TEPLOTNI_PLAN*

Tento funkční blok je naprogramován tak, aby odpovídal současné praxi, kdy se reguluje dle topného plánu, ale pokud je nastavena nová manuální teplota, má tato teplota vyšší prioritu až do okamžiku další změny topného plánu. Samozřejmě, že pokud není povoleno regulovat podle žádaného topného plánu, objeví se na výstupu hodnota manuální teploty.

Dále je pro každý termostat vyhodnoceno blokování regulace blokovacími okenními kontakty. Jestliže má blokovací kontakt v proměnné *Negace_vstupu* na patřičném místě logickou nulu, pak kontakt v sepnutém stavu blokuje. V opačném případě kontakt blokuje v rozepnutém stavu. Dále algoritmus ovládá jednotlivé ventily. Pokud není nastavena blokace termostatu patřící k danému ventilu a je povolena regulace, pak se vyhodnotí regulátory jednotlivých ventilů. Když se objeví spádová hrana u povolení regulace nebo blokace okenními kontakty, ventil se uzavře. Posléze se vyhodnotí podmínky pro stavy kotlů. V případě, že byl pro daný kotel v posledních deseti sekundách alespoň jeden ventil pootevřen, kotel bude sepnut. Nakonec program obslouží požadavky z nadřazené aplikace pro přímé ovládání ventilů a kotlů. Tyto požadavky program vykoná pouze v případě, že na termostatu, který je přiřazen k požadovanému ventilu, případně kotli, není zapnuta regulace. Pokud termostat reguluje, není povoleno měnit stavy jeho ventilů a kotlů.

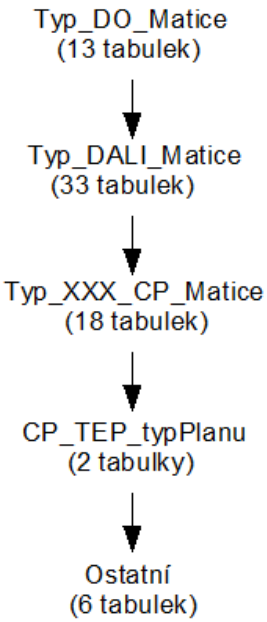
4.10 Algoritmy správy modelu

4.10.1 Program *MODBUS*

Aplikace v automatu využívá pro veškerou komunikaci s nadřazeným vizualizačním systémem rozhraní MODBUS/TCP. Pomocí tohoto rozhraní se načítá, případně vypisuje, model automatu, čte a simuluje aktuální stav komponent. O velikosti, rozčlenění a adresování paměti pojednává kapitola 4.4. V této kapitole popisuji program *MODBUS*, který zajišťuje výměnu dat přes rozhraní MODBUS/TCP při požadavku čtení či zápisu celého modelu chování automatu.

Jak jsem uvedl výše, vzhledem k velikosti adresovatelné paměti jsem realizoval načítání a vypisování modelu pomocí sdílených tabulek. Vznikly tak celkem čtyři sdílené tabulky, které jsou doplněny bytovou proměnnou určující typ tabulky ve sdílené paměti. Program *MODBUS* ošetřuje tyto sdílené tabulky a podle požadavků nadřazeného programu driveru řídí přenos dat modelu.

Program *MODBUS* se dá rozdělit na dvě poloviny. V první je řešeno načítání modelu do automatu, ve druhé výpis modelu z automatu. Obě části jsou téměř identické, rozdílný je pouze směr zápisu. Při načítání modelu se data kopírují ze sdílených matic do patřičných matic v paměti, při výpisu modelu je tomu obráceně. Obě části mají stejnou posloupnost řízení, která je naznačena na obrázku 4.14.



Obrázek 4.14 – Posloupnost vyhodnocení sdílených tabulek

Nejdříve je vyhodnocena sdílená tabulka pro komponenty digitálního výstupu. Dle hodnoty v přičleněné bytové proměnné program přenese data mezi patřičnou a sdílenou tabulkou. Pokud tato bytová proměnná má hodnotu rovnu nule nebo větší než je počet vytvořených tabulek pro tyto komponenty, vyhodnotí se požadovaný typ sdílené matice pro DALI komponenty. A pokud opět bytová proměnná určující typ dat v této sdílené tabulce má hodnotu rovnu nule nebo větší než počet tabulek určených pro DALI komponenty, vyhodnotí se požadovaný typ sdílené matice pro vazby komponent s časovými plány. Po tabulkách s vazbami časových plánů se řídí přenos dat časových nebo topných plánů. Pokud i zde je hodnota nespecifikovaná či rovná nule, pak se ošetří všechna ostatní pole, která nejsou umístěna v adresovatelné paměti, ale mají zde pouze svůj obraz. Jedná se buď o proměnné z remanentní paměti, nebo pole booleovských proměnných, která jsou z důvodu úspory paměti přenášena po bytech. Po přenesení dat mezi tabulkami se nakonec vynuluje bitová proměnná požadující načtení nebo výpis modelu. Díky tomu může nadřazený program poznat, že přenos žádané tabulky byl dokončen.

V této sekvenci ošetření má zvláštní význam bytová proměnná *Typ_DALI_Matice*. Zde se kromě hodnot podle očíslovaných tabulek DALI komponent objevuje navíc hodnota 255. Tím se v případě načítání modelu z nadřazené aplikace zadá automatu požadavek, aby bylo do všech DALI zařízení uloženo přiřazení jednotlivých zařízení do skupin a hodnoty scén. Při výpisu modelu naopak automat čeká, až bude přes sběrnici přečteno přiřazení jednotlivých zařízení do skupin a hodnoty scén ze všech zařízení DALI. Pokud jsou všechna data ze sběrnice přečtena, program *MODBUS* resetuje proměnnou *Vypis_Modelu* a tím informuje nadřazený program driveru, že data v adresovatelné paměti jsou aktuální.

Ostatní pole mají svou pevnou adresu a mohou být proto přímo přečtena nebo přepsána.

4.10.2 Program *SOUBORY*

Práce se soubory umístěnými na vnější paměťové kartě flash zajišťuje knihovna SysLibFile.lib. Ta obsahuje základní funkce pro práci se soubory. V mé aplikaci jsem použil následující funkce:

- ***SysFileOpen*** a ***SysFileClose*** pro otevření a zavření souboru
- ***SysFileRead*** a ***SysFileWrite*** pro čtení a zápis bytového pole do souboru
- ***SysFileDelete*** pro smazání starého souboru

Funkce *SysFileRead* a *SysFileWrite* čtou nebo zapisují do souboru zadaný počet bytů od zadané adresy. A tak, abych mohl zapsat celý model do paměti, musím nejdříve vytvořit pole typu byte, do kterého postupně překopíruji všechny matice a proměnné modelu. Toto pole je uloženo do binárního souboru a v případě požadavku je zase načteno.

Překladač nepovolí vytvoření bytového pole s délkou větší než 24kB. Proto jsem byl nucen vytvořit celkem tři soubory, do kterých se již kompletní model vejde. Soubory mají název „Modelx“, kde x značí číslici pořadí souboru. Remanentní proměnné samozřejmě do flash paměti neukládám.

Program *SOUBORY* tak plní jednoduchou úlohu. Při požadavku na zapsání modelu do vnější paměti flash program nejdříve smaže staré soubory. Poté připraví bytové pole, které zapíše do prvního souboru. Následuje přepsání bytového pole a jeho zápis do druhého souboru a pak se stejným způsobem uloží i třetí soubor. Nakonec program resetuje požadavek na zapsání modelu.

Při požadavku na přečtení modelu z vnější paměti, funguje program přesně obráceně. Nejdříve se přečtou bytová pole z jednotlivých souborů a z nich se zrekonstrují patřičné matice modelu. Nakonec program resetuje požadavek na čtení modelu.

4.11 Ostatní algoritmy

4.11.1 Program *POCATECNI_NASTAVENI*

Po restartu automatu je třeba nastavit ofsety pro digitální signály vstupů a výstupů a také nastavit komponenty do stejného stavu jako byly před tímto restartem. To je zajištěno v programu *POCATECNI_NASTAVENI*, který je vyvolán událostí restartu automatu.

Program *POCATECNI_NASTAVENI* pomocí funkcí z knihovny mod_com.lib nastaví ofsety digitálních signálů a poté se postará o překopírování dat z remanentní paměti do patřičných obrazů v adresovatelné části paměti. Na závěr program nastaví požadavek o přečtení modelu z vnější paměti FLASH a příznaky k uvedení výstupních komponent do předchozího stavu. K tomuto účelu slouží pole *VychoziStavON* a *VychoziStavOFF*. Pokud byla komponenta v předchozím stavu zapnutá, pak má na patřičném místě v remanentní paměti logickou jedničku a tato jednička se promítne i do pole *VychoziStavON*. Řídící program v hlavním cyklu automatu pak vyhodnotí příznak a zapne komponentu. Podobná pole existují i pro DALI světla.

4.12 Sběrnice DALI

4.12.1 Koncept ovládání zařízení DALI

Mojí snahou bylo začlenit sběrnici DALI do aplikace automatu WAGO tak, abych nemusel příliš měnit dosavadní koncepci programu. Chtěl jsem docílit, aby komponenty DALI byly stejně jako komponenty digitálních výstupů řízeny na základě událostí ze vstupních signálů, časových plánů a funkce Fantom. Navíc muselo platit, že komponenty DALI budou plně konfigurovatelné z nadřazeného datového modelu systému DAMIC.

Nejdříve jsem se pokusil vytvořit koncept řízení komponent DALI tak, že bych zcela ignoroval možnost přiřadit světla do skupin a v programu tak řídil jen jednotlivá světla. Pokud by událost na jedné komponentě měla řídit více světel, jednoduše by program řízení vyslal přes sběrnici více příkazů najednou. Výhodou by bylo, že nadřazený datový model by tvořil skupiny světel na základě přiřazení DALI světel ke stejným událostem, jako je tomu v případě digitálních výstupních komponent. Pro uživatele by bylo takové ovládání více intuitivní.

Ovšem můj první návrh ztroskotal na kapacitě DALI sběrnice. Při realizace tohoto principu ovládání se příkaz, který byl poslaný několika komponentám DALI světel, projevil postupným a pozvolným prováděním příkazu po jednotlivých světlech. Namísto, aby se například daná světla rozsvítila najednou nebo alespoň v akceptovatelně rychlém sledu, rozsvěcelo se pomalu jedno světlo po druhém. Při velkém počtu světel se někdy i některé

nerozsvítilo. To rozhodně neodpovídalo chování světel, jaké bychom v budovách očekávali. Vzhledem k těmto zkušenostem jsem koncept řízení přepracoval.

V novém konceptu jsem se rozhodl, že pod komponentou DALI budu uvažovat nejen samotné světlo, ale také skupinu. DALI komponenty budou řízeny z hlavního cyklu automatu jako ostatní komponenty. Dále jsem se rozhodl vytvořit čtvrtý proces *SpravaDALI*, který bude mít středně dlouhou dobu periody a který bude sloužit pro řízení toku informací po sběrnici DALI. Tento proces bude periodicky číst stavy světel, číst a zapisovat hodnoty scén, zapisovat nebo číst přiřazení světel do skupin.

Vzhledem k nízké kapacitě sběrnice je nutno zajistit, aby nebylo po sběrnici posíláno více příkazů najednou. Mým předpokladem je, že v praktickém použití je velice nízká pravděpodobnost vzniku dvou událostí ovládající komponenty DALI ve stejném okamžiku. Tedy že hlavní program řízení v jednom okamžiku nevyšle příkazy pro více než jednu komponentu DALI. S tím také souvisí poučení uživatele, aby nevytvářel stejné vazby různých DALI komponent. Chce-li ovládat jednou událostí více světel, měl by nejdříve vytvořit patřičnou skupinu a onou událostí ovládat skupinu. Dále je nutné synchronizovat posílání příkazů po sběrnici mezi hlavním cyklem a procesem spravujícím DALI sběrnici. Pokud hlavní cyklus vyšle libovolné komponentě DALI řídící příkaz, přeruší se na potřebnou chvíli v procesu *SpravaDALI* vykonávání programu. Řídící příkazy tedy budou mít jasnou přednost.

Specialitou DALI světel je možnost použití scén. Každé světlo má prostor pro 16 scén a v každé může být uložena jiná hodnota jasu. Světla tak mohou skokově měnit svou intenzitu jasu v závislosti na volané scéně. Výhodné je to hlavně u skupin, kdy se rázem změní intenzita jasu u několika světel. Použití scén v mém programu bude mít dvě omezení.

Zaprvé snížím počet možných scén na osm. Myslím si, že je to stále dostatečné množství scén pro běžné použití v domovní automatizaci. Důvodem ani tak není uspoření paměti, ale uspoření času potřebného pro zápis hodnot scén do všech zařízení DALI.

Zadruhé mezi scénami se bude moci přecházet jen na následující scénu ve vzestupném pořadí. Důvodem je přání používat jen jedinou událost pro přepínání scén bez možnosti výběru konkrétní.

Změna nastavení scén bude možná i během klasického používání komponent DALI v domovní automatizaci. Pokud uživatel změní jas světla, nová hodnota se automaticky uloží jako nová hodnota jasu v aktuální scéně.

4.12.2 Použité funkční bloky

Pro svůj záměr jsem se rozhodl použít níže popsané funkční bloky z knihovny DALI_02.lib.

FbDALI_Joblist

Funkční blok *FbDALI_Joblist* zajišťuje veškerou komunikaci s DALI modulem 750-641 připojeným k hlavní procesní jednotce automatu WAGO. Tomuto funkčnímu bloku jsou posílány všechny příkazy směřujících na DALI sběrnici a naopak odpovědi ze sběrnice jsou tímto blokem zprostředkovány ostatním blokům. Vstupní parametr *bModule_750_641*, který se dále objevuje ve všech ostatních funkčních blocích, adresuje konkrétní modul zapojený v automatu. Výstupním parametrem *feedback* je číselný kód značící stav DALI modulu. I tento parametr se objevuje u všech ostatních funkčních bloků. Kódy jsou popsány v manuálu ke knihovně DALI_02.lib [10].

Ve své aplikaci předpokládám pouze jeden DALI modul zapojený k procesní jednotce. Proto se v mé aplikaci objevuje jediná instance tohoto bloku, která je volána v rychlém procesu *SpravaVstupuVystupu*, aby dokázala reagovat na příkazy z programů volaných ve zbylých procesech. Vstupní parametr *bModule_750_641* musí být u všech použitých funkčních bloků shodně nastaven na hodnotu „1“.

FbDALI_Master

Funkční blok *FbDALI_Master* umožňuje posílat k jednotlivým světlům nebo skupinám základní příkazy specifikované normou DIN IEC 60929. Přehled všech možných signálů je uveden v manuálu [10]. V tabulce 4.5 jsem uvedl mnou použité příkazy.

Příkaz	Číslo příkazu pro jednotlivé světlo	Číslo příkazu pro skupinu světel
Zhasnout	0	300
Zvýšit intenzitu jasu	1	301
Snížit intenzitu jasu	2	302
Rozsvítit na minimální jas	6	306
Přejít na scénu 1-8	16-23	316-323

Tabulka 4.6 – Použité příkazy pro funkční blok *FbDALI_Master*

Vstupní parametr *bAddress* adresuje komponentu, které bude příkaz popsaný číslem v parametru *iCommand* patřit. Rozeznání, zda je příkaz určen světlu nebo skupině, se provádí pomocí čísla příkazu. Příkazy pro skupinu začínají od čísla 300. Pokud příkaz potřebuje vstupní hodnotu, je uvedena v parametru *bCommandValue*. Mezi důležité parametry také patří vstupně-výstupní parametr *xStartDaliMaster*. Vzestupnou hranou na tomto parametru vyšleme požadavek na provedení příkazu. Po úspěšném provedení funkční blok tuto proměnnou resetuje. Výstupní parametr *bQueryValue* je určen pro návratovou hodnotu v případě, že příkaz znamenal dotaz.

Ve své aplikaci jsem použil tento funkční blok uvnitř mnou vytvořeného funkčního bloku *SVETLO_DALI* pro řízení DALI komponent, jehož výhodou je přítomnost pouze jednoho funkčního bloku pro základní ovládání DALI komponenty.

FbDALI_ConfigDevice

Funkční blok *FbDALI_ConfigDevice* slouží pro nastavení jednotlivých zařízení DALI. Mezi možnosti nastavení patří maximální a minimální úroveň intenzity jasu světla, rychlost změny jasu, úroveň jasu při zapnutí a zejména nastavení příslušností zařízení do skupin DALI.

Funkční blok *FbDALI_ConfigDevice* jsem využil pouze pro nastavení nebo přečtení příslušností DALI zařízení ke skupinám. Z množství vstupních a výstupních parametrů mě tedy budou zajímat pouze parametry *bAddress* určující adresu zařízení, *bGroupAddress_8_1* a *bGroupAddress_16_9* určující požadované příslušnosti světla ke skupinám, *bGroups_8_1* a *bGroups_16_9* informující o příslušnostech světla ke skupinám, jak je uloženo v zařízení. Bitový vstupní parametr *xSetGroup*, resp. *xQuery*, slouží pro příkaz nastavení, resp. přečtení, příslušností světla ke skupinám.

Příslušnost světla ke skupinám se uvádí ve dvou bytových parametrech. V prvním z nich hodnoty jednotlivých bitů značí, zda světlo náleží do skupiny 1 až 8, a ve druhé, zda náleží do skupiny 9 až 16.

Tento funkční blok je volán v programu *MONITORING_DALI* při požadavku uložení příslušností světel ke skupinám přímo do fyzických zařízení dle nahraného modelu nebo při požadavku načtení příslušností světel do nadřazeného datového modelu.

FbDALI_Config_Scene

Funkční blok *FbDALI_ConfigScene* slouží pro zápis hodnot scén do fyzických zařízení DALI. Ze vstupních parametrů budou pro mě zajímavé parametry *bAddress* a *bSceneNumber* určující číslo scény, do které se nahraje hodnota z parametru *bDimmValue*.

Parametr *bDimmValue* je interpretován jako procentuální rozsvícení a proto může nabývat pouze hodnot v intervalu 0-100. Vzestupná hrana na parametru *xSet* způsobí start zápisu hodnoty.

Tento funkční blok je volán v programu *MONITORING_DALI* při požadavku uložení hodnot scén do fyzických zařízení dle nahraného datového modelu. Protože aktuální hodnoty scén jsou pomocí funkčního bloku *FbDALI_StatusDimmValue* periodicky načítány a zapisovány do patřičné tabulky, při požadavku výpisu modelu do nadřazeného datového modelu lze vypsat rovnou tuto tabulku.

FbDALI_StatusDimmValue

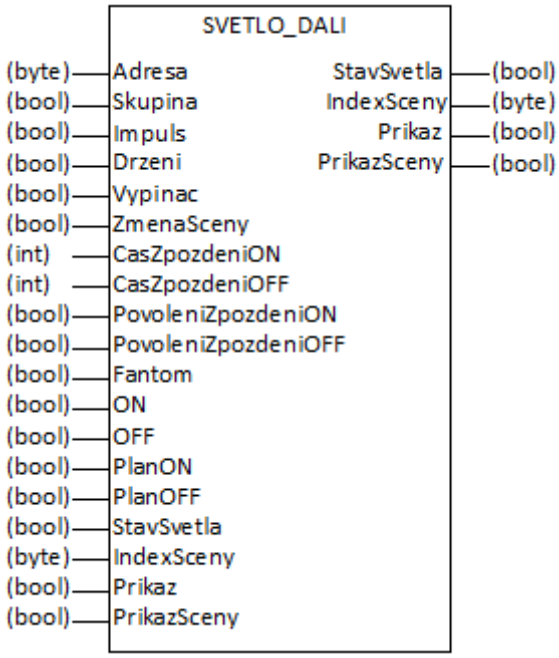
Funkční blok *FbDALI_StatusDimmValue* slouží pro periodické čtení aktuálního stavu a intenzity jasu světla. Vstupní parametr *xEnable* povoluje periodické čtení, *bShortAddress* udává adresu dotazovaného světla a *tCycleTime* je čas periody opakování dotazu. Z výstupních parametrů je důležitý parametr *xStatus* informující o stavu světla a *bDimmValue*, který obsahuje aktuální hodnotu intenzity jasu. Parametr *xReady* slouží pro indikaci aktivního dotazu.

Tento funkční blok má jedinou instanci, která je volána v programu *MONITORING_DALI*. Nevyužil jsem její samostatné periodické volání, ale při náběžných hranách parametru *xReady* měním postupně adresu zařízení a tak zjišťuji stav všech světel na sběrnici DALI.

4.12.3 Funkční blok *SVETLO_DALI*

Pro řízení komponent DALI jsem vytvořil vlastní funkční blok, který bude mít několik stejných parametrů jako program *DO_KOMPONENTA* z důvodu využití funkčního bloku *VYHODNOCENI_VSTUPU*. Ovšem pro DALI komponenty je třeba navíc vyhodnotit události pro změnu jasu a změnu scény. Tento funkční blok se jmenuje SVETLO_DALI a je zobrazen na obrázku 4.15

Funkční blok *SVETLO_DALI* se v závislosti na svých vstupních parametrech může dostat do pěti stavů. Stav „0“ je základní. V tomto stavu blok vyhodnocuje požadavky na akce v pořadí popsaném obrázkem 4.16.

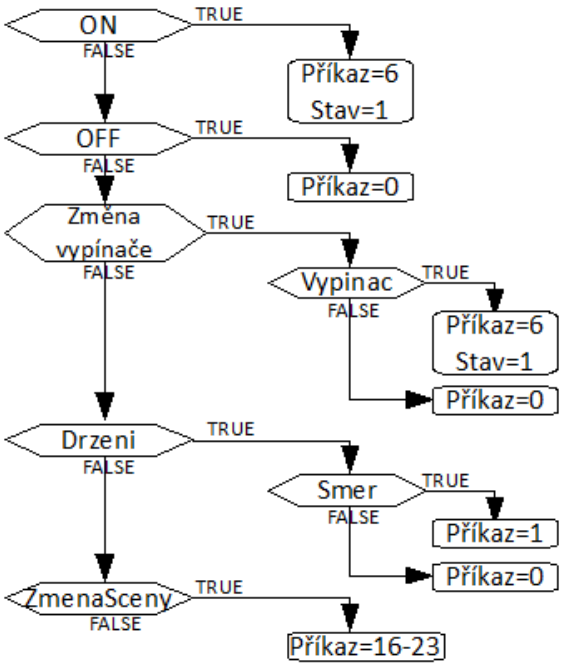


Obrázek 4.15 - Funkční blok *SVETLO_DALI*

Akci rozsvícení DALI komponenty jsem rozdělil na dvě části. V prvním kroku je třeba zajistit zapnutí komponenty a ve druhém rozsvícení komponenty na správnou hodnotu jasu. Pokud zapneme komponentu na minimální jas, vyhneme se tak nepříjemnému bliknutí komponenty. Pokud funkční blok *SVETLO_DALI* vyhodnotí aktivní požadavek na zapnutí komponenty, nejdříve vyšle příkaz pro zapnutí komponenty do minimální intenzity jasu a přejde do stavu „1“. V tomto stavu čeká na potvrzení o provedení příkazu a poté vyšle příkaz k přechodu na konkrétní scénu a přejde do stavu „2“. Zde také čeká na potvrzení o provedení příkazu a poté se vrátí do stavu „0“.

Stavy „3“ a „4“ slouží pro zpožděné zapnutí a vypnutí. Pokud funkční blok vyhodnotí aktivní požadavek na zapnutí se zpožděním, přejde do stavu „3“, ve kterém vyčká zadaný čas a teprve poté vyšle příkaz k zapnutí komponenty do minimální intenzity jasu a přejde do stavu „1“. Pro zpožděné vypnutí automat čeká ve stavu „4“.

Aby mohl být funkční blok použit i pro ovládání skupin, objevuje se mezi vstupy parametr *Skupina*. Pokud je na tomto vstupu logická jednička, jsou čísla příkazů posunuty o hodnotu 300. Pokud slouží funkční blok *SVETLO_DALI* pro řízení skupiny světel, je nutné také zajistit, aby se změna čísla aktuální scény u skupiny promítla i do proměnné s číslem aktuální scény u jednotlivých světel příslušejících ke skupině.



Obrázek 4.16 - Stav 0 funkčního bloku *SVETLO_DALI*

Parametr *AktualniStav* je vstupně-výstupní proměnná, do které si funkční blok ukládá předpokládaný stav komponenty. Mohl bych použít globální proměnnou s přečtenými stavy komponent z procesu *SpravaDALI*, ale přečtení reálných stavů světel trvá značně dlouhou dobu. Proto použiji tyto předpokládané stavy, aby mohla být světla DALI v krátké době za sebou rozsvícena a zhasnuta.

Parametr *Drzeni* slouží pro vyhodnocení události dlouhého stisku a komponenta reaguje změnou intenzity jasu. Při vzestupné hraně této proměnné se neguje proměnná indikující, zda jas bude sílit nebo zeslabovat. Protože by při simulaci změny jasu z nadřazené aplikace nastala situace, že by se směr změny jasu při každém požadavku měnil, je přítomen parametr *SimulaceDrzeni*. Pokud je na tento parametr přivedena logická jednička, směr jasu se po dobu deseti sekund nezmění. To bude mít za následek, že pokud nadřazená aplikace do deseti sekund vyšle nový požadavek na změnu jasu, jas se bude měnit ve stejném směru.

Parametr *Prikaz* je vstupně-výstupní proměnná, ve které funkční blok v případě vyslání řídícího příkazu indikuje zabrání sběrnice. Rychlé zablokování sběrnice je zajištěno nastavením 0.bitu globální proměnné *DALI_STATUS*. K resetování tohoto bitu dojde za předpokladu, že všechny komponenty v daném cyklu nepřistoupily na sběrnici, tedy nenastavili parametr *Prikaz*.

Parametr *PrikazSceny* má podobný význam jako parametr *Prikaz*. Funkční blok nastaví tuto proměnnou při poslání příkazu na změnu scény. Tato proměnná zůstane nejméně další dvě vteřiny nastavena. Pokud by tato proměnná nebyla použita, hrozilo by během změny scén zapsání nesprávných hodnot jasů na místa dle již nové aktuální scény.

4.12.4 Popis potřebných tabulek

Pro správu DALI komponent je potřeba několik specifických tabulek a proměnných. Samozřejmostí jsou tabulky definující vazby mezi DALI komponentami a událostmi, tabulky s časy zpoždění DALI komponent a tabulky pro simulace akcí rozsvícení, změny scény a změny jasu. Také existují tabulky pro uvedení DALI komponent do původního stavu a stavu dle nahraného modelu. Tabulka 4.7 přehledně popisuje další speciální tabulky.

Název tabulky	Rozměr tabulky	Popis
DALI_Svetla_stav	Pole typu BOOL o délce 64	Pole s aktuálními stavy jednotlivých DALI světel, které budou periodicky postupně zjišťovány pomocí dotazů posílaných po DALI sběrnici. Pole bude uloženo v zálohované paměti (retain), aby mohl být obnoven původní stav po restartu automatu. Pole má svůj obraz v adresovatelné části paměti.
DALI_AktualniScena	Vektor typu BYTE o délce 64	Vektor s aktuálními čísly scén u jednotlivých DALI světel. Vektor bude uložen v zálohované paměti (retain), aby po restartu automatu zůstaly aktivní původní scény. Pole má svůj obraz v adresovatelné části paměti.
DALI_AktualniScenaGrupy	Vektor typu BYTE o délce 16	Vektor s aktuálními čísly scén u DALI skupin. Vektor bude uložen v zálohované paměti (retain), aby po restartu automatu zůstaly aktivní původní scény. Pole má svůj obraz v adresovatelné části paměti.
DALI_Sceny	Matice typu BYTE o rozměrech 64x8	Matice, kde v řádku jsou hodnoty jasu všech scén jednotlivého světla. Počet řádků je roven maximálnímu počtu světel DALI, tedy 64.
DALI_Grupy_prirazeni	Matice typu BYTE o rozměrech 64x2	Matice, kde v řádku jsou dva byty nesoucí informaci o příslušnosti daného světla ke skupinám. Tyto dva byty jsou použity jako vstupní nebo výstupní parametry pro funkční blok *FbDALI_ConfigDevice*. Počet řádků je roven maximálnímu počtu světel DALI, tedy 64.
DALI_pouziteAdresy	Vektor typu BYTE o délce 8	Bitové pole uložené v bytové podobě, kde jednotlivé bity značí, zda je na dané adrese nastavené fyzické světlo DALI.

Tabulka 4.7 – Speciální tabulky pro správu DALI

4.12.5 Program MONITORING_DALI

Program *MONITORING_DALI* je volán v samostatném cyklickém procesu *SpravaDALI* s dobou periody 500ms. Tento program je naprogramován tak, aby spravoval DALI zařízení v době, kdy řídící program hlavního procesu nevysílá žádné příkazy. Program za běžných okolností postupně čte aktuální stavy a hodnoty jasů všech světel DALI. Tyto stavy zapisuje do remanentního pole *DALI_svetla_stav* a v případě změny hodnoty jasu uloží novou hodnotu jak do matice *DALI_Sceny*, tak do samotného zařízení. Dále má program na starosti čtení i zápis hodnot jasů všech scén a zápis příslušnosti světel ke skupinám při nahrání či výpisu modelu do nadřazené aplikace ovladače.

Pro komunikaci mezi hlavním procesem a procesem *SpravaDALI* jsem navrhnul proměnnou *DALI_Status*, která slouží jako pole příznaků a má strukturu dle tabulky 4.8.

DALI_Status	
číslo bitu	**Význam**
0	Blokování vykonávání programu *MONITORING_DALI* při vyslání příkazu z řídícího programu z důvodu malé kapacity sběrnice.
1	Zjištění změny jasu určitého světla a požadavek na zapsání nové hodnoty do fyzického zařízení.
2	Blokování zápisu hodnot přečtených jasů do tabulky *DALI_Sceny* během změny scény určitého světla. Mohlo by dojít k přepsání sousední buňky původní hodnotou.
3	Blokování zapisování DALI tabulek ze zálohované paměti do patřičných obrazů v adresovatelné paměti. Používá se při načítání modelu z nadřazeného programu, kdy se do obrazů nahrají požadované hodnoty, kterými se naopak přepíší zálohované tabulky.
4	Požadavek na přečtení příslušností světel ke skupinám ze zařízení DALI.
5	Hodnota požadavku na přečtení příslušností světel ke skupinám ze zařízení DALI v minulém cyklu vykonávání programu. Slouží pro detekci náběžné hrany.
6	Požadavek na uložení příslušností světel ke skupinám dle nahraného modelu.
7	Hodnota požadavku na uložení příslušností světel ke skupinám dle nahraného modelu v minulém cyklu vykonávání programu. Slouží pro detekci náběžné hrany.
8	Požadavek na uložení hodnot intenzit jasů v jednotlivých scénách do zařízení DALI dle nahraného modelu. A také uvedení světel do stavů dle nahraného modelu.
9	Hodnota požadavku na uložení hodnot intenzit jasů v jednotlivých scénách do zařízení DALI dle nahraného modelu v minulém cyklu vykonávání programu. Slouží pro detekci náběžné hrany.
10-15	Rezerva

Tabulka 4.8 – Struktura proměnné DALI_Status

Program *MONITORING_DALI* je vykonáván, pokud je v sestavě automatu přítomna karta DALI a na sběrnici naadresované alespoň jedno zařízení. Poté se vyhodnotí podmínka, zda není nastaven nultý bit proměnné *DALI_Status*. Pokud není nastaven, znamená to, že řídící program nevysílá žádný řídící příkaz a sběrnice DALI je volná pro správu DALI zařízení.

Základním úkolem správy zařízení DALI je postupné periodické čtení stavu a hodnot jasů jednotlivých světel. K tomuto účelu slouží funkční blok *FbDALI_StatusDimmValue*, u kterého se postupně mění adresa zařízení od jedné do počtu naadresovaných světel. Signálem pro inkrementaci adres je náběžná hrana výstupního parametru *xReady*. Při zjištění změny jasu oproti hodnotě uložené v tabulce *DALI_Sceny*, program nastaví 1.bit proměnné *DALI_Status*, který v následujícím cyklu programu namísto čtení stavu následujícího světla zajistí uložení nové hodnoty intenzity jasu pod aktuální číslo scény. K tomuto účelu jsem využil funkční blok *FbDALI_ConfigScene*. Po uložení nové hodnoty program přejde ke čtení stavu následujícího světla.

Koncepci čtení stavů a intenzit jasů přímo ze zařízení jsem zvolil proto, aby se změna intenzity jasu světla uživatelem promítla do modelu. Mým záměrem je ovládání hodnot intenzit pomocí prvků domovní automatizace, nikoli pouhým přepsáním nadřazeného datového modelu a jeho následným přehráním do automatu. Čtení stavů přímo z fyzické komponenty se při realizaci ukázalo jako praktické, neboť se občas světlo přes příkaz nerozsvítilo. Postupné čtení po jednom světle jsem zvolil z důvodu kapacity sběrnice. Vzhledem k tomuto způsobu čtení stavů světel se ve výsledné aplikaci objeví značné zpoždění mezi řídícím příkazem a zjištěním stavu v nadřazené aplikaci. Toto zpoždění bude tím delší, čím více naadresovaných světel na sběrnici bude přítomno. Při ovládání domovní automatice se toto zpoždění projeví zpožděním mezi změnou intenzity jasu a uložením této nové hodnoty do modelu automatu.

Postupné periodické čtení může být přerušeno celkem třemi požadavky. Priorita těchto požadavků je shodná s pořadím popisu. Prvním z nich s nejvyšší prioritou je požadavek na vyčtení příslušností světel ke skupinám přímo ze zařízení, který je indikován 4.bitem v proměnné *DALI_Status*. Tento požadavek se objeví při výpisu modelu automatu do nadřazené aplikace driveru. Druhým je naopak požadavek o zápis příslušností světel ke skupinám do zařízení, který je indikován 6.bitem proměnné *DALI_Status*. Tento požadavek se objeví při zápisu modelu automatu z nadřazené aplikace driveru. K těmto dvěma situacím využiji funkční blok *FbDALI_ConfigDevice*. Třetí požadavek, který přeruší pravidelné periodické čtení stavů světel, je požadavek o zapsání hodnot scén přímo do zařízení DALI, který je indikován 8.bitem proměnné *DALI_Status*. Tento požadavek se také objeví při zápisu modelu automatu z nadřazené aplikace. K této práci využívám funkční blok *FbDALI_ConfigScene*. Pořadí nahrávání hodnot jsem zvolil tak, aby se nejdříve u všech světel zapsaly intenzity jasů pro první scény, pak pro druhé scény, atd. Po této operaci dojde ještě k nastavení světel do stavů dle nahraného modelu.

4.12.6 DALI_Config

Jediná vlastnost, kterou nelze nastavit z nadřazeného datového modelu, je fyzická adresa zařízení DALI. Z tohoto důvodu jsem ponechal v aplikaci přítomný funkční blok *DALI_Config*, který spravuje obrazovky webové vizualizace potřebné pro veškerou konfiguraci DALI zařízení včetně fyzických adres jednotlivých světel. Dle naprogramovaných algoritmů je nutné dodržet, aby adresy byly po sobě jdoucí čísla od 1.

Mohlo by se zdát, že je tímto porušena koncepce úplné konfigurace modelu automatu pomocí nadřazené aplikace. Ale pokud si uvědomíme, že nastavení fyzické adresy bude v povinnostech technika, který bude systém domácí automatizace instalovat, a uživatel nebude mít práva přepojovat nebo měnit adresy komponent, pak koncepce porušena nebyla. Fyzické adresy budou na počátku nastaveny technikem a posléze se již měnit nebudou.

4.13 Technologie EnOcean

4.13.1 Koncepce zapojení technologie EnOcean

Z koncepce technologie EnOcean vyplývá, že ze sběrnice EnOcean mohu získat pouze vstupní signály, ať v digitální nebo analogové podobě. Proto jsem se rozhodl tuto technologii do systému zabudovat tak, aby senzory při své změně stavu mohly přepsat hodnoty vstupů, které se primárně čtou ze vstupních karet automatu. Tedy stejným způsobem jako nadřazená aplikace simuluje vstupní signály. Výhodou tohoto přístupu je, že nemusím přidávat tabulky chování automatu. Pro řídící programy se dané vstupy jeví, jako kdyby zde byl pouze vstup z fyzické karty. Také můžeme bez problémů sestavit systém tak, že na dané adrese bude jak signál z fyzické vstupní karty, tak ze sběrnice EnOcean. V tom případě platí ten signál, na kterém proběhla poslední změna. V případě změn obou signálů ve stejný okamžik, má větší prioritu signál ze vstupní karty.

Na větším modelu jsem měl k dispozici pouze jeden čtyřtlačítkový senzor EnOcean. Proto jsem ve své aplikaci vytvořil pouze správu digitálních hodnot ze sběrnice EnOcean.

4.13.2 Použité funkční bloky

FbEnoceanReceiver

Funkční blok *FbEnoceanReceiver* zajišťuje veškerou komunikaci s EnOcean modulem 750-642 připojeným k hlavní procesní jednotce automatu WAGO. Tento funkční blok poskytuje ostatním blokům data z přijatých paketů. Vstupní parametr *bModule_750_642* adresuje konkrétní modul zapojený v automatu. Výstupní parametr

typEnocean se objevuje u všech ostatních funkčních bloků jako vstupní parametr. Více informací v manuálu ke knihovně EnOcean_04.lib [11].

V mé aplikaci je tento funkční blok vykonáván v rychlém procesu *SpravaVstupuVystupu*.

FbShow_ID_By_Button

Funkční blok *FbShow_ID_By_Button* slouží k nalezení identifikačního čísla EnOcean komponenty spínače. Blok na svém výstupu *dwID* zobrazí z přijatého paketu identifikační číslo EnOcean komponenty spínače, ale pouze v případě, že je na komponentě stisknut určitý počet tlačítek najednou. Tento počet je dán vstupním parametrem *bButton_Count*.

V mé aplikace se tento funkční blok vyskytuje v programu *EnOcean_Konfigurace*.

FbButtonSelectChannel

Pro svůj záměr jsem potřeboval funkční blok, pomocí kterého bych mohl k libovolnému digitálnímu vstupu přiřadit konkrétní fyzické tlačítko EnOcean komponenty. Po prostudování knihovny EnOcean_04.lib jsem takovýto funkční blok objevil.

Parametry tohoto funkčního bloku jsou:

- *typEnocean* – Připojení funkčního bloku přijímače EnOcean paketů
- *dwID* – Identifikační číslo EnOcean spínače
- *bButton* – Číslo tlačítka na fyzické komponentě spínače EnOcean
- *tTimeOut* – Čas, po který může být daný vstup maximálně sepnut (globálně nastaveno 20 sekund)
- *xButton* – Bitová hodnota daného tlačítka na fyzické komponentě spínače EnOcean (výstupní parametr)

4.13.3 Program *EnOcean_tlacitka*

Na základě uvedených parametrů funkčního bloku *FbButtonSelectChannel* jsem sestrojil program následovně. Pro každý digitální vstup existuje jedna instance tohoto funkčního bloku, jehož výstupní hodnoty se ukládají do pole *EnOcean_VystupniPoleDI*. Při změně této hodnoty se změna zapíše do vstupních dat. Vstupní parametry *dwID*, resp. *bButton*, každé instance jsou načítány z pole *EnOcean_PoleID*, resp. *EnOcean_PoleCiselTlac*, ve kterém má každá instance na své pozici nakonfigurované identifikační číslo EnOcean komponenty, resp. číslo tlačítka. Tím snadno docílíme obecného přiřazení konkrétního tlačítka na libovolný digitální vstup. Konfiguraci můžeme vykonávat pomocí k tomuto účelu naprogramované nadřazené aplikace.

4.13.5 Program *EnOcean_Konfigurace*

Program En_Ocean_Konfigurace slouží, jak název napovídá, pro nakonfigurování instancí funkčních bloků *FbButtonSelectChannel*. Pro jeho vykonávání je třeba nastavit 0.bit proměnné *EnOcean_Status*. Tato proměnná je adresovatelná, aby provedení konfigurace EnOcean komponent mohla provést i k tomu určená nadřazená aplikace.

K nalezení identifikačního čísla EnOcean komponenty spínače slouží funkční blok *FbShow_ID_By_Button* rovněž z knihovny EnOcean_04.lib. Ten zobrazí z přijatého paketu identifikační číslo EnOcean komponenty spínače v případě, že je na komponentě stisknut najednou určitý počet tlačítek. Tento počet je dán vstupním parametrem *bButton_Count*, v mém programu pevně nastaveným na hodnotu „2“. Tedy při současném stisku dvou tlačítek na jedné komponentě EnOcean, se na výstupu funkčního bloku zobrazí její identifikační číslo.

Pro nalezení čísla tlačítka už jsem musel být rafinovanější. Pro tuto záležitost již neexistuje žádný funkční blok. Z manuálů jsem věděl, že s více jak osmi tlačítky se EnOcean spínače nevyrábí. Proto jsem vytvořil pole s osmi instancemi funkčního bloku *FbButtonSelectChannel*. Na vstupní parametry *dwID* jsem přivedl nalezené identifikační číslo a na parametry *bButton* postupně čísla od jedné do osmi. Při stisku už pouze jednoho tlačítka na dané komponentě se na výstupu patřičné instance objeví logická jednička a v proměnné *EnOcean_FindedNoButton* se objeví číslo instance, nebo-li tlačítka.

Nadřazená aplikace, která bude provádět konfiguraci, bude fungovat následovně. Nejdříve nastaví 0.bit proměnné *EnOcean_Status* na logickou jedničku. Tím se spustí vykonávání programu *EnOcean_Konfigurace*. Zažádá uživatele o současný stisk dvou tlačítek na jednom spínači EnOcean. Poté požádá o stisk pouze jednoho tlačítka. Tím se program dozví identifikační číslo komponenty včetně čísla tlačítka a může tyto hodnoty uložit na patřičná místa v polích *EnOcean_PoleID* a *EnOcean_PoleCiselTlac*. Tím je k danému vstupu přiřazeno konkrétní tlačítko spínače EnOcean.

4.14 KNX

4.14.1 Důvody nezavedení sběrnice KNX do aplikace

Dle zadání jsem se snažil v aplikaci využít i sběrnici KNX. Ovšem při bližším studiu jsem se přesvědčil, že využití této sběrnice v mé aplikaci je nesmyslné. Pokusím se zde uvést důvody.

Komponenty na bázi sběrnice KNX se programují v jednotném prostředí pojmenovaném ETS3. Každá komponenta má již své kanály, které se spojují do skupin a vytvářejí tak spolu vazby. Karta KNX automatu WAGO má výhodu, že si své proměnné můžeme dle potřeb nejdříve naprogramovat. Pokud tedy připojujeme PLC WAGO do systému domovní automatizace se sběrnicí KNX, měli bychom vědět předem, jaké vazby budeme chtít tvořit a podle toho naprogramovat dané komunikační objekty. To znamená v aplikaci automatu WAGO v prostředí CoDeSys vytvořit definovaný počet funkčních bloků daných typů, vyexportovat sys-xml soubor, a naimportovat jej do prostředí ETS3. V tomto prostředí posléze vytvořit skupinové adresy dle žádaných vazeb a nahrát tyto skupiny do všech zainteresovaných komponent. Takto vypadá praktické využití sběrnice KNX s automatem WAGO.

Koncepce mého programu říká, že vazby komponent by měly být kompletně nakonfigurovány pouhým nahráním nadřazeného datového modelu a ne dalším nastavováním v prostředí ETS3. To znamená, že by nadřazená aplikace driveru musela buď umět přistoupit a přeprogramovat všechna zařízení na sběrnici KNX, nebo by existovalo obecné nastavení sběrnice KNX a veškeré vazby by obsluhoval automat. To by vedlo k degradaci principu decentralizovaného řízení a veškerá energie a prostředky k dosažení tohoto cíle by ztratily význam. Komponenty KNX by obsahovali logiku pro řízení, kterou by vůbec nevyužívali a stali se tak pouze pasivními členy domovní automatizace. Ale oproti klasickým pasivním prvkům používaných automaty WAGO by byly několikanásobně dražší. Jedinou výhodou by snad bylo, že by od automatu nemuseli vést vodiče ke každé komponentě zvlášť, ale všechny by byly spojené jednou sběrnicí.

Po prostudování literatury [17] a manuálu knihovny KNX_Standart.lib [13] jsem si uvědomil další velký problém. Pro obecnou aplikaci bych musel vzít v úvahu velké množství datových typů. Například jen jednobitové proměnné mají 14 různých datových typů. Tabulka 4.9 uvádí přehled všech datových typů jednobitových proměnných. Abych vytvořil obecná spojení pro každý datový typ, stálo by to značné paměťové prostředky. A vzhledem k předchozím argumentům by přitom zcela postrádaly smysl.

ID	Název
1001	DPT_Switch
1002	DPT_Bool
1003	DPT_Enable
1004	DPT_Ramp
1005	DPT_Alarm
1006	DPT_BinaryValue
1007	DPT_Step
1008	DPT_UpDown
1009	DPT_OpenClose
1010	DPT_Start
1011	DPT_State
1012	DPT_Invert
1013	DPT_DimDendStyle
1014	DPT_InputSource

Tabulka 4.9 – Datové typy jednobitových proměnných

4.14.2 Realizace řízení KNX na velkém modelu

Velký model domovní automatizace, který jsem měl pro vývoj aplikace k dispozici, obsahuje množství prvků na KNX sběrnici. Nechtěl jsem připustit, aby množství komponent na panelu bylo při prezentaci mé aplikace nefunkčních, a tak jsem do aplikace automatu přidal program pro výměnu dat s KNX. Aplikace je ovšem připravena „na míru“ pro konkrétní model a jednotlivé komponenty jsem degradoval na pouhé pasivní členy.

V modelu je osm tlačítek, jeden vypínač, dva termostaty a žaluzie. Vytvořil jsem odpovídající funkční bloky a nechal vyexportovat patřičný sys-xml soubor. Tento soubor jsem naimportoval do prostředí ETS3. Zde jsem sestavil skupinové adresy, které s komponentami na sběrnici vytvořili dvojice vysílač-přijímač. Přečtené signály ze senzorů jsem spojil s konkrétními adresami na vstupních polích. Tím jsem do aplikace přidal více tlačítek, které mohou vytvářet události stisknutí, dvojitého stisknutí nebo držení. Z obou KNX termostatů jsem využil hodnotu aktuální teploty jako komponentu teplotního čidla a hodnotu požadované teploty jako komponentu manuálního vstupu požadované teploty.

Pouze žaluzie mají využití jako aktuátor. Propojil jsem tedy vstupní parametry daných funkčních bloků s adresami konkrétních digitálních výstupů. Komponenta žaluzie připojená na sběrnici KNX se řídí pomocí dvou komunikačních objektů. Jedním se ovládá pohyb žaluzií mezi hraničními body a druhým objektem se ovládá pohyb žaluzií o jeden krok. Tedy pro ovládání žaluzií jsem potřeboval dvě výstupní adresy. Navrhnul jsem tedy dvě samostatné komponenty digitálního výstupu.

První komponenta označená konstantou *ZALM_typ* řídí pohyb žaluzií mezi hraničními body. Použil jsem pro ni funkční blok *DO_KOMPONENTA*, kde zapnuto znamená dolní polohu, neboť zabudované žaluzie KNX využívají logickou jedničku pro směr dolů. Pro vyhodnocení událostí příslušejících dané komponentě budou klasicky využity tabulky výstupních komponent a funkční blok *VYHODNOCENI_UDALOSTI*.

Druhá komponenta označená konstantou *ZALS_typ* řídí pohyb žaluzií o jeden krok. Je pro ni využit také funkční blok *DO_KOMPONENTA*, kde zapnuto znamená krok směrem dolů. Pro vyhodnocení událostí příslušejících dané komponentě jsou také klasicky využity tabulky výstupních komponent. Ovšem oproti předchozí komponentě, výstupní hodnota značí pouze žádaný směr pro pohyb žaluzií o jeden krok. U této komponenty je třeba zajistit signál, který při opakované události žádající krok stejným směrem vyšle nový paket přes sběrnici KNX, neboť každý paket přiměje žaluzie k jednomu kroku.

Program pojmenovaný *RIZENI_ZALUZII* obsahuje pouze algoritmus pro vyhodnocení požadovaného stavu žaluzií. Teprve program *KNX* obsahuje funkční bloky sloužící k přijímání nebo vysílání paketů po sběrnice KNX. A jak jsem výše napsal, přijaté signály se kopírují na konkrétní adresy vstupního pole a výstupní telegramy posílají hodnoty z konkrétních výstupů. Program *KNX* je volán z procesu *SpravaVstupuVystupu*.

Kapitola 5

Úprava datového modelu

5.1 Popis problému

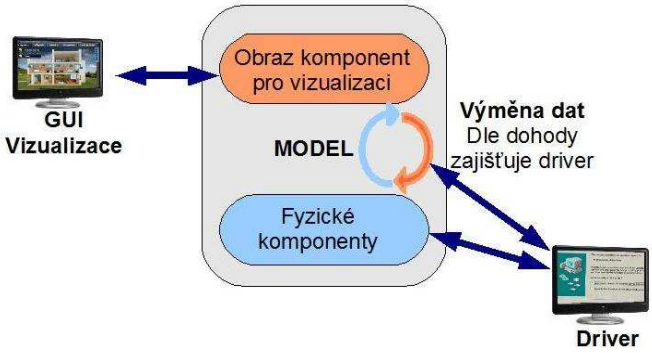
Mým úkolem bylo upravit stávající datový model systému DAMIC, aby více odpovídal způsobu práce s daty používanými v aplikaci automatu WAGO. Do svých úprav jsem zařadil i poznatky a návrhy Ing. Ondřeje Fialy, se kterým jsem změny konzultoval.

Protože je třeba strukturu systému za běhu vhodně udržovat v paměti a ukládat data ve formátu použitelném i jinými softwary, je model navržen jako XML Schéma. Ve své práci jsem musel poupravit základní strukturu schématu podle nových požadavků popsaných v následující kapitole a tím i následně změnit adresaci, která se používá pro dotazování se serveru na data v modelu. Na závěr jsem musel v datovém modelu vytvořit proměnné tak, abych v nich uchoval všechna potřebná data pro správné nastavení aplikace WAGO.

5.2 Požadavky na model

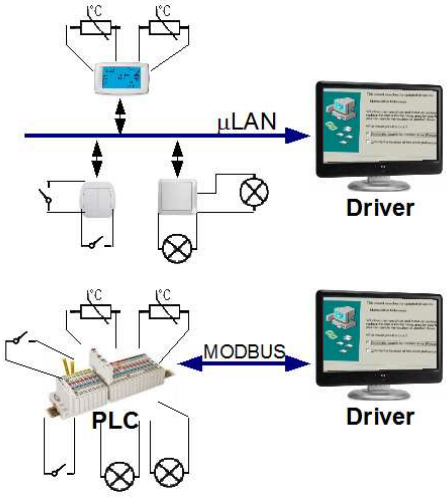
5.2.1 Rozdělení fyzické a vizuální části modelu

Data v modelu bylo třeba rozdělit na část fyzických komponent, se kterou budou pracovat klienti typu driver, a část vizuálních komponent, se kterou budou pracovat klienti grafického rozhraní. Důvodem je univerzálnost pro grafické uživatelské programy, které jednotně zobrazí komponenty na monitoru. Obrázek 5.1 ukazuje schéma takového rozdělení. O propojení dat z těchto dvou oblastí se podle domluvy stará driver.



Obrázek 5.1 – Schéma rozdělení modelu

Komponentou rozumíme konkrétní akční výstup či vstup do systému (tedy konkrétní světlo, ventilátor, tlačítko, čidlo, atd.), které má svou hodnotu a parametry. Umístění ve fyzické části odpovídá reálnému umístění komponent v zařízeních a reálnému umístění zařízení v sítích. Ve vizuální části jsou všechny vizuální komponenty pro grafické uživatelské rozhraní v jedné skupině. Tyto vizuální komponenty mají jednotný vzor a mají odkaz na fyzické komponenty, se kterými jsou spjati.



Obrázek 5.2 – Struktura fyzického rozmístění komponent

Pro jasné pochopení umístění fyzických komponent ve struktuře modelu je třeba se podívat na obrázek 5.2. Fyzickými komponentami zamýšlíme konkrétní světla, tlačítka a čidla (na obrázku označeny schématickými značkami). Ta jsou připojena k zařízením. Zařízením rozumíme „krabičku“, která fyzické komponenty ovládá. Může jím být elektronická karta, která sdružuje více světel či tlačítek, nebo také řídící jednotka PLC. Zařízení jsou připojena k síti, skrze kterou komunikují s driverem a případně i mezi sebou. Proto je základním elementem síť, která sdružuje zařízení, a jednotlivá zařízení obsahují fyzické komponenty. V modelu se tak udržuje obraz skutečného uspořádání komponent.

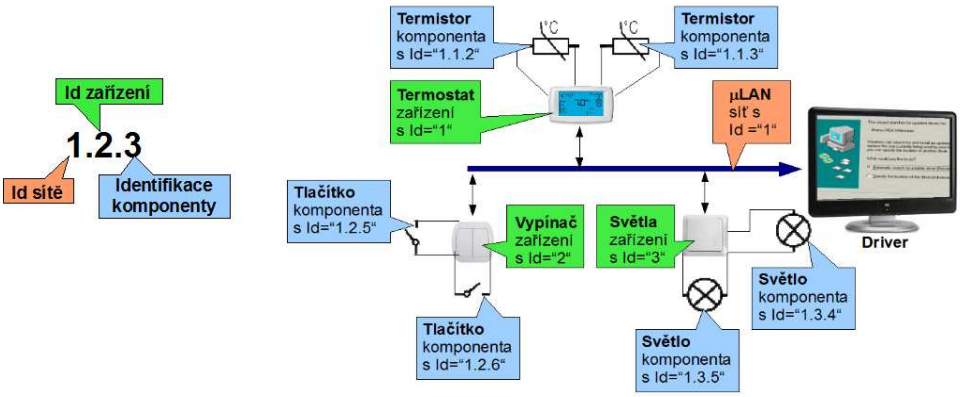
Vizuálními komponentami rozumíme spjaté komponenty s fyzickou částí modelu určených pro aplikace umožňující vizualizaci systému. Mají jednotný tvar a své obecné typy nezávislé na konkrétní realizaci (např. světlo, tlačítko, vypínač, termostat,…). Tím bychom měli docílit univerzálního uživatelského ovládání, kde například všechna světla mají stejnou ikonku a ovládání.

5.2.2 Vazby mezi komponentami

V modelu musí být uchovávány informace o všech vazbách mezi komponentami. Vazbou je myšleno například, které tlačítko způsobí rozsvícení daného světla. Navíc musí být jednoznačně řečeno, jaká událost na tlačítku (např. krátký stisk, dlouhý stisk, dvojklik, atd.) vyvolá rozsvícení světla. Tedy vazby musí propojit vyvolanou událost u dané komponenty s určitou akcí u druhé komponenty.

5.2.3 Adresování v modelu

Každý element ve schématu musí obsahovat atribut *Id* s jednoznačným identifikačním číslem v rámci své nadřazené skupiny a ve schématu se adresuje jen pomocí těchto *Id*. Pro generování nových *Id* musí server klientům poskytovat patřičnou funkci.



Obrázek 5.3 – Tvorba identifikačního řetězce pro fyzické komponenty

Odlišné značení komponent se projevuje ve fyzické části modelu. Jelikož se na několika různých místech v modelu objevují reference na konkrétní fyzické komponenty, bylo třeba zajistit jednoznačnou identifikaci komponenty bez ohledu na její konkrétní umístění v zařízení a síti (viz kapitola 5.2.1 Rozdělení fyzické a vizuální části modelu). Proto bylo rozhodnuto, že identifikátor nebude číslo, ale řetězec čísel, který v sobě zahrne identifikační číslo sítě i zařízení. Pro oddělení čísel byla zvolena tečková notace. První číslo v řetězci bude značit Id patřičné sítě, poté bude následovat tečkou oddělené druhé číslo značící Id patřičného zařízení. Pak bude opět následovat tečka a třetím a zároveň posledním číslem v řetězci určíme jednoznačně komponentu v zařízení. Tím zajistíme jednoznačnou identifikaci fyzických komponent v modelu. Na obrázku 5.3 je uveden příklad sestrojení identifikačního řetězce pro fyzickou komponentu.

5.2.4 Předobraz fyzických komponent

Dalším požadavkem bylo, aby všechny fyzické komponenty měly svůj předobraz ve společné sekci modelu. Jak později uvidíme, tato sekce se jmenuje *DataTypes* a obsahuje vzory všech použitelných fyzických komponent s implicitními hodnotami atributů. Program driveru při vkládání nových komponent vybere patřičný vzor komponenty, nastaví správné atributy a vloží do patřičného místa v modelu. Tím se zamezí, aby komponenty stejného typu měly rozdílné struktury a zároveň, aby driver nemusel generovat celou jejich strukturu. Navíc to umožní zadávat stejné implicitní hodnoty atributů.

5.2.5 Akce a události:

Komponenty mohou generovat události nebo vykonávat akce. Seznam podporovaných událostí nebo akcí musí mít každá komponenta vypsané ve svých vlastnostech. Po rozvaze bylo zvoleno, že seznam bude ve formátu textového řetězce obsahující pojmenování událostí či akcí oddělených znakem „|" („pipe").

Vazby mezi komponentami nejsou dány jenom zdrojovou a cílovou komponentou, ale také událostí na zdrojové komponentě a akcí na cílové komponentě, která má být událostí vyvolána. Ukazatelem na události či akce je pořadové číslo v řetězci. Důvodem je nezávislost na použitém jazyku v projektu. Při změně jazyka stačí jen přeložit řetězec, veškeré vazby zůstanou zachovány.

Řetězec podporovaných událostí či akcí je překopírován do spjaté vizuální komponenty. Aplikace pro vizualizaci tak mohou z tohoto řetězce vytvořit seznam a nabídnout jej uživateli. Uživatel tak má možnost vzdáleně vyvolávat události či akce na reálných komponentách. Důvod tohoto předávání podporovaných událostí a akcí je, že může nastat případ, kdy typ vizuální komponenty z jednoho driveru nemusí podporovat vyspělejší akce stejného typu z druhého driveru. Tedy například světlo z jednoho driveru nebude podporovat stejné akce jako světlo z vyspělejšího driveru. Navíc při vložení nového typu vizuální komponenty nemusíme nutně přeprogramovat aplikaci vizualizace. Ta zobrazí defaultní symbol pro neznámou komponentu a pomocí kontextové nabídky může stále posílat příkazy.

5.2.6 Ostatní požadavky

Model měl dále obsahovat prostor pro ukládání dat jednotlivých driverů.

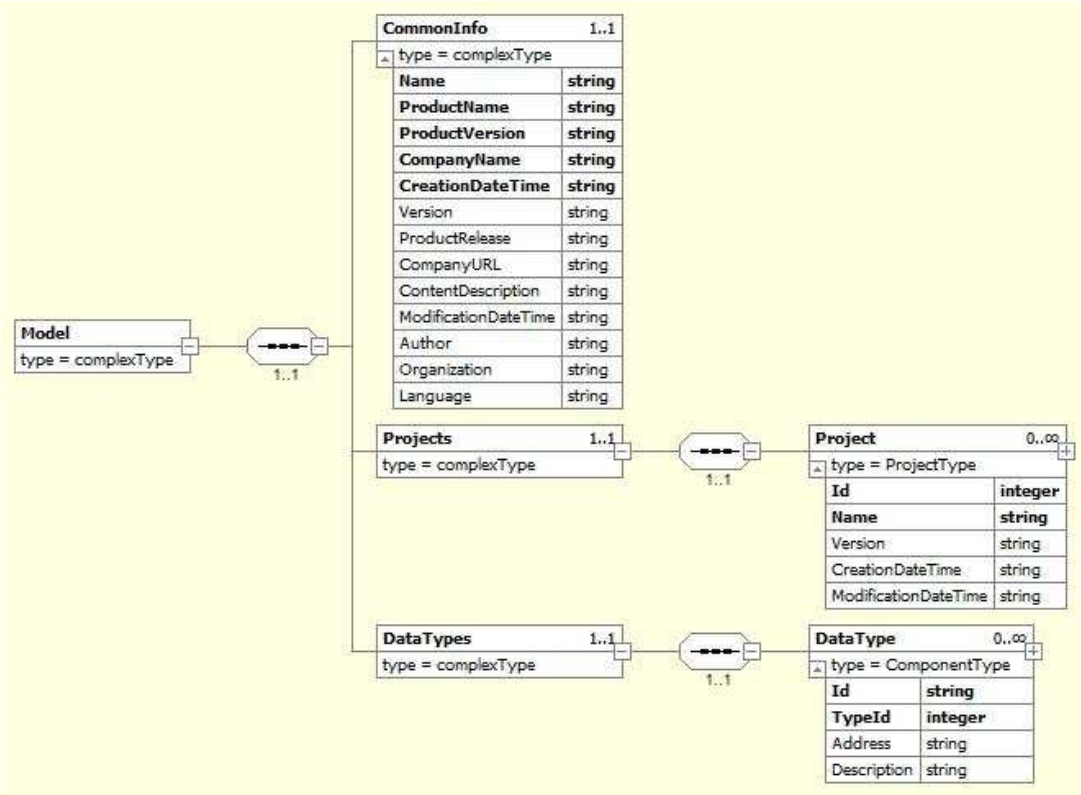
Komponenty mělo být možné filtrovat do skupin, například podle typu či společného umístění (např. obývací pokoj, atd.).

V modelu musí být spravovány i obrazovky pro vizualizaci. Každá obrazovka může obsahovat několik stránek s různým obrázkem a každá stránka si musí pamatovat, které vizuální komponenty k ní patří a kde jsou na ní rozmístěny.

5.3 Realizace modelu

5.3.1 Základní elementy modelu

Model je navržen jako XML schéma, kde se kořenový element jmenuje *„Model"*. Model obsahuje tři základní povinné elementy: *CommonInfo, Projects* a *DataTypes*.



Obrázek 5.4 – Základní struktura modelu

Sekce *CommonInfo*, jak název napovídá, ukládá obecné informace o modelu. Je zde hlavně uložen název modelu s verzí, jméno společnosti a čas vytvoření. Dále jsou zde uloženy například údaje o autorovi, jazyku, atd.

Sekce Projects je nejdůležitější částí modelu. Zde jsou uloženy jednotlivé projekty domovní automatizace, jejichž počet je neomezen. Projekt by měl být vázán na bytovou jednotku (byt nebo rodinný dům). Tím je možné pomocí serveru s jedním modelem vzdáleně spravovat i několik bytů či domů.

Sekce *DataTypes* obsahuje předobrazy všech typů komponent použitelných v projektech. Z těchto předobrazů se v projektech vytváří nové fyzické komponenty, které už mají vazbu na konkrétní reálnou komponentu v budově. V této sekci jsou instance typu *ComponentType*.

5.3.2 Sekce *Projects*

Tato sekce je určena pro elementy typu *ProjectType*, ve kterých jsou uložena veškerá data a nastavení pro správné fungování a vizualizaci domovní automatizace. Tabulka 5.1 přibližuje strukturu tohoto typu.

Atribut	Povinný	Popis
Id	ANO	Unikátní číslo projektu pro adresaci, jednoznačné v sekci Projects.
Name	ANO	Jméno projektu.
Version	ANO	Verze projektu.
CreationDateTime	NE	Datum a čas vytvoření projektu.
ModificationDateTime	NE	Datum a čas poslední změny projektu.
Element	**Povinný**	**Popis**
Drivers	ANO	*Sekce, která obsahuje informace o driverech používaných v projektu včetně jejich proměnných a dat potřebných pro konfiguraci a ovládání. Obsahuje instance typu DriverType.*
RealNets	ANO	*V této sekci jsou obrazy fyzických sítí. Na tyto sítě jsou připojena jednotlivá fyzická zařízení. Sekce obsahuje instance typu RealNetType.*
Relations	ANO	*Sekce obsahuje informace o vazbách mezi fyzickými komponentami. Vazby jsou definovány pomocí událostí řídících členů a akcí akčních členů. Instance jsou typu RelationType.*
Filtrations	ANO	*Sekce obsahující vizuální komponenty seskupené do skupin podle určitého společného znaku. Instance jsou typu FiltrationType.*
Visualitions	ANO	*Sekce obsahující informace o obrazovkách pro vizualizaci. Každá obrazovka může mít několik stránek. Instance jsou typu VisualisationType.*
GlobalVariables	ANO	*Sekce obsahuje definice globálních proměnných. Instance jsou typu VariableType.*
VirtualComponents	ANO	*Sekce obsahuje virtuální komponenty v projektu, které nejsou vázány na konkrétní zařízení. Virtuálními komponentami mohou být topné plány, časové plány, atd. Instance jsou typu ComponentType.*
VisualComponents	ANO	*Sekce obsahuje jednotný obraz komponent pro vizualizaci. S touto sekcí pracují klienti typu vizualizace, kteří zajišťují jednotné zobrazení a chování komponent stejného typu. Instance jsou typu VisualComponentType.*

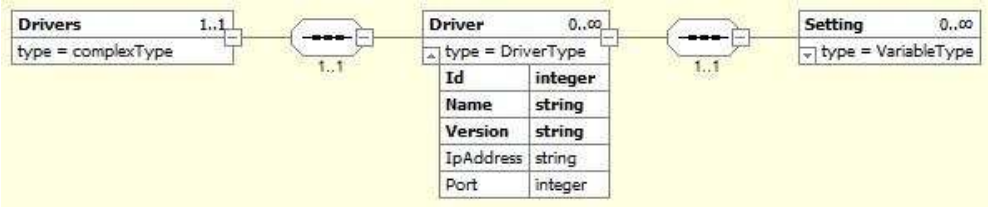
Tabulka 5.1 – Popis typu ProjectType

Dále popíšu jen nejdůležitější elementy potřebné pro svou práci. Podrobný popis datového modelu je uveden v příloze A.

Elementy typu *ProjectType*:

DriverType

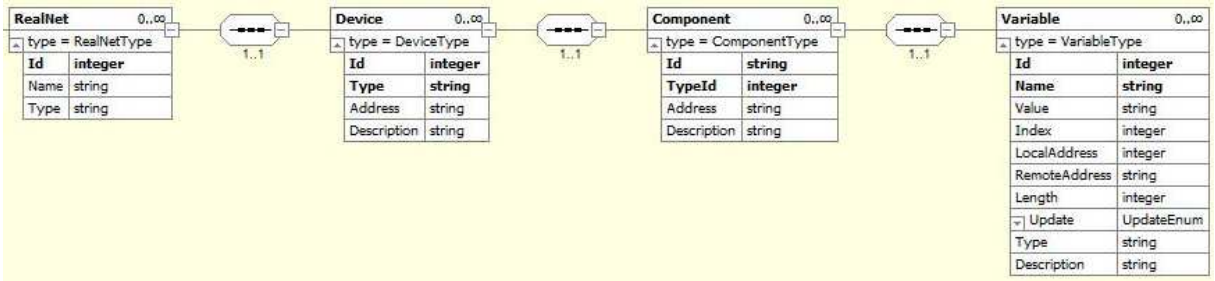
Sekce *Drivers* slouží pro uložení informací použitých driverů. Drivery si zde mohou ukládat své proměnné a data potřebná pro konfiguraci a ovládání systému. Elementy *Setting* v této sekci jsou typu *VariableType*.



Obrázek 5.5 – Struktura elementu typu *DriverType*

RealNetType

Sekce *RealNets* obsahuje obraz fyzických zařízení a jejich komponent. Struktura sekce je navržena tak, aby dávala jasnou představu o reálném zapojení komponent v systému. Síť je reprezentována elementem *RealNet*, zařízení elementem *Device* a komponenta elementem *Component*.



Obrázek 5.6 – Struktura elementu typu *RealNetType*

ComponentType

Elementy *Component* reprezentují reálné komponenty systému, včetně informací o jejich aktuálním stavu. *Component* obsahuje neomezený počet elementů typu *VariableType*, které reprezentují proměnné náležící komponentě.

Každá komponenta by měla mít proměnnou *Value* reprezentující její aktuální hodnotu a proměnnou *Behaviours*, která bude obsahovat řetězec pojmenovaných událostí či akcí oddělených znakem „|“ („pipe“). Na tento řetězec je odkazováno ze sekce Relations a je kopírován do sekce *VisualComponents*, do elementů *VisualComponent*, atributu *Behaviours*.

```
- <RealNet Id="1" Name="ModbusL1" Type="Modbus">
  - <Device Id="1" Type="Wago" Address="192.168.100.10" Description="PLC Wago na IP adrese 192.168.100.10">
    - <Component Id="1.1.1" TypeId="8" Address="3" Description="Topeni v loznici">
        <Variable Id="1" Name="Value" Value="False" Type="Bool" Description="Hodnota vystupu" />
        <Variable Id="2" Name="Behaviours" Value="otevrit/zavrit" Type="String" Description="Udalosti pro komponentu ventil" />
        <Variable Id="3" Name="TypeOfRegulation" Value="1" Type="String" Description="Typ regulace (1=PID,2=Hystereze)" />
        <Variable Id="4" Name="TypeOfAddress" Value="DO" Type="String" Description="Typ adresy" />
        <Variable Id="5" Name="Negation" Value="True" Type="String" Description="Negace ovladaci hodnoty" />
      </Component>
```

Obrázek 5.7 – Příklad instance typu *ComponentType*

VariableType

Element pro uložení proměnné. Objevuje se v elementech *Driver*, *Component*, *GlobalVariable* a *DataType*.

Atribut	Povinný	Popis
Id	ANO	*Unikátní identifikační číslo proměnné jednoznačné v daném elementu, v němž jsou umístěni.*
Name	ANO	Jméno proměnné.
Value	NE	Hodnota proměnné.
Index	NE	Index proměnné.
LocalAddress	NE	Lokální adresa proměnné.
RemoteAddress	NE	Vzdálená adresa proměnné.
Lenght	NE	Velikost proměnné.
Update	NE	Možnost změny proměnné.
Type	NE	Typ proměnné.
Description	NE	Bližší popis proměnné.

Tabulka 5.2 – Popis typu *VariableType*

RelationType

V sekci *Relations* jsou uloženy veškeré informace o vazbách mezi komponentami. Element *Relation* reprezentuje jednu vazbu mezi komponentami. Vazba není dána jen zdrojovou a cílovou komponentou, ale také událostí na zdrojové komponentě a akcí na cílové komponentě, která má být událostí vyvolána.

Atribut	Povinný	Popis
Id	ANO	Unikátní identifikační číslo jednoznačné v sekci *Relations*
From	ANO	Zdrojová komponenta. Obsahuje Id fyzické komponenty, která bude vyvolávat událost.
To	ANO	Cílová komponenta. Obsahuje Id fyzické komponenty, která bude vykonávat akci.
FromEvent	ANO	Pořadové číslo zdrojové události v řetězci proměnné *Behaviours*.
ToAction	ANO	Pořadové číslo cílové akce v řetězci proměnné *Behaviours*.
Name	NE	Jméno vazby.

Tabulka 5.3 – Popis typu *RelationType*

```
<Relation Id="21" From="1.1.22" To="1.1.3" FromEvent="1" ToAction="1"
 Name="Tlacitko leve v detskem pokoji:kliknout->Impulsni svetlo v detskem
 pokoji:rozsvitit" />
```

Obrázek 5.8 – Příklad instance typu *RelationType*

VisualComponentType

Jak bylo napsáno v požadavcích na model, vizuální část komponent má být oddělená a nezávislá na struktuře fyzických komponenent. Program pro uživatelskou vizualizaci domovní automatizace pracuje pouze v části modelu, kde všechny komponenty mají stejný vzor *VisualComponent*.

Atribut	Povinný	Popis
Id	ANO	Identifikující číslo vizuální komponenty jednoznačné v sekci *VisualComponents*
Driver_id	ANO	Odkaz na identifikátor driveru, který bude vizuální komponentu obsluhovat. Obsluhou je myšlena výměna dat mezi vizuální a fyzickou komponentou.
Real_id	ANO	Odkaz na identifikátor fyzické komponenty v sekci *RealNets*, se kterou je vizuální komponenta spjata.
Type	ANO	Typ vizuální komponenty (např. světlo,tlačítko,…). Je doporučeno zde volit domluvené jednotné anglické názvy (např. light, button,…) shodné pro všechny vizuální aplikace
Behaviours	ANO	Seznam událostí či akcí podporovaných fyzickou komponentou oddělené znakem '|' („pipe"). Tento řetězec by měl být napsán v jazyce používaném v projektu, neboť z tohoto řetězce si vizualizační aplikace vytvoří seznam akcí a zobrazí jej jako nabídku uživateli.
Status	NE	Status vizuální komponenty.
Value	NE	Hodnota vizuální komponenty (string z důvodu obecnosti). Zapisuje ji driver, vizualizace čte.
Behaviour	NE	Požadavek na vyvolání události nebo provedení akce u spjaté fyzické komponenty vyvolané zásahem uživatele. Zapisuje ji vizualizace, driver ji čte a maže.
Description	NE	Bližší popis vizuální komponenty.
VisualName	NE	Jméno vizuální komponenty, které bude zobrazováno v aplikaci pro vizualizaci.

Tabulka 5.4 – Popis typu *VisualComponentType*

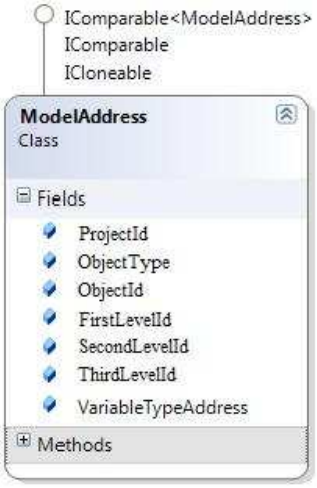
5.4 Adresování dat v modelu

5.4.1 SharedModel

Instance třídy *SharedModel* je sdílený objekt umístěný na serveru. Konfigurace serveru je nastavena tak, že instance této třídy se vytvoří při prvním připojení klienta k serveru a poté všichni další klienti přistupují k tomuto objektu. Třída *SharedModel* implementuje rozhraní *ISharedModel*, přes které přistupují všichni klienti.

5.4.2 Adresování

K adresování jednotlivých částí v třídě *Model* je určená třída *ModelAddress* (viz obrázek 5.9). Správně poskládaná adresa se předává metodám poskytovaných serverem a ty vrácí požadovaná data.



Obrázek 5.9 – Třída *ModelAddress*

Postup vytvoření adresy je následující:

1. Nejprve zvolíme požadovaný návratový typ – vlastnost *ObjectType*. Pro pole i jednu položku je návratový typ stejný. Návratové typy volíme z výběrového seznamu *ModelTypesEnum*.
2. Pokud načítáme jiný typ než *CommonInfo* nebo *DataType*, vyplníme identifikační číslo projektu – *ProjectId*.
3. Vyplníme identifikační čísla všech objektů v řadě následujícím způsobem: *ObjectId* nastavíme na ID hledaného objektu a jestliže má nadřazený jiný objekt, nastavíme proměnné *FirstLeveId*, *SecondLevelId* a případně i *ThirdLevelId* na identifikační čísla nadřazených objektů.
4. Pokud je *ObjectType* nastaveno na *VariableType*, je nutné ještě nastavit typ proměnné *VariableTypeAddress* určující umístění proměnné. Tedy zda se zajímáme o proměnnou driveru, komponenty nebo globální komponenty. Typ proměnné volíme z výběrového seznamu *VariableTypeElementEnum*.
5. Pokud načítáme komponentu ze sekce *DataType*, adresujeme dle atributu *TypeId*.

Obrázek A.21 v příloze A popisuje postup vytváření adresy včetně příkladů. Začneme od prvku *Model* a po čáře pokračujeme až ke konečnému prvku. Do adresy zapíšeme všechny položky uvedené v okénku nad prvkem, který jsme prošli. Pro konečný prvek uvedeme položky pouze ve spodním okénku. Pokud nás zajímá konkrétní objekt, musíme přidat ještě položku ObjectId.

5.5 Nejdůležitější metody poskytované serverem

5.5.1 Metody *GetObject*, *GetObjectList* a *GetObjectListByAttr*

```
public object GetObject(ModelAddress Address, bool plain)
public object GetObjectList(ModelAddress Address, bool plain)
public object GetObjectListByAttr(ModelAddress Address, string property, string value, bool plain)
```

Obrázek 5.10 – Metody *GetObject, GetObjectList* a *GetObjectListByAttr*

Metoda *GetObject* vrací z modelu dle nastavené adresy jeden konkrétní objekt. Metoda *GetObjectList* vrací z modelu dle nastavené adresy seznam objektů. Metoda *GetObjectListByAttr* vrací z modelu seznam objektů dle nastavené adresy a požadované hodnoty atributu. Parametr *property* slouží pro zadání jména atributu a parametr *value* slouží pro zadání hledané hodnoty atributu. Obrázek 5.10 znázorňuje definice metod. Parametr *plain* značí, zda chceme objekty bez jejich podelementů či nikoli. Hodnota „TRUE“ vrací objekty bez podelementů.

5.5.2 Metoda *GetAddressByComponentId*

```
public ModelAddress GetAddressByComponentId(string projectId, string componentId)
```

Obrázek 5.11 – Metoda *GetAddressByComponentId*

Jelikož se na několika různých místech v modelu odkazuje na fyzickou komponentu dle jejího identifikačního řetězce, slouží tato metoda pro transformaci identifikačního řetězce na adresu fyzické komponenty. Parametr *projectId* slouží pro zadání identifikačního čísla projektu, aby adresa byla úplná. Parametr *componentId* je určen pro zadání identifikačního řetězce žádané komponenty. Návratovým typem metody je úplná adresa fyzické komponenty. Obrázek 5.11 znázorňuje definici metody.

5.5.3 Metoda *SetObject*

```
public bool SetObject(ModelAddress Address, object modelObject)

public bool SetObject(ModelAddress Address, object modelObject, out string newId)
```

Obrázek 5.12 – Přetížená metoda *SetObject*

Server poskytuje přetíženou metodu *SetObject*, která buď přidá nový objekt do modelu, nebo přepíše starý objekt novými daty (update objektu).

Přepsání starého objektu:

```
public bool SetObject(ModelAddress Address, object modelObject)
```

Obrázek 5.13 – Přetížená metoda *SetObject* pro přepsání starého objektu

Metoda *SetObject* se dvěmi vstupními parametry vrací booleovskou hodnotu, zda se podařilo objekt v modelu přepsat. Obrázek 5.13 znázorňuje definici metody. Parametr *Address* slouží pro adresování cílového objektu, parametr *modelObject* je komponenta s novými daty.

Přidání nového objektu:

```
public bool SetObject(ModelAddress Address, object modelObject, out string newId)
```

Obrázek 5.14 – Přetížená metoda *SetObject* pro přidání nového objektu

Metoda SetObject se třemi vstupními parametry vrací booleovskou hodnotu, zda se podařilo přidat nový objekt do modelu. Obrázek 5.14 znázorňuje definici metody. Parametr *Address* slouží pro adresování cílové oblasti nové komponenty, parametr *modelObject* je vytvořená nová komponenta, která nemusí mít zadanou vlastnost *Id*. Ve výstupním parametru *newId* se objeví identifikační číslo, které server přidělil nové komponentě.

V případě vkládání komponent do sekce *Device*, vrátí metoda identifikační řetězec v tečkové notaci. V případě vkládání komponent do sekce *DataTypes* vrací metoda identifikační číslo z atributu *TypeId*.

5.5.4 Metoda *SetValue*

```
public bool SetValue(ModelAddress Address, string AttributeName, string Value)
```

Obrázek 5.15 – Metoda *SetValue*

Metoda SetValue nastaví atributu objektu zadaného v parametru *AttributeName* na hodnotu uvedenou v parametru *Value*. Objekt je adresován pomocí parametru *Address*. Metoda vrací booleovskou hodnotu, zda se podařilo u daného objektu přepsat uvedený atribut.

5.5.5 Metoda *RemoveObject*

```
public bool RemoveObject(ModelAddress Address)
```

Obrázek 5.16 – Metoda *RemoveObject*

Metoda *RemoveObject* odstraní objekt z modelu dle adresy zadané v parametru *Address*. Metoda vrací booleovskou hodnotu, zda se podařilo daný objekt odstranit.

5.6 Realizace instancí

5.6.1 Elementy *Variable*

Při realizaci nadřazené aplikace driveru jsem postupně vytvářel i jednotlivé obrazy fyzických komponent v datovém modelu. Každý typ komponenty potřeboval ukládat své potřebné proměnné pomocí elementů *Variable*. Ve výsledku se ukázalo, že existují elementy, které se vyskytují ve většině komponent. Kromě výše popsaných proměnných *Value* a *Behaviours*, se u všech komponent systému WAGO vyskytuje i proměnná *TypeOfAddress*, která upřesňuje typ adresy. Tuto proměnnou jsem přidal během realizace aplikace driveru, neboť atribut *Address* obsahuje pouze číselnou hodnotu a v aplikaci bylo potřeba rozpoznat několik typů adres používaných v automatu WAGO. Tabulka 5.5 uvádí použité typy adres.

Typ adresy	Označení adresy
Digitální vstupy	DI
Digitální výstupy	DO
Analogové vstupy	AI
Analogové výstupy	AO
Světla DALI	DALI
Skupiny DALI	DALIGROUP
Topné plány	HP
Časové plány	TP
Funkce Fantom	FANTOM
Funkce Východu a západu slunce	SP
Termostat	THERMOSTAT

Tabulka 5.5 – Typy adres použité v systému WAGO

Častá proměnná je také *Negation*, která indikuje, zda se má signál před, případně po, zpracováním řídícím algoritmem znegovat.

Při realizaci nadřazené aplikace driveru jsem se také rozhodl udělat výjimku a pro adresaci elementů *Variable* používat jména proměnných namísto identifikačních čísel. Důvodem byla možnost editovat proměnné u všech komponent bez nutnosti přeprogramovat hotové algoritmy.

5.6.2 Proměnné náležící k driveru systému WAGO

Při vytváření proměnných potřebných k uchování veškerých dat potřebných pro správné nastavení aplikace WAGO se vyskytla data, která nepříslušela k žádné komponentě, ale platila pro celou aplikaci automatu. Tyto proměnné uchovávají informace o chybových příznacích, době pro vyhodnocení událostí na vstupních signálech, maximálních a skutečných počtech vstupů, výstupů a plánů. Rozhodl jsem se tyto hodnoty uložit jako proměnné příslušející k driveru, který se v nadřazené aplikaci identifikuje pomocí názvu a IP adresy automatu. K těmto proměnným přibyly další dvě, které indikují požadavky pro přečtení nebo zápis datového modelu do automatu.

5.6.3 Zavedení řetězců u proměnných *Behaviours*

Vztahy mezi komponentami se realizují způsobem, kdy se u první komponenty vybere ze seznamu událost a u druhé komponenty se ze seznamu vybere patřičná akce. Seznam jak pro události, tak akce, je uložen jako textový řetězec v proměnné *Behaviours*. Tím vznikne dvojice událost-akce, která reprezentuje vazbu mezi těmito dvěma komponentami.

Při zavádění proměnných *Behaviours* u jednotlivých komponent jsem se rozhodl vytvořit řetězce tak, aby všechny kombinace dvojic událost-akce odpovídaly tabulkám zavedených v aplikaci WAGO a zároveň, aby všechny tabulky z aplikace WAGO byly reprezentované jednou dvojicí. Vzhledem ke struktuře tabulek v aplikaci WAGO jsem byl nucen některé události či akce spojit do jedné položky seznamu. Například aplikace WAGO umožňuje světlu být jedním tlačítkem pouze zapínáno a jiným vypínáno. Ale funkce Fantom lze nastavit pouze tak, že je jedním tlačítkem jak zapínána, tak vypínána. Proto komponenta Fantom má v proměnné *Behaviours* jen jednu položku se spojenými akcemi zapnutí i vypnutí. Dále jsem se rozhodl, že každou položku z řetězce *Behaviours* bude možné v automatu WAGO simulovat pomocí nadřazené aplikace driveru. Tabulka 5.6 uvádí přehled zavedených řetězců *Behaviours* u jednotlivých komponent. Znak „/“ váže události či akce do jedné položky a teprve znak „|“ („pipe“) rozděluje řetězec na jednotlivé položky.

Bohužel se mi nepodařilo dodržet záměr na vytvoření položek seznamu chování tak, aby všechny dvojice událost-akce odpovídaly tabulce uvnitř automatu. Kromě nekompatibilních dvojic komponent, jako například tlačítko-topný ventil, je výjimkou událost dlouhého stisku, která se dá přiřadit pouze ke změnám jasu světel, které tuto akci podporují.

Typ komponenty	Vytvořený řetězec Behaviours
Světlo	rozsvitit|zhasnout|rozsvitit se zpozdenim|zhasnout se zpozdenim
Ventilátor	rozbehnout|zastavit|rozbehnout se zpozdenim|zastavit se zpozdenim
Topný ventil	otevrit/zavrit
Kotel	zapnout/vypnout
Impulsní světlo	rozsvitit|zhasnout|rozsvitit se zpozdenim|zhasnout se zpozdenim|Zmena jasu
Tlačítko	kliknout|dvojkliknout|dlouze kliknout
Vypínač	zapnout/vypnout
Okenní kontakt	sepnout/rozepnout
PIR obvod	zapnout/vypnout
Termostat	regulace zapnout/vypnout
Manuální zadání teploty	pozadovat teplotu
DALI světlo	rozsvitit|zhasnout|rozsvitit se zpozdenim|zhasnout se zpozdenim| Zmena sceny|Zmena jasu
DALI skupina	rozsvitit|zhasnout|rozsvitit se zpozdenim|zhasnout se zpozdenim| Zmena sceny|Zmena jasu
Žaluzie	stahnout|vytahnout|stahnout se zpozdenim|vytahnout se zpozdenim
Žaluzie krokové	krok dolu|krok nahoru|krok dolu se zpozdenim|krok nahoru se zpozdenim
Fantom	Zapnout/Vypnout
Topný plán	Zapnout/Vypnout
Časový plán	Zapnout/Vypnout
Slunečný plán	Vychod|Zapad

Tabulka 5.6 – Přehled zavedených řetězců Behaviours

Kapitola 6

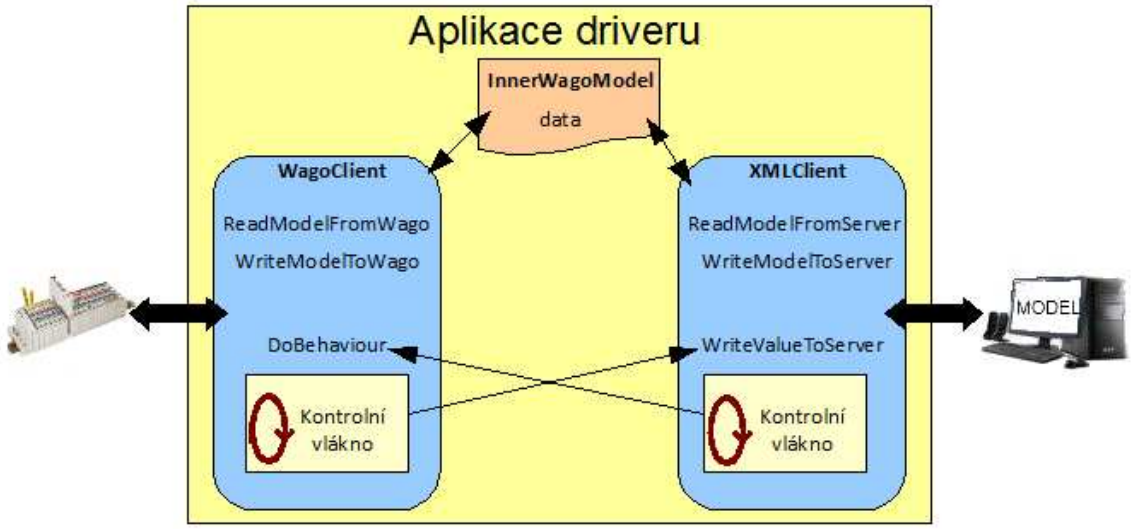
Nadřazená aplikace driver pro WAGO

6.1 Koncepce aplikace

Nadřazená aplikace driveru zajišťuje komunikaci mezi aplikací v automatu WAGO, popsanou ve 4. kapitole, a serverem systému DAMIC, který spravuje datový model domovní automatizace. Nadřazená aplikace umí přepsat nebo vyčíst model z automatu, kontrolovat změny stavů komponent automatu, které promítne do nadřazeného datového modelu, a simulovat události i akce požadované ze systému DAMIC.

Aplikaci jsem programoval v jazyce C# v prostředí Microsoft Visual Studio 2008, kde jsem své řešení připojil k multiprojektu DAMIC a využil již zde naprogramovaného klienta pro komunikaci se serverem.

Aplikaci driveru lze rozdělit na dvě části. Třída *WagoClint* reprezentuje první část a obsahuje potřebné metody pro řízení komunikace s automatem WAGO přes rozhraní MODBUS/TCP. Třída *XMLClient*, reprezentující druhou část, obsahuje metody pro řízení komunikace se serverem. Tyto dvě třídy si předávají informace o modelu pomocí další veřejné třídy *InnerWagoModel*, která obsahuje pouze proměnné a konstruktor. Změny hodnot komponent v automatu a požadavky simulací ze serveru se předávají pomocí delegátů.



Obrázek 6.1 – Koncepce aplikace driveru

Při vývoji byla aplikace nejdříve vytvářena jako klasická okenní aplikace, aby mohly být třídy *WagoClient* a *XMLClient* dobře odladěny. Posléze byly tyto třídy vloženy do aplikace naprogramované jako služba systému Windows.

Aplikace potřebuje pro správné fungování zadaná správná klíčová data, jako například IP adresu s portem automatu WAGO, adresu serveru, přístupové údaje a jméno projektu. Tyto údaje jsou uložené v souboru „App.config“ v adresáři aplikace.

6.2 Třída InnerWagoModel

Třída *InnerWagoModel* slouží v podstatě jako datová struktura pro uložení modelu. Obsahuje proměnné, jednorozměrná i dvourozměrná pole, která přesně odpovídají proměnným a polím používaných v automatu WAGO. Jsou zde uložena data o použitých komponentách, jejich stavech a vazbách s jinými komponentami. Data čtená z automatu se bez úprav ukládají do této třídy a teprve z ní se interpretují do datového modelu DAMIC. A naopak data z datového modelu se interpretují do této třídy a pak bez úprav zapisují do automatu.

Třída neobsahuje žádné metody, jen přetížený konstruktor. Konstruktor bez parametrů inicializuje všechna pole s nulovou délkou. Druhý konstruktor obsahuje parametry potřebné pro inicializování polí s odpovídající délkou. Těmito parametry jsou zadané maximální počty vstupů, výstupů a plánů. Proměnné jsou v obou případech nulové.

6.3 Rozhraní IComponent

6.3.1 Popis rozhraní

Při řízení aplikace se objevuje několik činností společných pro všechny komponenty, které potřebují být vykonány u každého typu komponenty trochu jinak. Například všechny komponenty potřebují interpretovat data ze třídy InnerWagoModel do datového modelu, ale v něm má každý typ komponenty vlastní strukturu.

Proto bylo užitečné pro každý typ komponenty vytvořit vlastní třídu, která bude implementovat rozhraní *IComponent*. V tomto rozhraní jsou definovány čtyři společné metody popsané v následujících kapitolách, které musí třída aplikovat.

6.3.2 Metoda *CreateComponentInModel*

Metoda *CreateComponentInModel* (viz obrázek 6.2) slouží k vytvoření elementu komponenty v datovém modelu na serveru. Parametr *baseClient* je odkaz na klienta, který zajišťuje komunikaci se serverem. Druhý parametr *param* je pole obecného typu object pro předávání hodnot komponent. Metoda se používá při nahrávání datového modelu na server, kde data modelu byla vyčtena z automatu. Postup aplikovaných metod je stejný. V datovém modelu se zkopíruje patřičný datový typ z oblasti *DataTypes* a jeho proměnné se vyplní předanými parametry.

```
bool CreateComponentInModel(ref BaseClient baseClient, object[] param);
```

Obrázek 6.2 – Metoda *CreateComponentInModel*

6.3.3 Metoda *CreateComponentInDevice*

Metoda *CreateComponentInDevice* (viz obrázek 6.3) slouží k sestavení komponenty v modelu automatu. Parametr *innerWagoModel* je odkaz na třídu zajišťující uložení dat modelu. Druhý parametr *param* je pole obecného typu object pro předávání hodnot komponenty. V tomto parametru je uložena hlavně komponenta z datového modelu. Metoda se používá při nahrávání datového modelu do automatu. Metoda přečte proměnné z předané instance komponenty a podle nich vyplní patřičná pole v odkazované instanci *InnerWagoModel*.

```
bool CreateComponentInDevice(ref InnerWagoModel innerWagoModel, object[] param);
```

Obrázek 6.3 – Metoda *CreateComponentInDevice*

6.3.4 Metoda *DoBehaviour*

Aplikace musí dokázat simulovat události či akce komponent v automatu. Simulace se většinou provádí zápisem hodnot do patřičné paměti automatu, v některých případech i daným sledem. Parametr *nbBehaviour* určuje událost komponenty. Značí se číslem, které odpovídá umístění operace v řetězci *Behaviours* u komponenty v datovém modelu. Parametr *mBClient* odkazuje na klienta zajišťující komunikaci s automatem. Třetí parametr je pole obecných proměnných typu object sloužící opět pro předávání potřebných hodnot.

```
bool DoBehaviour(int nbBehaviour, ref ModbusClient mBClient, object[] param);
```

Obrázek 6.4 – Metoda *DoBehaviour*

6.3.5 Metoda *GetTypeComponent*

Aby třída *WagoClient* ve své metodě *DoBehaviour*, která zajišťuje vykonání simulace, poznala, která komponenta má být simulována a tak jí dokázala vložit správné parametry, existuje metoda *GetTypeComponent*. Nemá žádné parametry a výstupem je textový řetězec s názvem komponenty. Podle tohoto řetězce aplikace identifikuje typ komponenty a předá jí tak správné parametry.

```
string GetTypeOfComponent();
```

Obrázek 6.5 – Metoda *GetTypeOfComponent*

6.4 Klient pro řízení komunikace s automatem

6.4.1 Třída ModbusClient

Pro komunikaci s automatem přes rozhraní MODBUS/TCP jsem chtěl nejdříve využít již některé veřejně dostupné naprogramované knihovny. Našel jsem knihovnu pojmenovanou NMODBUS.dll. Ale při jejím studiu jsem poznal, že nejjednodušší způsob, aby klient co nejvíce odpovídal potřebám mé aplikace, bude si klienta naprogramovat sám s použitím TCP klienta z knihovny Visual Studia.

Ve své aplikaci budu potřebovat přes rozhraní MODBUS/TCP číst i zapisovat bytová pole a nastavovat konkrétní bity v paměti automatu. Vytvořil jsem třídu *ModbusClient* a implementoval v ní tři metody, které budu pro svou aplikaci potřebovat. Tyto metody zajišťují přenos dat přes rozhraní MODBUS/TCP s patřičně vytvořenými příkazy dle protokolu.

```
class ModbusClient
public byte[] mbread(ushort referenceNumber, ushort wordCount)
public bool mbwriteMultiReg(ushort referenceNumber, ushort wordCount, byte[] register)
public bool mbwriteSingleBit(ushort ByteReferenceOffset, ushort BitReferenceNumber, bool bit)
```

Obrázek 6.6 – Třída *ModbusClient*

Protokol MODBUS adresuje v zařízení registry délky word. Proto mají ve třídě *ModbusClient* metody čtení a zápisu parametr *wordCount*, který znamená počet požadovaných „wordů“ pro tuto operaci. Je třeba také dát pozor na správnou adresu proměnných v automatu. Pro automaty WAGO 750-841 a 750-849 jsou rozsahy adres uvedeny v tabulce 3.2.

Ve své aplikaci budu potřebovat posílat rozsáhlá data, ale protokol MODBUS nedovoluje v jednom příkazu přečíst nebo zapsat více než 125 „wordů“. Metody třídy *ModbusClient* ovšem nemají žádná omezení délky datového pole a v případě potřeby si umí pole rozdělit, případně sloučit, po úsecích s délkou 125 „wordů“ a využít tak několika příkazů protokolu MODBUS.

Metoda *mbwriteSingleBit* slouží pro nastavení konkrétního bitu v paměti automatu. V parametru *ByteReferenceOffset* se zadá adresa registru, kde se nachází požadovaný bit, jehož umístění v registru je zadáno parametrem *BitReferenceNumber*. Na první bit v registru se odkazuje hodnotou nula. Číslo bitu ovšem nemusí být v intervalu 0 až 15, ale v případě většího čísla se odpovídající adresa registru dopočte. Posledním parametrem metody je požadovaná hodnota bitu.

Jelikož standard MODBUS používá pro reprezentaci dat způsob „Big-endian“, je třeba před zápisem nebo po přečtení bytového pole přehodit pořadí sudých a lichých bytů. K tomu slouží vnitřní funkce *RotateByteInWords*.

6.4.2 Třída *WagoClient*

Třída *WagoClient* obsahuje tři důležité metody. Dvě slouží pro čtení a zápis modelu do automatu, kde je u obou parametrem instance třídy *InnerWagoModel*. Třetí metoda *DoBehaviour* zajišťuje vykonání simulací událostí nebo akcí komponent v automatu. Využití této metody je popsáno v kapitole 6.6.1.

```
class WagoClient
public void WriteModelToWago(InnerWagoModel innerWagoModel)
public void ReadModelFromWago(out InnerWagoModel innerWagoModel)
public void DoBehaviour(IComponent component, int nbBehaviour, object[] param)
```

Obrázek 6.7 – Třída WagoClient

Dále instance třídy *WagoClient* spravuje vlákno, ve kterém běží funkce pro detekci změn hodnot komponent v automatu. Vlákno hlídá změny stavů vstupních i výstupních komponent, stavů virtuálních komponent jako jsou plány či funkce Fantom, reálných počtů připojených karet automatu a chybových příznaků automatu. Při detekci změny kterékoli hodnoty vlákno vyvolá funkci předanou v konstruktoru delegátem. Tato funkce má za úkol přepsat patřičnou proměnnou v datovém modelu uloženém na serveru.

Aby mohly metody třídy komunikovat s automatem, inicializuje si třída v konstruktoru globální instanci třídy *ModbusClient*, která spravuje připojení zadané IP adresou a portem. Tato IP adresa a port jsou vstupními parametry konstruktoru třídy *WagoClient*. Pokud nastane problém s připojením, aplikace se pokouší znovu připojit každých 5 sekund.

Pro správou komunikaci obsahuje třída *WagoClient* proměnné s hodnotami adres jednotlivých polí v automatu. Tyto adresy slouží pro protokol MODBUS a proto mají rozdílná čísla než adresy v automatu. Pro řídící jednotky typu 750-84x je začátek adresovatelné paměti na adrese 12288. Jak bylo psáno výše v kapitole 4.4.3, existují dva druhy adres. Konstantní pro proměnné s pevně definovanou adresou a proměnné adresy, které si třída sama vyčíslí dle přečtených hodnot maximálních počtů v automatu. Aby mohla třída *WagoClient* správně spočítat adresy, musí být pořadí adres pevně definováno. Toto pořadí je uvedeno v příloze C i s modbusovými adresami použitými v mé aplikaci. V elektronické podobě této přílohy (příloha H) jsou definovány vzorce pro přepočet adres při změně zadaných maximálních počtů.

Metoda *WriteModelToWago*

Metoda *WriteModelToWago* slouží pro zápis modelu do automatu přes rozhraní MODBUS. Pro rychlejší odezvu automatu, nastaví metoda v automatu nejdříve příznak pro blokování vykonávání řídících programů. Posléze jsou posílána pole, která se předávají přes sdílené tabulky v adresovatelné paměti automatu. Metoda zapíše pole na adresu sdílené tabulky, identifikuje typ pole ve sdílené tabulce, nastaví v automatu příznak pro čtení modelu a posléze čeká, až tento příznak automat vynuluje. To je signál, že pole bylo automatem zpracováno. Pořadí posílaných polí odpovídá pořadí při vyhodnocování sdílených tabulek v programu *MODBUS* aplikace automatu WAGO. Po odeslání dat přes sdílená pole se pošlou všechna ostatní pole, která mají svou adresu v paměti automatu a počká se opět na potvrzení o zpracování. Na závěr se vynuluje příznak pro blokování vykonávání řídících programů.

Metoda ReadModelFromWago

Metoda *ReadModelFromWago* slouží pro přečtení modelu z automatu přes rozhraní MODBUS. Pro rychlejší odezvu automatu, nastaví metoda v automatu nejdříve příznak pro blokování vykonávání řídících programů. Posléze jsou čtena pole, která se předávají přes sdílené tabulky v adresovatelné paměti automatu. Metoda nastaví typ požadovaného pole ve sdílené tabulce, zapíše do automatu příznak pro výpis modelu a posléze čeká, až tento příznak automat vynuluje. To je signál, že pole bylo automatem připraveno a metoda si přečte data ze sdílené paměti. Pořadí požadovaných polí odpovídá pořadí při vyhodnocování sdílených tabulek v programu *MODBUS* aplikace automatu WAGO. Po přečtení dat ze sdílených polí se přečtou všechna ostatní pole, která mají svou adresu v paměti automatu. Na závěr se vynuluje příznak pro blokování vykonávání řídících programů.

Při čtení sdílených tabulek pro zařízení DALI, se jako poslední požadovaný typ pole objeví hodnota 255. Ta slouží jako požadavek přečtení přiřazení světel do skupin, jenž automat čte přímo z fyzických zařízení DALI. Metoda zde čeká, až automat přečte data ze všech zařízení.

6.5 Klient pro řízení komunikace se serverem

6.5.1 *BaseClient*

Pro komunikaci se serverem aplikace používá již naprogramovaného klienta *BaseClient* z multiprojektu DAMIC. Tento klient poskytuje přístup do modelu spravovaného na serveru. Více informací o této třídě je v [2].

6.5.2 Třída *XMLClient*

Třída *XMLClient* obsahuje tři důležité metody. Dvě slouží pro čtení a zápis modelu na server, kde u obou je parametrem instance třídy *InnerWagoModel*. Třetí metoda *WriteValueToServer* zajišťuje zapsání změn stavů komponent a důležitých vlastností automatu do datového modelu. Využití této metody je popsáno v kapitole 6.6.2.

```
class XMLClient
public void WriteModelToServer(InnerWagoModel innerWagoModel)
public void ReadModelFromServer(out InnerWagoModel innerWagoModel)
public void WriteValueToServer(string type, int numberAddress, object[] value)
```

Obrázek 6.8 – Třída *XMLClient*

Dále instance třídy *XMLClient* spravuje vlákno, ve kterém běží funkce pro detekci požadavků na simulaci komponent v automatu. Tyto požadavky se objevují v datovém modelu u vizuálních komponent v atributu *Behaviour*. Při detekci požadavku na komponentě, která náleží spravovanému automatu, vlákno vyvolá metodu předanou v konstruktoru delegátem. Tato metoda má za úkol zařídit simulaci komponenty v automatu.

Použití vlákna není moc vhodné, hodilo by se spíše využít generování událostí serverem a obsloužení zaregistrované metody. Bohužel se mi nepodařilo patřičně server přeprogramovat, a tak jsem zvolil využití vlákna jako nouzové řešení.

Aby mohly metody třídy komunikovat se serverem, inicializuje se v konstruktoru globální instance třídy *BaseClient*, která poskytuje s datovým modelem spojení dle vyplněných atributů v souboru „App.config“. Při inicializaci klienta dojde také k inicializaci datového modelu pro spuštěnou aplikaci. To znamená, že se aplikace pokusí v hierarchii modelu najít svou síť a zařízení a pokud neexistují, tak je vytvoří. Pokud nastane problém s připojením, aplikace se pokouší znovu připojit každých 5 sekund.

Metoda *WriteModelToServer*

Metoda *WriteModelToServer* slouží pro transformaci a zápis modelu třídy *InnerWagoModel* do datového modelu, který je spravován na serveru. Metoda nejdříve v datovém modelu smaže veškeré komponenty a relace patřící obsluhovanému automatu.

Poté do připraveného datového modelu, na základě dat v předaném modelu třídy *InnerWagoModel*, zapíše vlastnosti driveru a vygeneruje znova všechny komponenty a jejich vazby.

Metoda ReadModelFromServer

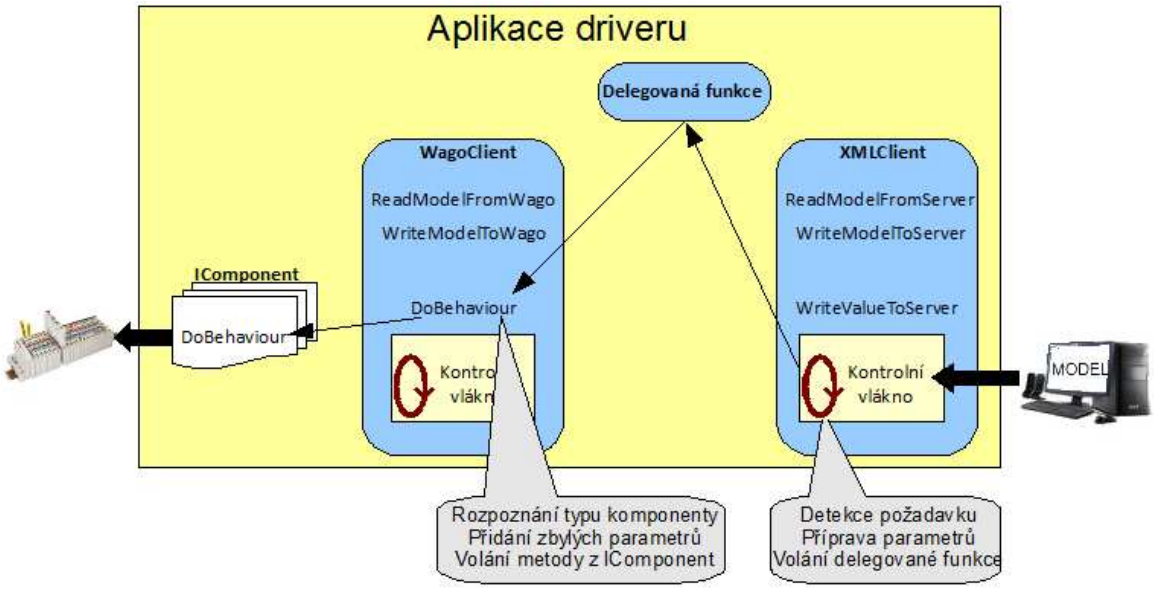
Metoda *ReadModelFromServer* slouží pro přečtení a transformování datového modelu umístěného na serveru do podoby modelu třídy *InnerWagoModel*. Metoda nejdříve z datového modelu přečte vlastnosti driveru, podle kterých vytvoří instanci třídy *InnerWagoModel* s nulovými hodnotami. Tuto instanci posléze postupně přepisuje na základě přečtených dat z datového modelu.

6.6 Komunikace

6.6.1 Postup simulace událostí nebo akcí komponent

Žádost o simulaci chování komponenty se objeví v datovém modelu u vizuální komponenty v atributu *Behaviour*. V tomto atributu se objeví číslo, které specifikuje požadovanou simulaci fyzické komponenty. Požadavky jsou detekovány v samostatném vlákně instance třídy *XMLClient*. Postup zpracování simulace je znázorněn na obrázku 6.9.

Při detekci požadavku, je volána delegovaná metoda předaná v konstruktoru třídy. Její parametry jsou předtím patřičně připraveny. Parametr *component* obsahuje dle typu komponenty, jejíž obraz ve vizuální části požaduje simulaci, odpovídající instanci třídy s implementovaným rozhraním *IComponent*. Parametr *nbBehaviour* obsahuje číslo požadované simulované operace a třetím parametrem je pole obecného typu object, které slouží pro předání ostatních parametrů potřebných k vykonání simulace. Každá komponenta vyžaduje odlišné parametry, ale většinou se jedná o adresy komponent.



Obrázek 6.9 – Postup zpracování požadavku na simulaci

```
public delegate void XMLValueChange(IComponent component, int nbBehaviour, object[] param);
```

Obrázek 6.10 – Delegát pro simulaci chování komponenty

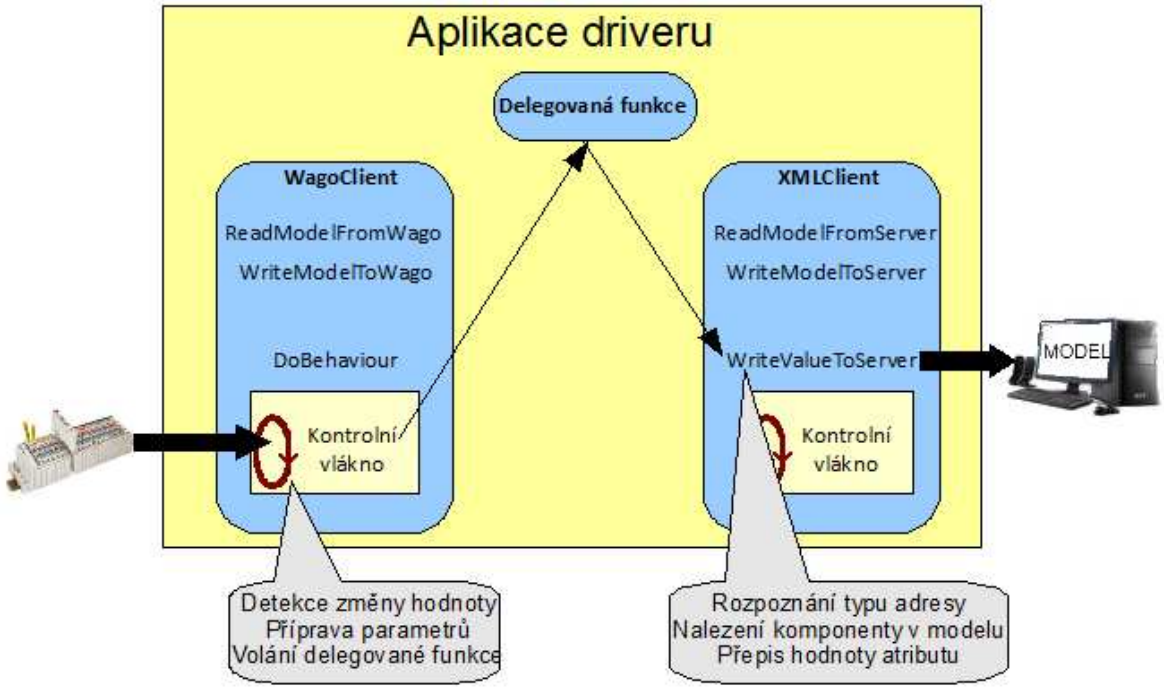
Metoda, na kterou delegát odkazuje, v sobě obsahuje jen volání metody *DoBehaviour* třídy *WagoClient* se stejnými parametry. Obrázek 6.10 znázorňuje definici tohoto delegáta. Metoda *DoBehaviour* rozpozná typ komponenty díky definované metodě *GetTypeOfComponent* a na základě vráceného typu přidá do parametru *param* další hodnoty, které jsou známy až uvnitř třídy *WagoClient*. Těmito parametry jsou převážně adresy proměnných v automatu. Posléze je volána metoda *DoBehaviour* v předané instanci komponenty, aby na základě vložených parametrů provedla simulaci chování komponenty.

Ještě je nutné sdělit, že kromě požadavků na simulaci jsou vyhodnocovány i požadavky na přenos celého modelu ze serveru do automatu a obráceně. Při zjištění takovéhoto požadavku, vlákno zavolá jinou delegovanou metodu, která vykoná patřičné operace spojené s přenosem modelu. Tedy konstruktor třídy *XMLClient* ve svých parametrech obsahuje celkem tři odkazy na metody. Jeden pro požadavky simulací, jeden pro přenos modelu z automatu na server a třetí pro přenos modelu opačným směrem.

6.6.2 Postup zápisu do modelu při změně stavu komponent

Kontrolní vlákno uvnitř třídy *WagoClient* periodicky čte stavy a hodnoty všech komponent. Pokud funkce vlákna detekuje změnu na některé komponentě, voláním delegované metody zařídí zápis nové hodnoty do datového modelu spravovaného serverem. Postup zpracování detekované změny je uveden na obrázku 6.11.

Při detekci změny hodnoty u některé komponenty vlákno zavolá delegovanou metodu předanou v konstruktoru třídy *WagoClient*. Před tímto voláním jsou patřičně připraveny parametry pro tuto metodu. Parametr *address* značí adresu komponenty, na které došlo ke změně. Pokud komponenta adresu nemá, je předáno číslo nula (např. komponenta funkce Fantom). Parametrem *type* se pomocí textového řetězce specifikuje typ adresy, protože pouze hodnotou adresy nelze přesně identifikovat komponentu (např. vstupní i výstupní komponenty mohou mít adresu číslo 5). Posledním parametrem je opět pole obecného typu object pro předání zbylých potřebných parametrů, většinou nové hodnoty komponenty.



Obrázek 6.11 – Postup zpracování změny u komponenty v automatu

```
public delegate void ValueChange(string type, int address, object[] values);
```

Obrázek 6.12 – Delegát pro přepis hodnoty komponenty v datovém modelu

Metoda, na kterou delegát odkazuje, v sobě obsahuje jen volání metody *WriteValueToServer* třídy *XMLClient* se stejnými parametry. Obrázek 6.12 znázorňuje definici tohoto delegáta. Metoda *WriteValueToServer* na základě typu a hodnoty adresy identifikuje komponentu v datovém modelu a zařídí přepis jejího patřičného atributu. Ve většině přídech se jedná o proměnnou *Value*.

Na závěr je ještě nutné připomenout, že kromě změn stavů či hodnot komponent vlákno detekuje i změny počtů momentálně připojených karet a jejich vstupních a výstupních signálů. Dále jsou detekovány i změny chybových příznaků automatu. Při zjištění změn tohoto typu je volána stejná delegovaná metoda jako při změnách komponent. Změny jsou označeny patřičným textovým řetězcem a metoda *WriteValueToServer* zapíše tyto změny do datového modelu k vlastnostem driveru.

Kapitola 7

Výsledky práce

7.1 Zhodnocení

Navržené aplikace jsem testoval na dvou modelech domovní automatizace, které jsou na katedře k dispozici.

Obrázek 7.1 – Modely domovní automatizace

Aplikace pro automaty WAGO byla do obou modelů nahrána bez nastavené hardwarové konfigurace. Výstupní komponenty, jako například světla, byly ovládány dle nakonfigurovaných tabulek uložených v automatu. Řízení vytápění také probíhalo v souladu s požadovanou koncepcí. Tlačítka EnOcean se v systému chovala naprosto stejně jako klasická tlačítka. Problémy ovšem nastaly při řízení komponent DALI. Funkční blok *FbDALI_Master* někdy neresetoval vstupně-výstupní parametr *xStartDALIMaster* a tím neodblokoval program pro správu DALI zařízení. Tento problém jsem se snažil alespoň částečně vyřešit hlídáním maximálního času, po který může být parametr *xStartDALIMaster* nastaven. Tím se alespoň odblokuje sběrnice DALI pro řízení její správy. Problémy se občas objevily i při zápisu příslušnosti světel do skupin nebo scén přímo do fyzických zařízení DALI, kdy některé hodnoty nebyly zapsány.

Nadřazená aplikace driveru, která byla nejdříve navržena jako klasická okenní aplikace, posléze jako služba systému Windows, řídila komunikaci mezi aplikací v automatu WAGO a serverem spravujícím upravený sdílený datový model. Driver úspěšně řídil výpis i zápis datového modelu do automatu a bez problému spravoval požadavky na simulaci u všech komponent. Driver byl dále naprogramován tak, aby se v případě ztráty komunikace pokoušel znova navázat spojení. Toto chování bylo úspěšně otestováno.

Bohužel se při testování na reálných modelech projevila značná výpočetní náročnost aplikace, která má za následek dlouhou dobu základního cyklu a tím i dlouhé odezvy sytému. V následující kapitole se budu snažit najít příčiny. V kapitole 7.3 bude popsán test aplikace na nejnovější řídící jednotce WAGO typu 750-881, která byla katedře k tomuto účelu zapůjčena.

7.2 Rozbor výpočetní náročnosti

Při testování na reálných modelech se největším problémem ukázala značná doba odezvy systému. Při maximálním počtu komponent vstupů, výstupů a plánů při testech na malém modelu se například doba mezi stiskem tlačítka a rozsvícením světla pohybovala v rozmezí 5 až 6 sekund, což je pro použití v domovní automatizaci neakceptovatelné. V následujících odstavcích se pokusím analyzovat příčiny.

Pro analýzu doby odezvy je nutné se podívat na strukturu aplikace. Při stisku tlačítka aplikace nejdříve vyhodnocuje vyvolanou událost. Vzhledem k algoritmu detekce je toto vyhodnocení zpracováváno tak, že se vyčká čas zadaný v modelu automatu pro detekci dvojstisku a posléze se vyhodnotí, zda došlo k jednomu či dvěma kliknutím nebo zda je tlačítko stále drženo. Vyhodnocené události se musí posléze předat hlavnímu programu, který na jejich základě řídí výstupní komponenty. Toto předání proběhne vždy na začátku hlavního programu. Uvážíme-li nejhorší situaci, že k vyhodnocení události dojde v okamžiku těsně po předání událostí, pak jeden cyklus hlavního programu jsou události jen uloženy a čekají na převzetí a během druhého cyklu se teprve vyhodnotí výstupní komponenty. Tedy čas odezvy se zvýší o dobu až dvou cyklů hlavního procesu. K celkovému času je ještě zapotřebí přičíst čas potřebný k vykonání akce na výstupní komponentě.

Předpokládal jsem, že nejzásadnější příčinou prodloužení doby základního cyklu je častý přístup k bitům bytových proměnných pomocí tečkové notace, neboť tato metoda nepatří mezi standardní operace systému. Tím, že se nejedná o standardní operaci systému WAGO, může být tato operace kontrolérem méně podporována a tím být časově náročnější. Standardním postupem by mělo být využití polí typu bool.

Abych svoji hypotézu potvrdil, provedl jsem test. Upravil jsem aplikaci pro automat tak, aby funkční blok *VYHODNOCENI_UDALOSTI* pracoval s poli typu bool. Vzhledem k paměťové náročnosti těchto polí popsané v kapitole 4.4.4 jsem byl nucen ponechat pouze hlavní program s programem řízení osvětlení a všechny ostatní procesy, programy a patřičné proměnné odstranit. I přes toto odstranění jsem byl ještě nucen snížit maximální počet digitálních výstupů na 40. Maximální počet digitálních vstupů zůstal zachován na hodnotě 128. Posléze jsem připravil druhou aplikaci, která měla stejnou strukturu, ale funkční blok *VYHODNOCENI_UDALOSTI* pracoval stejně jako originální aplikace s tečkovou notací.

Nastavil jsem na všech výstupních komponentách typ „světlo“, aby byl funkční blok volán co možná nejvíce, a porovnal časy základních cyklů. Změřené časy jsou uvedeny v tabulce 7.1 v prvním řádku.

	Původní aplikace s tečkovou notací	**Upravená aplikace s poli typu BOOL**
Aplikace s řízením osvětlení se 40 komponentami typu světlo a 128 digitálními vstupy	646ms	556ms
Aplikace pouze s hlavním programem	3ms	3ms
Aplikace s programem řízení osvětlení bez volání funkčního bloku VYHODNOCENI_UDALOSTI	9ms	9ms
Aplikace s řízením osvětlení se 40 komponentami typu světlo a 64 digitálními vstupy	334ms	289ms

Tabulka 7.1 – Změřené časy základních cyklů

Rozdíl časů 89ms není tak markantní, jak jsem očekával. Pro dvojnásobný počet výstupů by byl rozdíl patrně okolo 178ms, což vzhledem k době základního cyklu původní aplikace, která se pohybovala kolem 2 sekund, není velká úspora. Pokračoval jsem v testech a zkusil rozpoznat příčinu časové náročnosti. Upravil jsem postupně aplikace tak, aby byl jednou volán jen hlavní program, posléze i program řízení, ze kterého jsem ovšem odstranil funkční blok *VYHODNOCENI_VSTUPU*. Posléze jsem porovnal všechny změřené časy.

Ze změřených výsledků lze vyvodit, že příčinou značné časové náročnosti aplikace je algoritmus vyhodnocování událostí, který je vykonáván ve funkčním bloku *VYHODNOCENI_UDALOSTI*. Poslední test, při kterém se změnila konstanta maximálního počtu digitálních vstupů na polovinu (64), a tím se zkrátily i časy základních cyklů téměř přesně o polovinu, potvrzuje, že největší vliv na dobu základního cyklu aplikace má právě algoritmus, který vyhodnocuje události u jednotlivých komponent. Časová náročnost je přímo úměrná zadanému maximálnímu počtu digitálních vstupů a počtu vyhodnocovaných komponent. Zda je použitá v algoritmu tečková notace nebo standardní pole typu bool, nemá na dobu základního cyklu významný vliv.

7.3 Testování aplikace na řídící jednotce 750-881

Vzhledem k výsledkům této práce byl katedře zapůjčen nejnovější model řídící jednotky WAGO s označením 750-881, který obsahuje rychlejší procesor než doposud testované jednotky. Důvodem zapůjčení bylo otestování, o kolik se zkrátí doba základního cyklu, pokud aplikace poběží na této jednotce.

I v nové řídící jednotce jsem nakonfiguroval aplikaci s maximálním počtem výstupních komponent typu světlo a povolil vykonávání pouze řídícího programu osvětlení. Posléze jsem porovnal doby cyklů nové jednotky 750-881 a jednotky 750-841. Následující tabulka 7.2 uvádí výsledek testu.

	Původní aplikace s tečkovou notací na jednotce 750-841	**Původní aplikace s tečkovou notací na jednotce 750-881**
Aplikace s řízením osvětlení se 40 komponentami typu světlo a 128 digitálními vstupy	646ms	148ms
Aplikace s řízením osvětlení s 80 komponentami typu světlo a 128 digitálními vstupy	1348ms	258ms

Tabulka 7.2 – Výsledky testů na nejnovější jednotce 750-881

Výsledky testu jsou velice slibné. Doba základního cyklu se několikanásobně zkrátila a na testovaném malém modelu umožnila při maximálním počtu komponent zkrátit čas odezvy do 1 sekundy! Musím poznamenat, že čas čekání na vyhodnocení události byl opět nastaven na 500ms. Tyto výsledky lze již v aplikacích domovní automatizace akceptovat.

Bohužel jsem během testu zjistil skutečnost, že do nové jednotky nelze bez potíží nahrát aplikaci bez nastavené hardwarové konfigurace. Řídící jednotka nepovolí zapsat data do oblasti fyzických výstupů, pokud na adrese není nakonfigurovaná výstupní karta. Toto omezení platí, i pokud si chceme vynutit hodnotu přímo na výstupní adrese z prostředí CoDeSys. Pro plnohodnotné použití aplikace na nových řídících jednotkách by bylo vhodné toto omezení odstranit, k čemuž by byla potřebná podrobnější studie problému.

Kapitola 8

Závěr

Výsledkem mé práce je navržený systém, skládající se ze dvou aplikací, pro řízení domovní automatizace pomocí programovatelných automatů WAGO. Pro navržený systém byla upravena struktura datového modelu včetně zapracování poznatků, které se objevily při používání staré struktury.

Aplikace pro automaty umožňuje komunikaci s nadřazenou aplikací přes rozhraní MODBUS/TCP a plnou uživatelskou konfiguraci systému pomocí nahraného modelu. Aplikace řídí ovládání světel, ventilátorů a žaluzií na základě vyvolaných událostí tlačítek, vypínačů, časových programů včetně funkce východu a západu slunce a funkce Fantom. Dále aplikace řídí vytápění místnosti dle manuálně nebo programem zadané teploty, kde vytápění lze blokovat okenními kontakty. Řízení vytápění je prováděno topnými ventily, jejichž poloha je ovládána signálem PWM, a na základě poloh ventilů je ovládán i stav kotle. V aplikaci je zakomponována technologie DALI a EnOcean pro bezdrátová tlačítka. Technologie KNX se ukázala být pro tuto aplikaci nevhodná.

Nadřazená aplikace driveru je naprogramována jako služba systému Windows a umožňuje výměnu dat mezi automatem a serverem, který spravuje sdílený datový model domovní automatizace. Driver řídí zápis dat do automatu dle datového modelu a také naopak generování datového modelu z dat vyčtených z automatu. Dále v datovém modelu udržuje aktuální hodnoty proměnných a v případě objevení požadavku, zajistí simulaci chování komponent automatu.

Při testech systému na reálných modelech domovní automatizace byly zjištěny dva nedostatky. Zaprvé při správě sběrnice DALI se některé funkční bloky knihovny DALI_02.lib nechovají korektně. Buď nezapíší správně hodnotu do zařízení, nebo neindikují vyhotovení příkazu. Druhým nedostatkem systému se ukázala být časová náročnost aplikace pro řídící jednotky WAGO. Jako hlavní příčinu se podařilo označit algoritmus pro vyhodnocování požadavků na chování jednotlivých komponent. Ovšem při vyzkoušení aplikace na nejnovější řídící jednotce 750-881 se podařilo zkrátit časy na akceptovatelné hodnoty.

Literatura

[1] Nývlt, Ondřej: *Automatický návrh řízení pro domovní automatizaci.* Diplomová práce, FEL ČVUT, 1. vydání, 2008.

[2] Kubita, Ivo: *Vizualizační prostředí pro domovní automatizaci.* Diplomová práce, FEL ČVUT, 1. vydání, 2009.

[3] Kubita, Ivo: *Dokumentace konfiguračního a vizualizačního nástroje,* FEL-ČVUT, 1. vydání, 2009.

[4] Krejza, Pavel: *Demonstrátor automatizace budov.* Diplomová práce, FEL ČVUT, 1. vydání, 2010.

[5] Pokorný, Jaroslav a kolektiv: *XML technologie. Principy a aplikace v praxi*, GRADA Praha, 1. vydání, 2008, 272 s., iSBN 978-80-247-2725-7.

[6] Swales, Andy: *OPEN MODBUS/TCP* SPECIFICATION, Schneider Electric, 1. vydání, 1999.

[7] *User Manual for PLC Programming with CoDeSys 2.3.,* Smart Software Solutions, Version 1.0, 2003.

[8] WAGO: *Library for Building Automation: Function Block Description for Time – Switching Function*, 28.8.2007.

[9] WAGO: *WAGO-I/O-PRO 32 library: ModbusEthernet_04.lib*, Version 1.1.0

[10] WAGO: *Library for Building Automation: Function Block Description for DALI – Master Module 750-641,* WAGO, 2008.

[11] WAGO: *Library for Building Automation: Function Block Description for EnOcean Radio Receiver 750-642*, 23.1.2008.

[12] WAGO: *Library for Building Automation: Module Description for KNX IP Master*, 28.1.2008.

[13] WAGO: *Library for Building Automation: Module Description for KNX/EIB*, 28.1.2008.

[14] WAGO: *WAGO–I/O-SYSTEM 750: Ethernet TCP/IP 750-841*, Manual, Version 1.2.1.

[15] WAGO: *WAGO–I/O-SYSTEM 750: KNX IP Controller750-849*, Manual, Version 1.0.7.

[16] WAGO: *WAGO–I/O-SYSTEM 750: Programmable Fieldbus Controller ETHERNET 750-881*, Manual, Version 1.0.0.

[17] kolektiv autorů: *KNX základní kurs.* Konnex, 1. vydání, 2009

[18] WAGO: *KNX StarterKit navod pro oziveni.pdf*. 2010.

[19] Kottnauer, Jakub: *Vlákna v C#.,* 1. vydání, 2008

[20] [online]: *3S Software Alliance.* 2010, (http://www.3s-software.com).

[21] [online]: *DALI standard.* 2010, (http://www.dali-ag.org).

[22] [online]: *EnOcean.* 2010, (http://www.enocean.com).

[23] [online]: *EnOcean alliance.* 2010, (http://www.enocean-alliance.org).

[24] [online]: *KNX/Konnex Association*. 2010, (http://www.knx.org).

[25] [online]: *Wago* 2010, (http://www.wago.com).

Přílohy:

Tištěné:

A – Podrobný popis upraveného datového modelu
B – Nastavení velikosti oblastí remanentní a adresovatelné paměti
C – Adresace proměnných

Na CD:

D – Zdrojový kód aplikace automatu
E – Zdrojový kód aplikace nadřazené služby
F – Soubor .xsd definice upraveného datového modelu
G – Datový model pro fyzické modely domovní automatizace
H – Zdrojový soubor .xls pro adresaci proměnných
I – KNX databáze pro fyzický velký model

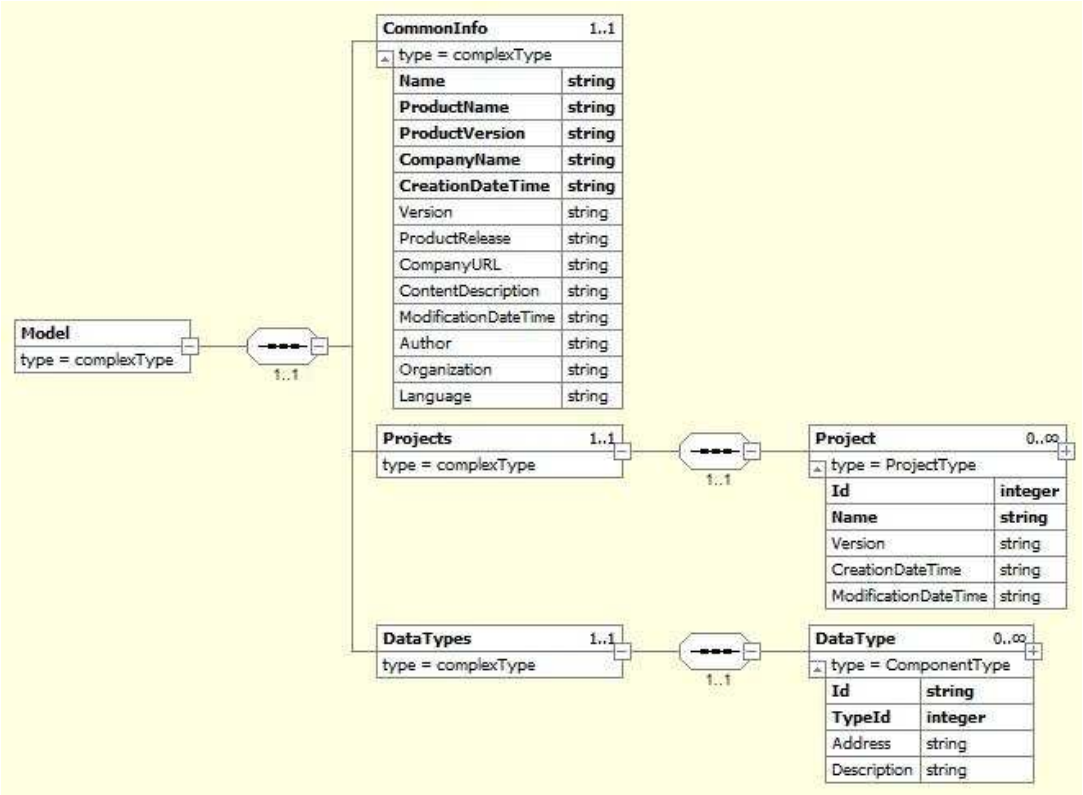
Příloha A

Podrobný popis upraveného datového modelu

Obsah

1) Realizace modelu

Model je navržen jako XML schéma, kde se kořenový element jmenuje Model. Model obsahuje tři základní povinné elementy: *CommonInfo, Projects* a *DataTypes*. Tyto elementy budou podrobně popsány v následujících kapitolách.



Obrázek A.1 - Model

Sekce *CommonInfo*, jak název napovídá, ukládá obecné informace o modelu. Jsou zde uloženy například údaje o autorovi modelu, času vytvoření, informace o verzi a jazyku, atd.

Sekce *Projects* obsahuje neomezený počet projektů domovní automatizace. Projekt by měl být vázán na bytovou jednotku (byt nebo rodinný dům). A tak je možné pomocí jednoho serveru s modelem vzdáleně spravovat i několik bytů či domů.

Sekce *DataTypes* obsahuje všechny předobrazy komponent používaných v projektech. Z těch se v projektech vytváří nové fyzické komponenty, které už mají vazbu na konkrétní reálnou komponentu v budově.

1.1 CommonInfo

Sekce *CommonInfo* obsahuje 13 atributů, z nichž pět je povinných (pozn. povinné atributy jsou vyznačeny tučně).

CommonInfo	**1..1**
type = complexType	
Name	**string**
ProductName	**string**
ProductVersion	**string**
CompanyName	**string**
CreationDateTime	**string**
Version	string
ProductRelease	string
CompanyURL	string
ContentDescription	string
ModificationDateTime	string
Author	string
Organization	string
Language	string

Obrázek A.2 - CommonInfo

<u>atributy:</u>

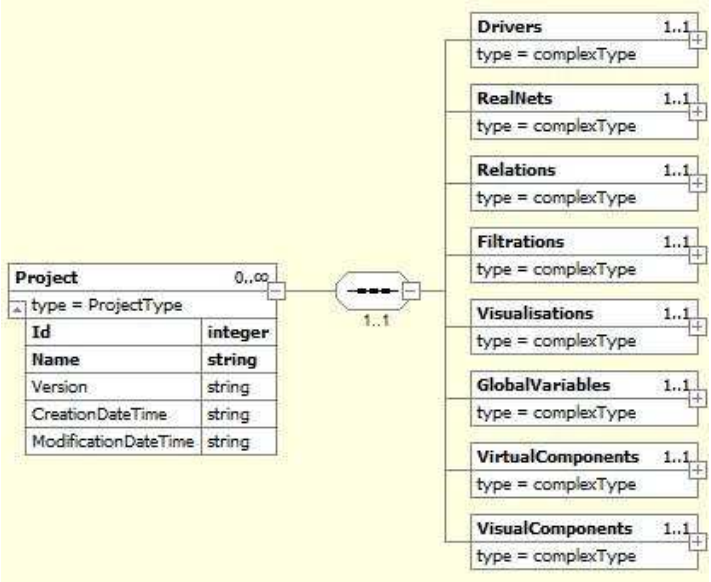
- **Name** *Jméno modelu*
- **ProductName** *Jméno produktu*
- **ProductVersion** *Verze produktu*
- **CompanyName** *Jméno firmy, která model vytvořila*
- **CreationDateTime** *Datum a čas vytvoření modelu*
- Version *Verze modelu*
- ProductRelease
- CompanyURL *Webová adresa firmy, která model vytvořila*
- ContentDescription *Bližší popis obsahu modelu*
- ModificationDateTime *Datum a čas poslední změny v modelu* ***(mění driver, který úpravu provedl)***
- Author *Autor modelu*
- Organization *Organizace využívající model*
- Language *Použitý jazyk v modelu*

1.2 Projects

Sekce Projects je nejdůležitější částí modelu. Zde jsou uloženy jednotlivé projekty domovní automatizace. Počet projektů je neomezen. V následujících podkapitolách je popsána struktura projektu, elementu typu *ProjectType*.

1.2.1 Project *(ProjectType)*

Element *Project* je typu *ProjectType* a je základním „pilířem“ modelu. Zde jsou uložena veškerá data a nastaveni pro správné fungování a vizualizaci domovní automatizace. V následujících kapitolách si podrobně popíšeme jednotlivé sekce tohoto elementu.



Obrázek A.3 – Project

atributy:

- **Id** — *Unikátní číslo projektu pro adresaci, jednoznačné v sekci Projects*
- **Name** — *Jméno projektu*
- **Version** — *Verze projektu*
- CreationDateTime — *Datum a čas vytvoření projektu*
- ModificationDateTime — *Datum a čas poslední změny projektu* ***(mění driver, který úpravu provedl)***

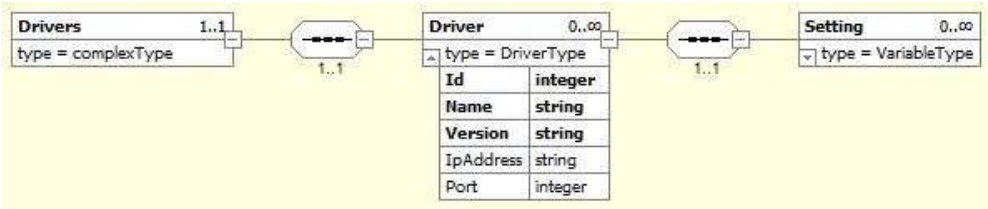
elementy:

- **Drivers** — *Sekce, která obsahuje informace o driverech používaných v projektu včetně jejich proměnných a dat potřebných pro konfiguraci a. Obsahuje instance typu DriverType.*
- **RealNets** — *V této sekci je obraz fyzické sítě. Na tyto sítě jsou připojena jednotlivá fyzická zařízení. Sekce obsahuje instance typu RealNetType.*
- **Relations** — *Sekce obsahuje informace o vazbách mezi fyzickými komponentami. Vazby jsou definována pomocí událostí řídících členů a akcí akčních členů. Instance jsou typu RelationType. Více v kapitole „1.2.4 Relations“.*

- **Filtrations** — *Sekce obsahující vizuální komponenty seskupené do skupin podle určitého společného znaku. Instance jsou typu FiltrationType.*
- **Visualitions** — *Sekce obsahující informace o obrazovkách pro vizualizaci. Každá obrazovka může mít několik stránek. Více v kapitole „1.2.6 Visualisations“. Instance jsou typu VisualisationType.*
- **GlobalVariables** — *Sekce obsahuje definice globálních proměnných. Instance jsou typu VariableType.*
- **VirtualComponents** — *Sekce obsahuje virtuální komponenty v projektu, které nejsou vázány na konkrétní zařízení. Virtuálními komponentami mohou být topné plány, časové plány, atd. Instance jsou typu ComponentType.*
- **VisualComponents** — *Sekce obsahuje jednotný obraz komponent pro vizualizaci. S touto sekcí pracují klienti typu vizualizace, kteří zajišťují jednotné zobrazení a chování komponent stejného typu. Instance jsou typu VisualComponentType. Více v kapitole „1.2.9 VisualComponents“.*

1.2.2 Drivers *(DriverType)*

Sekce *Drivers* může obsahovat neomezené množství elementů *Driver,* který je typu *DriverType* a který slouží pro uložení informací o použitém driveru. Drivery si zde mohou ukládat své proměnné a data potřebná pro konfiguraci systému. Elementy *Setting* v této sekci jsou typu *VariableType*, blíže popsané v kapitole „1.2.3.4 Variable *(VariableType)*“.



Obrázek A.4 – Drivers

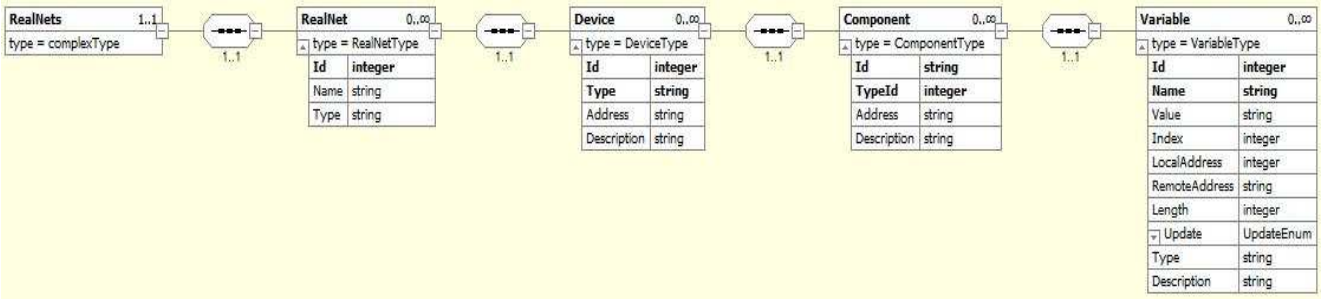
atributy:

- **Id** — *Unikátní identifikační číslo driveru jednoznačné v rámci sekce Drivers*
- **Name** — *Jméno driveru*
- **Version** — *Verze driveru*
- IpAddress — *IP adresa pro připojení k hardwaru*
- Port — *Port pro připojení k hardwaru*

elementy:

- Setting — *Elementy typu VariableType, které slouží pro uložení proměnných. Více v kapitole „1.2.3.4 Variable (VariableType)“.*

1.2.3 RealNets



Obrázek A.5 – RealNets a podsekce

Sekce *RealNets* obsahuje fyzické komponenty, obraz reálných komponent v budově. Struktura sekce je navržena tak, aby dávala jasnou představu o reálném zapojení komponent v systému. Toto uspořádání vychází z požadavku na model. Síť je reprezentována elementem *RealNet*, zařízení elementem Device a komponenta elementem *Component*.

1.2.3.1 RealNet *(RealNetType)*

Element pro reprezentaci reálné sítě, ke které jsou připojena zařízení.

atributy:

- **Id** — *Unikátní číslo identifikující konkrétní fyzickou síť, jednoznačné v sekci RealNets*
- Name — *Jméno fyzické sítě*
- Type — *Typ fyzické sítě*

elementy:

- Device — *Elementy reprezentující fyzická zařízení připojená k síti. Instance jsou typu DeviceType.*

1.2.3.2 Device *(DeviceType)*

Element pro reprezentaci reálného zařízení, ke které jsou připojeny komponenty.

atributy:

- **Id** — *Unikátní číslo identifikující konkrétní fyzické zařízení, unikátní v daném elementu RealNet*
- **Type** — *Typ fyzického zařízení (např. ULSW,ULMO, WAGO,…)*
- Address — *Adresa fyzického zařízení (hlavně pro Damic a podobná zařízení)*
- Description — *Bližší popis fyzického zařízení*

elementy:

- Component — *Elementy reprezentující fyzické komponenty připojené k zařízení. Instance jsou typu ComponentType*

1.2.3.3 Component *(ComponentType)*

Elementy reprezentující reálné komponenty systému, včetně informací o jejich aktuálním stavu.

Každá komponenta by měla mít proměnnou Value, reprezentující její aktuální hodnotu.

Každá komponenta, která bude vyvolávat události nebo mít své akce, musí mít element typu *VariableType* pojmenovaný „Behaviours“. Obsahem jeho atributu „Name“ musí být tedy „Behaviours“ a obsahem atributu „Value“ musí být řetězec pojmenovaných událostí či akcí oddělených znakem ’|’ („pipe“). Na tento řetězec je odkazováno ze sekce Relations a je kopírován do sekce VisualComponents, do elementů VisualComponent, atributu Behaviours. Důvodem je nezávislost na použitém jazyku v projektu. Obrázek A.6 ilustruje příklad. Více v patřičných kapitolách.

```
- <RealNet Id="1" Name="ModbusL1" Type="Modbus">
  - <Device Id="1" Type="Wago" Address="192.168.100.10" Description="PLC Wago na IP adrese 192.168.100.10">
    - <Component Id="1.1.1" TypeId="8" Address="3" Description="Topeni v loznici">
        <Variable Id="1" Name="Value" Value="False" Type="Bool" Description="Hodnota vystupu" />
        <Variable Id="2" Name="Behaviours" Value="otevrit/zavrit" Type="String" Description="Udalosti pro komponentu ventil" />
        <Variable Id="3" Name="TypeOfRegulation" Value="1" Type="String" Description="Typ regulace (1=PID,2=Hystereze)" />
        <Variable Id="4" Name="TypeOfAddress" Value="DO" Type="String" Description="Typ adresy" />
        <Variable Id="5" Name="Negation" Value="True" Type="String" Description="Negace ovladaci hodnoty" />
      </Component>
```

Obrázek A.6 - Příklad

atributy:

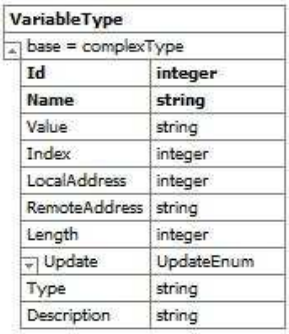
- **Id** — *Unikátní identifikující řetězec fyzické komponenty v tečkové notaci, které je unikátní v celé sekci RealNets.*
- **Type_id** — *Číslo ukazující na předobraz typu komponenty v sekci DataTypes*
- Address — *Adresa komponenty (hlavně pro systémy typu Wago a podobné)*
- Description — *Popis fyzické komponenty*

elementy:

- Variable — *Množina elementů umožňující ukládání proměnných a dat, které se váží k dané fyzické komponentě. Příkladem může být proměnná, která nese informaci o aktuální hodnotě komponenty (zapnuto/vypnuto). Instance jsou typu VariableType.*

1.2.3.4 Variable (VariableType)

Element pro uložení proměnné. Tento typ je použit pro elementy *Setting, Variable* a *GlobalVariable*.



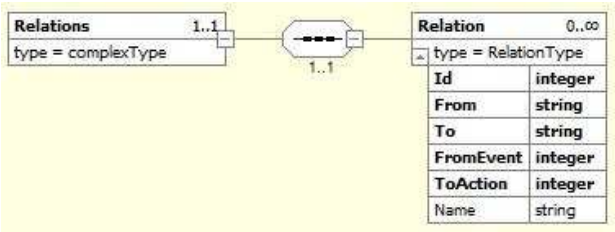
Obrázek A.7 – VariableType

atributy:

- **Id** *Unikátní identifikační číslo proměnné jednoznačné v daném elementu, v němž jsou umístěni.*
- **Name** *Jméno proměnné*
- Value *Hodnota proměnné*
- Index *Index proměnné*
- LocalAddress *Lokální adresa proměnné*
- RemoteAddress *Vzdálená adresa proměnné*
- Length *Velikost proměnné*
- Update *Možnost změny proměnné*
- Type *Typ proměnné*
- Description *Bližší popis proměnné*

1.2.4 Relations *(RelationType)*

V této sekci jsou uloženy veškeré informace o vazbách mezi komponentami. Element *Relation* je typu *RelationType* a reprezentuje jednu vazbu mezi komponentami. Vazba není dána jen zdrojovou a cílovou komponentou, ale také událostí na zdrojové komponentě a akcí na cílové komponentě, která má být událostí vyvolána.



Obrázek A.8 – Relations

Atributy *From* a *To* odkazují na identifikační řetězec fyzické komponenty a atributy *FromEvent* a *ToAction* odkazují na proměnnou komponenty, pojmenovanou Behaviours. Tedy každá komponenta, která bude vyvolávat události nebo mít své akce, musí mít tuto proměnnou typu *VariableType*. Obsahem jejího atributu „Name“ musí být tedy „Behaviours“ a obsahem atributu Value musí být řetězec pojmenovaných událostí či akcí oddělených znakem '|' („pipe“). Element *Relation* se odkazuje na tento řetězec a číslem určuje pořadové číslo události či akce, která se účastní vazby. Důvodem je nezávislost na použitém jazyku v projektu. Obrázek 13 ilustruje příklad elementu *Relation*.

```
<Relation Id="21" From="1.1.22" To="1.1.3" FromEvent="1" ToAction="1"
 Name="Tlacitko leve v detskem pokoji:kliknout->Impulsni svetlo v detskem
 pokoji:rozsvitit" />
```

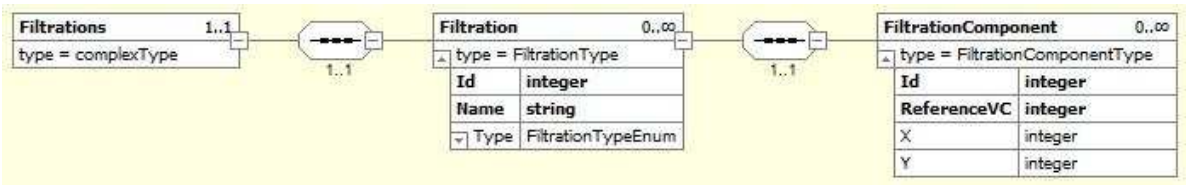
Obrázek A.9 – Příklad elementu Relation

atributy:

- **Id** — *Unikátní identifikační číslo jednoznačné v sekci Relations*
- **From** — *Zdrojová komponenta. Obsahuje Id fyzické komponenty,* která bude vyvolávat událost.
- **To** — *Cílová komponenta. Obsahuje Id fyzické komponenty, která bude vykonávat akci.*
- **FromEvent** — *Pořadové číslo zdrojové události v řetězci proměnné Behaviours.*
- **ToAction** — *Pořadové číslo cílové akce v řetězci proměnné Behaviours.*
- Name — *Jméno vazby*

1.2.5 Filtrations *(FiltrationType)*

Sekce obsahuje vizuální komponenty seskupené do skupin podle určitého společného znaku. Společným znakem může být stejný typ komponent (světla, termostaty,…) nebo umístění komponent ve společném reálném prostoru (kuchyně, obývací pokoj,…). Instance jsou typu *FiltrationType*.



Obrázek A.10 - Filtrations

atributy:

- **Id** — *Unikátní identifikační číslo filtru jednoznačné v sekci Filtrations*
- **Name** — *Jméno filtru*
- Type — *Typ filtrace (např. světla, ventilátory, pokoj, dům,…)*

1.2.5.1 FiltrationComponent *(FiltrationComponentType)*

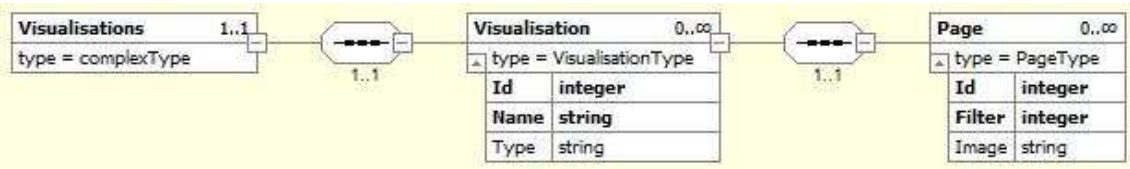
Element odkazující na určitou vizuální komponentu. Množina těchto elementů tvoří skupinu se společným znakem. Společným znakem může být i příslušnost na společné stránce obrazovky vizualizace. Proto má typ *FiltrationType* atributy X a Y, které značí celočíselné umístění komponenty na stránce.

atributy:

- **Id** — *Unikátní identifikační číslo jednoznačné v daném elementu Filtration*
- **ReferenceVC** — *Odkaz na vizuální komponentu v sekci VisualComponents*
- X — *X-ová celočíselná souřadnice umístění na stránce obrazovky*
- Y — *Y-ová celočíselná souřadnice umístění na stránce obrazovky*

1.2.6 Visualisations *(VisualisationType)*

Sekce obsahující informace o obrazovkách pro vizualizaci. Každá obrazovka může mít několik stránek a stránka zobrazuje určitou skupinu komponent definovanou v sekci *Filtrations*. Ideou této struktury je, aby pokud si uživatel zvolí vizualizovat např. svůj dům, aplikace mu nabídla různé pohledy (stránky) zobrazení, např. přehledově celý dům, detailně jednotlivé místnosti, atd., a on mohl mezi nimi plynule přecházet. Element *Visualisation* tedy zapouzdřuje společné stránky vizualizace.



Obrázek A.11 - Visualisations

atributy:

- **Id** — *Unikátní identifikační číslo v sekci Visualisations*
- **Name** — *Jméno obrazovky vizualizace*
- Type — *Typ vizualizace*

1.2.6.1 Page *(PageType)*

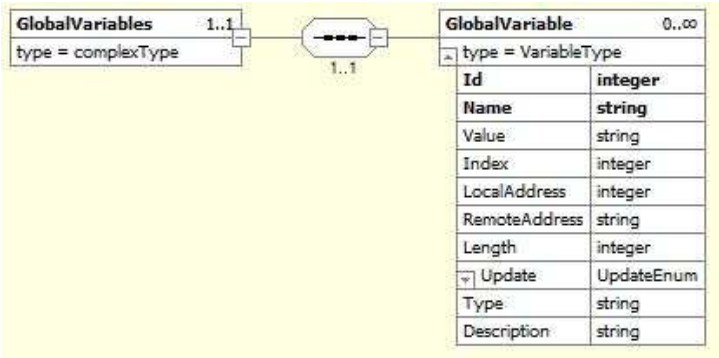
Element *Page* je typu *PageType* a ukládá v sobě informaci o vzhledu stránky, kterou si uživatel může prohlížet a upravovat. Atribut Image slouží pro uložení cesty k podkladovému obrázku a atribut Filter odkazuje na skupinu komponent, které jsou na stránce umístěny. Tato skupina komponent je definována pomocí elementu *Filtration* v sekci *Filtrations*.

atributy:

- **Id** — *Unikátní identifikační číslo jednoznačné pro daný element Visualisation*
- **Filter** — *Reference na element Filtration, skupinu komponent, které jsou na stránce zobrazeny*
- Image — *Obrázek na stránce pro uživatelovu vizualizaci*

1.2.7 GlobalVariables

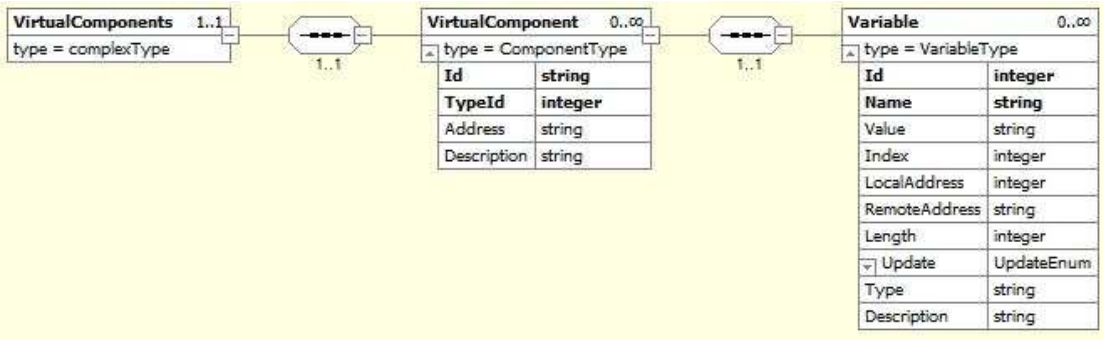
Sekce *GlobalVariables* slouží pro ukládání globálních proměnných platných v celém projektu nemající příslušnost ke konkrétnímu driveru nebo komponentě. Kterýkoli klient si zde může uložit své proměnné. Elementy v této sekci jsou typu *VariableType*, popsaných v kapitole „1.2.3.4 Variable *(VariableType)*“.



Obrázek A.12 – GlobalVariables

1.2.8 VirtualComponents

Sekce obsahuje virtuální komponenty v projektu, které nejsou vázány na konkrétní zařízení. Virtuálními komponentami mohou být topné plány, časové plány, atd. Instance jsou typu *ComponentType*, které byly popsány v kapitole „1.2.3.3 Component *(ComponentType)*“. Jediným rozdílem ovšem je, že virtuální komponenty se budou adresovat pomocí identifikátoru Id, který nebude řetězec, ale klasické číslo, jednoznačné v sekci *VirtualComponents*.



Obrázek A.13 – VirtualComponents

atributy:

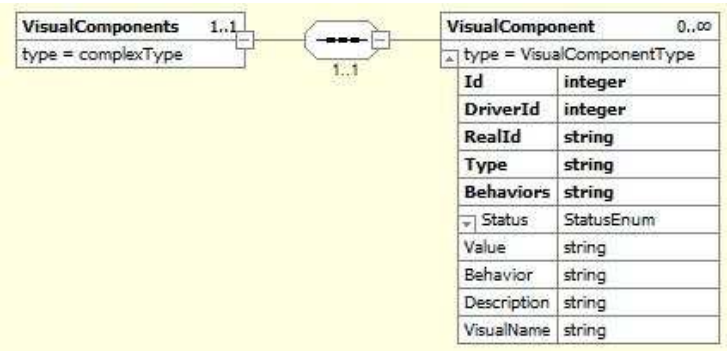
- **Id** — *Unikátní číslo virtuální komponenty jednoznačné v sekci VirtualComponents.*
- **Type_id** — *Číslo ukazující na předobraz typu komponenty v sekci DataTypes*
- Address — *Adresa, pro virtuální komponty pravděpodobně bez významu*
- Description — *Bližší popis virtuální komponenty*

elementy:

- Variable — *Množina elementů umožňující ukládání proměnných a dat, které se váží k dané virtuální komponentě. Instance jsou typu VariableType, popsané v kapitole „1.2.3.4 Variable (VariableTypes)“.*

1.2.9 VisualComponents

Sekce *VisualComponents* obsahuje elementy *VisualComponent,* které jsou typu *VisualComponentType*. Jak bylo řečeno v požadavcích na model, vizuální část komponent má být oddělená a nezávislá na struktuře fyzických komponenent. Program pro uživatelskou vizualizaci domovní automatizace pracuje pouze v části modelu, kde všechny komponenty mají stejný vzor - *VisualComponent*.



Obrázek A.14 - VisualComponents

Typ VisualComponentType má atribut odkazující na identifikátor driveru (*DriverId*), který zajišťuje výměnu dat mezi vizuální a spjatou fyzickou komponentou. Dále musí mít také identifikátor zmíněné spjaté fyzické komponenty (RealId). Atribut *Type* určuje typ vizuální komponenty. Podle tohoto typu aplikace zvolí ikonu pro vykreslení na obrazovce a přiřadí ji jméno podle atributu *VisualName*.

Atribut *Status*, má pouze dva stavy: „active“ a „nonactive“. Aktivní komponenta je ta, která je dostupná k ovládání skrze driver. Neaktivní komponenta je ta, která se stala z provozních důvodů nedostupnou, např. vyndání karty z PLC či ztráta komunikace zařízení. Driver takovou vizuální komponentu nesmí smazat, i kdyby byla adresně mimo rozsah. Tedy výpadek komponenty nesmí způsobit smazání nastavení, driver pouze přepíše atribut Status na „nonactive“. Driver maže vizuální komponenty v případě nahrání modelu z hardwaru na server.

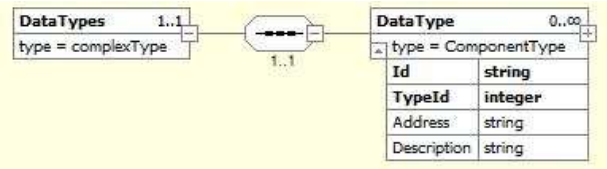
V atributu Behaviours je uložen řetězec podporovaných událostí či akcí komponenty. Řetězec tvoří pojmenované jednotlivé události či akce oddělená znakem ‘|’ („pipe”). Požadavek uživatele na vyvolání události nebo akce se napíše do atributu Behaviour. Driver si požadavek přečte a po zpracování smaže.

atributy:

- **Id** — *Identifikující číslo vizuální komponenty jednoznačné v sekci VisualComponents*
- **Driver_id** — *Odkaz na identifikátor driveru, který bude vizuální komponentu obsluhovat. Obsluhou je myšlena výměna dat mezi vizuální a fyzickou komponentou.*
- **Real_id** — *Odkaz na identifikátor fyzické komponenty v sekci RealNets, se kterou je vizuální komponenta spjata.*
- **Type** — *Typ vizuální komponenty (např. světlo,tlačítko,…). Je doporučeno zde volit domluvené jednotné anglické názvy (např. light, button,…) shodné pro všechny vizuální aplikace*
- **Behaviours** — *Seznam událostí či akcí podporovaných fyzickou komponentou oddělené znakem ‘|‘ („pipe“). Tento řetězec by měl být napsán v jazyce používaném v projektu, neboť z tohoto řetězce si vizualizační aplikace vytvoří seznam akcí a zobrazí jej jako nabídku uživateli. Příklad řetězce pro českou mutaci „kliknout|dvojkliknout|dlouze kliknout“.*
- Status — *Status vizuální komponenty.*
- Value — *Hodnota vizuální komponenty (string z důvodu obecnosti). Zapisuje ji driver, vizualizace čte. např. 0,1, true, false, 32123,…*
- Behaviour — *Požadavek na vyvolání události nebo provedení akce u spjaté fyzické komponenty vyvolané zásahem uživatele. Zapisuje ji vizualizace, driver ji čte a maže.*
- Description — *Bližší popis vizuální komponenty*
- VisualName — *Jméno vizuální komponenty, které bude zobrazováno v aplikaci pro vizualizaci.*

1.3 DataTypes

Sekce *DataTypes* obsahuje všechny základní typy komponent s výchozími daty používaných v projektech. Elementy v této sekci (*DataType*) jsou tedy typu *ComponentType*, popsaným v kapitole „Component *(ComponentType)“*. Z této sekce drivery vybírají typ komponenty, zkopírují si ji, změní některá data a vkládají připravenou komponentu do sekce *Projects*.



Obrázek A.15 – DataTypes

Elementy v této sekci jsou adresovány pomocí atributu *TypeId*.

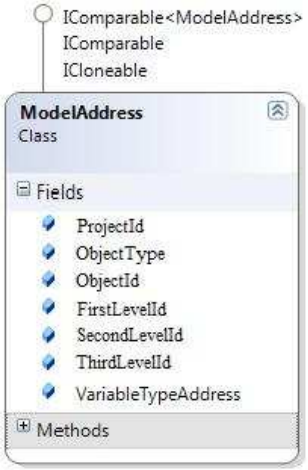
2) Adresování ve společném modelu

2.1 SharedModel

Instance třídy „SharedModel“ je sdílený objekt umístěný na serveru. Konfigurace serveru je nastavena tak, že instance této třídy se vytvoří při prvním připojení klienta k serveru a poté všichni další klienti přistupují k tomuto objektu. Třída „SharedModel“ implementuje rozhraní „ISharedModel“, přes které přistupují všichni klienti.

2.2 Adresování

K adresování jednotlivých částí v třídě „Model“ je určená třída „ModelAddress“ – viz Obrázek A.16. Strávně poskládaná adresa se předá metodám „GetObject“, „GetObjectList“ nebo „GetObjectListByAttr“ a metody vrátí požadovanou hodnotu.



Obrázek A.16 – Třída ModelAddress

Postup vytvoření adresy je následující:

1. Nejprve zvolíme požadovaný návratový typ – vlastnost „ObjectType“. Pro pole i jednu položku je návratový typ stejný. Návratové typy volíme z výběrového seznamu ModelTypesEnum. Například pro získání pole zařízení nebo jednoho zařízení nastavíme „ObjectType“ na „ModelTypesEnum.DeviceType“.
2. Pokud načítáme jiný typ než „CommonInfo“ nebo „DataType“, vyplníme identifikační číslo projektu – „ProjectId“.
3. Dále vyplníme identifikační čísla všech objektů v řadě následujícím způsobem: „ObjectId“ nastavíme na jméno hledaného objektu a pokud má nadřazený jiný objekt nastavíme „FirstLeveId“, „SecondLevelId“ a případně i „ThirdLevelId“ na identifikační čísla nadřazených objektu.
4. Pokud je „ObjectType“ nastaveno na „VariableType“, je nutné ještě nastavit typ proměnné „VariableTypeAddress“ na umístění proměnné. Tedy zda se zajímáme o proměnnou driveru, komponenty nebo globální komponenty. Typ proměnné volíme z výběrového seznamu VariableTypeElementEnum. Například pro získání proměnné driveru nastavíme „VariableTypeAddress“ na „VariableTypeElementEnum.Driver“.
5. Pokud načítáme typ „DataType“, adresujeme dle atributu TypeId.

Obrázek A.21 popisuje postup vytváření adresy. Začneme od prvku model a po čáře pokračujeme až ke konečnému prvku. Do adresy zapíšeme všechny položky uvedené v okénku nad prvkem, který jsme prošli. Pro konečný prvek uvedeme položky pouze ve spodním okénku. Pokud nás zajímá **konkrétní objekt**, musíme přidat ještě položku **ObjectId**.

Příklady vytvoření adresy

a) Adresa fyzické komponenty s Id=“3.5.4“

```
address = new ModelAddress()
{
    ProjectId = "1",
    ObjectType = ModelTypesEnum.ComponentType,
    FirstLevelId = "3",
    SecondLevelId = "5",
    ObjectId = "3.5.4",
};
```

Obrázek A.17 – Příklad adresy 1

b) Adresa proměnné u fyzické komponenty s Id="10"

```
address = new ModelAddress()
{
    ProjectId = "1",
    ObjectType = ModelTypesEnum.VariableType,
    VariableTypeAddress = VariableTypeElementEnum.Component,
    FirstLevelId = "3",
    SecondLevelId = "5",
    ThirdLevelId = "3.5.4",
    ObjectId = "10",
};
```

Obrázek A.18 – Příklad adresy 2

c) Adresa stránky vizualizace s Id="3"

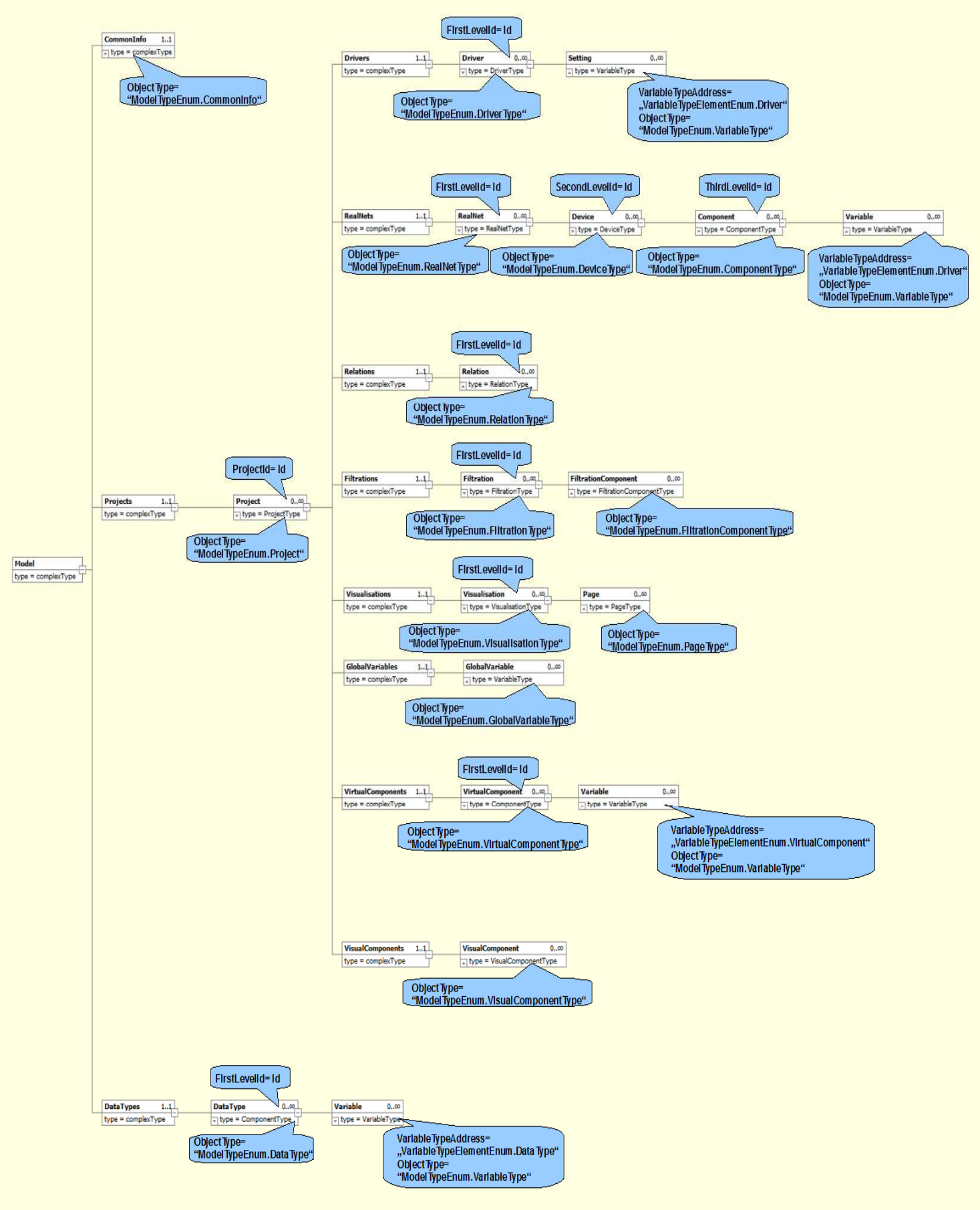
```
address = new ModelAddress()
{
    ProjectId = "1",
    ObjectType = ModelTypesEnum.PageType,
    FirstLevelId = "2",
    ObjectId = "3",
};
```

Obrázek A.19 – Příklad adresy 3

d) Adresa komponenty z DataType s TypeId="6"

```
address = new ModelAddress()
{
    ObjectType = ModelTypesEnum.DataType,
    ObjectId = "6",
};
```

Obrázek A.20 – Příklad adresy 4



Obrázek A.21 – Tvorba adresy

Příloha B

Nastavení velikosti oblastí remanentní a adresovatelné paměti

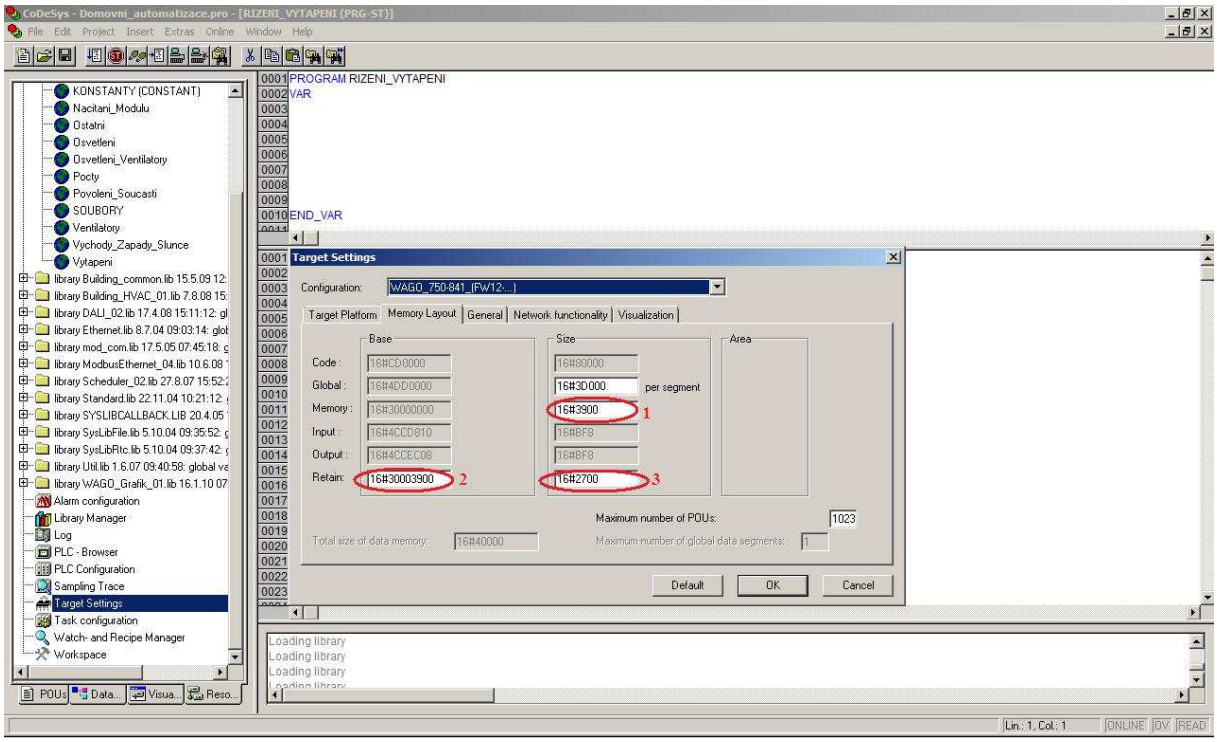
Obsah

Úvod

Během své práce na PLC Wago, jsem narazil na potřebu použít adresovatelnou pamět a přístupu na tuto paměť pomocí protokolu MODBUS. Dále jsem potřeboval použít část paměti jako zálohovanou, tedy remanentní. Protože nalezení potřebných údajů o struktuře a nastavení paměti mi trvalo značnou dobu, rozhodl jsem se popsat zjištěné skutečnosti do tohoto krátkého dokumentu a ušetřit tím mým případným následovníkům čas.

Dialogové okno pro nastavení paměti

Pro programování a konfiguraci PLC Wago se využívá program CoDeSys. Pro konfiguraci paměti automatu v tomto prostředí je třeba otevřít patřičné dialogové okno. Zvolíme z levé lišty záložku *„Resources“* a dále dvakrát poklepeme na položku *„Target Settings“*. V objeveném okně zvolíme záložku *„Memory Layout“*. Následující obrázek tento postup přibližuje.



Obrázek B.1 – Dialogové okno pro nastavení paměti PLC

Záložka obsahuje dvanáct textových polí s hodnotami dle zvoleného typu centrály PLC. Jen čtyři pole můžeme měnit. V našem zájmu budou tři pole zvýrazněná na obrázku.

Rozdělení paměti

Nejprve je nutné nalézt v manuálu patřičné centrály velikost paměti. V anglicky psaných dokumentech bývá uvedena pod položkou Non-Volatile memory.

Technical Data	
Number of I/O modules	64
with bus extension	250
Fieldbus	
Max. input process image	2 Kbytes
Max. output process image	2 Kbytes
Input variables (max.)	512 bytes
Output variables (max.)	512 bytes
Configuration	via PC
Program memory	512 Kbytes
Data memory	256 Kbytes
Non-volatile memory (retain)	24 Kbytes (16 Kbytes retain, 8 Kbytes flag)
Voltage supply	DC 24 V (-25 % ... +30 %)
Max. input current (24 V)	500 mA
Efficiency of the power supply	87 %
Internal current consumption (5 V)	300 mA
Total current for I/O modules (5 V)	1700 mA
Isolation	500 V system/supply
Voltage via power jumper contacts	DC 24 V (-25 % ... +30 %)
Current via power jumper contacts (max.)	DC 10 A

Obrázek B.2 – Velikost paměti k dispozici pro adresovatelnou a zálohovanou oblast

Pouze tuto paměťovou velikost máme k dispozici pro součet velikostí adresovatelné a zálohované oblasti. Tato paměť se dělí na dvě části. První část se využívá jako adresovatelná oblast, kde proměnným můžeme přiřadit konkrétní adresu, a druhá část jako zálohovanou (remanentní) oblast, kterou ovšem už adresovat nelze. Hranici mezi těmito oblastmi můžeme dle svých požadavků posouvat.

Adresovatelná část paměti

První částí v této paměti je adresovatelná oblast. Ta začíná od pevně dané adresy, která se může lišit podle typu centrály. V případě centrály Wago_750-841_(FW12..) je tato adresa 12288 neboli 3000hex. Požadovanou velikost adresovatelné oblasti zadáváme v hexadecimálním kódu do textového okna v řádku „Memory“, sloupec „Size“ (na obrázku 1 označeno jako pole 1). Pokud tedy požadujeme velikost adresovatelné paměti 14592Bytu (3900h), zapíšeme do textového pole „3900“.

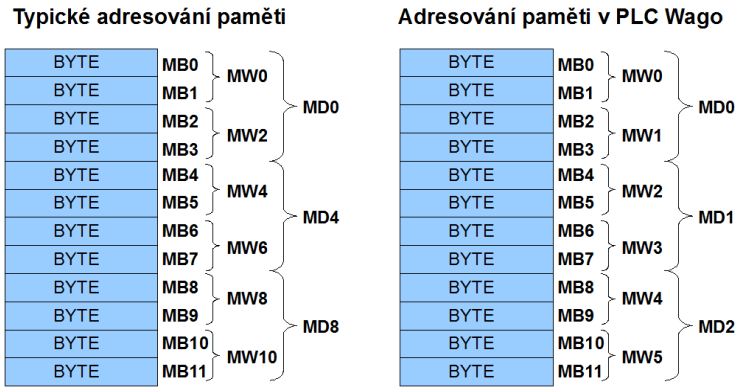
V programu automatu adresujeme od nuly (první byte MB0). Příklad užití adresace je uveden na obrázku 3. Důležitou informací je způsob číslování adres.

```
0018  (*Promenna jedna - komentar*)
0019  promenna1 AT %MB0: INT;
```

Obrázek B.3 – Příklad adresace proměnné v prostředí CoDeSys

Pokud chceme tuto oblast paměti adresovat pomocí např. protokolu MODBUS, musíme přičíst daný offset. Tedy opět pro případ centrály Wago_750-841_(FW12..) je tento offset 12288 neboli 3000hex. Pokud chceme číst proměnnou, kterou jsme uložili v programu na adrese MB0, zadáme do protokolu číslo registru 12288 (3000hex). Vnější adresování se děje po 16 bitových registrech, tedy číslují se „wordy“. Např. MB10 je na adrese 12293 (12288+5).

Poznámka: Na rozdíl od jiných běžných PLC, Wago čísluje pro každý datový typ po sobě jdoucími čísly. Jasnější popis poskytuje obrázek 4.

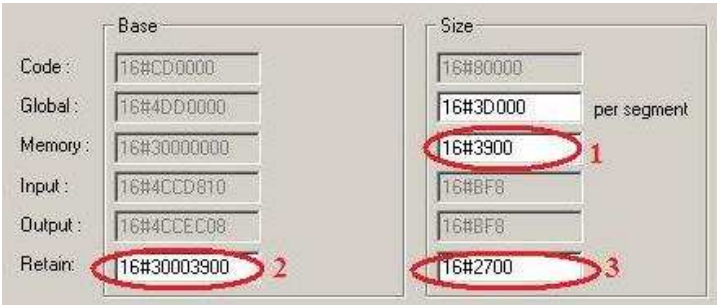


Obrázek B.4 – Způsob adresování proměnných v programu PLC

Číslování bytových proměnných je stejné. Pro typ word (2byty) ve většině PLC začíná adresace MW0 následované MW2,MW4, atd. Tedy číslo wordu se shoduje s adresou spodního bytu. PLC Wago čísluje wordy a byty zvlášť. Tedy typ word začíná MW0, následuje MW1,MW2,MW3. To samé pravidlo je použito i pro typy proměnných s délkou 4bytů. Při volbě adresy proměnné musíme tedy dávat dobrý pozor, aby se oblast proměnné nepřekrývala s druhou proměnnou jiného typu.

Remanentní část paměti

Pro část remanentní paměti se zadává počátek a velikost oblasti. Počátek se zadává v textovém poli 2. Je důležité, aby tato adresa byla rovna nebo větší než nastavená velikost adresovatelné paměti, o které jsme se zmínili v minulé kapitole. Pokud dojde k překrytí, program nás nenechá adresovat překrývající se část paměti.



Obrázek B.5 – Nastavení remanentní oblasti

Do této paměti jsou ukládány proměnné označené v deklaraci jako „RETAIN“. Automat si jejich proměnné zálohuje a pamatuje si je i po výpadku napájení.

Příloha C

Adresace proměnných

Název proměnné / konstanty v automatu	Název proměnné / konstanty v driveru	Typ a rozměry	Hodnota konstanty	Informace o proměnné/konstantě	čtení	zápis	Velikost proměnné v bytech	Počáteční adresa DEC MODBUS pro funkci FC3 po wordech	Koncová adresa DEC MODBUS pro funkci FC3 po wordech		Adresa použitá ve Wagu		
Vstupy a výstupy											**byty**	**word**	**double**
Vstupy_analog		pole INT o rozměru 1..POCET_VSTUPU_ANALOG		Pole INT čísel, jež obsahují přímý přístup na analogové vstupy, např. aktuální teplota na senzorech násobenou deseti. Pozor na obsazení adres kartami DALI, EnOcean a některými dalšími.	r	-		**0**			**IB0**	**IW0**	**ID0**
Vstupy		pole BOOL o rozměru 1..POCET_VSTUPU ()		Pole logických proměnných obsahující přímý přístup na fyzické digitální vstupy automatu. Proměnné začínají na adrese hned za analogovými vstupy.	r	-		**200**			**IB200**	**IW100**	**ID50**
Vystupy_analog		pole INT o rozměru 1..POCET_VYSTUPU_ANALOG		Pole INT čísel, jež obsahují přímý přístup na analogové výstupy. Pozor na obsazení adres kartami DALI, EnOcean a některými dalšími.	r	w		**512**			**QB80**	**QW0**	**QD0**
Vystupy		pole BOOL o rozměru 1..POCET_VYSTUPU ()		Pole logických proměnných obsahující přímý přístup na fyzické digitální výstupy automatu. Proměnné začínají na adrese hned za analogovými výstupy.	r	w		**712**			**QB200**	**QW100**	**QD50**
KONSTANTNÍ ADRESY													
Počty karet													
obraz_POCET_KARET_VSTUPU	**ADDRESSOF_DIGITCARDSCOUNT**	1 BYTE	**16**	Položka udávající počet karet digitálních vstupů použitých v programu. Jedná se o obraz konstanty systému.	r	-	1	**12288**	**12288**		**MB0**	MW0	MD0
obraz_POCET_KARET_VYSTUPU	**ADDRESSOF_DIGITCARDSCOUNT**	1 BYTE	**10**	Položka udávající počet karet digitálních výstupů použitých v programu. Jedná se o obraz konstanty systému.	r	-	1	**12288**	**12288**		**MB1**	MW0	MD0
obraz_POCET_KARET_VSTUPU_ANALOG	**ADDRESSOF_ANALOGCARDSCOUNT**	1 BYTE	**6**	Položka udávající počet karet analogových vstupů použitých v programu. Jedná se o obraz konstanty systému.	r	-	1	**12289**	**12289**		**MB2**	MW1	MD0
obraz_POCET_KARET_VYSTUPU_ANALOG	**ADDRESSOF_ANALOGCARDSCOUNT**	1 BYTE	**6**	Položka udávající počet karet analogových výstupů použitých v programu. Jedná se o obraz konstanty systému.	r	-	1	**12289**	**12289**		**MB3**	MW1	MD0
Reálné počty v automatu													
RealnyPocetKaretVstupu	**ADDRESSOF_REALDIGITCARDSCOUNT**	1 BYTE		Proměnná udávající kolik karet digitálních vstupů je ve skutečnosti zařazeno v systému (PLC). Tento počet musí být menší nebo rovno než obraz_POCET_KARET_VSTUPU. Pokud ne, pak se nahlásí v PLC chyba.	r	-	1	**12290**	**12290**		**MB4**	MW2	MD1
RealnyPocetKaretVystupu	**ADDRESSOF_REALDIGITCARDSCOUNT**	1 BYTE		Proměnná udávající kolik karet digitálních výstupů je ve skutečnosti zařazeno v systému (PLC). Tento počet musí být menší nebo rovno než obraz_POCET_KARET_VYSTUPU. Pokud ne, pak se nahlásí v PLC chyba.	r	-	1	**12290**	**12290**		**MB5**	MW2	MD1
RealnyPocetKaretAnalogVstup	**ADDRESSOF_REALANALOGCARDSCOUNT**	1 BYTE		Proměnná udávající kolik karet analogových vstupů je ve skutečnosti zařazeno v systému (PLC). Tento počet musí být menší nebo rovno než obraz_POCET_KARET_VSTUPU_ANALOG. Pokud ne, pak se nahlásí v PLC chyba.	r		1	**12291**	**12291**		**MB6**	MW3	MD1
RealnyPocetKaretAnalogVystup	**ADDRESSOF_REALANALOGCARDSCOUNT**	1 BYTE		Proměnná udávající kolik karet analogových výstupů je ve skutečnosti zařazeno v systému (PLC). Tento počet musí být menší nebo rovno než obraz_POCET_KARET_VYSTUPU_ANALOG. Pokud ne, pak se nahlásí v PLC chyba.	r	-	1	**12291**	**12291**		**MB7**	MW3	MD1
RealnyPocetVstupu	**ADDRESSOF_REALDIGITINCOUNT**	1 WORD		Položka udávající skutečný počet digitálních vstupů zařazených v PLC.	r	-	2	**12292**	**12292**		MB8	**MW4**	MD2
RealnyPocetVystupu	**ADDRESSOF_REALDIGITOUTCOUNT**	1 WORD		Položka udávající skutečný počet digitálních výstupů zařazených v PLC.	r		2	**12293**	**12293**		MB10	**MW5**	MD2
RealnyPocetVstupuAnalog	**ADDRESSOF_REALANALOGINCOUNT**	1 WORD		Položka udávající skutečný počet analogových vstupů zařazených v PLC.	r		2	**12294**	**12294**		MB12	**MW6**	MD3
RealnyPocetVystupuAnalog	**ADDRESSOF_REALANALOGOUTCOUNT**	1 WORD		Položka udávající skutečný počet analogových výstupů zařazených v PLC.	r	-	2	**12295**	**12295**		MB14	**MW7**	MD3
RealnyPocetKaretDALI	**ADDRESSOF_REALCOUNTSDALI**	1 BYTE		Proměnná udávající kolik DALI karet je ve skutečnosti zařazeno v systému (PLC). Tento počet musí být menší nebo rovno než 1. Pokud ne, pak se nahlásí v PLC chyba.	r	-	1	**12295**	**12295**		**MB16**	MW8	MD4
RealnyPocetSvetelDALI	**ADDRESSOF_REALCOUNTSDALI**	1 BYTE		Položka udávající skutečný počet naadresovaných světel v systému DALI v PLC - velikost je menší nebo rovno 64.	r	-	1	**12296**	**12296**		**MB17**	MW8	MD4
RealnyPocetKaretEnOcean	**ADDRESSOF_REALCOUNTSENOCEAN**	1 BYTE		Proměnná udávající kolik EnOcean karet je ve skutečnosti zařazeno v systému (PLC). Tento počet musí být menší nebo rovno 1. Pokud ne, pak se nahlásí v PLC chyba.	r	-	1	**12297**	**12297**		**MB18**	MW9	MD4
Rezerva	**Rezerva**	1 BYTE		Rezerva	-	-	1	**12297**	**12297**		**MB19**	MW9	MD4
Počet plánů													
obraz_POCET_PLANU	**ADDRESSOF_REALCOUNTSOFPLANS**	1 BYTE	24	Proměnná udávající s kolika plány je počítáno v automatu PLC (Topné a časové plány).	r	-	1	**12298**	**12298**		**MB20**	MW10	MD5
Rezerva	**Rezerva**	19 BYTE		Rezerva	-	-	19	**12298**	**12307**		MB21	MW10	MD5
Řízení modelu													

Nacteni_Modelu	**ADDRESSOF_CONTROLOFMODEL**	1 BOOL		Požadavek na načtení modelu z MODBUS komunikace - z vizualizace.	r	w	0	**12308**	**12308**		MB40	**MX20.0**	MD10
Vypis_Modelu	**ADDRESSOF_CONTROLOFMODEL**	1 BOOL		Požadavek na načtení modelu do MODBUS komunikace - do vizualizace.	r	w	0	**12308**	**12308**		MB40	**MX20.1**	MD10
Blok_Rizeni	**ADDRESSOF_CONTROLOFMODEL**	1BOOL		Požadavek na blokaci řídícího algoritmu.	r	w	0	**12308**	**12308**		MB40	**MX20.2**	MD10
Rezerva	**Rezerva**	13 BOOL		Rezerva	-	-	0	**12290**	**12309**		MB40	**MX20**	MD10
Chyby v automatu													
Rezerva	**Rezerva**	1 BOOL		Rezerva	-	-	0	**12310**	**12310**		MB44	**MX22.0**	MD11
Kontrola_OK	**ADDRESSOF_SYSTEMERRORS**	1 BOOL		Kontrola konfigurace automatu proběhla úspěšně.	r	-	0	**12310**	**12310**		MB44	**MX22.1**	MD11
Chyba_modulu	**ADDRESSOF_SYSTEMERRORS**	1 BOOL		Nastala chyba konfigurace a je zastaveno řízení.	r		0	**12310**	**12310**		MB44	**MX22.2**	MD11
Chyba_konfigurace	**ADDRESSOF_SYSTEMERRORS**	1 BOOL		Chybně zadána konfigurace HW systému.	r		0	**12310**	**12310**		MB44	**MX22.3**	MD11
nekompatibilni_karta	**ADDRESSOF_SYSTEMERRORS**	1 BOOL		Nalezena nekompatibilní karta v automatu.	r	-	0	**12310**	**12310**		MB44	**MX22.4**	MD11
prilis_DI	**ADDRESSOF_SYSTEMERRORS**	1 BOOL		Příliš digitálních vstupů v automatu.	r	-	0	**12310**	**12310**		MB44	**MX22.5**	MD11
prilis_DO	**ADDRESSOF_SYSTEMERRORS**	1 BOOL		Příliš digitálních výstupů v automatu.	r		0	**12310**	**12310**		MB44	**MX22.6**	MD11
prilis_AI	**ADDRESSOF_SYSTEMERRORS**	1 BOOL		Příliš analogových vstupů v automatu.	r	-	0	**12310**	**12310**		MB44	**MX22.7**	MD11
prilis_AO	**ADDRESSOF_SYSTEMERRORS**	1 BOOL		Příliš analogových výstupů v automatu.	r	-	0	**12310**	**12310**		MB44	**MX22.8**	MD11
prilis_DALI	**ADDRESSOF_SYSTEMERRORS**	1 BOOL		V automatu je více než jedna karta DALI.	r		0	**12310**	**12310**		MB44	**MX22.9**	MD11
prilis_EnOcean	**ADDRESSOF_SYSTEMERRORS**	1 BOOL		V automatu je více než jedna karta EnOcean.	r		0	**12310**	**12310**		MB44	**MX22.10**	MD11
Rezerva	**Rezerva**	3 BOOL		Rezerva	-	-	0	**12310**	**12310**		MB44	**MX22**	MD11
Chyba_pozice_DALI	**ADDRESSOF_SYSTEMERRORS**	1 BOOL		Karta DALI je na nesprávné pozici v konfiguraci automatu. Musí být za všemi vstupními analogovými kartami.	**r**	-	0	**12310**	**12310**		MB44	**MX22.14**	MD11
Rezerva	**Rezerva**	1 BOOL		Rezerva	-	-	0	**12310**	**12311**		MB44	**MX22**	MD11
nekompatibilni_karta_pozice	**ADDRESSOF_COMPATIBLEERRORPOSITION**	260 string		Pozice nekompatibilní karty v konfiguraci automatu.	**r**	-	260	**12312**	**12441**		MB48	**MW24**	MD12
Rezerva	**Rezerva**			Rezerva	-	-	0	**12442**	**12576**		MB308	**MW154**	MD77
Čas v automatu													

rok	**ADDRESSOF_TIMEYEAR**	1 WORD		Rok v automatu	**r**	-	2	**12577**	**12577**		MB578	**MW289**	MD144
mesic	**ADDRESSOF_TIMEMONTH**	1 WORD		Měsíc v automatu.	**r**	-	2	**12578**	**12578**		MB580	**MW290**	MD145
den	**ADDRESSOF_TIMEDAY**	1 WORD		Den v automatu (den v měsíci).	r	-	2	**12579**	**12579**		MB582	**MW291**	MD145
hodina	**ADDRESSOF_TIMEHOUR**	1 WORD		Hodina v automatu.	r	-	2	**12580**	**12580**		MB584	**MW292**	MD146
minuta	**ADDRESSOF_TIMEMINUTE**	1 WORD		Minuta v automatu.	**r**	-	2	**12581**	**12581**		MB586	**MW293**	MD146
sekunda	**ADDRESSOF_TIMESECOND**	1 WORD		Sekunda v automatu	**r**	-	2	**12582**	**12582**		MB588	**MW294**	MD147
Doba pro dvojklik													
Doba_Double	**ADDRESSOF_TIMEFORDOUBLE**	1 BYTE		Proměnná udávající čas v desetinnách sekundy, kdy se čeká, zda je signál kliknutí, dvojkliknutí nebo držení.	r	w	1	**12583**	**12583**		**MB590**	MW295	MD147
Rezerva	**Rezerva**	1 BYTE		Rezerva	-	-	1	**12583**	**12583**		MB591	MW295	MD147
EnOcean nastavení													
EnOcean_Status	**ADDRESSOF_ENOCEAN_STATUS**	1 BYTE		Požadavek pro hledání ID komponent. (1 = požadavek aktivní, 0 = požadavek neaktivní)	r	w	1	**12584**	**12584**		MB592	**MX296**	MD148
EnOcean_FindedNoButton	**ADDRESSOF_ENOCEAN_FOUNDNRBUTTON**	1 BYTE		Číslo konfigurovaného tlačítka.	r	w	1	**12585**	**12585**		**MB594**	MW297	MD148
Rezerva	**Rezerva**	1 BYTE		Rezerva	-	-	1	**12585**	**12585**		MB595	MW297	MD148
EnOcean_FindedID	**ADDRESSOF_ENOCEAN_FOUNDID**	1 DOUBLE		ID konfigurované EnOcean komponenty.	r	w	4	**12586**	**12587**		MB596	MW298	**MD149**
Komponenta Východ a západ slunce													
obraz_VZS_Status	**ADDRESSOF_SUNSYSTEM**	1 BOOL		Informace o stavu funkce VZS	r	-	0	**12588**	**12588**		MB600	**MX300.0**	MD150
VZS_Simulace	**ADDRESSOF_SUNSYSTEM**	1 BOOL		Simulace zapnutí a vypnutí funkce VZS.	-	w	0	**12588**	**12588**		MB600	**MX300.1**	MD150
VZS_SimulaceVychod	**ADDRESSOF_SUNSYSTEM**	1 BOOL		Simulace východu slunce.	-	w	0	**12588**	**12588**		MB600	**MX300.2**	MD150
VZS_SimulaceZapad	**ADDRESSOF_SUNSYSTEM**	1 BOOL		Simulace západu slunce.	-	w	0	**12588**	**12588**		MB600	**MX300.3**	MD150
obraz_VZS_letni_cas	**ADDRESSOF_SUNSYSTEM**	1 BOOL		Proměnná indikující letní čas.	r	w	0	**12588**	**12588**		MB600	**MX300.4**	MD150
Rezerva	**Rezerva**	1 BYTE		Rezerva	-	-	1	**12588**	**12588**		MB600	MX300	MD150
VZS_casova_zona	**ADDRESSOF_ZONETIME**	1 BYTE		Proměnná s hodnotou časové zóny.	r	w	1	**12589**	**12589**		MB602	**MW301**	MD150
VZS_zemepisna_delka	**ADDRESSOF_LONGITUDE**	1 DOUBLE		Hodnota zeměpisné délky, kde se automat nachází.	r	w	4	**12590**	**12591**		MB604	MW302	**MD151**
VZS_zemepisna_sirka	**ADDRESSOF_LATITUDE**	1 BOOL		Hodnota zeměpisné šířky, kde se automat nachází.	r	w	4	**12592**	**12593**		MB608	MW304	**MD152**
VZS_vychod_hodina	**ADDRESSOF_SUNRISEHOUR**	1 WORD		Povolení spouštění algoritmů řízení DALI svítidel, také zároveň indikuje zařazení DALI karty do systému. Odpovídá nastavení z vizualizace "Konfigurace systemu".	r	-	2	**12594**	**12594**		MB612	**MW306**	MD153
VZS_vychod_minuta	**ADDRESSOF_SUNRISEMINUTE**	1 WORD		Povolení spouštění algoritmů řízení DALI svítidel, také zároveň indikuje zařazení DALI karty do systému. Odpovídá nastavení z vizualizace "Konfigurace systemu".	r	-	2	**12595**	**12595**		MB614	**MW307**	MD153
VZS_zapad_hodina	**ADDRESSOF_SUNSETHOUR**	1 WORD		Povolení spouštění algoritmů řízení DALI svítidel, také zároveň indikuje zařazení DALI karty do systému. Odpovídá nastavení z vizualizace "Konfigurace systemu".	r	-	2	**12596**	**12596**		MB616	**MW308**	MD154

VZS_zapad_minuta	**ADDRESSOF_SUNSETMINUTE**	1 WORD		Povolení spouštění algoritmů řízení DALI svítidel, také zároveň indikuje zařazení DALI karty do systému. Odpovídá nastavení z vizualizace "Konfigurace systemu".	r	-	2	**12597**	**12597**		MB618	**MW309**	MD154
PROMĚNNÉ ADRESY													
obraz_vstupy	**ADDRESSOF_DIGITINPUTS**	Pole BOOL délky POCET_VSTUPU		Pole obsahujici hodnoty binárních vstupů, které vyhodnocuje program řízení modelu.	r	-	16	**12788**	**12795**		**MB1000**	MW500	MD250
DI_Prirazeni_Typy	**ADDRESSOF_TYPESOFDI**	POLE BYTE délky POCET_VSTUPU		Pole obsahující typ vstupních komponent.	r	w	128	**12796**	**12859**		**MB1016**	MW508	MD254
DI_Prirazeni_Podtypy	**ADDRESSOF_SUBTYPESOFDI**	Pole BOOL délky POCET_VSTUPU		Pole obsahující podtyp vstupních komponent.	r	w	16	**12860**	**12867**		**MB1144**	MW572	MD286
Negace_vstupu	**ADDRESSOF_NEGATIONOFDI**	Pole BOOL délky POCET_VSTUPU		Pole obsahující informaci, zda se má vstupní hodnota pro program řízení modelu znegovat.	r	w	16	**12868**	**12875**		**MB1160**	MW580	MD290
simulovane_vstupy	**ADDRESSOF_SIMULATIONDI**	Pole BOOL délky POCET_VSTUPU		Pole sloužící pro simulaci vstupů z vizualizace.	-	w	16	**12876**	**12883**		**MB1176**	MW588	MD294
obraz_Funkcni_vystupy	***ADDRESSOF_DIGITOUTPUTS***	*Pole BOOL délky POCET_VYSTUPU*		*Pole obsahující hodnoty binárních výstupů, které vyhodnotil program řízení modelu.*	*r*	-	*10*	**12884**	**12888**		**MB1192**	MW596	MD298
DO_Prirazeni_Typy	***ADDRESSOF_TYPESOFDO***	*POLE BYTE délky POCET_VYSTUPU*		*Pole obsahující typ výstupních komponent.*	*r*	*w*	*80*	**12889**	**12928**		**MB1202**	MW601	MD302
DO_Prirazeni_podtypy	***ADDRESSOF_SUBTYPESOFDO***	*Pole BOOL délky POCET_VYSTUPU*		*Pole obsahující podtyp výstupních komponent.*	*r*	*w*	*10*	**12929**	**12933**		**MB1282**	MW641	MD322
Negace_vystupu	***ADDRESSOF_NEGATIONOFDO***	*Pole BOOL délky POCET_VYSTUPU*		*Pole obsahující informaci, zda se má výstupní hodnota vypočtená programem řízení znegovat.*	*r*	*w*	*10*	**12934**	**12938**		**MB1292**	MW646	MD323
Simulovane_Vystupy	***ADDRESSOF_SIMULATIONDO***	*Pole BOOL délky POCET_VYSTUPU*		*Pole sloužící pro simulaci výstupů z vizualizace.*	-	*w*	*10*	**12939**	**12943**		**MB1302**	MW651	MD327
AI_Vstupy	**ADDRESSOF_ANALOGINPUTS**	Pole WORD délky POCET_VSTUPU_ANALOG		Pole obsahujici hodnoty analogových vstupů, které vyhodnocuje program řízení modelu.	r	-	48	**12944**	**12967**		MB1312	**MW656**	MD328
AI_PrirazeneTypy	**ADDRESSOF_TYPESOFAI**	POLE BYTE délky POCET_VSTUPU_ANALOG		Pole obsahující typ vstupních analogových komponent.	r	w	24	**12968**	**12979**		**MB1360**	MW680	MD340
AI_SimulovaneVstupy	**ADDRESSOF_SIMULATIONAI**	Pole WORD délky POCET_VSTUPU_ANALOG		Pole sloužící pro simulaci analogových vstupů z vizualizace.	-	w	48	**12980**	**13003**		MB1384	**MW692**	MD346
AO_PrirazeneTypy	**ADDRESSOF_TYPESOFAO**	Pole BYTE délky POCET_VYSTUPU_ANALOG		Pole obsahující typ výstupních analogových komponent.	r	w	24	**13004**	**13015**		**MB1432**	MW716	MD358
AO_SimulovaneVystupy	**ADDRESSOF_SIMULATIONAO**	Pole WORD délky POCET_VYSTUPU_ANALOG		Pole sloužící pro simulaci analogových výstupů z vizualizace.	-	w	48	**13016**	**13039**		MB1456	**MW728**	MD364
Typ_DO_matice	***ADDRESSOF_OSVVNT_TYPEOFMATRIX***	*1 BYTE*		*Promenná, která značí typ matice ve sdílené matici.*	*r*	*w*	*2*	**13040**	**13040**		**MB1504**	MW752	MD376
obraz_DO_MapaSdilena	***ADDRESSOF_OSVVNT_SHAREDMATRIX***	*POLE BYTE délky POCET_VYSTUPU* POCET_VSTUPU*		*Sdílená matice pro chování DO_komponent.*	*r*	*w*	*1280*	**13041**	**13680**		**MB1506**	MW753	MD378
DO_MapaZpozdeniON	***ADDRESSOF_OSVVNT_DELAYONTIME***	*Pole WORD délky POCET_VYSTUPU*		*Pole obsahující časy v sekundách zpožděného zapnutí pro výstupní komponenty.*	*r*	*w*	*160*	**13681**	**13760**		MB2786	**MW1393**	MD698
DO_MapaZpozdeniOFF	***ADDRESSOF_OSVVNT_DELAYOFFTIME***	*Pole WORD délky POCET_VYSTUPU*		*Pole obsahující časy v sekundách zpožděného vypnutí pro výstupní komponenty.*	*r*	*w*	*160*	**13761**	**13840**		MB2946	**MW1473**	MD738
obraz_FantomStatus+FantomSwitch	***ADDRESSOF_FANTOMSTATUS***	*Pole BOOL délky 16*		*Pole sloužící pro ovládání funkce fantom z vizualizace.*	*r*	*w*	*2*	**13841**	**13841**		MB3106	**MW1553**	MD778
obraz_OSV_MapaFantom	**ADDRESSOF_FANTOMMAPOSV**	Pole BOOL délky POCET_VYSTUPU		Pole obsahujici informaci, která světla jsou ovládána funkcí Fantom.	r	w	10	**13842**	**13846**		**MB3108**	MW1554	MD777
obraz_DALI_MapaFantom	**ADDRESSOF_FANTOMMAPDALI**	POLE BOOL délky POCET_VYSTUPU		Pole obsahujici informaci, která DALI světla jsou ovládána funkcí Fantom.	r	w	8	**13847**	**13850**		**MB3118**	MW1559	MD781
obraz_DALI_GrupaMapaFantom	**ADDRESSOF_FANTOMMAPDALI_GROUP**	Pole BOOL délky POCET_VYSTUPU		Pole obsahujici informaci, které DALI skupiny jsou ovládány funkcí Fantom.	r	w	2	**13851**	**13851**		**MB3126**	MW1563	MD783
obraz_OSV_MapaFantomImpuls	**ADDRESSOF_FANTOMMAPIMPULSE**	Pole BOOL délky POCET_VSTUPU		Matice pro ovládání funkce Fantom - impulsní ovládání.	r	w	16	**13852**	**13859**		**MB3128**	MW1564	MD782
obraz_OSV_MapaFantomDoubleImpuls	**ADDRESSOF_FANTOMMAPDOUBLE**	Pole BOOL délky POCET_VSTUPU		Matice pro ovládání funkce Fantom - dvojklik ovládání.	r	w	16	**13860**	**13867**		**MB3144**	MW1572	MD786
obraz_OSV_MapaFantomVypinac	***ADDRESSOF_FANTOMMAPSWITCH***	*Pole BOOL délky POCET_VSTUPU*		Matice pro ovládání funkce Fantom - úrovňové ovládání.	*r*	*w*	*16*	**13868**	**13875**		**MB3160**	MW1580	MD790
CP_FantomMapa	***ADDRESSOF_TPANDFANTOM_MAP***	*1 BYTE*		*Proměnná označující, který časový plán ovládá funkci Fantoma.*	*r*	*w*	*2*	**13876**	**13876**		**MB3176**	MW1588	MD794
VZS_FantomMapa_XXX	***ADDRESSOF_SUNANDFANTOM_MAP***	*Pole BOOL délky 16*		*Proměnné pro spojení funkce východu a západu slunce s funkcí Fantom.*	*r*	*w*	*2*	**13877**	**13877**		MB3178	**MW1589**	MD796
Typ_XXX_CP_matice	***ADDRESSOF_TIMEPLANS_TYPEOFMATRIX***	*1 BYTE*		*Promenná, která značí typ matice ve sdílené matici.*	*r*	*w*	*2*	**13878**	**13878**		**MB3180**	MW1590	MD795
obraz_XXX_CP_MapaSdilena	***ADDRESSOF_TIMEPLANS_SHAREDMATRIX***	*Pole BYTE délky POCET_VYSTUPU*		*Sdílená matice pro chování DO_komponent dle časových plánů.*	*r*	*w*	*80*	**13879**	**13918**		**MB3182**	MW1591	MD797
obraz_CP_SwitchON	**ADDRESSOF_TP_SIMULATION**	*Pole BOOL délky POCET_BYTU_PLANU*		*Proměnné sloužící pro ovládání časových plánů z vizualizace.*	r	w	4	**13919**	**13920**		**MB3262**	MW1631	MD817
CPTEP_typPlanu	**ADDRESSOF_TPANDHEAT_TYPEOFMATRIX**	1 BYTE		*Promenná, která značí typ matice ve sdílené matici.*	r	w	2	**13921**	**13921**		**MB3266**	MW1633	MD818
CPTEP_sdilena	**ADDRESSOF_TPANDHEAT_SHAREDMATRIX**	Pole datového typu TEPLOTNI_PLAN délky		*Sdílená matice pro uložení či načtení časových plánů pro vizualizaci.*	r	w	4032	**13922**	**15937**		**MB3268**	MW1634	MD817
TEP_prirazeni_senzoru	**ADDRESSOF_HEATING_ALLOCATEDSENSORS**	Pole WORD délky POCET_VYSTUPU		Pole označující spojení termostatů a ventilů.	r	w	160	**15938**	**16017**		MB7300	**MW3650**	MD1825
TEP_Termostat_Kotel	**ADDRESSOF_HEATING_ALLOCATEDTHERMOSTATANDFURNACE**	Pole WORD délky POCET_VYSTUPU		Pole označující spojení termostatů a kotlů.	r	w	160	**16018**	**16097**		MB7460	**MW3730**	MD1865
TEP_MapaTopnychPlanu	***ADDRESSOF_HEATING_HEATINGPLANTOTHERMOSTAT***	*Pole BYTE délky POCET_VSTUPU_ANALOG*		Pole označující spojení termostatů a topných plánů.	*r*	*w*	*24*	**16098**	**16109**		**MB7620**	MW3810	MD1905

TEP_ManualTeploty_Vstup	*ADDRESSOF_HEATING_INPUTFORMANUALTEMPERATURE*	*Pole WORD délky POCET_VSTUPU_ANALOG*		*Pole označující spojení termostatů a manuálních vstupů.*	*r*	*w*	*48*	**16110**	**16133**		MB7644	**MW3822**	MD1911
TEP_typ_regulace	*ADDRESSOF_HEATING_TYPEOFREGULATION*	*Pole BYTE délky POCET_VYSTUPU*		*Pole obsahující typy regulací ventilů.*	*r*	*w*	*80*	**16134**	**16173**		**MB7692**	MW3846	MD1923
TEP_typ_termoventil	*ADDRESSOF_HEATING_TYPEOFVALVE*	*Pole BYTE délky POCET_VYSTUPU*		*Pole obsahující typy ventilů (zda je otevřen v nule nebo v jedničce).*	*r*	*w*	*80*	**16174**	**16213**		**MB7772**	MW3886	MD1943
obraz_TEP_povoleni_regulace	*ADDRESSOF_HEATING_ALLOWOFREGULATION*	*Pole BOOL délky POCET_BYTU_VSTUPU_ANALOG*		*Pole označující povolení regulace u termostatu pro vizualizaci.*	*r*	*w*	*4*	**16214**	**16215**		**MB7852**	MW3926	MD1963
obraz_TEP_Manual_Teploty	ADDRESSOF_HEATING_MANUALSTEMPERATURE	Pole WORD délky POCET_VSTUPU_ANALOG		Pole sloužící pro manuální vstupy z vizualizace.	-	w	48	**16216**	**16239**		MB7856	**MW3928**	MD1964
obraz_HP_SwitchON	ADDRESSOF_HEATING_PLANON	POLE BYTE délky POCET_VYSTUPU		Proměnná sloužící pro ovládání topných plánů z vizualizace.	r	w	4	**16240**	**16241**		**MB7904**	MW3952	MD1976
obraz_DALI_svetla_stav	ADDRESSOF_DALI_LIGHTSTATUS	Pole BOOL délky 64		Pole obsahující stavy DALI světel.	r	w	8	**16242**	**16245**		**MB7908**	MW3954	MD1977
obraz_DALI_ActualScene	ADDRESSOF_DALI_ACTUALSINGLESCENES	Pole BYTE délky 64		Pole obsahující aktuální čísla scén u jednotlivých DALI scén.	r	w	64	**16246**	**16277**		**MB7916**	MW3958	MD1979
obraz_DALI_ActualSceneGroup	ADDRESSOF_DALI_ACTUALGROUPSCENES	Pole BYTE délky 16		Pole obsahující aktuální čísla scén u DALI skupin scén.	-	w	16	**16278**	**16285**		**MB7980**	MW3990	MD1995
obraz_DALI_Scenes	*ADDRESSOF_DALI_SCENEVALUES*	*Pole BYTE délky 64*8*		*Pole obsahující hodnoty binárních výstupů, které vyhodnotil program řízení modelu.*	*r*	*w*	*512*	**16286**	**16541**		**MB7996**	MW3998	MD1999
DALI_pouziteAdresy	*ADDRESSOF_DALI_USEDADDRESS*	*Pole BOOL délky 64*		*Pole označující, na kterých adresách jsou přiřazena světla.*	*r*	-	*8*	**16542**	**16545**		**MB8508**	MW4254	MD2127
Typ_DALI_matice	*ADDRESSOF_DALI_TYPEOFMATRIX*	*1 BYTE*		*Promenná, která značí typ matice ve sdílené matici.*	*r*	*w*	*2*	**16546**	**16546**		**MB8516**	MW4258	MD2129
obraz_DALI_MapaSdilena	ADDRESSOF_DALI_SHAREDMATRIX	Pole BOOL délky 64*POCET_VSTUPU		*Sdílená matice pro chování DALI komponent.*	r	w	1024	**16547**	**17058**		**MB8518**	MW4259	MD2131
DALI_Grupy_prirazeni	ADDRESSOF_DALI_ALLOCATEDGROUPS	POLE BOOL délky 64*16		Pole přiřazující DALI světla do skupin.	r	w	128	**17059**	**17122**		**MB9542**	MW4771	MD2387
DALI_SimSingle	ADDRESSOF_DALI_SIMULATIONSINGLE	Pole BOOL délky 64		Pole sloužící k ovládání DALI světel z vizualizace (zapnuto-vypnuto)	-	w	8	**17123**	**17126**		**MB9670**	MW4835	MD2419
DALI_SimGroupy	ADDRESSOF_DALI_SIMULATIONGROUP	Pole BOOL délky 16		Pole sloužící k ovládání DALI skupin z vizualizace (zapnuto-vypnuto)	-	w	2	**17127**	**17127**		**MB9678**	MW4839	MD2421
DALI_SimSceneSingle	ADDRESSOF_DALI_SIMULATIONSCENESINGLE	Pole BOOL délky 64		Pole sloužící k ovládání DALI světel z vizualizace (změna scény)	-	w	8	**17128**	**17131**		**MB9680**	MW4840	MD2420
DALI_SimSceneGroupy	*ADDRESSOF_DALI_SIMULATIONSCENEGROUP*	Pole BOOL délky 16		Pole sloužící k ovládání DALI skupin z vizualizace (změna scény)	-	*w*	*2*	**17132**	**17132**		**MB9688**	MW4844	MD2422
DALI_SimSceneSingleLong	*ADDRESSOF_DALI_SIMULATIONLONGCLICK*	Pole BOOL délky 64		Pole sloužící k ovládání DALI světel z vizualizace (zapnuto-vypnuto)	-	*w*	*8*	**17133**	**17136**		**MB9690**	MW4845	MD2424
DALI_SimSceneGroupyLong	*ADDRESSOF_DALI_SIMULATIONLONGCLICKGROUP*	Pole BOOL délky 16		Pole sloužící k ovládání DALI skupin z vizualizace (změna jasu)	-	*w*	*2*	**17137**	**17137**		**MB9698**	MW4849	MD2426
DALI_MapaZpozdeniON	*ADDRESSOF_DALI_DELAYONTIME*	*Pole WORD délky 64*		*Pole obsahující časy v sekundách zpožděného zapnutí pro DALI světla.*	*r*	*w*	*128*	**17138**	**17201**		MB9700	**MW4850**	MD2425
DALI_MapaZpozdeniOFF	*ADDRESSOF_DALI_DELAYOFFTIME*	*Pole WORD délky 64*		*Pole obsahující časy v sekundách zpožděného vypnutí pro DALI světla.*	*r*	*w*	*128*	**17202**	**17265**		MB9828	**MW4914**	MD2457
DALI_GrupaZpozdeniON	ADDRESSOF_DALIGROUP_DELAYONTIME	*Pole WORD délky 16*		*Pole obsahující časy v sekundách zpožděného zapnutí pro DALI skupiny.*	r	w	32	**17266**	**17281**		MB9956	**MW4978**	MD2489
DALI_GrupaZpozdeniOFF	ADDRESSOF_DALIGROUP_DELAYOFFTIME	*Pole WORD délky 16*		*Pole obsahující časy v sekundách zpožděného vypnutí pro DALI skupiny.*	r	w	32	**17282**	**17297**		MB9988	**MW4994**	MD2497
EnOcean_PoleCiselTlac	ADDRESSOF_ENOCEAN_ARRAYOFNRBUTTONS	Pole BYTE délky POCET_VSTUPU		Pole označující číslo EnOcean tlačítek, které jsou přiřazeny ke vstupům.	r	w	128	**17298**	**17361**		**MB10020**	MW5010	MD2505
EnOcean_PoleID	ADDRESSOF_ENOCEAN_ARRAYOFIDS	Pole DOUBLE délky POCET_VSTUPU		Pole označující ID EnOcean komponenty, které jsou přiřazeny ke vstupům.	r	w	512	**17362**	**17617**		MB10148	MW5074	**MD2537**
Simulovane_Vystupy_Drzeni	ADDRESSOF_SIMULATIONDOLONG	Pole BOOL délky POCET_VYSTUPU		Pole pro simulaci drzeni pro výstupní komponenty	-	w	10	**17618**	**17622**		**MB10660**	MW5330	MD2665
TEP_propojeni	ADDRESSOF_HEATING_ALOCATEDBLOCKSENZORES	Pole BOOL délky POCET_VSTUPU_ANALOG*POCET_VSTUPU		Matice přiřazených ventilů k termostatům	r	w	384	**17623**	**17814**		**MB10670**	MW5335	MD2669