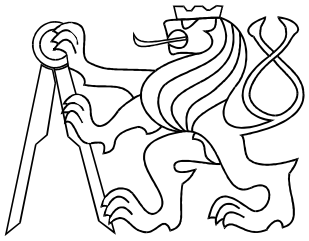
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky



Diplomová práce

Řídicí systém chovu ryb

Praha, 2004 **Jan Kelbel**

Prohlášení

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

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne: ....

Podpis

Poděkování

Na tomto místě bych rád poděkoval lidem, kteří mi při vytváření této práce pomáhali nalézt správnou cestu. Především vedoucímu diplomové práce Ing. Pavlu Burgetovi, dále pak Ing. Petru Smolíkovi za rady týkající se knihovny ORTE.

Také bych chtěl poděkovat svým rodičům za podporu, které se mi od nich dostávalo po celou dobu studia.

Abstrakt

Tato práce se zabývá návrhem a realizací softwarové části řídicího systému pro model automatizované rybí farmy. Při tom využívá možností dostupného svobodného softwaru použitím operačního systému Linux, a také knihoven ACE a ORTE. Pro implementaci bylo využito objektových vlastností jazyka C++. Výsledkem je modulární řídicí program, který umožňuje komunikaci s nadřazeným řídicím systémem přes internet.

Abstract

This thesis is concerned with design and implementation of software part for automatic fish breed control. Potency of use of free software in control is exploited, using Linux as operating system and libraries ACE and ORTE. Object oritented features of programming language C++ were used. The result is a modular control program, that provides means for communication with superior control system.

Obsah

1 Úvod

Tato práce je součástí projektu IFiBO [4], jehož cílem je přizpůsobení existující řídicí technologie používané v chemickém průmyslu k použití v automatizovaném provozu pro chov tržních ryb. V rámci tohoto projektu byla navržena a ve Výzkumném ústavu rybářském ve Vodňanech postavena pokusná automatizovaná rybí farma.

Úkolem této práce je vytvoření řídicího systému pro demonstrační přípravek automatizovaného chovu ryb, který představuje zmenšený model automatizované rybí farmy. Záměrem je použít pro realizaci levné komponenty, a prozkoumat tak možnosti snížení nákladů na pořízení rybí farmy.

Tato práce se zabývá především vytvořením programové části řídicího systému. Je tedy zaměřená na použití levného programového vybavení. Slovem levný je zde myšleno použití svobodného programového vybavení šířeného pod GNU General Public Licencí [9] nebo podobnou licencí.

Mezi výhody tohoto řešení patří nulová pořizovací cena, neboť většina softwaru se zmíněnými licencemi se dá pořídit zadarmo či za manipulační poplatek. Oproti tomu u proprietárního programového vybavení se většinou platí za každou použitou instalaci daného produktu. Vlastně se platí pouze za licenci, za „pronájem“. Z podstaty svobodného softwaru plyne i to, že máme k dispozici zdrojové kódy programového vybavení, a nic nám nebrání v jejich úpravě pro vlastní potřebu. Pokud například zkrachuje dodavatel některé části programového vybavení, můžeme na jeho potřebná rozšíření najmout jinou firmu.

Nevýhody plynoucí z použití svobodného softwaru jsou způsobeny relativní absencí open-source projektů zabývajících se automatizací a řízením. Je proto nutné většinu programových komponent řídicího systému vytvořit od nejnižší úrovně představující API operačního systému. Přesto existují projekty vytvářející různé knihovny, které ulehčují vytvoření řídicího systému.

Tento dokument popisuje jednotlivé kroky vytváření řídicího systému. V kapitole 2 je popsán model rybí farmy, který je naším řízeným systémem. Následující kapitola 3 obsahuje popis hardwarových komponent použitých při realizaci řídicího systému. Kapitola 4 pojednává o použitých open-source komponentách pro softwarovou část řídicího programu. Další kapitoly se věnují popisu návrhu řídicího programu (kapitoly 5 a 6), a jeho realizaci (kapitoly 7 a 8) v podobě vytvořených tříd.

2 Řízená technologie

Cílem této práce je vyvinout řídicí systém pro demonstrační model rybí farmy pro intenzivní chov ryb, který byl vytvořen v rámci diplomové práce [1] na Katedře řídicí techniky. Model byl upraven a byly přidány některé funkce. Tato kapitola obsahuje popis řízeného systému a zmiňuje se o možném způsobu řízení tohoto systému.

2.1 Intenzivní chov ryb

Pokud nahlédneme do slovníku, najdeme u slova *intenzivní* tento výklad: „*ekon.* založený na dokonalejší technice, lepší organizaci výrobního procesu“.

Jak píše např. [2], intenzivní chov ryb je poměrně moderní zemědělská technologie, která usiluje o produkci maximálního množství tržních ryb na co nejmenším prostoru při minimálních nákladech na výrobu. Toho se dosahuje právě nasazením automatizace.

Pro intenzivní chov ryb je typické, že se ryby chovají v nádržích, ve kterých se udržují hydrochemické parametry optimální pro život a růst ryb. K tomu se používá voda proudící nádržemi, která tak funguje jako transportní médium tepla, chemických látek rozpuštěných ve vodě i mechanických nečistot.

Mezi hlavní hydrochemické parametry patří:

- Teplota — optimum obvykle mezi 20 a 30 °C.
- pH — optimum 6–8, jeho kolísání se nepříznivě projevuje na zdraví ryb.
- O_2 — optimální množství kyslíku rozpuštěného ve vodě je 5–10 mg/l.
- NH_3 — toxická forma amoniaku. Amoniak je do vody vylučován rybami jako produkt jejich metabolismu. Jeho množství je závislé na teplotě a pH vody, snažíme se ho minimalizovat.

Regulace hodnot základních hydrochemických parametrů v nádržích dovoluje, aby byly tyto nádrže osazeny s vyšší hustotou ryb, něž je běžná v rybnících.

Intenzivní chov zvířat má jistě nedostatky z etického hlediska, ale tím se tato práce nezabývá.

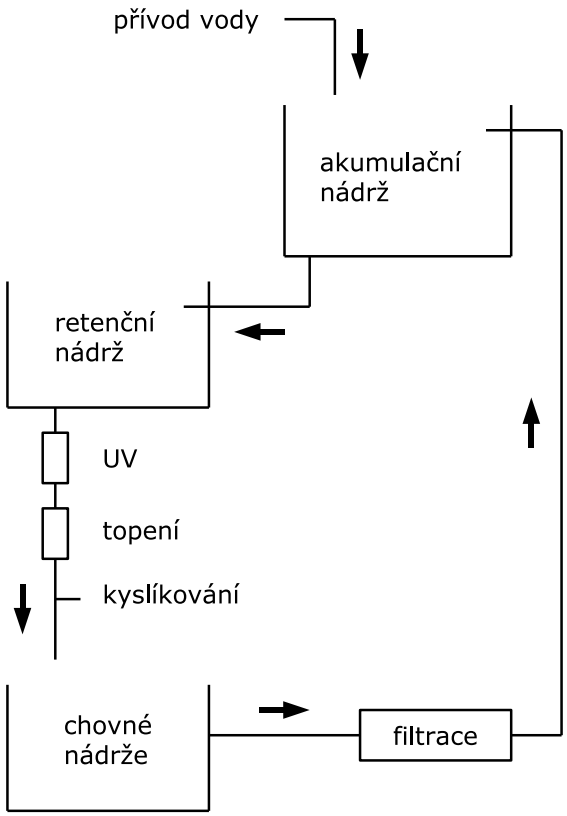
2.2 Technologie rybí farmy

Jak již bylo zmíněno, voda chovnými nádržemi protéká. Podle způsobu nakládání s vodou můžeme rozlišit dva typy rybích farem:

- Průtokový systém. V tomto případě se voda, která odtéká z chovných nádrží do nich již nevrací.
- Recirkulační systém. Zde se odpadová voda z nádrží upravuje a znovu se pouští do chovných nádrží.

Řízený systém je modelem recirkulačního systému intenzivního chovu ryb, a tak se zde budeme zabývat pouze jím. Na obrázku 2.1 vidíme jeho základní schéma.

Chovné nádrže jsou hlavní částí systému. V nich jsou drženy ryby, a tak je zde požadavek na dodržování optimálních podmínek k jejich životu. Z retenční nádrže se do chovných nádrží přivádí čerstvá voda, která se zbavuje mikroorganismů ozařováním UV světlem, a dále se podle potřeby ohřívá a obohacuje kyslíkem.



Obrázek 2.1. Schéma recirkulačního systému

Z chovných nádrží se voda odvádí do filtrů, kde se nejprve zbaví mechanických nečistot a následně v biologickém filtru působením mikroorganismů i některých chemických nečistot. Vyčištěná voda se čerpá do akumulační nádrže, kde se z vnějšího zdroje doplňují úbytky vody způsobené především výparem.

2.3 Řízený systém

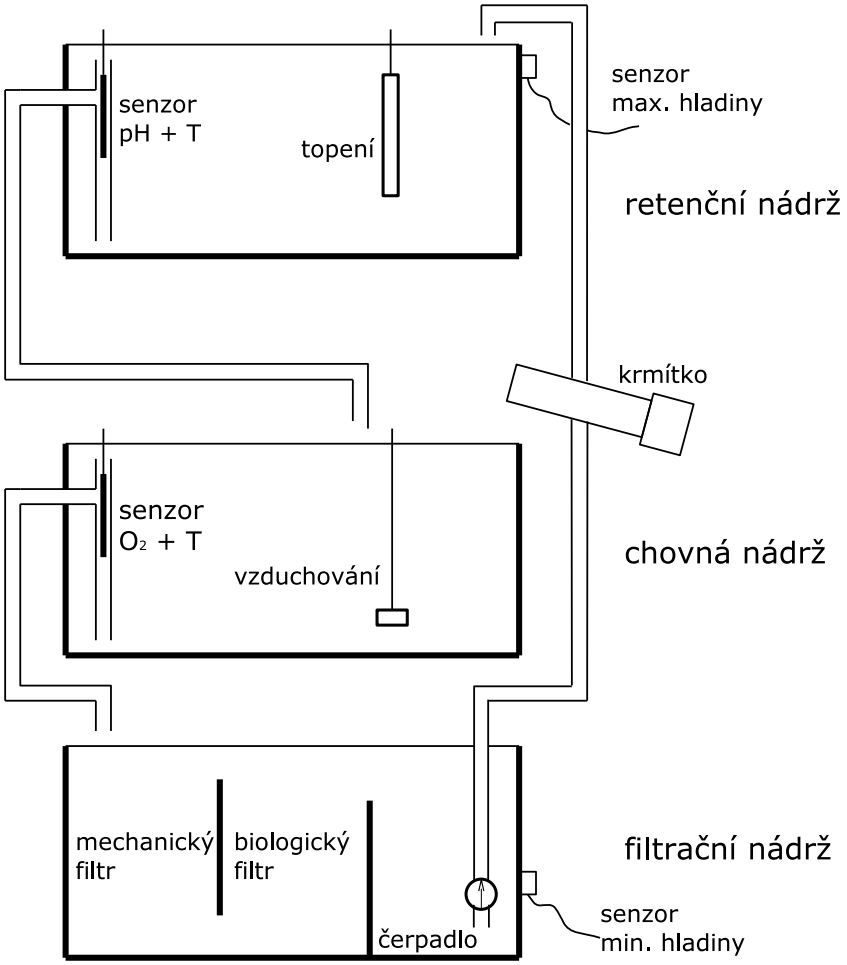
V rámci projektu Ifibo byla ve Výzkumném ústavu rybářském ve Vodňanech vytvořena automatizovaná rybí farma používající recirkulační systém.

Systém, jehož řízení je náplní této práce, je zjednodušeným modelem recirkulačního systému, zmenšenou verzí rybí farmy ve Vodňanech. Má stejné uspořádání nádrží, řídicí systém má stejnou strukturu s použitím sběrnice RS-485. Tento model byl vytvořen v rámci diplomové práce [1] a je schematicky znázorněn na obrázku 2.2. Je složen ze tří nad sebou umístěných nádrží. V horní, retenční nádrži se voda ohřívá. Přepadem odtéká do prostřední, chovné nádrže, kde jsou umístěny ryby. Zde se provádí vzduchování. Z chovné nádrže voda odtéká, také přepadem, do spodní, filtrační nádrže. Ta je příčkami rozdělena na tři části, z nichž první ve směru proudění vody slouží k mechanické filtraci a druhá představuje biologický filtr. Přefiltrovaná voda odtéká do třetí části filtrační nádrže, odkud je čerpadlem vytlačena do horní nádrže.

Senzor teploty a pH je umístěn v odtoku retenční nádrže, v odtoku chovné je senzor teploty a kyslíku. Akční členy jsou realizované běžnými akvaristickými komponentami a jsou spínány pomocí reléového výstupního modulu Advantech ADAM-4060.

Dále jsou na modelu instalovány dva kapacitní senzory hladiny od firmy Turck: jeden umístěný v horní nádrži – jeho úkolem je hlídání maximální hladiny jako ochrana před přetečením nádrže. Druhý umístěný u čerpadla ve spodní nádrži – hlídání minimální hladiny jako ochrana před během čerpadla naprázdno.

Aby byl model plně automatický, byl ještě rozšířen o automatické krmítko vyvinuté a vyrobené v rámci diplomové práce [2], které s řídicím systémem komunikuje také po sběrnici RS-485.



Obrázek 2.2. Model recirkulačního systému

2.4 Dynamika řízeného systému

Původním záměrem bylo provést identifikaci parametrů systému, a podle toho navrhnout a nastavit regulátor. Podle [3] jsem vytvořil stavový popis systému. Následující matice představují model teplotní dynamiky systému:

$$\boldsymbol{A}_T = \operatorname{diag}\begin{bmatrix} 1/V_1 \\ 1/V_2 \\ 1/V_3 \end{bmatrix} \cdot \begin{bmatrix} -F & 0 & F \\ F & -F & 0 \\ 0 & F & -F \end{bmatrix} - \operatorname{diag}\begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

$$\boldsymbol{B}_T = \operatorname{diag}\begin{bmatrix} 1/V_1 \\ 1/V_2 \\ 1/V_3 \end{bmatrix} \cdot \begin{bmatrix} \beta_1 & 1/c & 0 & 0 \\ \beta_2 & 0 & 1/c & 0 \\ \beta_3 & 0 & 0 & 1/c \end{bmatrix}$$

$$\boldsymbol{C}_T = [\,1 \quad 1 \quad 0\,], \qquad \boldsymbol{D}_T = 0,$$

kde jednotlivé nádrže jsou očíslovány shora dolů, tj. retenční nádrž má číslo 1, a jednotlivé konstanty znamenají:

V_i jsou objemy jednotlivých nádrží.
β_i jsou konstanty určující rychlost výměny tepla mezi nádrží a prostředím
F udává objemový průtok vody nádržemi, v litrech za sekundu
c je tepelná kapacita vody

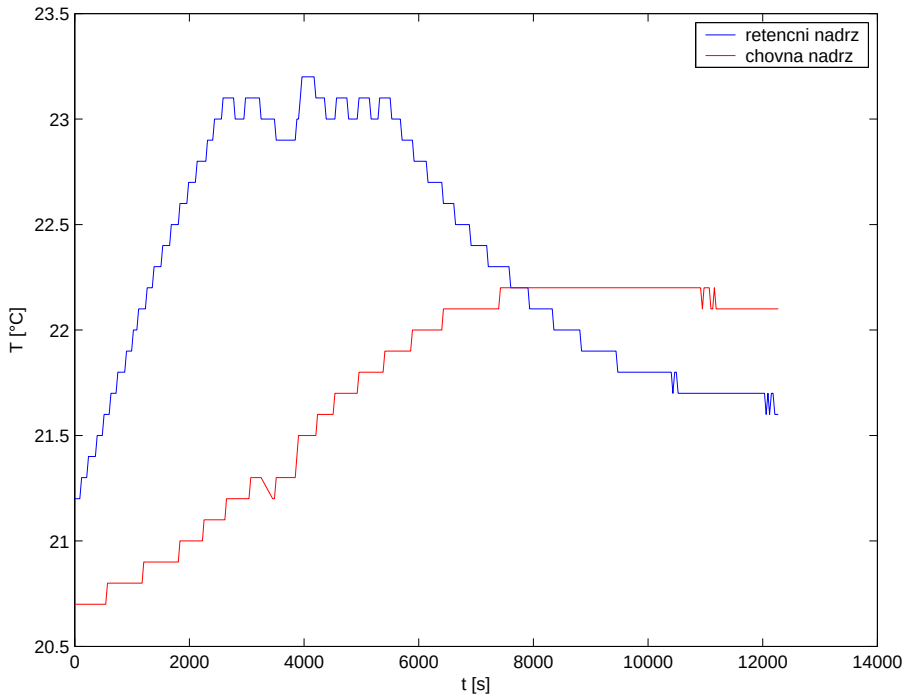
Stavy systému jsou teploty v jednotlivých nádržích, $\boldsymbol{x}_T = [\,T_1 \quad T_2 \quad T_3\,]^T$. Vektor vstupů vypadá následovně: $\boldsymbol{u}_T = [\,T_0 \quad P \quad 0 \quad 0\,]^T$, kde T_0 je teplota okolního prostředí a P je výkon topení v první nádrži.

Řídicí systém neobsahuje senzor okolní teploty, která však figuruje ve stavovém popisu jako vstupní veličina. Nejedná se ale o vstupní veličinu použitelnou pro řízení, nýbrž o chybovou vstupní veličinu – řídicí systém nedisponuje prostředky k jejímu ovládání. V případě, kdybychom okolní teplotu považovali za konstantní vstup, vnesli bychom navíc do systému nelinearitu. Identifikace takového systému by byla obtížná. Vzhledem k tomu, že takto vynaložené úsilí by nepřineslo přílišné zlepšení oproti použití jednoduchého dvoupolohového regulátoru, neboť daný systém má veliké časové konstanty, opustili jsme záměr použít pro řízení matematický model.

Pomalá odezva systému je způsobena málo výkonnými akčními členy, hlavně pak čerpadlem, které nestačí dostatečně rychle obměňovat vodu v nádržích. V retenční nádrži instalovaný topný výkon 700 W se jeví jako dostatečný.

Popis vytvořeného regulátoru je v kapitole 7.4. Vzhledem k tomu, že při regulaci teploty vody v nádrži se provádí akční zásah (ohřev) v jiné nádrži než ve které je řízena teplota, dochází ke zpoždění, které je díky málo výkonému čerpadlu znatelné. Proto je dvoupolohový regulátor doplněn o hlídání teploty vody v retenční nádrži a tím zmírnění regulačních překmitů.

Příklad regulace je uveden na obrázku 2.3. Vzhledem k dobám odezvy systému, které jsou na obrázku vidět, byla zvolena vzorkovací perioda řízení v řádu jednotek sekund, ale bylo by dostatečné i použití periody v desítkách sekund.



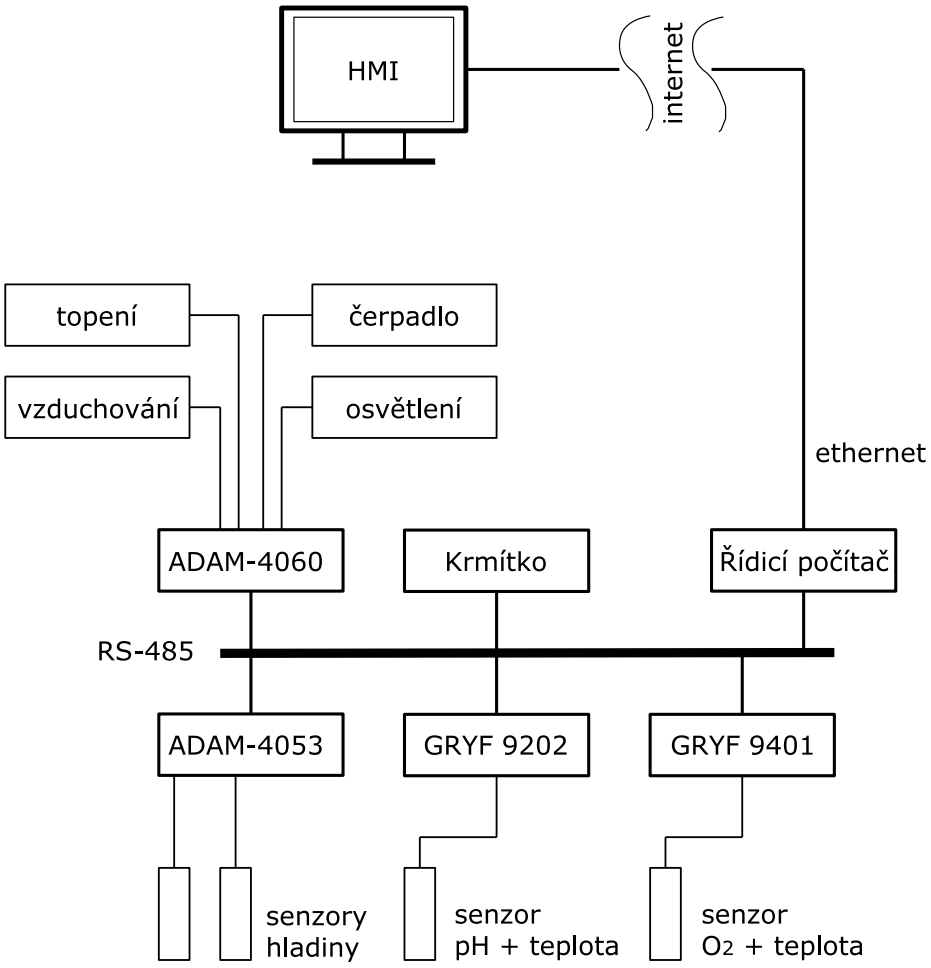
Obrázek 2.3. Záznam regulace teploty vody

3 Řídicí systém

V této kapitole je stručně popsán řídicí systém použitý k řízení modelu rybí farmy. Podrobnější informace viz [1], [2] a firemní literatura výrobců jednotlivých zařízení.

Použitý řídicí systém je distribuovaný. Jednotlivá zařízení spolu komunikují po sběrnici RS-485, kde je řízení přístupu metodou master-slave. Zvolená přenosová rychlost 9600 b/s je vzhledem k malému objemu přenášených dat dostačující. Schéma řídicího sytému viz obrázek 3.1.

Jako master je ke sběrnici připojen řídicí počítač. Ten jediný může zahájit komunikaci a ostatní zařízení pouze odpovídají na jeho příkazy. Těmito slave zařízeními jsou reléový výstupní číslicový modul ADAM-4060 od firmy Advantech [6], vstupní číslicový modul ADAM-4053, inteligentní měřicí moduly GRYF 9202 (Stacionární pH metr – teploměr) a GRYF 9401 (Stacionární oxy metr – teploměr) [7], a automatické krmítko, výsledek práce [2].



Obrázek 3.1. Blokové schéma řídicího systému

Výstupní číslicový modul slouží ke spínání akčních členů a disponuje čtyřmi relé. Je k němu připojen vzduchovací motorek, oběhové čerpadlo, světelné těleso a na jeden kanál (jedno relé) i všechna topná tělesa. Ke vstupnímu číslicovému modulu jsou připojeny kapacitní senzory hladiny. Detaily zapojení viz ref priloha-zapojeni Do měřicích modulům GRYF jsou zapojeny kombinované senzory pH+teplota a kyslík+teplota.

Řídicí počítač je přes ethernet spojen s nadřazeným řídicím systémem, kde je k dispozici uživatelské rozhraní – HMI.

3.1 Řídicí počítač

Jako řídicí počítač byl vybrán miniaturní počítač Dimm-PC od firmy Mite [5]. Jedná se o počítač tvořený základní deskou typu mitePC-F, na kterou je zapojen jeden CPU modul a jeden I/O modul. Na CPU modulu, konkrétně typu Dimm-PC/520I, se nachází procesor AMD Elan SC520 s jádrem AM5x86 běžícím na 133MHz, 16MB paměti RAM a 16MB flash disk, I/O řadič pro dvě linky COM, LPT, FDD, jeden IDE kanál a také pro sběrnici I^2C. Na I/O modulu je umístěn řadič pro rozhraní ethernet, a VGA s 1MB video paměti.

Veškeré konektory pro rozhraní poskytované řadiči na modulech CPU a I/O se nacházejí na základní desce. Pro nás je důležitý konektor pro ethernet (10BaseT) a konektor rozhraní COM2, který je proveden s optickým oddělením signálů a je určen pro připojení sběrnice RS-485 nebo RS-422.

Velikost flash disku přítomného na CPU modulu nemusí být některých případech dostačující. Pak je možné Dimm-PC rozšířit o držák CompactFlash karet, který se připojí na volný IDE konektor na základní desce.

CPU modul i I/O modul jsou vyrobeny firmou JUMPtec. Ta vybavuje CPU moduly hardwarovými vlastnostmi, které nejsou přístupné pomocí standardních API. Pro přístup k těmto funkcím existuje knihovna JIDA32, která nabízí ovládání watchdogu, nastavování parametrů LCD displeje, přístup ke sběrnici I^2C. CPU modul je také vybaven technologií JUMPtec Remote Control (JRC), což je rozšíření PC BIOSu, které nabízí možnost přesměrovat určité funkce BIOSU přes sériový port. Pomocí JRC se instaluje programové vybavení na flash disk Dimm-PC, viz Příloha C.

K napájení Dimm-PC se používá stejnosměrné napětí 24V běžné pro zařízení automatizační techniky.

3.2 Komunikační protokol na sběrnici RS-485

Všechna zařízení používají pro komunikaci jednoduchý znakově orientovaný protokol. Komunikace je typu master-slave s jedním master zařízením (řídicí počítač). Každé slave zařízení je identifikováno svojí adresou, pro jejíž reprezentaci je použito jednoho bytu, a může tedy nabývat hodnot 0–255.

Komunikaci zahajuje master vysláním datagramu obsahujícího příkaz, a očekává odpověď od zařízení určeného adresou. Zařízení jsou povinna odpovědět v určitém časovém limitu, jinak ho master považuje za porouchané.

Zařízení používají stejnou strukturu datagramu, která je následující:

(řídicí znak)(adresa)(data)(kontrolní součet)CR

Řídicí znak (1 Byte) určuje typ zprávy. Příkazy od mastera používají řídicí znaky `%`,`$` a `#`, odpovědi jsou uvozeny znakem `!`,`?` nebo `<`. Adresa (2 B) určuje v případě příkazu od mastera adresáta, v případě odpovědi se vyskytuje jen v určitých případech a označuje slave zařízení, které tu odpověď vysílá. Velikost té části zprávy, která obsahuje data, je specifická pro každý typ zprávy a určité zařízení. Master zde

vysílá samotný kód příkazu, slave odpovídá např. naměřenou hodnotou. Kontrolní součet je nepovinný, a v našem řídicím systému jsou všechna zařízení nastavena na nepoužívání kontrolního součtu. Zprávu ukončuje znak CR (carriage return, 0x0D).

3.3 Digitální vstupní modul ADAM-4053

Modul Advantech ADAM-4053 je vstupní číslicový modul disponující šestnácti logickými vstupy. Ty jsou uzpůsobeny pro dva způsoby určení vstupní úrovně. V prvním případě (nazvaném angl. dry contact) znamená připojení napěťové úrovně GND logickou nulu, a rozpojení je logická jednička. Druhým způsobem (wet contact) je řízení napěťovou úrovní připojenou ke svorce; logické nuly docílíme připojením napětí 0–2 V, logickou jedničku napětím 4–30 V. Moduly ADAM řady 4000 mohou být napájeny napětím v rozsahu 10–30 V stejnosměrných.

Na tento modul jsou připojeny oba spínací kapacitní senzory hladiny.

3.4 Reléový výstupní modul ADAM-4060

Zařízení ADAM-4060 je inteligentní reléový výstupní modul ovládaný po sběrnici RS-485. Podle přijatých příkazů spíná nebo rozpíná celkem čtyři relé. Tato relé mohou spínat střídavá napětí do 250 V/0,3 A či stejnosměrná do 110 V/1,6 A. V řídicím systému je tento modul použit ke spínání topných těles, vzduchovacího motorku, oběhového čerpadla a osvětlovacího tělesa.

3.5 Měřící moduly GRYF

Pro sledování hydrochemických parametrů v nádržích jsou použity přístroje firmy GRYF. Jedná se o mikroprocesorem řízené inteligentní moduly schopné komunikace po sběrnici RS-485. K modulům jsou kabelem připojena kombinovaná čidla. Ta jsou vybavena chemickým senzorem pro měření pH (připojené k modulu GRYF 9202) a senzorem pro měření koncentrace kyslíku (připojené k modulu GRYF 9401). Druhým senzorem pro každou kombinaci je odporový teploměr Ni 1000. Ten slouží v prvé řadě pro získání informace potřebné pro korekci měření příslušného hydrochemického parametru, kterou provádí měřicí modul. Naměřená teplota je také využívána pro samotné řízení teploty vody v nádržích.

3.6 Krmítko

Automatické krmítko použité v řídicím systému je také připojené ke sběrnici RS-485. Pracuje na principu pístu, který vytlačuje krmivo z kontejneru. Pístem pohybuje šroub připojený ke krokovému motorku. Pokud chceme vsypat krmivo do chovné nádrže, pošleme krmítku příkaz, o kolik kroků se má otočit šroub pohybující pístem. Pro doplňování krmiva do kontejneru nabízí krmítko servisní polohu, kdy je natočeno otevřeným koncem kontejneru vzhůru.

4 Software

Tato kapitola pojednává o možnostech použití svobodného software pro účely vytvoření řídicího systému. Zaměřuje se na výběr operačního systému vyhovujícího nárokům plynoucím z vlastností řízeného systému. Dále je zde řešen výběr vhodného jazyka pro real-time aplikace, a následuje popis dvou knihoven použitých v projektu.

4.1 Operační systém

Požadavkem ze zadání je použít pro řízení real-time operační systém, navíc šířený pod open source licencí.

Když se řekne open source operační systém, většině zainteresovaných lidí se vybaví jako první jméno Linux. Ten je však vytvářen jako univerzálně použitelný operační systém a pro použití v real-time aplikacích není příliš vhodný, jednak kvůli nedeterministickému plánovači úloh, a také proto, že používá nepreemptivní jádro (v současné verzi 2.4), což obojí vede ke zvětšení latence[1] úloh.

Vzniklo ale množství projektů snažících se z Linuxu vybudovat real-time operační systém. Ty se dají rozdělit podle zvoleného přístupu do dvou skupin:

- Záplatování standardního linuxového jádra. Toto řešení nahrazuje standardní plánovač úloh vlastní verzí s více deterministickým algoritmem, upravuje jádro na částečně preemptivní přidáním oblastí, kde je možné přepnutí úloh. Tyto záplaty se pak mohou časem stát součástí standardního jádra.
- Použití real-time jádra pod jádrem Linuxu. V tomto případě je linuxové jádro spuštěno jako jeden z procesů (s nízkou prioritou) v malém real-time jádře, které přebírá kontrolu nad hardwarem a Linuxu poskytuje jeho softwarovou emulaci. Real-time úlohy pak běží jako konkurenční k samotnému Linuxovému jádru. Tento přístup vyžaduje, aby RT aplikace byla vytvořena jako tzv. modul, viz např. RTAI [25], která je jednou z implementací real-time jádra.

Existují také open source real-time operační systémy nemající s Linuxem nic společné, jako například eCos [26], který nabízí jednoprocesní, ale vícevláknové prostředí – nemá správu paměti, vše běží v jednom paměťovém prostoru. Více o real-time operačních systémech viz [24] a [10].

Všechny tyto hard real-time operační systémy jsou určené pro použití ve „světě oceli a rychlosti“. Model rybí farmy se ale dá zařadit spíše mezi soft real-time systémy, jelikož zpoždění reakce řídicího systému zde není fatální.

Pak převáží výhody plynoucí z jednoduchosti použití standardního Linuxového jádra nad jeho nedostatkem real-time vlastností. Tyto nedostatky se navíc dají částečně odstranit.

Nyní nový Linux řady 2.6 přináší mnoho nových vlastností, viz [23]. Třebaže ještě není pravým real-time operačním systémem, získal vylepšení, která ho dělají vhodným pro embedded zařízení a soft real-time. Mezi ty podstatná patří preemptivní jádro, použití efektivnějšího algoritmu pro plánování úloh a použití nové implementace POSIX vláken – Native POSIX Thread Library [11], která je v porovnání se stávající implementací vláken LinuxThreads výkonnější, a ve větší míře dodržuje standard POSIX.

Linuxové jádro obsahuje i jednoduchý real-time plánovač úloh, někdy též označený jako „soft real-time POSIX scheduler“, který používá statickou prioritu procesů

[1] Latence je časový rozdíl mezi okamžikem kdy má být úloha spuštěna či dokončena a okamžikem, kdy se tak opravdu stane.

a nabízí dva real-time plánovací algoritmy v souladu s POSIX.4 standardem, viz [20]. Je to FIFO algoritmus (SCHED_FIFO), kdy je vždy spuštěn proces první ve frontě s nejvyšší prioritou, a běží tak dlouho, jak uzná za vhodné. Algoritmus Round-Robin (SCHED_RR) se od FIFO liší tím, že procesy se shodnou prioritou se automaticky po určité době přepínají.

Protože je možné špatně fungujícím nebo záškodnickým programem s vysokou real-time prioritou vyřadit z provozu celý systém, je k nastavení real-time priority procesu nutné tento proces spouštět s právy superuživatele.

Firma Mite dodává vlastní distribuci Linuxu nazvanou MiteLinux se standardním jádrem založenou na distribuci Debian, upravenou pro provoz na Dimm-PC. Nejdůležitější úpravou je zde použití ramdisku pro ukládání dočasných souborů, a tím umožnění připojování flash disku v módu jen pro čtení, neboť tyto disky mají omezený počet zapisovacích cyklů. Pro úsporu místa potřebného pro

4.2 Programovací jazyk

Volba vhodného programovacího jazyka je dalším důležitým krokem k úspěchu projektu. Pro posuzování kvalit jazyka jsem použil informace z knihy [12], kde jsou popsány vlastnosti, které by měl mít programovací jazyk použitý pro vývoj RT aplikací.

Důraz je zde kladen na dobrou *zabezpečenost* jazyka, která je určena tím,do jaké míry lze chyby v programu detekovat při kompilaci nebo příslušnými moduly operačního systému při běhu programu. Obecně by jazyk měl být schopen detekovat většinu chyb hned při kompilaci, než až při běhu programu, a to ze dvou důvodů. Za prvé proto, že čím dříve jsou chyby odhaleny, tím dříve mohou být odstraněny. Za druhé, detekce chyb při běhu programu snižuje efektivitu, neboť do programu je nutné vložit instrukce zajišťující určité kontroly. Hlavním prostředkem k dosažení vysoké zabezpečenosti na úrovni kompilace je typová kontrola dat.

Typová kontrola však neodhalí například chyby v logické struktuře programu. Počet takových chyb však klesá, jestliže jazykové konstrukce vedou, či dokonce nutí programátora k psaní přehledných a dobře strukturovaných algoritmů. Jazyk s dobrou zabezpečeností musí být jazykem dobře strukturovaným, čitelným a srozumitelným. Snadná *čitelnost* programů přináší řadu výhod: snižuje náklady na dokumentaci, umožňuje lepší pochopení programu a snadnější odhalení chyb, zjednodušuje údržbu programu. Čitelnost výsledného programu však není jen věcí jazyka, ale vyžaduje také systematické úsilí programátora.

Mezi další požadované vlastnosti RT jazyků patří *flexibilita*, tj. možnost vyjádřit všechny potřebné operace prostředky daného jazyka. Požadavek vysoké flexibility je však v rozporu s požadavkem zabezpečenosti, a je proto potřeba zvolit rozumný kompromis. *Jednoduchostí* jazyka není myšleno to, že nebude obsahovat složitější konstrukce, ale že je třeba, aby na výrazové prostředky jazyka nebyly kladeny již žádné další omezení, podmínky a dodatky, které by si programátor musel pamatovat. Potřeba *portability* jazyka je také zřejmá, neboť je výhodné, když vytvořený program není pevně spjat s jedním hardwarem. V praxi je ale dosažení skutečné portability značně obtížné, zvláště v RT systémech, kde je žádoucí maximální využití technických možností počítače.

V RT systémech se také často požaduje vysoká výpočetní výkonnost. Jazyk proto musí umožnit efektivní implementaci. Pokles ceny technických prostředků a růst ceny programového vybavení má ale za následek, že požadavek vysoké *efektivity* již není považován za prvořadý.

Přestože z [12] a [13] vyplývá, že pro RT aplikace je nejvhodnější jazyk ADA, omezil jsem se na rozhodování pouze mezi jazyky C a C++, jednak z důvodu, že tyto

jazyky již částečně ovládám, a také proto, že v prostředí UNIXu je stále nejběžnějším jazykem C. Bude tudíž nejspíše více podporován, například množstvím dostupných knihoven.

Jelikož už víme, že efektivita jazyka není v našem případě primární, ztrácí jazyk C svojí výhodu. Oproti tomu jazyk C++ disponuje vlastnostmi, které ho dělají vhodnějším pro RT aplikace, také viz srovnání [13].

Vyšší zabezpečenost jazyka C++ je dána tím, že používá silnější typovou kontrolu, nedovoluje používat funkce bez deklarací, při překladu provádí name mangling[1] identifikátorů, jehož výsledkem je *type-safe linkage* (viz [14] kapitola 7).

Objektově orientovaný přístup v programování se kladně projevuje na čitelnosti výsledného kódu, např. zvyšuje modularitu programu, odděluje implementaci od veřejného rozhraní. Protože je ale i v C++ možné psát nečitelný kód, je pro zlepšení čitelnosti vhodné dodržovat nějaký styl programování, například [15].

Při použití jazyka C++ v RT aplikacích je však nutné být opatrný při použití některých konstrukcí, které mohou zapříčinit vyšší nároky na paměť či zpomalení systému – objektové datové proudy, nebo se chovají nedeterministicky – vytváření a destrukce dynamických proměnných, dynamická identifikace typů (run time type identification, RTTI), výjimky. Na druhou stranu C++ umožňuje použití inline funkcí, které mohou zvýšit efektivitu aplikace. Vhodnost jejich použití však musí programátor uvážit.

4.3 Programovací prostředí

Jazyky C a C++ nedisponují samy o sobě konstrukcemi pro vytváření vláken, prostředky pro meziprocesní komunikaci, apod. Používají k tomu systémové knihovny, které jsou ale závislé na použitém operačním systému. Této překážky přenositelnosti aplikace se můžeme zbavit použitím knihovny zapouzdřující API operačního systému, v případě objektově orientovaného jazyka použitím objektové knihovny. Její třídy se podle jejich účelu označují jako adaptéry (wrapper) nebo fasády (facade) [17] a jejich použití přináší kromě již zmíněného zvýšení portability následující výhody:

Stručnější a robustnější programovací rozhraní – fasáda zapouzdřuje nízkoúrovňové funkce do metod objektů, jejichž použití je jednodušší, a odstraňuje fádnost programování s nízkoúrovňovými funkcemi, čímž redukuje možnost chyby programátora. Vylepšení udržovatelnosti aplikace – objektově orientovaný přístup se podílí na zlepšení struktury programu. Je jednodušší aplikaci porozumět nebo ji udržovat na základě jejího logického návrhu než jejího fyzického návrhu. Z použití OOP plyne i vylepšení modularity, znovupoužitelnosti a konfigurovatelnosti aplikace.

Nevýhodou tohoto řešení s použitím objektů jako prostředníků je, že se nízkoúrovňové funkce nevolají přímo, což vede na komplikovanější výsledný spustitelný kód. Avšak jazyky jako C++, umožňující použití inline funkcí, dovedou tyto třídy implementovat bez znatelné režie.

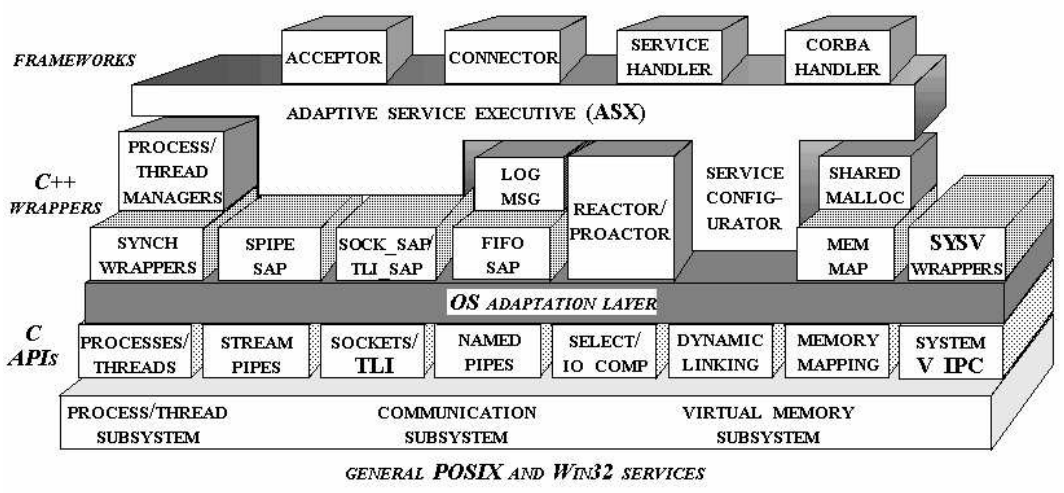
Jelikož existuje množství již hotových objektových knihoven tohoto typu, rozhodl jsem se použít jednu z nich.

[1] Name mangling je transformace identifikátorů funkcí na nový tvar. Provádí ji překladač pro rozlišení jednotlivých způsobů volání funkcí, a přidává také informace o typu parametrů. Důvodem pro name mangling je zvýšení bezpečnosti kódu a možnost provádění přídavných kontrol při linkování

4.4 Knihovna ACE

ACE (the ADAPTIVE Communication Environment) [16] je rozsáhlá objektová knihovna poskytovaná pod Open Source licencí. Zvolil jsem ji pro její vyzrálost – na jejím vývoji se za zhruba deset let podílelo více jak tisíc lidí, a také proto, že je podporována z komerčního sektoru i užívána v něm. ACE byla portována na množství platforem zahrnující Microsoft Windows, většinu UNIXů, real-time operační systémy (např. LynxOS, VxWorks, ChorusOS, QnX), OpenVMS, MVS OpenEdition, a CRAY.

Knihovna ACE je navržena s vícevrstvou strukturou, viz obrázek 4.1.



Obrázek 4.1. Struktura knihovny ACE (převzato z [16])

Nejnižší vrstvou je Vrstva přizpůsobení se operačnímu systému (*OS adaptation layer*), která není objektová; obsahuje pouze adaptační funkce, které již používají API daného OS. Tyto funkce jsou pojmenovány shodně s POSIX API, jsou však umístěny do jmenného prostoru `ACE_OS`. Tato vrstva umožňuje jednotný přístup k systémovým funkcím z těchto oblastí OS API:

- Paralelizace a synchronizace – API pro multitasking a multithreading, pro synchronizaci procesů.
- Meziprocesní komunikace (IPC) a sdílená paměť – API pro lokální i vzdálenou meziprocesní komunikaci a pro sdílenou paměť.
- Mechanizmy pro obsluhu událostí (event demultiplexing) – API pro synchronní a asynchronní obsluhu událostí.
- Explicitní připojování dynamických knihoven (explicit dynamic linking).
- Mechanizmy pro přístup k souborovému systému – API pro manipulaci se soubory a adresáři.

OS adaptation layer je jedinou vrstvou, jejíž implementace se mění v závislosti na použité platformě.

Fasádová vrstva (*wrapper facade layer*) obsahuje třídy, které zapouzdřují funkce a data do objektových rozhraní. V aplikacích se dají kombinovat tyto skupiny tříd:

- Třídy pro paralelizaci a synchronizaci – abstrakcí základních mechanizmů pro multiprocesing, multitasking a synchronizaci vznikají nové objekty s vyšší úrovní abstrakce, například aktivní objekt.

- Třídy pro IPC a souborový systém – ACE adaptéry zapouzdřují mechanizmy pro lokální i vzdálenou meziprocesní komunikaci, jako jsou například BSD sockety, roury, a také API pro manipulaci se souborovým systémem.
- Třídy pro správu paměti – tyto třídy umožňují spravovat dynamickou paměť (alokovanou na haldě) i paměť sdílenou mezi procesy.

Tato vrstva poskytuje většinu vlastností dostupných z předchozí vrstvy, avšak ve formě typově bezpečné a jednodušší k použití. To také redukuje úsilí potřebné k naučení se knihovnu správně používat.

Vrstva komponent (*framework layer*) spojuje a obohacuje třídy z předchozí vrstvy, a vytváří tak již polohotové komponenty. Doplněním těchto komponent o chování specifické pro danou aplikaci lze jednoduše vytvořit funkční program, nebo alespoň jeho část. Tato vrstva obsahuje tyto komponenty:

- Komponenty pro obsluhu událostí – objekty ACE Reactor a Proactor provádějí obsluhu událostí tím, že spouští obslužné rutiny jako odezvu na příchozí události.
- Komponenty pro inicializaci služeb – objekty ACE Acceptor a Connector oddělují inicializační část komunikace od vlastní komunikace.
- Komponenty pro konfiguraci služeb – ACE Service Configurator podporuje dynamickou konfiguraci služeb při startu nebo za běhu aplikace.
- Komponenty pro hierarchicky vrstvené datové proudy – zjednodušují vývoj komunikačních aplikací.
- ORB adaptéry – použitím ORB adaptérů může být ACE spojena s CORBA implementacemi.

Komponenty z této vrstvy ulehčují vývoj komunikačního programového vybavení, které může být aktualizováno a rozšiřováno bez nutnosti modifikovat, znovu překládat a sestavovat, a často ani restartovat běžící aplikaci.

Každá vrstva používá třídy z nižších vrstev. V ACE je tedy obvykle možné danou úlohu vyřešit více jak jedním způsobem, a je na programátorovi, jakou úroveň abstrakce zvolí. Při použití funkcí z nejnižší vrstvy získáme přenositelnou aplikaci, styl programování je stejný jako při přímém použití OS API, a je zaručeno, že efektivita programu se nezmění. Použití tříd z vyšších vrstev usnadní programátorovi práci.

Nyní popíšeme podrobněji tři oblasti z knihovny ACE, které jsou v řídicím programu nejvíce potřebné. Jsou to prostředky pro synchronizaci procesů a vláken, prostředky pro vytváření vláken a mechanizmy pro obsluhu událostí.

4.4.1 Třídy pro synchronizaci procesů a vláken

Knihovna ACE obsahuje několik tříd použitelných pro synchronizaci. Tyto třídy lze rozdělit do následujících kategorií:

- Třídy ACE Lock
- Třídy ACE Guards
- Třída ACE Condition

Třídy ACE Lock

Mezi tyto třídy patří adaptéry základních synchronizačních mechanizmů operačního systému, což jsou mutexy a semafory. Tyto adaptéry jsou k dispozici v různých variantách. Mutexy i semafory mohou být obecné, nebo určeny pouze pro synchronizaci vláken (třídy `ACE_Thread_Mutex` a `ACE_Thread_Semaphore`) nebo pro synchronizaci procesů (třídy `ACE_Process_Mutex` a `ACE_Process_Semaphore`). Pro mutexy

také existují třídy implementující read-write zamykání, které dovoluje přístup více čtenářů najednou. Tyto třídy mají ve jméně `RW`, například `ACE_RW_Mutex` a pokud systémové API read-write mutexy nenabízí, je tato vlastnost součástí implementace třídy.

Dále v této kategorii nalezneme třídu `ACE_Token`, která nabízí funkci rekurzivního mutexu, tj. vlákno, které token vlastní, ho může zamykat vícekrát, aniž by bylo blokováno. Navíc tato třída zajišťuje u ostatních vláken požadujících získat zámek jejich řazení do FIFO fronty.

Třídy `ACE_Lock` a `ACE_Lock_Adapter` umožňují volbu typu zámku při běhu programu. Jelikož je k této funkčnosti použita pozdní vazba s virtuálními metodami, je toto řešení méně efektivní z hlediska výkonu aplikace.

Pro přestavu o způsobu použití mutexů následuje příklad jejich aplikace v programu. Vytvoříme třídu `Citac`, která obsahuje jediný atribut `pocet`, a nabízí metodu pro jeho inkrementaci. S pomocí synchronizačních mechanizmů lze pak pocet inkrementovat z více vláken a máme zaručeno, že dostaneme správný výsledek.

Jako první je uveden příklad s použitím POSIX Thread API mutexu, o kterém se více dočteme v [20] nebo [21].

```
#include <pthread.h>

class Citac
{
public:
  // konstruktor - inicializace mutexu a poctu
  Citac(void) : pocet(0) { pthread_mutex_init(&mutex,0); }
  // destruktor - uvolnění systémových zdrojů
  ~Citac(void) { pthread_mutex_destroy(&mutex); }
  void Inkrementuj(void);
private:
  pthread_mutex_t mutex;          // mutex
  int pocet;                      // atribut chráněný mutexem
}

void
Citac::Inkrementuj(void)
{
  pthread_mutex_lock(&mutex);    // získání mutexu
  pocet++;                       // inkrementace atributu
  pthread_mutex_unlock(&mutex);  // uvolnění mutexu
}
```

Nyní řešení stejného problému s použitím třídy `ACE_Thread_Mutex`:

```
#include "ace/Thread_Mutex.h"

class Citac
{
public:
  // konstruktor - inicializace poctu
  Citac(void) : pocet(0) {}
  // destruktor postačuje překladačem generovaný
  void Inkrementuj(void);
private:
```

```
  ACE_Thread_Mutex mutex;          // mutex
  int pocet;                       // atribut chráněný mutexem
};

void
Citac::Inkrementuj(void)
{
  mutex.acquire();                // získání mutexu
  pocet++;                        // inkrementace atributu
  mutex.release();                // uvolnění mutexu
}
```

Jak je vidět, v tomto případě se již nemusíme starat o inicializaci a destrukci mutexu. K tomu jako bonus získáme lepší čitelnost kódu. A jak ukázáno v [18], tak je toto řešení dokonce výkonnější než samotné použití fukcí z API.

Třídy ACE Guards

Třídy ze skupiny ACE Lock v porovnání s OS API nabízejí elegantní rozhraní pro synchronizaci, zůstává v nich ale potenciální nebezpečí, že zapomeneme zámek uvolnit.

Třídy ACE Guard jsou navrženy tak, aby automaticky získávaly i uvolňovaly zámek, čímž je zvýšena robustnost aplikace. Jedná se vlastně o šablony, jejichž parametrem je třída zámku. Fungují na tom principu, že v konstruktoru je zavolána metoda pro získání zámku, a v destruktoru se zámek uvolňuje. Parametrem konstruktoru je objekt zámku, který se má použít pro chránění oblasti platnosti objektu třídy Guard.

Použití je ilustrováno v následujícím příkladě:

```
#include "ace/Synch.h"

class Citac
{
public:
  // konstruktor - inicializace poctu
  Citac(void) : pocet(0) {}
  void Inkrementuj(void);
private:
  ACE_Thread_Mutex mutex;          // mutex
  int pocet;                       // atribut chráněný mutexem
};

void
Citac::Inkrementuj(void)
{
  // získání mutexu vytvořením instance třídy ACE_Guard
  ACE_Guard<ACE_Thread_Mutex> guard(mutex);
  pocet++;                         // inkrementace atributu
  // uvolnění mutexu je automatické
}
```

Třída ACE Condition

Tato třída zapouzdřuje synchronizační primitivum nazvané podmínková proměnná (condition variable). Tu je vhodné použít například v případě, kdy je vykonávání programové smyčky podmíněno hodnotou nějaké proměnné, viz [21]. Pokud tuto funkčnost vytvoříme pouze mutexem který bude chránit danou proměnnou proti současnému přístupu z více vláken, bude nutné vždy zkontrolovat hodnotu, a v případě, že nevyhovuje podmínce, po chvíli provést kontrolu znovu. Tento přístup se nazývá polling.

Oproti tomu, pokud použijeme podmínkovou proměnnou, pak v případě, že při kontrole zjistíme, že hodnota nevyhovuje podmínce, zablokujeme provádění tohoto vlákna do té doby, než jiné vlákno hodnotu změní a tuto změnu signalizuje. Potom můžeme kontrolu provést znovu.

Podmínková proměnná umožňuje vlákno atomicky zablokovat a testovat proměnnou chráněnou mutexem, dokud její hodnota nebude vyhovovat podmínce. Vlákno musí mutex vlastnit před tím, než zavolá funkce pro signalizaci nebo čekání na podmínkovou proměnnou. Pokud je podmínka neplatná, vlákno se atomicky uspí a zároveň uvolní mutex. Když pak jiné vlákno změní podmínku, může vlákna na této podmínkové proměnné čekající vzbudit tím, že zasignalizuje tuto podmínkovou proměnnou. Čekající vlákno při probuzení znovu získá mutex a znovu vyhodnotí, zda proměnná vyhovuje podmínce.

Odlišnot v použití podmínkové proměnné z POSIX Thread API a z ACE je podobná jako u mutexů, na příkladě si tedy ukážeme pouze použití šablony z ACE – `ACE_Condition<MUTEX>`. Jejím parametrem mohou být pouze třídy `ACE_Thread_Mutex`, `ACE_Recursive_Thread_Mutex` nebo `ACE_Null_Mutex`.

V prvním vlákně vytvoříme mutex a podmínkovou proměnnou. Dále tam budeme čekat na signalizaci podmínkové proměnné a testovat její hodnotu.

```
#include "ace/Synch.h"

// vytvoření mutexu
ACE_Thread_Mutex mutex;
// vytvoření podmínkové proměnné, používá mutex
ACE_Condition<ACE_Thread_Mutex> cond(mutex);
// samotná proměnná obsahující hodnotu
bool pokracuj = false;
...
// čekání na signalizaci
mutex.acquire();
while(!pokracuj)
  cond.wait();
// nyní už pokracuj == true
mutex.release();
```

V druhém vlákně se zapíše do proměnné `pokracuj` nová hodnota, a signalizuje se podmínková proměnná.

```
...
mutex.acquire();                // získání mutexu
pokracuj = true;                // změna hodnoty proměnné
cond.signal();                  // signalizace podmínkové proměnné
mutex.release();                // uvolnění mutexu
...
```

4.4.2 Třídy pro vytváření vláken

Vlákna lze pomocí ACE vytvářet několika způsoby. Zde ukážeme dva z nich.

Třída ACE_Thread

Použití třídy `ACE_Thread` vede na postup podobný tomu, který se používá při použití OS API. Zde jako příklad uvedeme vytvoření POSIXového vlákna. Pro nové vlákno potřebujeme vytvořit funkci pro vstupní bod vlákna, což je analogie funkce main() pro hlavní vlákno procesu. Ukazatel na ni předáme jako parametr funkci `pthread_create()`, a v procesu se vytvoří nové vlákno, ve kterém se tato naše funkce pro vstupní bod vlákna spustí.

Vytváření vláken zde ukážeme na příkladu, kdy hlavní vlákno vypisuje na standardní výstup periodicky znak `'o'` a námi vytvořené vlákno vypisuje `'x'`. Nejdříve řešení s POSIX API:

```
// vstupní bod vlákna
void* thread_function(void*)
{
  while(1)
    putc('x'); // vypisování znaku 'x'
  return NULL;
}

// hlavní program
int main(void)
{
  pthread_t thread_id;
  // vytvoření nového vlákna
  pthread_create(&thread_id, NULL, &thread_function, NULL);
  // vypisování znaku 'o'
  while(1)
    putc('o');
  return 0;
}
```

Nyní ukážeme stejný problém řešený užitím třídy `ACE_Thread`. Vlákno se zde vytváří statickou metodou `ACE_Thread::spawn()`.

```
#include "ace/Thread.h"

// vstupní bod vlákna
void* thread_function(void*)
{
  while(1)
    putc('x'); // vypisování znaku 'x'
  return NULL;
}

// hlavní program
int main(void)
{
  ACE_thread_t  thread_id;
  ACE_hthread_t thread_handle;
```

```
  // vytvoření nového vlákna
  ACE_Thread::spawn((ACE_THR_FUNC)thread_function,
                    0,
                    THR_NEW_LWP|THR_JOINABLE,
                    &thread_id,
                    &thread_handle);
  // vypisování znaku 'o'
  while(1)
    putc('o');
  return 0;
}
```

Jak je vidět, kód se spíše stal složitějším. Kromě ID vlákna se zde objevuje i handle vlákna. Přesto má vytváření vláken pomocí ACE výhody, lze tak jednodušeji manipulovat s větším množstvím vláken najednou, k čemuž slouží třída `ACE_Thread_Manager`.

Třída ACE_Task

Použití třídy `ACE_Thread` vede na ne-objektově orientovanou dekompozici programu, neboť tato třída funguje spíše jen jako prostředek k zabalení funkcí pro manipulaci s vlákny do společného class scope (rozsahu platnosti).

V kontrastu s tím, třída `ACE_Task` pracuje s objekty, a jednodušeji se s ní uvažuje při návrhu objektového programu. Proto je ve většině případů lepší použít pro vytváření vláken v programu třídy odvozené od `ACE_Task`. Kromě té výhody, že její použití vede na lepší OO program, nemusíme se již starat o vytváření globálních nebo statických funkcí představujících vstupní bod vlákna, jelikož ten je teď běžnou metodou třídy.

Třída `ACE_Task` může být použita jako:

- Vyšší úroveň vlákna (z hlediska struktury programu), nazvaná Úloha (Task)
- Aktivní objekt

Vytváření Úlohy

Jak už bylo zmíněno dříve, pro vytvoření úlohy nebo aktivního objektu musíme děděním z třídy `ACE_Task` vytvořit vlastní třídu. V této třídě pak implementujeme svojí metodu pro vstupní bod vlákna, což je metoda `svc()`. Dále vytvoříme implementaci obslužných metod pro inicializaci a ukončení běhu vlákna (nazvané `open()` a `close()`). A do té inicializační můžeme vložit zavolání metody `activate()`, která aktivuje objekt, tj. vytvoří samotné vlákno.

Jednoduchost a přehlednost použití této třídy je vidět na následujícím příkladu.

```
#include "ace/Task_Base.h"

class PracovniVlakno : public ACE_Task_Base
{
public:
  int open(void*)               // inicializační metoda
  {
    activate();
    return 0;
  }
  int close(ulong) {return 0;} // ukončovací metoda
  int svc(void)                // vstupní bod vlákna
```

```
  {
    // zde se bude pracovat
    return 0;
  }
};

int main(void)
{
  ...
  PracovniVlakno vlakno;        // objekt vlakno
  vlakno.open(0);               // spuštění vlákna
  ...
}
```

Jednotlivé úlohy spolu mohou komunikovat zasíláním zpráv. Toho je využito při návrhu aktivního objektu.

Aktivní objekt

Účelem Aktivního objektu je oddělit vykonávání metody od jejího volání. Pro vytvoření vlákna používá třídu `ACE_Task`. Metody Aktivního objektu jsou volány stejně, jako by to byl obyčejný objekt. Rozdíl je v tom, že volaná metoda se provádí ve vlákně zapouzdřeném v tomto objektu, a ne ve vlákně, odkud byla metoda zavolána. To je výhodné tehdy, pokud chceme vytvořit třídy, u kterých se jejich uživatel nebude muset zabývat vlákny a jejich synchronizací.

Jelikož je vytvoření Aktivního objektu poněkud náročnější, nebudeme se tím zde dále zabývat.

4.4.3 Mechanizmy pro obsluhu událostí

Knihovna ACE nabízí pro obsluhu událostí dvě komponenty – Reactor a Proactor. Zde se podíváme jen na některé z vlastností první jmenované komponenty.

V řídicím programu je potřeba určité úlohy spouštět periodicky, viz kapitola 5. K tomu můžeme použít právě `ACE_Reactor`.Tato komponenta nabízí metody pro zaregistrování obslužných funkcí pro signály, vstupy nebo výstupy, výjimky a pro časovače.

Popíšeme zde vytvoření časově spouštěné funkce pomocí `ACE_Reactor`. Nejprve se děděním od třídy `ACE_Event_Handler` vytvoří objekt obsahující metodu `handle_timeout()` s naším kódem pro obsluhu události. Poté se vytvoří objekt třídy `ACE_Reactor`. Jeho metodou `schedule_timer()` se nastaví časovač a náš objekt pro obsluhu události se zaregistruje pro obsluhu jeho vypršení. Dále již stačí zavolat metodu `handle_events()` objektu třídy `ACE_Reactor` a ten začne obsluhovat vypršení časovače.

4.5 Knihovna ORTE

Jak již bylo zmíněno, komunikace s nadřazeným řídicím systémem bude realizována po Internetu. Zde se nabízejí možnosti od nejnižší úrovně tvořené přímým použitím BSD soketů, přes RPC až po CORBA.

Přímé použití soketů vyžaduje vytvoření si nějakého protokolu komunikace, což by mohlo být náplní celé diplomové práce. Na druhou stranu ne všechny protokoly používající sokety jsou vhodné pro real-time, třeba kvůli jejich komplikovanosti nebo použití TCP kanálu, jehož implementace bývá nedeterministitcká.

Rozhodl jsem se použít protokol Real-Time Publish-Subscribe (RTPS), konkrétně jeho open source implementaci ORTE (OCERA Real-Time Ethernet) [19], která je k dispozici jak pro Linux, tak pro MS Windows.

RTPS je protokolem aplikační vrstvy vytvořený nad UDP protokolem. Umožňuje komunikaci typu publish-subscribe, kdy jsou data přenášena pouze jedním směrem, od vydavatele (publisher) k odběrateli (subscriber).

V komunikaci se angažují tyto objekty:

- Manažer – speciální objekt, správce komunikace
- Vydavatel – dodává data nazývaná publikace
- Odběratel – odebírá data publikovaná vydavatelem

Manažer je nezávislý proces. Jeho úkolem je správa komunikace. Na lokálním počítači předává data od vydavatelů jejich odběratelům. Komunikace po síti se vzdáleným strojem probíhá pouze v úrovni Manažer–Manažer.

Data, která nabízí vydavatel, se nazývají publikace. Pro příjem dat publikace si odběratel vytváří předplatné – subskripci. Publikace a subskripce jsou v ORTE identifikovány v síti dvěma znaky. Prvním je název tématu (topic) a druhým je jméno typu (typename). Je povoleno mít více publikací se stejným identifikátorem. Potom jsou odběrateli doručeny data od všech vydavatelů.

Při použití knihovny ORTE je v klientské aplikaci nejprve nutné zavolat inicializační funkci `ORTEInit()`. Následně se funkcí `ORTEDomainAppCreate()` vytvoří v aplikaci ORTE doména, což je zjednodušeně řečeno vlákno spravující samotnou komunikaci. V této doméně se potom funkcí `ORTEPublicationCreate()` vytvářejí publikace a funkcí `ORTESubscriptionCreate()` subskripce.

Názvy pro téma i pro typ dat jsou voleny uživatelem. Typ dat je nutné v ORTE doméně zaregistrovat, a určit tak, jak se s daným typem má zacházet. Dají se tak registrovat i serializační a deserializační funkce. Ty převádějí data z formy, v které jsou uloženy v operační paměti, do datového proudu představeného znakovým řetězcem, a zpět. Jejich použití je vhodné například tehdy, pokud chceme v jedné publikaci vysílat celý obsah struktury s více prvky.

U vytvářené publikace lze nastavit několik parametrů. Lze tím také zvolit, zda se bude publikace odesílat automaticky s nastavenou periodou, nebo zda bude nutné její odeslání zajistit explicitně zavoláním funkce `ORTEPublicationSend()`. U subskripce je možné zvolit, zda bude přijímat publikace ihned, nebo zda je bude nutné zavolat deserializační funkci pro získání dat s datového proudu explicitně (což se provádí zavoláním funkce `ORTESubscriptionPull()`).

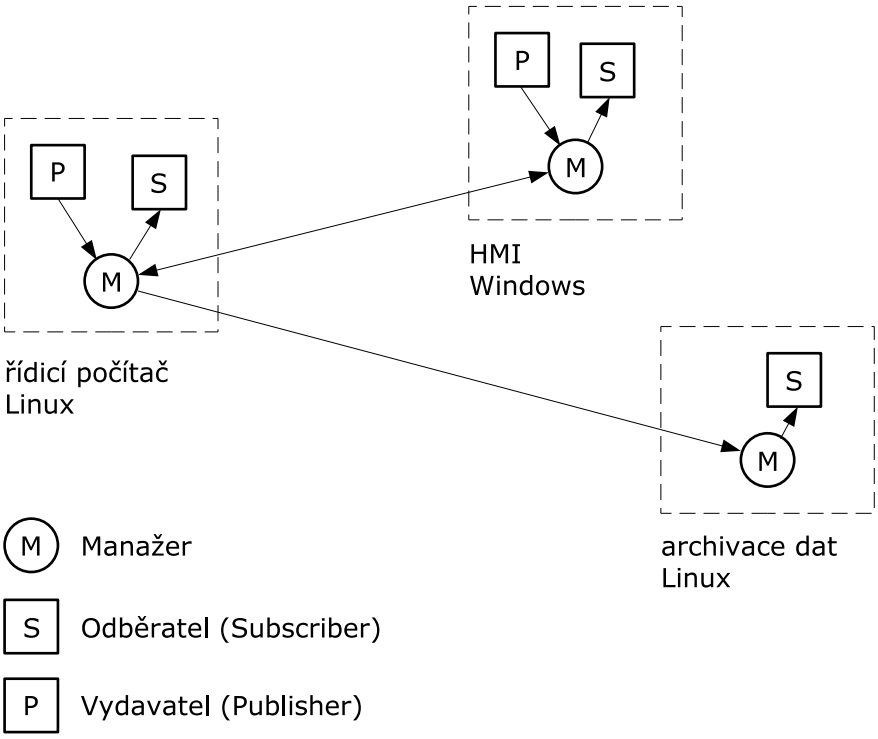
Při vytváření publikace nebo subskripce je možné pro ně ještě zaregistrovat CallBack funkce. Ty jsou automaticky zavolány při odesílání nebo přijímání dat, ale pokud příjem nebo vyslání dat zajistíme explicitně zavoláním již zmíněných funkcí, CallBack funkce spuštěny nejsou. SendCallBack funkce jsou určeny pro přípravu dat před jejich odesláním, RecvCallBack pro provedení nějaké akce po příjmu dat, například informování jiné části programu.

Explicitní a implicitní vysílání nebo přijímání dat se také liší tím, v kterém vlákně se provádí serializační/deserializační funkce a CallBack funkce. Při explicitním spuštěním přenosu dat se toto provádí ve vlákně, odkud byla daná funkce zavolána. Při automatickém přenosu se o to stará ORTE doména, a serializační/deserializační funkce i CallBack funkce jsou spouštěny ve vlákně vlastněném doménou.

Funkce knihovny ORTE nejsou plně reentrantní. Při implementaci tříd používajících ORTE (viz kapitola 7.6) je nutné jako reakci na příchozí data tyto data zapsat do paměti řídicího programu a následně je vyslat ostatním stanicím. To všechno je prováděno uvnitř RecvCallBack funkce. Uvnitř CallBack funkce ale není možné

volat funkce pro odeslání publikací, které se nacházejí ve stejné ORTE doméně, jako tato subskripce. Proto jsou v řídicím programu použity domény dvě – jedna pro příjem a druhá pro vysílání.

Na obrázku 4.2 je znázorněno, jak může vypadat použití ORTE v tomto projektu. Na řídicím počítači běží ORTE Manažer, a zasílá ostatním Manažerům publikace procesních dat. Ty přijímá Manažer na počítači s uživatelským rozhraním, a předává je Odběrateli, který je součástí programu pro vizualizaci. Uživatelské vstupy jsou zasílány zpět k Manažeru řídicího počítače. Dále tu běží počítač, na kterém je spuštěn pouze Odběratel, který data z publikací přijímá a archivuje. Na obrázku je také ilustrována možnost použití různých platforem.



Obrázek 4.2. Příklad použití ORTE

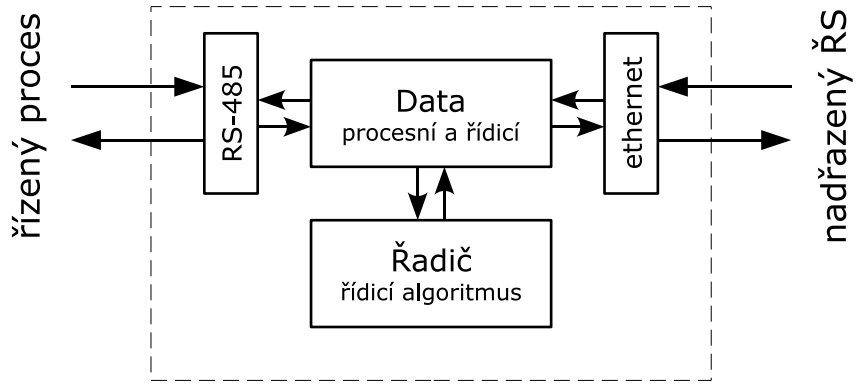
5 Řídicí program

Základním úkolem řídicího programu je poskytovat včasnou odezvu na vstupy. Ke splnění tohoto úkolu je nutné načíst data ze vstupů do paměti, zpracovat je a výsledky zapsat na výstupy. Tato kapitola se zabývá postupy při vytváření řídicího programu v multitaskingovém prostředí.

Program pro řízení našeho modelu můžeme rozdělit na tři hlavní části podle jejich funkce, viz obrázek 5.1:

- Komunikace s řízeným procesem. V našem případě tato komunikace probíhá pouze po sběrnici RS-485, ke které se přistupuje z jednoho sériového portu.
- Řadič zpracovávající procesní data. Bude obsluhovat několik na sobě nezávislých regulačních smyček – pro regulaci teploty, kyslíku apod.
- Komunikace s nadřazeným řídicím systémem po ethernetu pomocí internetových protokolů TCP/IP.

  Všechny tyto části si předávají informace přes množinu dat, kterou sdílejí.



Obrázek 5.1. Blokové schéma řídicího programu

Je také nutné navrhnout strukturu řídicího programu z hlediska rozvržení jednotlivých úloh do prováděcích toků. K tomu potřebujeme vědět, co nám nabízí operační systém.

5.1 Využití multitaskingu v řízení

V reálném světě se věci dějí paralelně. Oproti tomu v počítači se program zpracovává postupně (pokud není použito více procesorů). Existují však možnosti, jak od sebe oddělit jednotlivé úlohy tak, aby se jevily jako paralelní.

Použitím multitaskingového operačního systému se nám nabízí několik alternativ, jak navrhnout strukturu řídicího programu z hlediska organizace prováděcích toků. V [20] jsou popsány tyto případy:

- Použití jediného procesu. Ten obsluhuje veškeré vstupy a výstupy, provádí výpočty apod. Výhodou této metody je, že se vyhneme provozní režii plynoucí z použití více procesů. Ale pro více jak pár jednoduchých úloh může být výsledkem komplikovaný kód, jehož údržba a rozšiřování bude složité.
- Použití množství procesů. Multitaskingový OS dovoluje vytvořit vlastní proces pro každou činnost potřebnou v naší aplikaci. Odstraní se tím komplikovanost

předchozí metody, ale vzniká problém, jak vyřešit komunikaci mezi jednotlivými procesy. Navíc použití mnoha procesů způsobí sníženou výkonnost aplikace.

- Sloučení podobných úloh do jednoho procesu je kompromis mezi předchozími dvěma přístupy. Fyzicky se tak od sebe oddělí pouze logicky nesouvisející úlohy. Vhodnou kombinací úloh zachováme jednoduchost údržby a přitom udržíme na uzdě nároky aplikace na systémové prostředky.
- Použití signálů k emulaci multitaskingu. Obslužná rutina signálu může být považována za nezávislý asynchronní tok uvnitř procesu, spouštěný jako reakce na příchod signálu. Problémem tohoto řešení je, že ačkoli by se mohly jednotlivé obslužné rutiny zdát jako oddělené prováděcí toky, tak jimi nejsou.
- Použití vláken. Vlákna nabízejí jednodušší a efektivnější prostředky pro vytvoření množiny souvisejících úloh. Nesmíme ale zapomenout na synchronizaci přístupu jednotlivých vláken ke sdíleným proměnným.

Hlavním typem úlohy v řídicím programu je regulační smyčka. V programu může jedna programová smyčka obsluhovat jednu i více regulačních smyček. Jejího cyklického spouštění můžeme dosáhnout několika způsoby.

Cyklus může běžet neustále, což je ale nevhodné kvůli enormní zátěži systému a taky kvůli nemožnosti řídit vzorkovací periodu. Obzvláště problematické je použití tohoto řešení při více běžících procesech s regulačními smyčkami.

Lepším řešením je spouštění této smyčky s nějakou danou periodou. Pokud je toto použito v případě jednoho procesu obsahujícího více smyček s rozdílnou vzorkovací frekvencí, je nutné celý tento proces spouštět s frekvencí rovnou jejich nejmenšímu společnému násobku.

Existuje několik způsobů jak dosáhnout periodického spouštění programové smyčky s danou periodou. Jednou možností je jako poslední ve smyčce volat funkci na pasivní čekání (např. `sleep`). Nevýhodou tohoto řešení je, že pokud samotné vykonávání operací v regulační smyčce trvá nenulovou dobu, celková perioda vykonávání programové smyčky je o tuto, často neznámou dobu delší, a navíc může kolísat.

Kvalitnějším řešením je použití časovače. V momentě, kdy časovač vyprší, obdrží proces signál, který toto oznamuje. Na příchod signálu se proces musí připravit, a to buď tím, že pro něj zaregistruje obslužnou rutinu (obdoba interrupt service routine u HW přerušení), nebo může signál zablokovat jeho vložením do signálové masky procesu. Přijetí neošetřeného signálu má za následek ukončení běhu procesu.

5.2 Časovače

V Linuxu jsou k dispozici standardní UNIXové intervalové časovače. Jeden proces může použít tři časovače, které se však liší způsobem měření času, a tak je pro potřeby řízení použitelný pouze jeden – `ITIMER_REAL`.

Naštěstí v Linuxu můžeme také použít POSIX.4 intervalové časovače, i když o nich manuálové stránky mlčí. Ty nabízejí pro jeden proces minimálně 32 různých časovačů, jimž můžeme nastavit signál, který bude při jejich vypršení procesu zaslán (oproti standardním časovačům, které zasílají podle typu časovače vždy jeden určený signál, a tudíž při použití více jak jednoho časovače už nemáme možnost odlišit, který z nich vypršel).

Jak již bylo zmíněno, signály jsou asynchronní, a může se stát, že vykonávání programu přeruší v tu nejnevhodnější chvíli. Měly bychom se tedy v obslužné rutině signálu vyvarovat vstupně/výstupních operací a volání většiny systémových a knihovních funkcí. Volání všech funkcí řídicí smyčky z obslužné rutiny, která je periodicky spouštěna časovačem, sice vypadá efektně, avšak není to vhodné řešení.

Na druhé straně, mohlo by se zdát, že zablokováním signálu vlastně ničeho nedosáhneme, protože mu znemožníme, aby nás o vypršení časovače informoval. V případě POSIX real-time časovačů je však skutečnost taková, že se sice nebude vykonávat žádná obslužná rutina, ale existují funkce, které nám o příchodu signálu dají vědět tím, že se probudí z čekání. Toto řešení je v [20] označeno jako synchronní čekání na signál a použil jsem ho v následující třídě.

5.3 Třída pro POSIX časovače

V kapitole 4 je popsána komponenta `ACE_Reactor`. Ta sama o sobě nepodporuje nastavování časovačů na absolutní čas. Protože jsem se o POSIX časovačích a jejich výhodách dozvěděl dříve, než jsem měl možnost důkladněji prozkoumat knihovnu ACE, a navíc proto, že ACE vnitřně nepoužívá POSIX časovače, vytvořil jsem vlastní třídu zapouzdřující POSIX časovač, která se jmenuje `Ifibo_Timer`.

Inicializace časovače se provádí zavoláním metody `CreateTimer(int signalNumber)`, která vytvoří samotný časovač, nastaví číslo signálu, který se má zasílat po uplynutí nastavené doby časovače, a zablokuje příjem tohoto signálu jeho vložením do signálové masky procesu. Třída dále nabízí metody k nastavení času vypršení časovače. Metody `SetTime()` jsou určeny k nastavení relativního času a jdou použít pro jak pro jednorázový časovač, tak i periodický. Absolutní čas se nastavuje metodou `SetAbsTime()`, kde je možné si zvolit např. 11 hodin 55 minut dne 21. 5. 2004. Metoda `SetAbsTimeOfDay()` je určena k nastavení přesného času, ale se dnem zadaným relativně k dnešku.

Během provozu může uživatel nastavení časovače měnit, popřípadně nastavením hodnoty na nulu deaktivovat.

Metoda `GetTime()` vrací čas zbývající do uplynutí nastavené doby. Metoda `DeleteTimer()` slouží ke zrušení časovače a tím uvolnění systémových prostředků. Je automaticky volána z destruktoru, takže ve většině případů není nutné ji volat explicitně. Pro vlastní synchronní čekání na signál zaslaný po vypršení časovače se použije metoda `WaitForSignal()`.

Příklad použití třídy:

```
Ifibo_Timer  my_timer;
my_timer.CreateTimer(SIGRT_MIN);  //použití signálu SIGRT_MIN
my_timer.SetTime(5);              //interval 5 vteřin

while (1)
{
  //čekání na signál
  my_timer.WaitForSignal();

  //funkce kterou chceme periodicky spouštět
  PeriodicFunction();
}
```

6 Objektový model řídicího programu

Tato kapitola je úvodem do popisu vytvořeného řídicího programu. Jsou zde popsány jednotlivé skupiny vytvořených objektů, účel jejich použití a je zde naznačena jejich spolupráce v řídicím programu.

V závěru je popsáno, jakým způsobem jsou pojmenovány třídy, jejich metody, datové typy i soubory je obsahující.

Při návrhu objektové struktury programu jsem se inspiroval reálným systémem. Cílem je vytvoření objektů představujících jednotlivé části systému a jejich propojením vznikne v programu obraz celého řídicího systému.

Objekty v programu můžeme rozdělit do dvou skupin. V první skupině jsou objekty pro manipulaci s daty, které se starají o získávání informací, jejich zpracování, a odesílání. Jsou vytvořeny jako obecně použitelné komponenty.

Druhou skupinou jsou objekty zajišťující provoz řídicího systému. V těchto objektech je řídicí systém sestaven z jednotlivých komponent z první skupiny, a tyto objekty se také starají o spouštění jednotlivých fukcí řídicího systému, například periodické vzorkování vstupů apod.

6.1 Objekty tvořící datovou část programu

Tyto objekty jsou navrženy jako komponenty pro vytvoření datové části řídicího programu. Jsou nezávislé na konkrétním problému, který řeší řídicí program.

Jejich metody se omezují pouze na přenos a zpracování dat. V programu existuje několik skupin těchto objektů odlišujících se funkcí, pro kterou jsou určeny. Na obrázku 6.1 je šipkami znázorněno, jak si jednotlivé skupiny objektů v řídicím programu předávají data.

Moduly

První skupinou jsou objekty představující zařízení připojená na sběrnici RS-485, viz obrázek 3.1. Tyto komunikační moduly lze ještě jemněji rozdělit na vstupní a výstupní moduly.

Sériové zařízení

Sběrnice s moduly je k řídicímu počítači připojena přes sériové rozhraní. Existuje tedy objekt představující toto rozhraní, a na tento objekt se bude odkazovat objekt komunikačního modulu v případě, že chce zaslat nějakou zprávu reálnému modulu.

Datové bloky

Dále je nutné vytvořit objekty, které obsahují data. Jeden modul může nabízet několik veličin naměřených v procesu, např. moduly GRYF poskytují dvě hodnoty – teplotu a pH nebo koncentraci kyslíku. Druhým příkladem je reléový modul ADAM-4060, který spíná čtyři akční členy.

Pro návrh těchto datových tříd jsem si vzal jako inspiraci PROFIBUS-PA Profil [27], konkrétně jeho bloky Analog Iput Function Block (použitý například pro hodnotu teploty), Discrete Input Function Block (pro kapacitní čidlo hladiny) a Discrete Output Function Block (pro spínané akční členy).

Řadiče

Regulaci zajišťuje skupina objektů – řadičů. Ty přistupují k jednotlivým objektům s daty, jak pro čtení v případě vstupů, tak i pro zápis akčních zásahů na výstupy. V těchto objektech jsou implementovány regulační algoritmy.

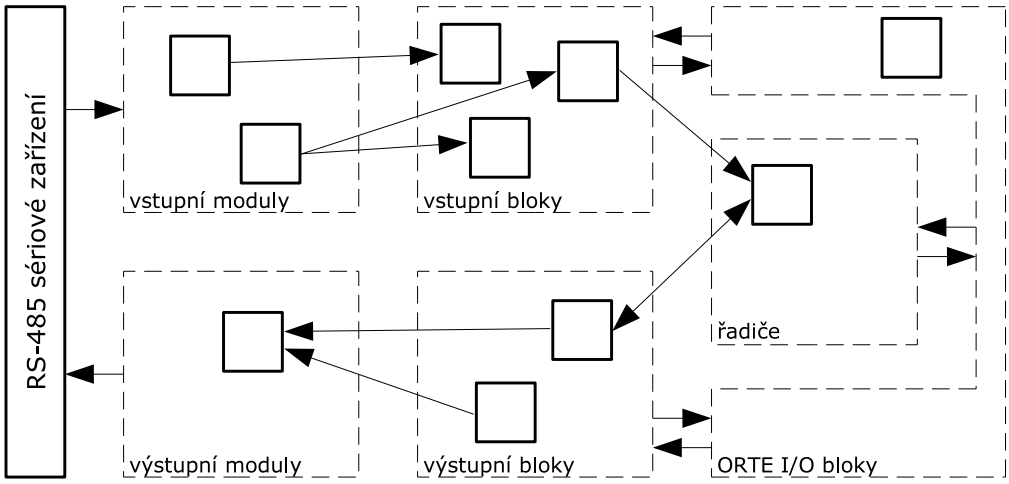
Objekty úloh

Vedle periodického řízení, které se použije například pro regulaci teploty, je v programu nutné také vytvořit část pro jednorázové spouštění úloh naplánovaných na určitý čas. Mezi tyto úlohy patří například rozsvěcení a zhasínání světel nebo krmení ryb.

Objekty z této skupiny reprezentují jednotlivé úlohy. Jsou v nich obsaženy údaje potřebné k vykonání úlohy.

Objekty pro komunikaci s nadřazeným ŘS

Tato skupina objektů již není nutná pro samotnou činnost řídicího programu. Zajišťuje spojení s nadřazeným řídicím systémem tím, že doplňuje datové bloky o rozhraní pro síťovou komunikaci. Každému objektu ze skupiny datových bloků náleží objekt, jehož úkolem je data tohoto bloku vysílat vzdáleným stanicím, a zároveň přijímat změny a zapisovat je. Pro komunikaci je využito funkcí knihovny ORTE.



Obrázek 6.1. Konstrukce datové části řídicího programu z jednotlivých komponent

6.2 Objekty pro konstrukci řídicího programu

Tyto objekty jsou narozdíl od předchozí skupiny víceméně specifické pro daný řídicí systém.

Spouštění jednotlivých funkcí řídicího programu během provozu je prováděno dvěma objekty. Jeden je určen pro periodické řízení a druhý pro spouštění naplánovaných úloh. Tyto dva objekty v sobě obsahují svůj vlastní prováděcí tok, jsou vytvořeny děděním od třídy `ACE_Task_Base`, viz kapitola 4.4.2. V těchto prováděcích tocích poběží programová smyčka řízená objektem časovače třídy `Ifibo_Timer` s použitím synchronního čekání na signál časovače, viz příklad na konci kapitoly 5.3.

Hlavním objektem řídicího programu je objekt třídy `Ifibo_System`. V něm jsou obsaženy všechny ostatní objekty, je zde provedena jejich inicializace a jejich propojením vzniká struktura řídicího programu.

Podrobný popis těchto tříd je v kapitole 8.

6.3 Jména tříd a jejich metod, datových typů

Tato práce je součástí projektu Ifibo. Názvy většiny tříd začínají tedy řetězcem `Ifibo_`. Výjimkou jsou třídy zapouzdřující funkce knihovny ORTE, které používají předponu `Orte_`. V obrázcích zobrazujících hierarchické uspořádání tříd jsou tyto předpony vynechány.

Při pojmenovávání metod objektů jsem využil možnosti přetěžování funkcí. Přístupové metody se nejmenují například `SetValue(int value)` a `GetValue(void)`, ale obě jsou nazvány stejně a liší se pouze parametry: `Value(int value)` a `Value(void)`.

Kvůli jednoduššímu aplikování změn použitých datových typů jsem pomocí konstrukce `typedef` vytvořil nové datové typy, a ty jsem použil při vytváření tříd. Tyto nové typy jsou definovány v souboru `Ifibo_types.h`. Zde je část vypsána jako příklad:

```
...
typedef unsigned char   StatusType;
typedef float           AnalogValueType;
typedef unsigned char   DiscreteValueType;
...
```

6.4 Jména souborů

Pro přehlednost jsou soubory pojmenovány shodně s názvem třídy, která je v nich obsažena. Platí pravidlo, že jedné třídě náleží jeden hlavičkový soubor s deklarací třídy a jeden soubor s implementací třídy, a tyto soubory se liší pouze příponou.

7 Třídy tvořící datové komponenty

V této kapitole jsou popsány třídy pro manipulaci s daty a pro jejich uchovávání v řídicím programu. Většina z nich je navržena jako znovupoužitelné komponenty určené k sestavení samotného řídicího programu.

7.1 Třída pro komunikaci přes RS-485

Třídou poskytující přístup na sběrnici RS-485 je `Ifibo_SerialDevice`. Jednomu fyzickému COM portu náleží jedna instance této třídy. Inicializační metoda `OpenDevice()` zajišťuje otevření souboru zařízení a nastavení parametrů pro komunikaci, jako je přenosová rychlost, používání paritních bitů apod. Analogicky, metoda `CloseDevice()` soubor zařízení uzavírá.

Samotnou komunikaci zajišťuje metoda `SendCommand()`, která vyšle danou zprávu a čeká na odpověď. Pro čekání je použita systémová funkce `select()`, a tak je zaručen návrat z metody v konečném čase i v případě, kdy neobdržíme žádnou odpověď. To může nastat například při poruše zařízení nebo i samotné sběrnice.

Pro vyloučení současného přístupu k tomuto objektu z více vláken obsahuje třída metody pro zamknutí a odemknutí jejího mutexu – (`AcquireMutex()` a `ReleaseMutex()`). Tyto metody se ale nevolají v žádné metodě třídy, jsou veřejné, a je nutné, aby zamykání/odemykání obstaral ten, kdo na zařízení přistupuje. Toto řešení je zvoleno z toho důvodu, aby bylo například možné zaslat bez přerušení několik příkazů na sběrnici.

Komunikace po sběrnici RS-485 má svoje pravidla – je použit pouze jeden pár vodičů sdílený všemi zařízeními, a je nutné zajistit aby v jednu chvíli vysílalo pouze jedno zařízení. Tento způsob komunikace se označuje jako half-duplex mód.

Před zahájením vysílání se aktivuje vysílač (budič linky), a jakmile je datagram odeslán, je třeba vysílač deaktivovat. Toto načasování je důležité hlavně pro master zařízení, protože po vyslání příkazu obvykle ihned následuje odpověď od adresovaného slave zařízení.

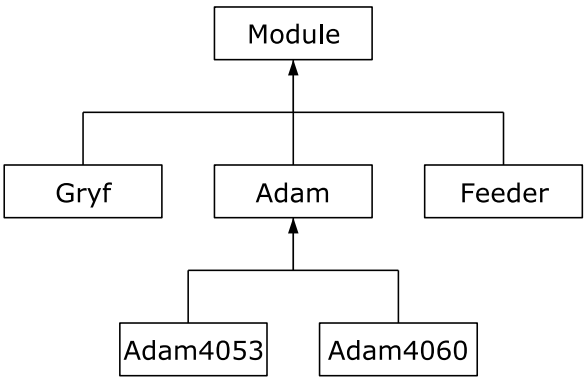
V Linuxu zatím neexistuje nějaký obecně použitelný ovladač, který by byl určen přímo pro RS-485. Použil jsem tedy standardní ovladač sériové linky, který dovoluje vytvořit si potřebné chování ve vlastním programu, viz [22]. Budič linky u portu RS-485 se u Dimm-PC spíná signálem RTS, se kterým nám ovladač umožňuje manipulovat.

Slabým místem tohoto řešení je, že standardní ovladač neposkytuje funkce pro spolehlivé určení přesné doby, kdy se vyšle poslední znak a je třeba deaktivovat vysílač. Bohužel už nezbyl čas na vytvoření si vlastního ovladače, který by toto časování prováděl spolehlivě. Na druhou stranu, po částečných nezdarech při testování funkčnosti této třídy na pracovní stanici s použitím externího převodníku RS-232–RS-485, při testování na Dimm-PC se ukázalo toto řešení se standardním ovladačem jako spolehlivé. V každém případě, metoda `SendCommand()` je, jak bylo zmíněno výše, vybavena mechanizmy zabraňujícími zaseknutí programu, když přenos dat skončí neúspěšně.

Jelikož třída `Ifibo_SerialDevice` vnitřně volá přímo POSIX funkce pro přístup k sériovému portu, je její přenositelnost omezená.

7.2 Třídy představující moduly na sběrnici RS-485

Tyto třídy jsou určeny k ovládání vstupních a výstupních modulů připojených na sběrnici RS-485. Všechny tyto moduly používají ke komunikaci jednoduchý protokol se stejnou strukturou datagramu, viz kapitola 3. Na obrázku 7.1 je znázorněno hierarchické uspořádání těchto tříd.



Obrázek 7.1. Hierarchie tříd představujících moduly

7.2.1 Třída Ifibo_Module

Společné vlastnosti modulů jsou shrnuty ve třídě `Ifibo_Module`, která je základní třídou. Adresa modulu se nastavuje metodou `SetAddress()`, a v případě potřeby ji můžeme z objektu přečíst metodou `GetAddress()`. Metoda `SetDevice()` se používá k nastavení sériového zařízení (objekt třídy `Ifibo_SerialDevice`), ke kterému je modul připojen.

Každý z modulů je kromě své adresy identifikován také názvem. Metoda `AcquireName()` slouží pro načtení jména modulu do datové struktury objektu. Následně můžeme tento řetězec obsahující jméno pomocí metody `GetName()` získat k dalšímu zpracování v programu.

Z třídy `Ifibo_Module` jsou děděním odvozené třídy pro jednotlivé moduly.

7.2.2 Třída Ifibo_Adam

Pro moduly ADAM je základní třída `Ifibo_Adam`, která obsahuje jejich společné atributy. Nyní obsahuje pouze metodu `Status()`, která vrací status modulu získaný podle jeho odezvy při poslední komunikaci.

7.2.3 Třída Ifibo_Adam4053

Digitální vstupní modul ADAM-4053 je v programu reprezentován třídou `Ifibo_Adam4053`. Ta nabízí pro přečtení hodnot na vstupech metodu `Read()`. Tyto hodnoty pak získáme zavoláním metody `Value(int channel)`, kde parametr `channel` je číslo kanálu v rozsahu 0–15.

7.2.4 Třída Ifibo_Adam4060

Reléovému výstupnímu modulu ADAM-4060 náleží třída `Ifibo_Adam4060`. Metodou `SetData()` se požadovaná výstupní data (všechna najednou) uloží do soukromého atributu třídy, a pak zavoláním metody `Write()` se tyto data vyšlou modulu.

7.2.5 Třída Ifibo_Gryf

Pro moduly GRYF je vytvořena jedna třída – `Ifibo_Gryf`, neboť moduly 9202 i 9401 se chovají totožně. Metoda `ReadChannel(int channel)` načte data z kanálu učeného parametrem `channel`, který může být 0 nebo 1.

Potom můžeme použít metody `Value(int channel)` a `Status(int channel)` pro přístup k naměřeným hodnotám a ke stavu udávajícímu jejich věrohodnost. Třída vrací naměřenou hodnotu v celočíselné podobě, tak jak ji přijala od modulu, takže teplotu je nutné vydělit deseti, pH a koncentraci kyslíku číslem 100.

7.2.6 Třída Ifibo_Feeder

Poslední třídou odvozenou z `Ifibo_Module` je třída `Ifibo_Feeder` určená pro ovládání krmítka. Metoda `ReadCurrentPosition()` načte ze zařízení současnou pozici pístu. Tuto hodnotu pak můžeme zjistit metodou `CurrentPosition()`.

K ovládání pohybu pístu slouží dvě metody, `MoveForward()` a `MoveBackward()`. Těm se jako parametr předává hodnota, o kolik kroků se má šroub poháněj ící píst otočit. Více informací o krmítku viz [2].

Třídy představující moduly byly implementovány bez použití systémových funkcí. V místech, kde je nutná přesná velikost datového typu, byly použity typy se zaručenou velikostí z knihovny ACE, například čtyřbytové celé číslo bez znaménka představuje typ `ACE_UINT32`. Tyto třídy jsou tedy přenositelné v rámci přenositelnosti knihovny ACE.

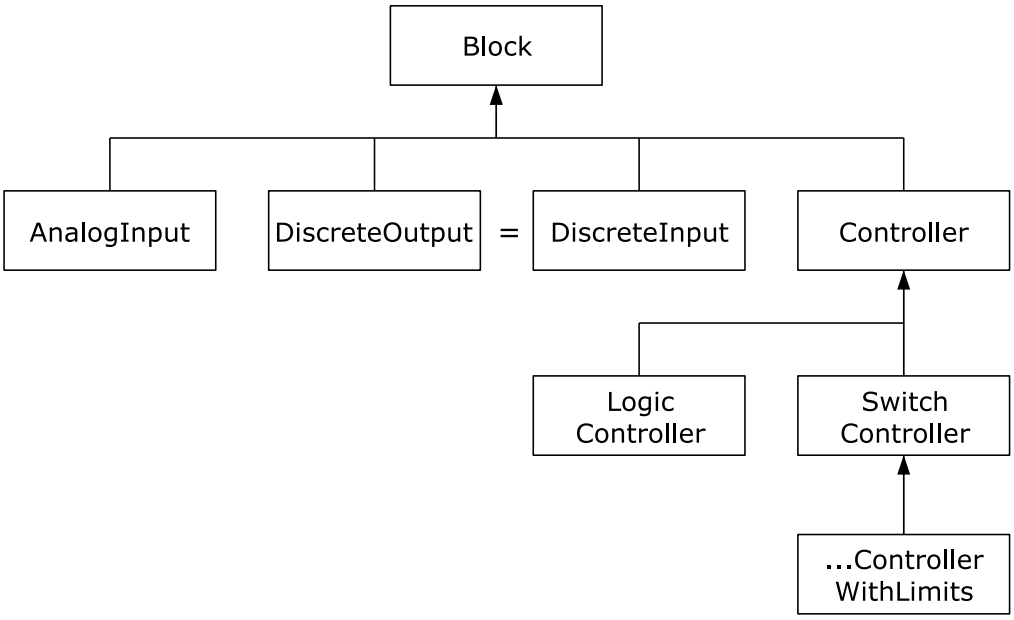
7.3 Třídy představující datové bloky

Hlavním účelem těchto tříd je vytvořit v programu objekty pro ukládání a čtení dat – datové bloky. Tyto bloky slouží pro výměnu dat mezi jednotlivými aktivními částmi řídicího programu. Vstupní moduly do nich zapisují naměřené hodnoty, regulátory tyto hodnoty čtou, a do jiných datových bloků pak zapisují hodnotu akčního zásahu.

Objekt *datový blok* zapouzdřuje spolu související údaje do jediného celku. Tyto data a nad nimi povolené operace (většinou přístupové metody) tvoří celistvou jednotku – což je vlastně charakteristika objektů.

K těmto blokům se bude přistupovat z více vláken, jsou proto doplněny o mutex jako prostředek zajišťující výlučný přístup k objektu. Vzniká tak vlastně objekt typu *monitor*, o kterém je zmínka např. v [12] nebo [24]. Druhým krokem k zajištění bezpečnosti ve vícevláknovém prostředí je použití modifikátoru `volatile` u přístupných[1] atributů.

[1] Přístupným atributem se zde myslí soukromý atribut, k němuž existují přístupové metody.



Obrázek 7.2. Hierarchie tříd pro datové bloky

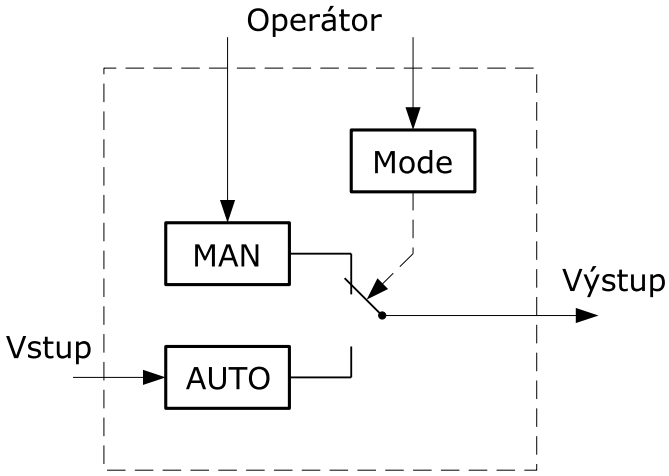
7.3.1 Třída Ifibo_Block

Základní třídou pro datové bloky je `Ifibo_Block`. V ní jsou obsaženy základní informace o bloku, konkrétně:

- Název bloku – řetězec určující název bloku, slouží k identifikaci bloku.
- Typ bloku – určuje typ daného bloku. Odstraňuje se tím nutnost použít run-time type identification pro zjištění, jakého typu daný objekt je.
- Identifikační číslo bloku – sekundární identifikační atribut, jednodušeji se používá v programu (porovnání čísel je jednodušší než porovnání řetězců).
- Režim provozu bloku (Mode)

Aby bylo možné do bloku vkládat hodnoty ručně, například jako manuální ovládání z operátorského stanoviště, je každý blok vybaven atributem *režim provozu*. Ten slouží k přepínání režimu bloku na automatický nebo manuální. Všechny proměnné, které závisejí na režimu, budou v bloku obsaženy ve dvou variantách – pro automatický provoz a pro manuální provoz. Pro každou z nich bude existovat přístupová metoda pro zápis. Přístupová metoda pro čtení bude jen jedna, a bude vracet hodnotu v závislosti na aktuálním režimu, viz obrázek 7.3. Do proměnné pro automatický provoz budou hodnoty zapisovat řadiče a vstupní moduly, do proměnné pro manuální provoz bude hodnoty vkládat objekt zajišťující komunikaci s nadřazeným řídicím systémem.

Pro vytvoření funkce monitoru obsahuje třída `Ifibo_Block` objekt třídy `ACE_Thread_Mutex` (viz kapitola 4.4.1) jako svůj privátní atribut. Ten se používá ve veřejných metodách k zabránění vícenásobnému přístupu k objektu. Aby manipulaci s ním mohly provádět i odvozené třídy, obsahuje tato třída tři metody – `AcuquireReadMutex()`, `AcuquireWriteMutex()` a `ReleaseMutex()`, které mají přístupová práva `protected`. Výhodou tohoto řešení je, že v odvozených třídách se nemusíme zabývat tím, jak se objekt mutexu jmenuje, a navíc to umožňuje změnit implementaci zamykání a odemykání mutexu.



Obrázek 7.3. Princip řízení režimu provozu bloku

Metody pro manipulaci s mutexem se, jak je z jejich názvu vidět, odlišují podle toho, zda chceme v dané kritické sekci pouze číst, nebo hodnoty i měnit. V knihovně ACE existuje třída `ACE_RW_Thread_Mutex`, kde je tato vlastnost implementována. Zjistil jsem ale, že manipulace s RW Mutexem spotřebuje zhruba dvojnásobek času oproti obyčejnému mutexu, který je tedy nakonec použit.

Pro lepší bezpečnost by bylo výhodné použít k zamykání a odemykání mutexů třídu `ACE_Guard`. Potom by se ale musela změnit přístupová práva objektu mutex na protected, neboť nevýhodou tohoto řešení je, že se objekt třídy `ACE_Guard` musí vytvářet přímo v místě použití, a nejdou pro to použít pomocné metody. V knihovně ACE je použití třídy `ACE_Guard` implementováno s použitím maker.

Přístupové metody pro čtení jsou deklarovány jako konstantní – nemodifikují datové složky objektu. Přitom také potřebují zajistit výlučný přístup k objektu. Aby bylo možné volat metody mutexu i uvnitř konstantních metod, je tento mutex deklarován s paměťovou třídou `mutable`.

7.3.2 Třída Ifibo_AnalogInput

Do objektů této třídy jsou ukládány analogové hodnoty získané měřením z procesu. Slovem „analogový“ je míněno, že je hodnota vícestavová, a v paměti je reprezentovaná číslem s pohyblivou řádovou čárkou. Tato třída je vytvořena podle bloku Analog Input Function Block z PROFIBUS PA Profilu, obsahuje ale pouze část jeho vlastností. Třída obsahuje tyto prvky:

- Hodnota (Value) – slouží k uložení vlastní hodnoty
- Stav (Status) – stav hodnoty
- Dolní mez (LoLimit) – hodnota dolní meze pro alarmy
- Horní mez (HiLimit) – hodnota horní meze pro alarmy
- Alarm – obsahuje stav alarmu
- Časová značka alarmu (Timestamp) – čas, kdy došlo k poslední změně stavu alarmu

Tato třída využívá možnosti přepínání režimu. Pro Hodnotu a její Stav je ve třídě po dvou atributech. Pro zápis se používají metody začínající slovem `Automatic` nebo `Manual` podle požadovaného chování. Například pro vložení hodnoty naměřené senzorem se použije metoda `AutomaticValue()`.

Pro čtení Hodnoty a Stavu slouží metody `Value()` a `Status()`. V nich je implementováno přepínání podle režimu provozu bloku.

Dolní a horní mez slouží k nastavení limitů, přes které by se Hodnota neměla dostat. V případě, že se tak stane, nastaví se Alarm a do časové značky se uloží aktuální čas.

7.3.3 Třída Ifibo_DiscreteOutput

Tato třída představuje požadované hodnoty pro akční členy. Dovoluje pouze logické řízení, akční člen může být pouze zapnut nebo vypnut. Třída obsahuje tyto prvky:

- Hodnota (Value) – požadovaná hodnota
- Stav (Status) – stav zařízení

Tato třída také využívá přepínání režimu. Lze tak manuálně ovládat přímo akční členy. Pro zápis se používají metody začínající slovem `Automatic` nebo `Manual` podle požadovaného chování.

Čtení Hodnoty a Stavu je stejné jako u třídy `Ifibo_AnalogInput` – metody `Value()` a `Status()` přepínají podle režimu provozu bloku zdroj, ze kterého načtou požadovaný údaj.

Stav představuje u této třídy stav samotného komunikačního modulu, tj. například pokud modul nereaguje, dozvíme se to z tohoto atributu.

7.3.4 Třída Ifibo_DiscreteInput

Jelikož třídu pro číslicový vstup bylo nutné vytvořit až v pozdější fázi projektu, a implementace třídy `Ifibo_DiscreteOutput` nabízí stejné vlastnosti, které by měla mít i tato třída, v souboru s deklarací se nachází pouze toto:

```
typedef Ifibo_DiscreteOutput Ifibo_DiscreteInput;
```

V PROFIBUS PA Profilu [27] jsou mezi funkčními bloky Discrete Input a Discrete Output jisté rozdíly. Pokud by bylo nutné implementovat i ty funkce, které tyto bloky odlišují, pak by již takovéto jednoduché řešení nestačilo.

7.4 Třídy představující regulátory

Hlavním úkolem těchto tříd je s daty manipulovat, a ne je ukládat. Přesto obsahují několik atributů, které jsou použity pro potřeby regulace – jsou to parametry regulátoru. Proto jsou tyto třídy také odvozeny ze základní třídy pro datové bloky, z `Ifibo_Block`.

7.4.1 Třída Ifibo_Controller

`Ifibo_Controller` je základní třídou pro všechny třídy regulátorů. Je to vlastně abstraktní třída, neboť obsahuje jedinou metodu, která je čistě virtuální. Tato metoda se jmenuje `CalculateControl()` a jejím účelem je vykonat řídicí zásah. Ne však tím, že by odeslala příkaz přímo modulu ovládajícímu akční členy. Ukládá výsledek z výpočtu řídicího algoritmu do určeného výstupního datového bloku.

Metoda `CalculateControl()` byla vytvořena jako virtuální proto, aby se zjednodušilo spouštění regulace. Například, může existovat seznam s x regulátory. V určitý časový okamžik se spustí regulační zásah. Program tedy projde celý seznam a u každého prvku spustí metodu `CalculateControl()` bez ohledu na typ konkrétního regulátoru.

7.4.2 Třída Ifibo_SwitchController

Tato třída implementuje dvoupolohový regulátor s hysterezí. Obsahuje dva atributy – požadovanou hodnotu (setpoint) a hysterezi (hysteresis). Přístupové metody k těmto atributům jsou nazvány `Setpoint()` a `Hysteresis()`. Vstupní veličina regulátoru je analogová, a také požadovaná hodnota a hystereze jsou typu `AnalogValueType`.

Třída dále obsahuje dva ukazatele, jeden na objekt třídy `Ifibo_AnalogInput` a druhý na `Ifibo_DiscreteOutput`. Příslušné objekty se určí (připojí) metodami `ConnectAnalogInput()` a `ConnectDiscreteOutput()`.

7.4.3 Třída Ifibo_SwitchControllerWithLimits

Tato třída byla vytvořena kvůli specifickým vlastnostem našeho řízeného systému, konkrétně řízení teploty vody. Požadavkem kladeným na řídicí systém je udržovat určenou teplotu v chovné nádrži. Problém ale nastává, pokud voda, která se ohřívá v retenční nádrži, nestačí dostatečně rychle přetékat do chovné nádrže. V retenční nádrži pak vystoupá teplota vody hodně nad požadovanou teplotu v chovné nádrži. Postupně se tato voda dostane do chovné nádrže, kde způsobí veliký regulační překmit. Než pak teplota v chovné nádrži klesne pod spínací úroveň dvoupolohového regulátoru, voda v retenční nádrži vychladne, a následně, až se dostane do chovné nádrže, způsobí tam zase veliký překmit do záporných hodnot regulační odchylky.

Proto byl regulátor `Ifibo_SwitchController` doplněn o hlídání stavu alarmů u druhého analogového vstupního bloku. Nastavením HiLimit a LoLimit mezí tohoto bloku lze pak snížit kmitání při regulaci.

7.4.4 Třída Ifibo_LogicController

Tato třída byla vytvořena pro účely jednoduchého logického řízení, pro řešení konkrétního problému v řídicím programu – řízení čerpadla. Vzhledem k tomu, že součástí řídicího systému jsou čidla hladiny (viz obrázek 2.2), je možné vytvořit v řídicím programu mechanizmus chránící horní nádrž před přetečením například kdyby se ucpal odtokový otvor nádrže. Také je možné chránit čerpadlo před během naprázdno, což by vedlo k jeho zničení.

Regulátor tvořený objektem této třídy řídí v programu běh čerpadla tím způsobem, že ho spustí pouze v tom případě, když čidlo maximální hladiny v retenční nádrži je nesepnuté a čidlo minimální hladiny ve filtrační nádrži sepnuté je.

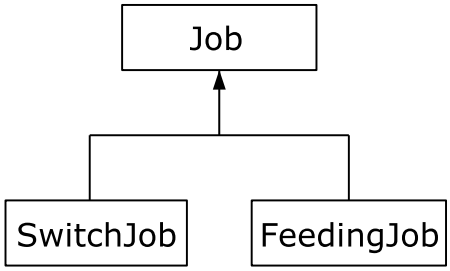
Třída obsahuje dva ukazatele na datové bloky typu `Ifibo_DiscreteInput` (čidla) a ukazatel na blok `Ifibo_DiscreteOutput` (čerpadlo).

7.5 Třídy pro časově spouštěné úlohy

Úlohy spouštěné jednorázově v určitý časový okamžik se neřadí k regulaci, ale k ovládání systému. Patří mezi ně spínání světel či automatické krmení ryb. Tyto třídy se dají považovat za obdobu regulátorů odvozených z třídy `Ifibo_Controller`. Také v sobě mají implementovánu metodu, která se spouští jako obsluha události – vypršení časovače.

Na rozdíl od regulátorů, které mají v řídicím programu své pevné místo a většinou jsou vytvářeny už při startu aplikace, třídy představující úlohy je výhodné v programu vytvářet dynamicky. Příkladem může být změna počtu krmných dávek během dne.

Proto musí v řídicím programu existovat objekt spravující tyto úlohy. Bude se starat o jejich vytváření a rušení, i o jejich plánování. Tento objekt se jmenuje `Ifibo_JobScheduler` a je popsán v následující kapitole 8. K objektům úloh se může přistupovat pouze prostřednictvím tohoto objektu – plánovače úloh. Proto je zbytečné, aby každý objekt představující úlohu vlastnil svůj mutex. Tyto třídy tedy nejsou odvozené z třídy `Ifibo_Block`.



Obrázek 7.4. Hierarchie tříd pro časově spouštěné úlohy

7.5.1 Třída Ifibo_Job

Třída `Ifibo_Job` je abstraktní, a je společnou třídou pro všechny třídy představující úlohy. Jsou v ní shrnuty společné vlastnosti nutné pro každý typ úlohy. Obsahuje tedy tyto atributy:

- Čas spuštění úlohy (Execution time)
- Typ úlohy (Job type)

Dále tato třída obsahuje čistě virtuální metodu určenou pro spuštění v nastaveném čase. Ta se jmenuje `Action()` a její implementace je provedena v třídách odvozených děděním. V řídicím programu jsou použity dvě třídy odvozené od `Ifibo_Job`, viz obrázek 7.4. Mechanizmus úloh je navržen tak, že úlohy jsou spouštěny každý den. Atribut Čas spuštění úlohy určuje čas od začátku dne, vyjádřený v sekundách. Pro jeho nastavení a nebo přečtení slouží metoda `ExecutionTime()`.

7.5.2 Třída Ifibo_SwitchJob

Úkolem této třídy je zařídit v nastavený okamžik sepnutí akčního členu. Informace potřebné k této akci jsou uloženy ve dvou atributech třídy:

- Požadovaný stav akčního členu (Value)
- Ukazatel na objekt třídy `Ifibo_DiscreteOutput`.

Metoda `Action()` u této třídy provádí pouze zapsání požadovaného stavu akčního členu do automatické hodnoty (viz režim provozu bloku v 7.3.1) datového bloku určeného ukazatelem. Samotné poslání příkazu k sepnutí akčního členu je nutné zařídit zvlášť. V řídicím programu je to vyřešeno tak, že tento příkaz zasílá objekt periodické řídicí smyčky – může tak vzniknout prodleva mezi spuštěním úlohy a samotným sepnutím akčního členu, která je veliká maximálně jako perioda řídicí smyčky.

7.5.3 Třída Ifibo_FeedingJob

Objekt této třídy představuje jedno krmení ryb během dne. Informace potřebné pro provedení této úlohy obsahuje v těchto atributech:

- Krmná dávka (Ration)
- Ukazatel na objekt třídy `Ifibo_Feeder`

U této třídy provede metoda `Action()` akční zásah, a není tedy nutné již provádět žádné další přenosy dat. Atribut krmná dávka se zde používá jako parametr pro metodu `MoveForward()` třídy `Ifibo_Feeder`. Určuje, o kolik kroků se má pootočit šroub poháněj ící píst krmítka.

7.6 Třídy postavené nad ORTE

Z objektů tříd již zmíněných v této kapitole je možné sestavit funkční datovou část řídicího programu. Takovému programu by ale chybělo spojení s nadřazeným řídicím systémem. Pro tento úkol jsou vytvořeny třídy z této skupiny.

7.6.1 Třídy pro datové bloky

Účelem těchto tříd je doplnění stávajících datových bloků o možnost komunikace přes internet. Jako prostředek této komunikace je použita knihovna ORTE. Toto nové komunikační rozhraní objektů datových bloků bylo vytvořeno způsobem nezasahujícím do již navržených tříd pro datové bloky. Přistupuje k nim pomocí již vytvořených přístupových metod. To zajišťuje výlučný přístup k datům, neboť tak není narušena funkce monitoru, a zároveň se tyto Orte třídy nemusejí zamykáním zabývat.

Protože ale není vhodné používat mutexy v serializačních a deserializačních funkcí, je v těchto třídách vytvořen přenos dat ve dvou krocích. Při zasílání publikace se v prvním kroku data přečtou z příslušného datového bloku pomocí jeho přístupové metody, a uloží se do atributu objektu třídy Orte. Následně se zavolá funkce `ORTEPublicationSend()` a data jsou serializační funkcí přemístěny do vyrovnávací paměti (datového proudu) a odeslány. Příjem dat je prováděn analogicky. Nejprve je spuštěna deserializační funkce, která data z vstupního datového proudu zapíše do atributu třídy Orte, a dále je spuštěna RecvCallBack funkce, která tyto data zapíše do určeného datového bloku. Jelikož je u všech subskripcí tříd Orte použit typ doručení `IMMEDIATE`, tak je příjem dat prováděn automaticky.

Z podstaty komunikace publish–subscribe vyplývá, že je to pouze jednosměrný přenos dat. V řídicím programu jsou ale data, která je potřeba přenášet oběma směry. Pro obousměrnou komunikaci je potřeba vytvořit pro každou přenášenou proměnnou dvojici publikace–subskripce.

Na obrázku 4.2 v kapitole 4.5 byla naznačena struktura sítě ORTE manažerů v řídicím systému. Komunikace zde bude probíhat způsobem, kdy manažer běžící na řídicím počítači bude přijímat publikace od ostatních manažerů v síti, a bude jim zasílat své publikace. Spojení mezi jednotlivými manažery bude tedy realizováno pouze přes řídicí počítač. V řídicím programu se tedy použije dvou ORTE domén, jedné pro příjem dat a druhé pro vysílání dat (jak je zmíněno v 4.5), aby bylo možné po příjmu dat od jednoho vzdáleného manažera tyto data obratem rozeslat ostatním v síti.

Data přijímaná od manažerů v nadřazeném řídicím systému jsou považována za uživatelské vstupy a tudíž Orte třídy zapisují tyto data do manuálních proměnných

datových bloků (viz režim provozu bloku popsaný u třídy `Ifibo_Block`, kapitola 7.3.1).

Třídy postavené nad ORTE mají stejnou hierarchii, jako třídy datových bloků, viz obrázek 7.2. Jsou nazvány podle tříd, ke kterým přísluší, a to tím způsobem, že k názvu třídy datového bloku je přidána předpona `Orte_`. Pro použití v řídicím programu jsou vytvořeny tyto třídy:

Orte_Ifibo_Block
Toto je základní třída pro všechny třídy vytvářející rozhraní Orte k třídám datových bloků.

Orte_Ifibo_AnalogInput
Tato třída je určena pro použití s třídou `Ifibo_AnalogInput`.

Orte_Ifibo_DiscreteOutput
Tato třída je určena pro použití s třídou `Ifibo_DiscreteOutput`, a také s třídou `Ifibo_DiscreteInput`, jelikož jsou tyto dvě třídy totožné (viz 7.3.4).

Orte_Ifibo_SwitchController
Tato třída nabízí přenos dat pro třídy `Ifibo_SwitchController` i pro `Ifibo_SwitchControllerWithLimits`, neboť obě tyto třídy mají atribut pro požadovanou hodnotu řízené veličiny a hysterezi spínání.

Všechny tyto třídy doplňují datové bloky o síťové rozhraní způsobem, který zde již byl popsán. Pro většinu přístupných atributů bloku vytvářejí publikaci i subskripci. Existují však pouze publikovaná data, což jsou například atributy týkající se alarmů.

Vytvoření prvků pro komunikaci pomocí ORTE se provádí zavoláním metody `CreateOrteCommunication()`, která je implementována ve všech těchto třídách. Uvedeme zde její deklaraci (konkrétně pro třídu `Orte_Ifibo_Block`) a popíšeme jednotlivé parametry.

```
CreateOrteCommunication(
    ORTEDomain* subscriptionDomain,
    ORTEDomain* publicationDomain,
    Ifibo_Block* pBlock,
    const char* subscriptionSuffix,
    const char* publicationSuffix,
    ORTERecvCallBack recvCallBack,
    ORTESendCallBack sendCallBack);
```

První dva parametry jsou ukazatele na ORTE domény, které budou použity pro vytváření subskripcí a publikací. Je nutné, aby to byly dvě různé domény. Další parametr je ukazatel na objekt datového bloku, ke kterému přidáváme toto síťové rozhraní. Dalšími dvěma parametry jsou konstantní ukazatele na řetězce určující příponu v názvu tématu pro subskripce a publikace.

Nyní odbočíme od popisu parametrů funkce `CreateOrteCommunication()`, neboť je nutné vysvětlit, jakým způsobem vytvářejí Orte třídy názvy jednotlivých témat komunikace. Jako základ názvu se bere jméno datového bloku. K tomu se připojí název samotného atributu, pro který je daná publikace nebo subskripce vytvořena. Na konec se umístí přípona získaná z parametrů `subscriptionSuffix` a `publicationSuffix`. Jednotlivé části názvu tématu jsou od sebe odděleny tečkou. Řetězec představující název tématu pak vypadá takto:

```
"JMENO_BLOKU.NAZEV_PROMENNE.PRIPONA"
```

Význam přípony v názvu tématu je takový, že dovoluje v síti odlišit ty publikace, které představují výstupní hodnoty od systému, od těch, co představují vstupní hodnoty. Přitom je zachováno to, že v názvu tématu je obsaženo jméno bloku a proměnné. Při vytváření publikací a subskripcí si uživatel vložením příslušných přípon zvolí, jaká data bude přijímat a jaká vysílat.

Poslední dva parametry metody pro vytvoření Orte rozhraní jsou ukazatele na uživatelské CallBack funkce. V druhém odstavci této kapitoly 7.6 je popsáno, že RecvCallBack u subskripce provádí přenos dat z objektu třídy Orte do datového bloku. Pokud uživatel chce při příjmu dat provést ještě nějakou specifickou akci a tuto funkci použije zde jako parametr, bude se tato funkce volat z RecvCallBack funkce, která je součástí třídy Orte[1].

Jelikož jsou publikace vytvořené v objektu třídy Orte zasílány pouze na příkaz, a ne automaticky, nejsou funkce SendCallBack v programu vůbec využity. Poslední parametr funkce `CreateOrteCommunication()` je tudíž pouze „do počtu“, a je zde pro případ, že by byl potřeba v budoucnu.

7.6.2 Třídy pro nadřazený řídicí systém

Kromě tříd vytvářejících Orte rozhraní u datových bloků byly vytvořeny ještě třídy určené pro použití mimo řídicí program, když nechceme vytvářet datové bloky. Tyto třídy používají pro ukládání přijímaných a vysílaných data pouze svoje atributy. Avšak narozdíl od předchozí skupiny tříd k těmto atributům nabízejí přístupové metody. Protože můžeme chtít vytvořit program, který bude pouze monitorovat data v řídicím systému, nebo naopak pouze řídit určité hodnoty, nabízejí tyto třídy dvě metody pro vytvoření komunikačního rozhraní ORTE. Jsou to `CreateOrtePublications()` pro vytvoření publikací a `CreateOrteSubscriptions()` pro vytvoření subskripcí a jejich definice je následující:

```
CreateOrteSubscriptions(
    ORTEDomain* d,
    const char* subscriptionSuffix,
    ORTERecvCallBack recvCallBack);

CreateOrtePublications(
    ORTEDomain* d,
    const char* publicationSuffix,
    ORTESendCallBack sendCallBack);
```

Význam jednotlivých parametrů je stejný jako u tříd pro datové bloky v předchozí kapitole 7.6.1. Rozdíl je pouze v tom, že nyní již neexistuje objekt, ze kterého by se při vytváření publikace či subskripce získal název datového bloku. Proto nabízejí tyto třídy metodu pro nastavení názvu bloku, kterou je nutno zavolat před samotným vytvořením rozhraní pro ORTE.

[1] Jelikož RecvCallBack funkce je obyčejná funkce, v třídách Orte je vytvořena jako statická metoda. Jako parametr se jí předává ukazatel `this`.

7.6.3 Třída Orte_Ifibo_JobScheduler

Tato třída doplňuje třídu `Ifibo_JobScheduler` o rozhraní knihovny Orte. Vzhledem k náročnosti vytvoření přístupu k dynamickému seznamu pomocí protokolu RTPS a vzhledem k nedostatku času, byla zvolena jednoduchá implementace s pevně daným počtem úloh v seznamu. Publikace a subskripce pro tyto úlohy byly pojmenovány podle účelu dané úlohy, například `LIGHT_ON` apod. Třída disponuje metodou `CreateOrteCommunication()` se stejnými parametry jako u tříd v kapitole 7.6.1. Předává se jí ukazatel na objekt třídy `Ifibo_JobScheduler`.

8 Třídy konstruující řídicí program

V této kapitole jsou popsány třídy obsahující prováděcí toky použité pro řízení, a objekt třídy `Ifibo_System` sestavující řídicí program z objektů datových komponent i těchto objektů s vlákny.

Tyto třídy jsou víceméně šité na míru řídicího programu, jejich znovupoužitelnost je omezená, a u třídy `Ifibo_System()` by se musela implementace změnit kompletně. V řídicím programu existuje od každé této třídy pouze jediná instance.

8.1 Třída Ifibo_PeriodicControlThread

Tato třída obsluhuje jednu periodickou řídicí smyčku. O datové struktuře řídicího programu nemá žádné informace. Obsahuje pouze ukazatel na objekt třídy `Ifibo_System`, a periodicky spouští jeho čtyři metody určené pro průběžné řízení, které mají na starost načtení hodnot ze senzorů, výpočet akčního zásahu, zápis akčního zásahu do ovládacích modulů pro akční členy, a zaslání nových hodnot přes ORTE ostatním stanicím. Tato jedna periodická řídicí smyčka ve skutečnosti může obsahovat více smyček, ale třída `Ifibo_PeriodicControlThread` je považuje za jedinou, neboť spouští jen ty zmíněné čtyři metody.

Tato třída je odvozena děděním od třídy `ACE_Task_Base` a řídicí smyčka je vytvořena v metodě `svc()`, viz příklad v kapitole 4.4.2.

Perioda spouštění řídicí smyčky je řízena objektem třídy `Ifibo_Timer`, synchronním čekáním na signál, jak je ukázáno na příkladu v kapitole 5.3. Vytvoření časovače zavoláním metody `CreateTimer()` je nutné provést ve vlákně, kde budeme na signál čekat, tj. uvnitř metody `svc()`. Je to z toho důvodu, že v LinuxThreads implementaci POSIX vláken jsou vlákna vlastně procesy a signály nejsou doručovány všem vláknům, ale pouze tomu jedinému, ve kterém se zavolá metoda pro vytvoření časovače.

8.2 Třída Ifibo_JobScheduler

Objekt této třídy představuje správce a plánovač časově spouštěných úloh popsaných v kapitole 7.5. Jeho úkolem je správa seznamu naplánovaných úloh, a zajištění spuštění metody `Action()` každé úlohy ze seznamu v nastavený čas. Třída `Ifibo_JobScheduler` je také odvozena od `ACE_Task_Base` a také obsahuje časovač, který používá pro načasování spouštění úloh.

Čas spuštění úlohy je součástí třídy `Ifibo_Job` a představuje čas od začátku dne v sekundách. Tyto úlohy jsou tedy plánovány pro každý den. Při startu programu se zjistí aktuální čas, dle toho se zjistí, jaká úloha bude následovat jako první, a pro ni se nastaví časovač. Třída `Ifibo_JobScheduler` rozlišuje úlohy na dva typy. Na úlohy provádějící stavovou operaci, například rozsvícení světla, které je po výpadku řídicího programu nutné provést znovu, a na úlohy provádějící operaci, kterou není vhodné opakovat po výpadku, například krmení ryb. Při inicializaci objektu `Ifibo_JobScheduler` se tedy také provedou metody `Action()` všech již dnes spuštěných úloh, které provádějí stavovou operaci.

Po inicializaci se již spustí samotná řídicí smyčka. Ta začíná čekáním na signál od časovače. Při vypršení časovače se spustí metoda `Action()` aktuální úlohy. Pak se naplánuje další úloha. Inkrementuje se atribut určující aktuální úlohu a její čas spuštění se použije k nastavení časovače. Ještě se kontroluje, zda právě provedená úloha nebyla poslední úlohou toho dne. Pokud ano, naplánuje se spuštění první úlohy v seznamu na následující den.

Tato třída je nyní implementována zjednodušeně. Veškeré objekty představující úlohy jsou vytvořeny při startu programu a jejich seznam v této třídě je statický, tvořen polem ukazatelů. Toto řešení bylo zvoleno z časových důvodů, neboť implementace dynamického seznamu a metod pro jeho úpravy by bylo složitější i při použití šablon ze standardní šablonové knihovny jazyka C++. Navíc vytvoření přístupu k dynamickému seznamu a umožnění jeho modifikace prostřednictvím ORTE by bylo náročné.

Třída `Orte_Ifibo_JobScheduler` je určena pro tuto zjednodušenou implementaci seznamu úloh, a doplňuje tyto pevně vytvořené úlohy o síťové rozhraní knihovny ORTE.

8.3 Třída Ifibo_System

Třída `Ifibo_System`, jak už název napovídá, představuje celý řídicí program, a existuje pouze jedna instance této třídy. V ní jsou obsaženy všechny další objekty představující jak datovou část, tak i vlákna zajišťující běh řídicího programu.

Pro inicializaci systému slouží metoda `Open()`. Zde jsou nastaveny parametry datových bloků, vytvořeny dvě ORTE domény, v nich zaregistrovány publikace a subskripce pro výměnu dat s nadřazeným řídicím systémem, a vytvořeny objekty obsahující vlákna. Hodnoty použité k inicializaci jsou nyní obsaženy v hlavičkových souborech. V souboru `Ifibo_configuration.h` jsou obsaženy data pro konfiguraci řídicího systému, například adresy jednotlivých komunikačních modulů na sběrnici. Pro inicializaci hodnot datových bloků, například limitních hodnot analogových vstupních bloků, jsou použita data ze souboru `Ifibo_datainit.h`.

Pro průběžné řízení jsou zde vytvořeny čtyři metody, které jsou volány z vlákna objektu třídy `Ifibo_PeriodicControlThread`. Jako první se volá metoda `ReadFromDevices()`, která zavoláním metod jednotlivých vstupních modulů načte vstupní data do programu, a uloží je do příslušných datových bloků. Následuje vykonání metody `CalculateControl()`, která zavolá stejnojmenné metody objektů představujících regulátory. V metodě WriteToDevices() jsou pak nově vypočtené hodnoty akčního zásahu poslány pomocí objektů výstupních modulů zpátky na sběrnici RS-485. Metoda `SendPeriodicPublications()` nakonec vyšle všechna data ostatním stanicím připojeným k lokálnímu ORTE manažeru.

V nynější implementaci třídy se v metodě `SendPeriodicPublications()` periodicky zasílají všechny hodnoty, ne pouze hodnoty změněné během jednoho řídicího cyklu. Jelikož těchto hodnot není veliké množství, tak to nevadí, a řeší se tím dva implementační problémy. Prvním by byla nutnost kontrolovat změny hodnot v jednotlivých blocích a podle toho vysílat data. Druhý problémem vzniká při připojování odběratelů hodnot publikací během provozu, kdy by tyto odběratelé museli zaslat řídicímu programu dotaz na aktuální stavy všech veličin v řídicím programu. Do budoucna by bylo vhodné toto vyřešit vytvořením speciální subskripce v řídicím programu, která by přijímala příkazy od vzdálených stanic. Jedním z příkazů by byla žádost o hodnoty všech proměnných v řídicím programu. Přesto by mělo zůstat periodické zasílání hodnot ze senzorů a nejspíš i stavů alarmů.

Komunikace mezi jednotlivými vlákny řídicího programu probíhá pouze přes objekty datových bloků, které mají implementováno vyloučení vícenásobného přístupu k datům. Žádná jiná synchronizace se zde neprovádí.

Na závěr uvedeme, jak vypadá funkce `main()` řídicího programu:

```
int main(void)
{
    Ifibo_System  system; // vytvoření objektu System
    system.Open();        // inicializace a spuštění
```

```
                        // řídicího programu

    // čekání na skončení všech vláken
    ACE_Thread_Manager::instance()->wait();
    return 0;
}
```

9 Závěr

Výsledkem této práce je funkční řídicí systém pro demonstrační model automatizovaného chovu ryb.

Pro vytvoření řídicího programu byl použito objektově orientovaného přístupu. To umožňuje jednoduché rozšíření programu o další funkce. Příkladem může být řízení chodu čerpadla logickým regulátorem, které bylo do programu přidáno v pozdní fázi projektu aniž bylo nutné cokoliv v již vytvořených třídách měnit (to se ovšem netýká třídy `Ifibo_System`).

V této práci byla vytvořena sada objektů – komponent pro konstrukci řídicího programu. Ty jsou navržené pro použití v obecném řídicím systému, nezávisle zde na řešeném problému. Z časových důvodů však byly tyto komponenty vytvořeny zjednodušené například v porovnáním se vzorem – PROFIBUS PA Profilem. Nebudou tudíž vyhovovat pro řešení úplně libovolného řídicího systému.

Použitím knihoven ACE a ORTE bylo dosaženo přenositelnosti řídicího programu na průnik množin jimi podporovaných platforem. Mezi ně patří Linux a MS Windows, což jsou v dnešní době dvě nejpoužívanější platformy. Primární je ale operační systém Linux, pro který je napsaná i jediná omezeně přenositelná třída `Ifibo_SerialDevice`.

Jak už bývá u softwarových projektů zvykem, vždy je co vylepšovat. Zde jsou popsány možné oblasti.

Pro vytvoření vizualizace řídicího systému by bylo výhodné použít některý ze standardních nástrojů používaných v automatizaci. Tyto aplikace, jmenovitě Wonderware Factory Suite nebo Iconics Genesis32, obsahují ovladač pro komunikaci s řídicím systémem přes rozhraní OPC. Vytvoření programu pro konverzi protokolů mezi ORTE a OPC je řešením tohoto problému. Jelikož OPC je v automatizaci rozšířený standard, zvýšila by se tak použitelnost řídicího programu, který by mohl být propojen s jinými řídicími systémy, aplikacemi archivujícími data apod.

V současné verzi řídicího programu jsou při startu programu nastaveny parametry řízení podle konfiguračního hlavičkového souboru. Jsou tedy součástí binárního souboru programu a pro jejich změnu je nutné program znovu přeložit. To není zase takový problém, jelikož změna parametrů pro řízení se provádí pouze při výměně chovaných ryb za jiné s odlišnými požadavky na teplotu vody, obsah kyslíku apod. Přesto by bylo výhodné doplnit třídu System o metodu, která by načetla soubor s konfigurací systému a podle něj nastavila řízení.

10 Použité zdroje

[1] Cíl, T.: *Komunikace pro průmyslovou automatizaci.* Diplomová práce. Praha, ČVUT, Elektrotechnická fakulta, Katedra řídící techniky, 2002. Vedoucí diplomové práce Dr. Ing. Zdeněk Hanzálek.

[2] Špinka, O.: *Demonstrační prostředí pro vestavěné systémy.* Diplomová práce. Praha, ČVUT, Elektrotechnická fakulta, Katedra řídící techniky, 2004. Vedoucí diplomové práce Dr. Ing. Zdeněk Hanzálek.

[3] Pachner, D: *Matlab simulation. Deliverable T1.1.* Project IST-2000-31 080 IFiBO, 2002.

[4] *Stránky projektu IFiBO*
`http://dce.felk.cvut.cz/ifibo`

[5] *MITE – MIkropočítačová TEchnika*
`http://www.mite.cz`

[6] *Advantech*
`http://www.advantech.com`

[7] *Gryf – elektronické přístroje*
`http://www.gryf.cz`

[8] *Operační systém Linux*
`http://www.linux.cz`

[9] *Stránky o svobodném software*
`http://www.gnu.cz`

[10] Abbott, D.: *Linux for Embedded and Real-time Applications.* Newnes, Amsterdam, 2003.

[11] Drepper, U. – Molnar, I.: *The Native POSIX Thread Library for Linux.*
`http://people.redhat.com/drepper/nptl-design.pdf.`

[12] Young, S. J.: *Programovací jazyky pro RT-aplikace.* Nakladatelství technické literatury, Praha, 1988.

[13] Wheeler, D. A.: *Ada, C, C++, and Java vs. The Steelman.*
`http://www.adahome.com/History/Steelman/steeltab.htm.`

[14] Eckel, B.: *Thinking in C++. Volume 1, 2nd Edition.* Prentice Hall, Upper Saddle River, 2000.

[15] *C++ Coding Standard*
`http://www.possibility.com/Cpp/CppCodingStandard.html`

[16] *The ADAPTIVE Communication Environment (ACE)*
`http://www.cs.wustl.edu/~schmidt/ACE.html`

[17] Schmidt, D. C.: *Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes.*
`http://www.cs.wustl.edu/~schmidt/PDF/wrapper-facade.pdf.`

[18] Schmidt, D. C.: *An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit.*
`http://www.cs.wustl.edu/~schmidt/PDF/ACE-concurrency.pdf.`

[19] *OCERA Project*
`http://www.ocera.org/`

[20] Gallmeister, B. O.: *POSIX.4: Programming for the Real World.* O'Reilly & Associates, Inc., 1995.

[21] Mitchell, M. – Oldham, J. – Samuel, A.: *Advanced Linux Programming.* New Riders Publishing, Boston, 2001.

[22] Sweet, M. R.: *Serial Programming Guide for POSIX Operating Systems. 5th Edition, 3rd Revision.*
`http://www.easysw.com/~mike/serial.`

[23] White, B.: *Linux 2.6: A Breakthrough for Embedded Systems. Revision Sep. 9, 2003.*
`http://linuxdevices.com/articles/AT7751365763.html.`

[24] Bruyninckx, H.: *Real Time and Embedded Guide, Revision 0.04, December 2002.*
`http://people.mech.kuleuven.ac.be/~bruyninc/rthowto.`

[25] *Realtime Application Interface*
`http://www.aero.polimi.it/~rtai`

[26] *Real-time operační systém eCos*
`http://sources.redhat.com/ecos`

[27] *PROFIBUS-PA Profile for Process Control Devices. Version 3.*
PROFIBUS Nutzerorganisation e. V., Karlsruhe, 1999.

Příloha A Obsah přiloženého CD

Na přiloženém CD jsou jednotlivých adresářích obsaženy tyto data:

`docs` obsahuje dokumenty týkající se použitých komponent řídicího systému v následujících podadresářích
- `advantech` – dokumentace k modulům Advantech ADAM
- `gryf` – dokumentace k měřicím modulům Gryf
- `krmitko` – popis komunikačního protokolu krmítka
- `mite` – dokumentace týkající se Mite Dimm-PC a MiteLinuxu

`doxygen` – dokumentace jednotlivých tříd řídicího programu vytvořená programem Doxygen

`dp` obsahuje elektronickou podobu tohoto dokumentu.
- `tex` – zdrojové soubory pro systém TEX
- `obrazky` – obrázky použité v tomto dokumentu, ve formátu OpenOffice

`jrc` obsahuje program JRC pro instalaci operačního systému na Dimm-PC přes sériový port.

`libraries` obsahuje zdrojové texty použitých knihoven
- `ace` – knihovna ACE, verze 5.4
- `orte` – knihovna ORTE, verze 0.2.2

`linux_image` obsahuje binární obraz flash disku Dimm-PC s nainstalovaným MiteLinuxem a se zprovozněným řídicím programem. Pro instalaci stačí použít program JRC.

`mereni` obsahuje data naměřená při regulaci na modelu

`source` obsahuje zdrojové texty pro řídicí systém.
- `src` – soubory se zdrojovými texty všech tříd. Zde je Makefile pro vytvoření řídicího programu
- `controller` – utilita pro vzdálené ovládání řídicího programu
- `monitor` – utilita pro vzdálené monitorování řídicího procesu
- `orte` – include soubory třídy ORTE

Příloha B Popis konfigurace řídicího systému

Adresy modulů na sběrnici RS-485

0x01 ADAM 4060
0x02 ADAM 4053
0x03 GRYF 9401
0x04 GRYF 9202
0x05 Krmítko

Připojení akčních členů k jednotlivým kanálům modulu ADAM 4060

0 Osvětlení
1 Topná tělesa
2 Vzduchovací motorek
3 Čerpadlo

Připojení kapacitních senzorů hladiny ke kanálům modulu ADAM 4053

0 Senzor maximální hladiny v retenční nádrži
1 Senzor minimální hladiny v nádrži u čerpadla

Označení svorkovnic

PE Ochraný vodič
N Střední vodič
L Fázový vodič 230 V střídavých

+24 Kladný pól stejnosměrného napětí 24 V
GND Zem stejnosměrného napětí

+D Kladný datový vodič sběrnice RS-485
-D Záporný datový vodič sběrnice RS-485

T Svorkovnice pro topná těles
S Svorkovnice pro osvětlení
Č Svorkovnice pro čerpadlo
V Svorkovnice pro vzduchovací motorek

Příloha C Instalace MiteLinuxu

Instalace MiteLinuxu z daného binárního obrazu flash disku se skládá ze tří kroků.

Inicializace spojení

Propojíme pracovní stanici na které běží MS Windows nebo DOS sériovým kabelem s DimmPC. U DimmPC použijeme port COM1. Na pracovní stanici spustíme program `jrc.exe` s parametry `connect`, použitým COM portem a požadovanou přenosovou rychlostí, například `>jrc connect COM2 115200`. Nyní restartujeme DimmPC. Za chvíli by měl program `jrc` ohlásit úspěšné propojení a skončit.

Zápis dat

Nyní už máme oba počítače propojeny, můžeme přenést data. K tomu použijeme program `jrc` s parametry `diskwrite 128` a jménem souboru s binárním obrazem flash disku, například `>jrc diskwrite 128 linux.img`. Začne přenos dat.

Ukončení spojení

Po dokončení přenosu dat ukončíme spojení příkazem `>jrc reboot`.