TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta strojní

Studijní obor : 3902T021 Automatizované systémy řízení ve strojírenství
Zaměření : Automatizace inženýrských prací

Katedra aplikované kybernetiky

Možnosti zpracování analogového a digitálního signálu s využitím PC a multifunkční PCI měřící karty

Possibilities of analog and digital signal processing with utilization of PC and multifunction PCI measuring card

Josef Machytka

Vedoucí diplomové práce : prof. Ing. Miroslav Olehla, CSc.

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta strojní

Katedra aplikované kybernetiky Studijní rok: 2005/06

ZADÁNÍ DIPLOMOVÉ PRÁCE

Jméno a příjmení: **Josef MACHYTKA**

Studijní program: **M2301 Strojní inženýrství**

Obor: **3902T021 Automatizované systémy řízení ve strojírenství**

Zaměření: **Automatizace inženýrských prací**

Ve smyslu zákona č. 111/1998 Sb. o vysokých školách se Vám určuje diplomová práce na téma:

Možnosti zpracování analogového a digitálního signálu s využitím PC a multifunkční PCI měřící karty

Zásady pro vypracování:
(uveďte hlavní cíle diplomové práce doporučené metody pro vypracování)

1. Základní rozbor možností vývojových prostředí MatLab, ControlWeb a LabView.
2. Vytipujte vhodné měřící I/O multifunkční karty pro analýzu nízkonapěťových analogových signálů pro dotykové měření teplot s využitím termočlánků typu K, J, E a dalších.
3. V jazyce C++ naprogramujte uživatelské rozhraní pro komunikaci s multifunkční měřící kartou, s vizualizací průběhu měřeného signálu v reálném čase s možnostmi:
   a) nezávislé on-line kalibrace jednotlivých kanálů,
   b) nastavení vzorkování snímání signálu,
   c) statistického vyhodnocování naměřených dat,
   d) ukládání/načítání dat měření, exportu dat v textovém formátu,
   e) ethernetové komunikace ze vzdáleného PC (model server-klient).
4. Funkčnost systému otestujte při provozním měření.

ANOTACE

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta strojní

Katedra aplikované kybernetiky

Studijní obor :	3902T021 Automatizované systémy řízení ve strojírenství
Studijní zaměření :	Automatizace inženýrských prací
Diplomant :	Josef Machytka
Téma práce :	Možnosti zpracování analogového a digitálního signálu s využitím PC a multifunkční PCI měřící karty Possibilities of analog and digital signal processing with utilization of PC and multifunction PCI measuring card
Rok obhajoby DP :	2006
Vedoucí DP :	prof. Ing. Miroslav Olehla, CSc.

Stručný výtah :

Práce si klade za cíl demonstrovat možnost použití programovacího jazyka C++, dynamicky linkovaných knihoven a příslušných hlavičkových souborů dodávaných výrobcem měřící karty, k vytvoření multiplatformně orientované aplikace umožňující komunikaci s měřící kartou a distribuci dat prostřednictvím počítačové sítě.

Abstract :

The target of this work is to demonstrate a possibility of use the C++ programming language, dynamically linked libraries and relevant header files distributed by producer of measuring card, to create a multiplatform oriented application enabling communication with measuring card and data distribution by means of computer network.

Obsah

Úvod

Vzhledem k přirozené snaze co nejlépe uspět na trhu se dnešní průmyslové podniky stále častěji potýkají s nutností nejen zvýšení produktivity a kvality výroby, ale také s nutností rychlé reakceschopnosti na měnící se trendy a požadavky zákazníků. S těmito skutečnostmi a také s ohledem na další běžné požadavky výroby jsou často bezpodmínečně spojeny vysoké nároky na odpovídající technické i softwarové vybavení. Nedílnou součástí výrobních linek jsou prvky ovládané nebo monitorované prostřednictvím analogových nebo digitálních napěťových signálů, jedná se především o měřící vybavení, nejrůznější snímače a akční členy založené na elektromotorickém pohonu. Možným klíčovým prvkem propojení uvedených signálů se softwarovým vybavením osobních počítačů jsou tzv. měřící karty. Softwarové aplikace potom ve spolupráci s měřícími kartami umožňují kromě čtení vstupů a ovládání výstupů provádět také složitější operace jako např. hlídání povolených mezí, automatické řízení, archivace a statistické zpracování dat. V současné době lze vzhledem k uvedeným účelům hovořit o používání několika rozšířených softwarových produktů a to konkrétně MATLAB, Control Web a LabVIEW. Tato softwarová vybavení poskytují značně sofistikované funkce, které ovšem mnohdy přesahují rámec méně náročných potřeb a s ohledem na vysokou cenu těchto produktů, která se může pohybovat řádově až ve statisících korun za jednu licenci, nemusejí být pořizovací náklady vždy úplně rentabilní.

Hlavním cílem této práce je tedy demonstrovat možnost použití programovacího jazyka C++, příslušných hlavičkových souborů a dynamicky linkovaných knihoven zajišťujících komunikaci s měřící kartou a dodávaných jejím výrobcem, k vytvoření vlastního kvalitativně jednoduššího, ovšem všechny konkrétní požadavky splňujícího, softwarového vybavení. Tvorba takové aplikace se může stát finančně dostupnějším řešením, které navíc není omezeno počtem použitelných licencí.

Práce je rozdělena do čtyř kapitol. První kapitola stručně popisuje základní možnosti softwarových vybavení MATLAB, Control Web a LabVIEW, druhá pojednává o možnostech používaných měřících karet a jejich použití s termočlánky. Třetí kapitola představuje stěžejní bod celé práce, tedy tvorbu vlastního softwarového vybavení dle požadavků zadání, a čtvrtá kapitola ověřuje funkčnost tohoto vybavení jeho nasazením na reálné provozní měření.

1. Popis vybraných softwarových produktů běžně používaných v technické praxi

1.1 MATLAB

Jedná se o špičkový programový produkt dalece přesahující možnosti pouhé komunikace s měřícími kartami za účelem provádění měření, regulace nebo vizualizace dat. Jeho použití je velice všestranné. Kromě zmíněného poskytuje možnosti pro nasazení nejen v mnoha oblastech průmyslu, ale také vědy, výzkumu i obecného vzdělávání. MATLAB v dnešní podobě po dlouholetém vývoji a s ním spojenému mimořádně rychlému výpočetnímu jádru s optimalizovanými algoritmy představuje zaslouženě určitý celosvětový standard v oblasti technických výpočtů, modelování a simulace nejrůznějších technických problémů. Poskytuje nejen mocné grafické a výpočetní nástroje, ale také spolu s vlastním objektově orientovaným programovacím jazykem, srovnatelným například s C nebo Fortran, také mimořádně rozsáhlé cílově orientované knihovny funkcí použitelné takřka ve všech oblastech lidské činnosti. Za naprostou samozřejmost jsou považovány možnosti pro síťovou distribuci dat a vzdálené využití výpočetního potenciálu MATLABu prostřednictvím MATLAB serveru. Bližší pohled na výpočetní jádro MATLABu potom ukazuje na velmi značný potenciál pro práci nejen s maticemi a vektory, ale také s dalšími složitějšími datovými typy, jakými jsou např. vícerozměrná pole reálných nebo komplexních čísel a struktury s prvky odlišných typů. Díky tomu lze vytvářet a efektivně pracovat s téměř libovolně složitými datovými strukturami. Výpočetní jádro je implementováno s využitím nejmodernějších knihoven LAPACK a ARPACK a je schopné se adaptivně přizpůsobovat hardwarovým možnostem konkrétního počítače, takže i na slabších strojích je dosahováno znatelného výkonu. Implementovaný algoritmus pro výpočet Fourierovy transformace je dokonce považován za nejrychlejší známý algoritmus pro tuto úlohu.

MATLAB je k dispozici pro všechny významné softwarové platformy, tedy operační systémy MS Windows, GNU/Linux i UNIX. Používá otevřenou architekturu, což nejen přispělo k jeho velkému rozšíření, ale také inspirovalo mnoho firem k vývoji distribucí vlastních produktů rozšiřujících výpočetní prostředí MATLABu o další knihovny nebo zajišťujících jeho propojení s dalšími aplikacemi. Jednotlivé knihovny funkcí se nazývají toolboxy a vzhledem k opravdu univerzálním možnostem použití MATLABu jsou s ohledem na výslednou cenu produktu samostatně dokoupitelné. Cena tohoto programového

vybavení se liší v závislosti na typu použití. Jiná cena je stanovena pro komerční využití, jiná pro vládní účely a jiná pro školní a vzdělávací účely. Řádově lze ovšem hovořit o nákladech v řádu statisíců korun. Speciální rozšiřující knihovny potom představují další desetitisíce a není žádnou výjimkou, aby výsledná investice činila několikamiliónový výdaj. Uvedené skutečnosti lze závěrem a bez jakékoliv nadsázky shrnout do konstatování, že MATLAB je ultimátní špičkový softwarový produkt v technické praxi s všestrannými možnostmi využití a nasazení.

1.2 Control Web

Control Web koncepčně vychází z osvědčené architektury svého předchůdce Control Panel a v kontrastu s programovým vybavením MATLAB představuje cenově dostupnější ovšem úzce orientovaný nástroj pro vývoj aplikací reálného času, nebo aplikací pracujících na principu reakce na změnu dat. Uvedený výrok nástroj by do značné míry mohl schopnosti tohoto produktu negativním způsobem zkreslit, ovšem skutečnost vypovídá naopak o kvalitách vysokých. Cílová orientovanost tu samozřejmě hraje svou roli, ale i v těchto stanovených mezích, kdy je řeč o zmíněném druhu aplikací, se jedná o univerzální softwarové vybavení. Snadno a poměrně rychle lze vytvářet vizualizační a řídící aplikace, aplikace sběru, ukládání a vyhodnocování dat a aplikace systému člověk-stroj. Objektově orientovaná komponentová architektura zajišťuje aplikacím vyvinutým v tomto prostředí široké nasazení s vysokou mírou pružnosti v přizpůsobování nejrůznějším konkrétním požadavkům. Vnitřní databáze těchto komponent obsahuje takřka všechny používané prvky řízení a vizualizace, dále alarmy, archívy, historické trendy a podobně. Množina těchto takzvaných virtuálních přístrojů ovšem není pevně dána a zabudována v systému, každý přístroj představuje dynamicky linkovanou knihovnu automaticky detekovanou při startu systému. Díky již zmíněné otevřené komponentové architektuře tedy není problém jejich množinu libovolně rozšiřovat. Vlastní aplikace vzniká propojením, nastavením vlastností a případným bližším programovým přizpůsobením jednotlivých virtuálních přístrojů, přičemž značnou část tvorby lze provádět prostřednictvím dialogů a průvodců. Není to samozřejmě jediná cesta, kdykoliv lze přecházet mezi textovým a grafickým návrhem aplikace. Control Web poskytuje podporu pro ovladačový standard DDE a OPC, ovšem implementuje také vlastní nativní ovladače schopné pro účely této aplikace pracovat efektivněji. Rozhraní těchto ovladačů je přitom plně dokumentováno a otevřeno, každý si tedy může napsat ovladač vlastní podle svých potřeb.

Stejně jako v předchozím případě i Control Web samozřejmě poskytuje nástroje pro síťovou distribuci dat prostřednictvím vestavěného http serveru a pro obecný vzdálený přístup k jakémukoliv virtuálnímu přístroji. Obecně řečeno všechny tyto systémy by takovéto možnosti měly poskytovat, protože do značné míry se jedná zároveň o systémy informační. Náklady spojené s pořízením Control Webu představují řádově desetitisíce korun.

1.3 LabVIEW

V případě již přes dvacet let vyvíjeného softwarového vybavení LabVIEW lze hovořit o určité podobnosti s předchozím produktem Control Web. Jedná se nejen o grafické prostředí určené pro vývoj řídících aplikací, aplikací sběru, archivace, vizualizace, analýzy a prezentace dat, ale také o prostředí založené na vlastním programovacím jazyku G umožňujícím vytváření programů ve formě blokových diagramů, přičemž lze programovat jakoukoliv úlohu. Díky koncepci tohoto systému je tvorba nových a úprava již existujících aplikací značně rychlá. Obdobně jako u předešlých popsaných produktů je zde využita knihovna funkcí a nástrojů a vzhledem k velkému rozšíření LabVIEW také výrobci nejrůznějšího měřícího a dalšího hardwarového vybavení tuto knihovnu rozšiřují o svoje produkty. Aplikace vytvořené pomocí LabVIEW jsou nazývány virtuálními přístroji (Virtual Instruments - VIs), protože jejich vhled a činnost je podobná skutečným přístrojům, nicméně s ohledem na jejich koncepci jsou také podobné funkcím konvenčních programovacích jazyků. LabVIEW obsahuje kompilátor jazyka G generující optimalizovaný kód, přičemž výkon výsledných aplikací je plně srovnatelný s výkonem aplikací vytvořených nízkoúrovňovými programovacími jazyky, jakým je například jazyk C. Komfort při vytváření aplikací je ale podstatně vyšší, protože programátor se zbavuje starostí s řadou syntaktických detailů konvenčního programování a může se plně soustředit na řešení zadaného problému. Náklady spojené se zakoupením licence LabVIEW představují až několik set tisíc korun.

2. Dotykové měření teplot s využitím měřících karet a termočlánků

S měřením teploty se v technické praxi lze pochopitelně setkat velmi často, přičemž dotykové měření je nejen běžnější a přesnější, ale také méně nákladné. Označení dotykové lze vysvětlit principem tohoto způsobu, protože měřící část snímače teploty je v přímém kontaktu s měřeným prostředím nebo předmětem. V opačném případě se jedná o měření bezdotykové.

2.1 Používané typy termočlánků

Jak známo termoelektrický článek neboli termočlánek je tvořen dvěma na konci spojenými vodiči z různých materiálů, což vede ke vzniku měřitelného rozdílu elektrických potenciálů, tedy elektrického napětí. Velikost tohoto napětí je závislá na použitém materiálu obou větví, teplotě měřícího spoje a rozdílu teplot měřícího a srovnávacího spoje, přičemž větve termočlánku musí být homogenní. Termoelektrické napětí nezávisí na případném třetím materiálu v obvodu, pokud tento má na obou spojích, kterými je do obvodu včleněn, stejnou teplotu. Kombinace materiálů pro výrobu termoelektrických článků mají vykazovat pokud možno velký a lineární přírůstek změny termoelektrického napětí v závislosti na teplotě, stabilitu údaje při dlouhodobém provozu a odolnost proti chemickým a mechanickým vlivům.

Typ	Termoelektrická dvojice	Rozsah měřitelných teplot [°C] (druhý údaj krátkodobě)
S	PtRh10 - Pt	0 až 1300 / 1600
R	PtRh13 - Pt	0 až 1300 / 1600
B	PtRh30 - PtRh6	300 až 1600 / 1800
M	Cu - k	-200 až 100
T	Cu - CuNi	-200 až 400
J	FeCu - Ni	-200 až 700 / 900
L	NiCr - CuNi	-200 až 600 / 800
E	NiCr - CuNi, ch - k	-100 až 700 / 900
K	NiCr - Ni, ch - a	-200 až 1000 / 1300
A	WRe5 - WRe20	0 až 2200 / 2500
N	nikrosil - nisil	-270 až 1300

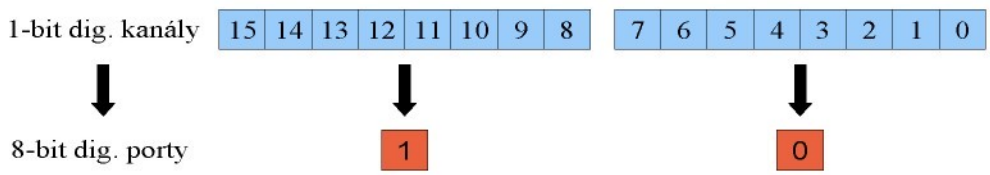
Tabulka 1: Základní vlastnosti používaných typů termočlánků.

U různých typů termočlánků je doporučeno různé pracovní prostředí (vakuum, oxidační, redukční, inertní), což ovšem z hlediska problému použití termočlánku s měřící kartou nehraje žádnou roli.

2.2 Základní popis možností běžně dostupných měřících karet

Měřící karty lze charakterizovat jako integrované obvody určené pro propojení obecných napěťových signálů se softwarovým vybavením osobních počítačů. Použití měřících karet je vzhledem k uvedenému účelu zřejmé a sahá nejen do oblasti průmyslové výroby, ale také například do zdravotnictví, nejrůznějších experimentálních měření vědeckého výzkumu a obecně i do informačních technologií.

Jednotlivé měřící karty se liší počtem a typem vstupních/výstupních kanálů, podporovaným rozsahem vstupního/výstupního napětí a případně počtem těchto rozsahů. Převod vstupního elektrického napětí na jeho digitální reprezentaci a opačně je realizován prostřednictvím A/D a D/A převodníků, přičemž je očekávána velmi vysoká přesnost převodu. Maximální podporovaný rozsah napětí se většinou řídí unifikovaným rozsahem pro tyto účely, tedy 0–5 V, 0–10 V v obou polaritách. Jednotlivé kanály lze rozlišit na analogové a digitální. Digitální reprezentace hodnoty na analogovém vstupu po A/D převodu je představována datovým typem s pohyblivou řádovou čárkou, ovšem v případě digitálních vstupů je situace jiná. Digitální kanály rozlišují pouze dva stavy, konkrétně stav 0 (false, nepravda) a stav 1 (true, pravda). Pokud je hodnota vstupního napětí nižší než stanovená mez (bývá 2,4 V), pak kanál hlásí stav 0, v opačném případě 1. Z uvedeného vyplývá, že digitální kanály lze samostatně smysluplně využít pouze pro monitorování (vstup) nebo ovládání (výstup) obecných dvoustavových prvků (např. dvoupolohová čidla a regulátory). Sdružením více digitálních kanálů lze rozlišit takový počet hodnot, jaký odpovídá maximální hodnotě reprezentované pomocí daného počtu bitů. Sdružením osmi těchto kanálů vzniká osmibitový digitální port schopný rozlišit hodnoty 0-255, tedy hodnoty v rozsahu 1 byte.



Obr. 1: Vztah digitálních kanálů a 8-bitových digitálních portů.

Na obrázku 1 je znázorněn případ měřící karty disponující šestnácti digitálními kanály, které lze použít samostatně nebo jako dva digitální porty. Samozřejmě je také možné použití osmi kanálů a jednoho portu.

Některé měřící karty kromě analogových/digitálních kanálů/portů poskytují určité další nástroje jako například čítače. Takové karty jsou již nazývány kartami multifunkčními. Rozdělení měřících karet podle typu sběrnice (PCI, PCI-e, ISA, USB atd.) není z hlediska systémového přístupu podstatné. Typ sběrnice pochopitelně odpovídá časové úrovni výroby a účelu daného hardwaru a souvisí s maximální rychlostí vzorkování, resp. s rychlostí přenosu daných informací do aplikačního prostředí. Z programátorského hlediska lze ještě uvést podporu některých karet pro obsluhu událostí, která bude blíže popsána dále.

2.3 Systémový přístup k problému použití termočlánku s měřící kartou

Systémový přístup k řešení obecného problému lze popsat jako snahu od sebe oddělit jednotlivé samostatné funkční celky a k danému problému přistoupit nezávisle z pozice každého z nich. Aplikace systémového přístupu na problém použití termočlánku s měřící kartou tedy odděluje první celek reprezentovaný měřící kartou od druhého celku reprezentovaného termočlánkem. Termočlánek je z energetického hlediska zdrojem napětí, měřící karta voltmetrem, pro který není druh zdroje podstatný. Jediným podstatným faktorem je velikost napětí, které do měřící karty příslušným analogovým kanálem vstupuje. Naopak termočlánku nezáleží na tom, zda je vzniklé termoelektrické napětí někde měřeno či ne.

2.4 Vhodné měřící karty pro použití s termočlánky

Jediným kritériem pro stanovení vhodnosti použití konkrétního termočlánku s konkrétní měřící kartou je vztah hodnoty vzniklého termoelektrického napětí s podporovaným vstupním rozsahem měřící karty. Vzhledem k tomu, že pro všechny typy termočlánků se termoelektrické napětí pohybuje v řádu milivoltů a měřící karty podporují rozsahy značně vyšší, lze říci, že danému kritériu odpovídají všechny standardní měřící karty a otázka zadání je do určité míry irelevantní. Jedinou výjimku tvoří pouze případ karet nepodporujících zápornou polaritu na analogovém vstupu při současném měření termočlánkem v takovém teplotním rozsahu, kdy je hodnota vzniklého termoelektrického napětí záporná.

3. Tvorba vlastního softwarového vybavení v jazyce C++

3.1 Počáteční stádium realizace

3.1.1 Analýza požadavků, stanovení jasných cílů

Vzhledem k charakteru problému a také z požadavků zadání vyplývá fakt, že se bude jednat o tzv. aplikaci reálného času, která kromě běžných očekávatelných možností, jakými jsou v tomto případě nastavení periody vzorkování, vizualizace dat, ukládání/načítání dat a jejich export v textovém formátu, musí poskytovat také rozšířené funkce pro online kalibraci vstupních hodnot a výpočet statistik. Současně musí být zajištěna možnost provádět komunikaci s měřící kartou vzdáleně prostřednictvím počítačové sítě, což už v této fázi aplikaci potenciálně předurčuje k rozdělení na více uzavřených celků. Zadání sice počítá pouze s prováděním měření, tj. čtení vstupních kanálů/portů, ale vzhledem ke snaze vytvořit obecně použitelnou aplikaci je možnost ovládání případného výstupu chápána jako samozřejmost. Ze stejného důvodu bude softwarové vybavení poskytovat určité rozšířené uživatelské nástroje pro nastavení parametrů vizualizace jednotlivých elementů grafu a jeho export do bitmapových datových formátů. V rámci vizualizace dat bude aplikace umožňovat sledování okamžitých (resp. posledních známých) hodnot číselně ve zvláštních oknech a definici rozšířených os s vlastními jednotkami a stupnicí. Výsledná aplikace nesmí sloužit pouze pro účely jednorázového měření/ovládání (dále jen měření), ale samozřejmě musí být schopna dříve uložená data otevřít a dále s nimi pracovat.

Pro účel vývoje zadaného softwarového vybavení byla poskytnuta měřící karta Advantech 1710HG (PCI). Tato karta disponuje šestnácti vstupními a dvěma výstupními analogovými kanály, dále šestnácti vstupními i výstupními digitálními kanály a pro uživatelské aplikace také jedním pevně nastaveným čítačem. Jak již bylo uvedeno, aplikace bude řešena univerzálně a nebude se tedy nevhodnou vnitřní konstrukcí omezovat na použití pouze jedné konkrétní měřící karty. Z tohoto důvodu je nutné vytvořit určitou jednotnou komunikační formu aplikace s měřící kartou a umožnit co nejpohodlnější a nejrychlejší přizpůsobení kartě jiné. Uvedený cíl lze realizovat systémem globálních definic popisujících vlastnosti měřící karty a univerzálním komunikačním rozhraním představovaným vlastním C++ objektem. Přizpůsobení jiné měřící kartě potom bude představovat pouze

úpravu implementace daného objektu a správné nastavení globálních definic vlastností měřící karty, k čemuž budou také k dispozici předpřipravené prázdné šablony. Realizace tohoto přizpůsobení je samozřejmě podmíněna dostupností potřebných knihoven a hlavičkových souborů dodávaných výrobcem konkrétní měřící karty.

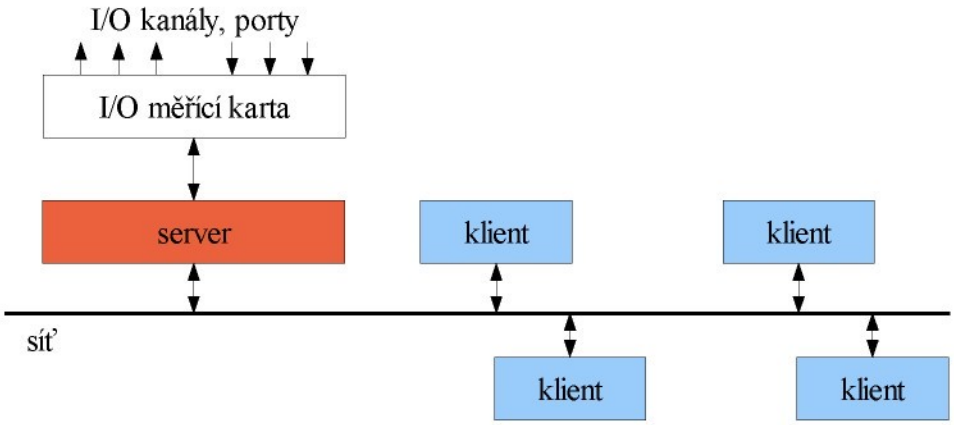
3.1.2 Obecné požadavky kladené na aplikace reálného času

Aplikace reálného času jsou takové aplikace, které určitým způsobem manipulují s daty nebo vykonávají příslušné algoritmy v závislosti na čase, konkrétně v přesných časových intervalech. Jedná se o tzv. deterministické časování. Z podstaty těchto aplikací nevyplývá jen jejich označení, ale také několik obecných kritérií, která musí být pokud možno splněna. Jedná se především o zajištění co nejpřesnějšího časování a o správnou interpretaci výsledků vzhledem k jejich možnému dopravnímu zpoždění směrem k uživateli. První z uvedených kritérií souvisí se stanovením správného okamžiku k provedení žádaného algoritmu, což se může jevit jako banální problém, ovšem ukazuje se, že ve specifických případech může být splnění tohoto kritéria komplikované. Nutnost zabývat se splněním druhého kritéria přichází v úvahu pouze v takových situacích, kdy je sice zajištěno správné časování, ovšem samotné vykonání daného algoritmu nebo případný přesun výsledných informací směrem na uživatelský výstup je potenciálně opožděné vzhledem k času události pro výkon daného algoritmu. Již v tuto chvíli je zřejmé, že právě s nutností splnit druhé uvedené kritérium se softwarové vybavení bude potýkat. Přesun informací jakoukoliv počítačovou sítí představuje i přes dnešní velmi vysoké kapacity těchto sítí potenciální dopravní zpoždění, které je samozřejmě nutné respektovat, i když se může řádově pohybovat pouze v milisekundách.

3.1.3 Rozdělení problému na dva uzavřené celky, server a klient

Jak již bylo naznačeno analýzou požadavků kladených na vytvářené softwarové vybavení, nutnost zajistit možnost vzdálené komunikace s měřící kartou je s ohledem na systémový přístup k tomuto problému víceméně předurčením rozdělení aplikace na dva uzavřené celky, tedy server a klient. Samozřejmě by bylo možné přistoupit k tomuto problému opačně a koncipovat aplikaci jako jeden velký celek s funkcemi pro komunikaci s měřící kartou, vizualizaci a zpracování dat a zároveň jejich distribuci prostřednictvím počítačové sítě, ale takovéto řešení se nejeví jako příliš vhodné, přestože by samozřejmě bylo

funkční. Rozdělením na server a klient je dosaženo určitého zjednodušení a zpřehlednění situace a v konečném důsledku samozřejmě i snížení paměťových nároků dílčích aplikačních procesů oproti případu řešení na bázi jednoho velkého celku. Proč by jedna v uvozovkách velká aplikace měla plnit konkrétní účel s tím, že její další funkce případně nejsou využity a zabírat v paměti řekněme dvojnásobný prostor, když podstatně menší aplikace od počátku koncipovaná pro tento jediný konkrétní účel zajistí stejné služby? Pochopitelně v současné době jsou výkony počítačů a jejich paměťové možnosti natolik vysoké, že uvedený důvod se stává téměř zanedbatelným, ovšem z již uvedeného hlediska systémového přístupu je převod problému síťové komunikace na dvě nezávislé aplikace rozhodně vhodnější a také standardní.



Obr. 2: Systémové rozdělení problému síťové komunikace na dvě nezávislé aplikace.

3.1.4 Komunikační schéma server - klient

Schéma komunikace mezi klientem a serverem představuje posloupnost komunikačních operací nutných k dosažení cílového stavu. Cílový stav může znamenat samotné připojení klienta k serveru, specifikaci detailů měření (perioda vzorkování, počet vzorků, použité kanály/porty, rozsah napětí analogových kanálů), zahájení vlastního měření, přerušení měření atd. Za určitých okolností jsou jisté operace (požadavky) nerelevantní a žádná ze stran komunikace se jich nesmí dopustit. Např. není možné, aby klient žádal přerušení měření, pokud toto měření předtím nebylo zahájeno nebo bylo korektně dokončeno. Ze stejného důvodu není možné, aby server zaslal klientovi zprávu s daty měření (s datagramem), pokud klient měření neprovádí. Uvedené a všechny další nerelevantní stavy musí být na obou stranách patřičně ošetřeny a vedou k ukončení komunikace s nahlášením příslušného

chybového kódu. Navržené komunikační schéma je znázorněno tabulkami 2 a 3, přičemž některé uvedené stavy jsou vzájemně kaskádově závislé a k některým nerelevantním stavům z technického důvodu nikdy nedojde.

Cílový stav/událost klienta	**Nutné předpoklady (ostatní stavy nerelevantní)**
připojen k serveru	odpojen od serveru, server žádost o připojení přijal
odpojen od serveru	připojen k serveru, případné probíhající měření automaticky přerušeno a specifikované detaily měření uvolněny
možnost specifikace detailů měření	připojen k severu, měření neprobíhá
možnost uvolnění specifikovaných detailů měření	specifikovány detaily měření, měření neprobíhá
možnost zahájení měření	specifikovány detaily měření, měření neprobíhá
možnost přerušení měření	měření probíhá
možnost nastavení hodnoty konkrétního výstupního kanálu/portu	specifikace detailů měření zahrnuje daný výstupní kanál/port, měření probíhá
server zaslal zprávu	připojen k serveru
server zaslal zprávu s daty měření	měření probíhá
server zaslal zprávu s potvrzením dokončeného měření	měření probíhá, byla přijata zpráva s posledním datagramem
server zaslal zprávu s reakcí na požadovanou akci	připojen k serveru, byla žádána konkrétní akce

Tabulka 2: Komunikační schéma server - klient, strana klienta.

Klient při pokusu o připojení k serveru obdrží zprávu s potvrzením úspěšného připojení, nebo zprávu s informací o vytížené kapacitě serveru. K pokusu o toto připojení může dojít pouze v případě odpojeného stavu. K vlastnímu zahájení měření (proces měření zahrnuje i možnost ovládání výstupu) je nutný předpoklad specifikovaných detailů měření, přičemž možnost této specifikace existuje pouze v případě připojeného stavu. Z uvedeného vyplývá, že jednotlivé stavy nebo události jsou kaskádově závislé. Pokud z nějakého důvodu dojde k odpojení klienta ze strany serveru, případné probíhající měření je tímto automaticky přerušeno a specifikované detaily měření daného klienta na straně serveru automaticky uvolněny.

Cílový stav/událost serveru	Nutné předpoklady (ostatní stavy nerelevantní)
server spuštěn	server zastaven
server zastaven	server spuštěn, automaticky provedeno případné odpojení všech klientů
konkrétní klient připojen k serveru	daný klient odpojen od serveru, kapacita serveru není překročena
konkrétní klient odpojen od serveru	daný klient připojen k serveru, neprovádí měření
konkrétní klient provedl specifikaci detailů měření	daný klient připojen k serveru, neprovádí měření
konkrétní klient uvolnil specifikované detaily měření	daný klient má provedenu specifikaci detailů měření, neprovádí měření
konkrétní klient zahájil měření	daný klient má provedenu specifikaci detailů měření, neprovádí měření
konkrétní klient přerušil měření	daný klient provádí měření
konkrétní klient žádá o nastavení hodnoty výstupního kanálu/portu	specifikace detailů měření daného klienta zahrnuje požadovaný výstupní kanál/port, daný klient provádí měření

Tabulka 3: Komunikační schéma server - klient, strana serveru.

3.1.5 Komunikační protokol, zabezpečení komunikace

Komunikační protokol v tomto případě představuje určitou množinu povolených zpráv (včetně stanovení dílčích pravidel), prostřednictvím kterých spolu budou obě aplikace komunikovat. Navržená struktura komunikačního protokolu samozřejmě ovlivňuje složitost implementace všech dílčích algoritmů pracujících s příchozími nebo odchozími zprávami. S ohledem na tyto skutečnosti je navržená struktura komunikačního protokolu představována jednoduchými textovými zprávami začínajícími identifikačním textem požadavku, následuje rourový znak |, který je pro názornost použit jako oddělovač, následuje informační část zprávy, jejíž struktura se může podle typu požadavku lišit a může také obsahovat další oddělovací znaky. Ukončení zprávy je provedeno textem END za posledním oddělovačem. Ukázka vybraných zpráv, které obě aplikace používají, je uvedena v tabulce 4, kompletní přehled je součástí příloh.

Všechny zprávy budou před vlastním odesláním zakódovány jednoduchým mechanismem, který různě posune ASCII hodnoty jednotlivých znaků a obrátí jejich pořadí.

Při přijetí zprávy pak bude aplikován odpovídající dekódovací algoritmus. Bylo by samozřejmě možné použít pokročilé šifrovací metody a certifikovaný přenos dat prostřednictvím zabezpečených protokolů, ale přistoupení k takto v uvozovkách ostrému řešení není v rámci těchto aplikací nutné. Obecné zajištění bezpečnosti komunikace je ovšem velmi důležité. Obzvlášť při komunikaci přes Internet, což je z hlediska hackerských útoků, přítomnosti různých virů a dalších možných negativních aspektů velmi nebezpečné prostředí, se jedná o nezbytnou nutnost. Obě strany (server i klient) musí veškeré příchozí zprávy kontrolovat na validitu, tzn. musí rozhodnout, zda je zpráva formulována podle stanovených pravidel a zda obsahuje smysluplné informace. Pokud nastane jakákoliv chyba, spojení bude okamžitě ukončeno s ohlášením příslušného chybového kódu. Přehled komunikačních chybových kódů obou aplikací je součástí příloh.

Směr	Zpráva, popis
S-K	`CONNECTION|ACCEPTED|END` - potvrzení úspěšného připojení k serveru
S-K	`CONNECTION|DECLINED_FULL|END` - spojení odmítnuto, kapacita serveru překročena
S-K	`CONNECTION|LIVING|END` - udržovací kontrolní zpráva
K-S	`CLIENT_SPECS|typ řetězec|typ řetězec|typ řetězec|typ bool|END` - specifikace vlastností klienta, obsahuje majoritní číslo verze, jméno uživatele, identifikační řetězec měřící karty a příznak připojení k lokálnímu serveru
S-K	`CLIENT_SPECS|OK|END` - specifikace vlastností klienta v pořádku
S-K	`CLIENT_SPECS|FALSE_VERSION|END` - majoritní verze obou aplikací neodpovídají, spojení je následně ukončeno
S-K	`CLIENT_SPECS|FALSE_CARD|END` - server byl zkompilován pro použití s jinou měřící kartou, spojení je následně ukončeno
K-S	`DETAILS_SPECIFY|typ double|typ long|pole bool|pole bool|...|pole short|END` - specifikace detailů měření, obsahuje periodu vzorkování, počet vzorků, příznaky použití jednotlivých kanálů/portů a kódy napěťových rozsahů analogových kanálů
S-K	`DETAILS_SPECIFY|OK|END` - specifikace detailů měření v pořádku
S-K	`DETAILS_SPECIFY|CONFLICT_typ řetězec|END` - specifikace detailů měření bez efektu z důvodu výstupního konfliktu s jiným klientem, zpráva obsahuje řetězec se seznamem konfliktů
K-S	`MEASURING_START|NULL|END` - klient žádá o zahájení měření

Tabulka 4: Popis vybraných zpráv navrženého komunikačního protokolu.

3.1.6 Funkční studie aplikace server (MIOCS)

Aplikace server nazvaná MIOCS (Multifunction Input Output Card Server) představuje serverovou část použitého modelu server/klient. Její základní úloha spočívá v zajištění vstupní a výstupní komunikace s měřící kartou a distribuce dat klientům prostřednictvím počítačové sítě. Vzhledem k tomu, že výstupní kanály/porty představují určité spojení s ovládáním akčních členů a jiných důležitých prvků a také vzhledem k tomu, že server je navržen s ohledem na možný vyšší počet současně připojených klientů, je nutné poskytovat přístup ke konkrétním výstupním kanálům/portům pouze jednomu klientovi současně a případné stejné požadavky od jiných klientů blokovat. Ke vstupním kanálům/portům mají přístup všichni klienti současně, navíc pro analogové vstupní kanály může každý klient specifikovat vlastní napěťový rozsah, ve kterém data požaduje. Doba měření je s ohledem na široké spektrum možných nasazení stanovena na 0,1 s až 50 dní, perioda vzorkování na 0,1 s až 5 dní.

Server musí monitorovat stav jednotlivých spojení s klienty. Některé síťové prvky na trase mezi serverem a klientem mohou spojení při delší nečinnosti automaticky ukončit, proto je vhodné všem klientům zasílat kontrolní udržovací zprávy, čímž je uvedený problém eliminován zejména v případě nastavené dlouhé periody vzorkování. Pokud nastane problém při procesu čtení vstupů nebo zápisu na výstupy měřící karty, server musí postiženým klientům zaslat informační zprávu o chybě měření a spojení ukončit. Stejnou operaci je nutné provést v případě poruchy časovače, která ovšem postihuje všechny připojené klienty.

Pro účely testování kvality síťového spojení a pro poskytnutí určité přesné reference bude server poskytovat možnost simulace průběhu na vstupních analogových kanálech a vstupních digitálních portech. Nebude-li měřící karta v systému nalezena, server bude pracovat v simulačním režimu, kdy jsou všechny kanály/porty simulované a žádná komunikace s měřící kartou neprobíhá.

Při připojení klienta server musí ověřit, zda jeho konfigurace odpovídá konfiguraci serveru, tj. zda je zkompilován pro použití se stejnou měřící kartou. Toto ověření spočívá v porovnání identifikačního řetězce měřící karty s očekávanou hodnotou.

Přesné časování, které je jedním z nutných kritérií aplikace reálného času, je vhodné implementovat přímo s využitím integrovaného čítače/časovače měřící karty, čímž se problém generování událostí v přesných časových intervalech částečně přenese mimo softwarové zpracování a není tolik závislý na momentálním zatížení operačního systému. Ovšem ne všechny měřící karty jsou vybaveny integrovaným čítačem/časovačem a server tedy musí poskytovat i časování softwarové, které je ovšem velmi vhodné implementovat do vlastního vlákna aplikačního procesu s nastavením vysoké priority, jak bude rozebráno dále.

Většina měřících karet poskytuje pro uživatelské účely jeden a některé dokonce více čítačů/časovačů. Každopádně je nutné spokojit se s případným jedním jediným dostupným čítačem/časovačem a aplikaci s ohledem na tuto skutečnost koncipovat, tzn. události získání dat a jejich zaslání klientům generovat s použitím jednoho externího hardwarového časování nebo jedné instance objektu softwarového časovače. Z uvedeného důvodu je zahájení měření konkrétního klienta nutné sesynchronizovat s událostmi časování, s čímž je spojena nutnost zajistit dostatečně krátký časový interval pro generování těchto událostí, aby zpoždění reálného zahájení měření bylo co možná nejkratší a pokud možno zanedbatelné.

Dalším důležitým problémem, na který je potřeba poukázat, je zacházení aplikace se získanými daty. Je zřejmé, že pokud během jedné časovací události nastane situace, kdy více klientů přistupuje ke stejnému vstupnímu kanálu/portu, bylo by značně neefektivní čtení daného kanálu/portu několikrát za sebou opakovat. Podstatně vhodnějším řešením je dočasný zápis hodnoty poprvé čteného kanálu/portu do paměti a s každým dalším požadavkem pouze navrácení této uložené hodnoty. Algoritmus každé nové časovací události musí paměť opět uvolnit, protože v ní uložená data již nemusí být aktuální.

Objekt univerzálního komunikačního rozhraní s měřící kartou bude implementovat obdobný algoritmus, ovšem zde se jedná o zcela jiný problém. Některé měřící karty totiž mohou být vybaveny funkcemi pro obsluhu událostí na vstupních kanálech/portech, což znamená, že jsou schopny informovat aplikaci (konkrétně tedy objekt univerzálního komunikačního rozhraní) o události změny. Obsluha této události spočívá v přečtení nové hodnoty a jejím zápisu do paměti, odkud bude čtena při jakémkoliv požadavku na přístup k danému vstupnímu kanálu/portu do doby, než karta ohlásí další událost, která vyvolá aktualizaci této uložené hodnoty. Je nutné si uvědomit, ze tento a předešlý princip, i když jsou si značně podobné, spolu nijak nesouvisí. Tento algoritmus řeší přístup ke kanálům/portům

měřící karty pouze na základě ohlášených událostí, kdežto algoritmus popsaný v předchozím odstavci řeší možnou úsporu přístupu k měřící kartě již na úrovni časovací události. V situaci, kdy více připojených klientů provádí měření s velmi krátkou periodou vzorkování, jsou samozřejmě výhody popsaných algoritmů zřejmé.

3.1.7 Funkční studie aplikace klient (MIOCC)

Aplikace klient nazvaná MIOCC (Multifunction Input Output Card Client) představuje klientskou část použitého modelu server/klient. Její základní úloha spočívá v komunikaci se serverem MIOCS, okamžité vizualizaci příchozích hodnot a poskytnutí nejrůznějších uživatelských nástrojů pro práci se získanými daty. Vizualizace příchozích hodnot je chápána jako postupné sestrojení grafického průběhu v závislosti na čase a také jako možnost sledovat příchozí hodnoty číselně. Toto číselné zobrazení bude možné otevřením speciálního okna zůstávajícího vždy navrchu, aby uživatel mohl současně provádět jinou činnost, a poskytujícího zároveň hodnoty hlavní i rozšířené osy. Stupnice uživatelsky definované rozšířené osy bude vzhledem k hlavní ose přesně určena dvěmi referenčními hodnotami a lineární interpolací.

Již bylo uvedeno, že i přes dnešní velmi vysoké kapacity počítačových sítí se jedná o přenosová média s potenciálním různě velkým dopravním zpožděním. Zprávy ze serveru tedy mohou dorazit opožděné, proto je nutné respektovat tzv. timeout (maximální povolený čas odezvy) a veškeré příchozí datagramy zpracovávat vzhledem k jejich skutečnému času vytvoření, který je přesně dán jejich indexem. Klient tedy nesmí daná data zakreslit na pozici odpovídající času přijetí, ale podle indexu datagramu vypočítat jeho skutečný čas vytvoření a ten pro zakreslení použít. Takovouto vhodnou koncepcí je samozřejmě docíleno pouze správné interpretace výsledku, nikoliv zamezení samotného dopravního zpoždění.

Vzhledem k požadavku zadání umožnit ukládání/načítání dat měření je vhodné v této aplikaci využít standardní architekturu dokument/pohled, která značně zjednodušuje a ulehčuje implementaci souvisejících algoritmů, a s ohledem na možné větší velikosti výstupního datového souboru tento vnitřně komprimovat. Při práci s uživatelem prostřednictvím dialogů je nutné kontrolovat správnost všech vstupních požadavků, případná nepovolená zadání ohlásit a znemožnit jejich přijetí. Data týkající se analogových kanálů

budou v paměti uložena jako datový typ double, ovšem je zřejmé, že data týkající se digitálních kanálů rozlišujících pouze dva stavy je vhodné do paměti ukládat jako datový typ bool. Data digitálních portů potom budou reprezentována datovým typem short.

Vzhledem ke skutečnosti, že perioda vzorkování může být značně dlouhá a server je povinnen pravidelně zasílat kontrolní udržovací zprávy, musí klient monitorovat stav síťové komunikace a v případě, že ze serveru nedorazí tato zpráva včas, je nutné ohlásit problém a ukončit spojení. Ostatně všechny takovéto hrubé chyby v komunikaci se serverem musí být náležitě ošetřeny.

Jedním z požadavků zadání je možnost online kalibrace příchozích vstupních hodnot. Již v tuto chvíli je řešení tohoto problému zřejmé a poměrně jednoduché. I když do měřící karty vstupují analogová napětí (pro analogový i digitální signál), jak bylo uvedeno jsou následně převedena na digitální data, se kterými lze na softwarové úrovni provádět přepočet podle nejrůznějších kritérií. Online kalibrace vstupních kanálů/portů tedy bude probíhat softwarovým přepočtem na straně klienta a s činností serveru nijak nesouvisí.

3.1.8 Multiplatformní řešení

Stav používaného hardwaru a softwaru, konkrétně tedy používaných architektur procesorů, hardwarových uspořádání a operačních systémů osobních počítačů se v poslední době značně změnil a stále se mění. Tato skutečnost je částečně dána příchodem nejrůznějších druhů kapesních počítačů a dalších značně sofistikovaných mobilních zařízení, která poskytují určité multiplatformní možnosti nebo přímo uživatelská prostředí nejrozšířenějších operačních systémů, tedy konkrétně MS Windows, různých distribucí GNU/Linuxu a dalších. Lze předpokládat, že tímto směrem bude věnováno čím dál více pozornosti a multiplatformní řešení aplikací se budou stávat nutnějšími a žádanějšími. Komerční softwarové společnosti na straně jedné i společnosti zabývající se vývojem a distribucí Open Source na straně druhé v tomto směru nalézají určité shodné tendence, i když základní přístup k podstatám jejich práce zůstává stejný. Je-li řečeno multiplatformní řešení, bylo by velmi omezené nahlížet na možné výhody pouze z hlediska kontrastu stolních počítačů a různých kapesních zařízení. Tato řešení poskytují univerzální jednotnou rovinu pro vývoj softwaru a jeho nasazení na mnoha různých hardwarových i softwarových platformách. Jedny a ty samé zdrojové texty

jsou jednoduše s použitím příslušných knihoven zkompilovány pro různé platformy a výsledné aplikace fungují.

I přesto, že většina podniků pro nasazení softwarových aplikací pravděpodobně používá osvědčená jednoplatformně orientovaná řešení, potenciální potřeba byť částečného, dočasného, výjimečného nebo zkušebního přechodu na jinou platformu stále existuje a obzvlášť v dnešní době rychlého technického vzestupu a vysoké softwarové kriminality by se neměla podceňovat. Navíc z hlediska snahy o co nejvyšší míru rentability nákladů vynaložených na vývoj softwarového vybavení se jako velmi vhodné jeví právě multiplatformní řešení s využitím jednoho z osvědčených Open Source projektů, mezi které patří obecně známé knihovny GTK, Qt a wxWidgets. Nejen z důvodu jejich více než desetiletého vývoje, ale také vzhledem k určitým kladným zkušenostem budou pro realizaci softwarového vybavení použity Open Source knihovny wxWidgets.

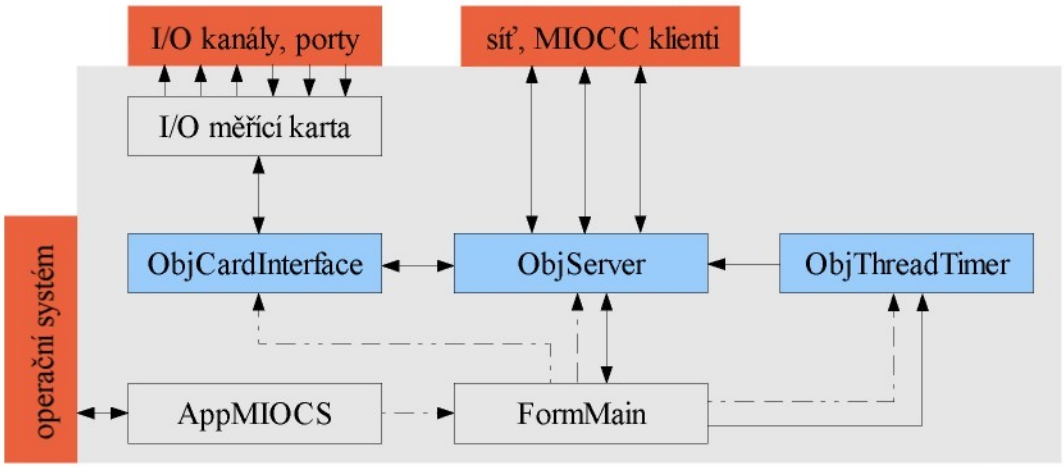
3.2 Realizace aplikace MIOCS (server)

Realizace aplikace je provedena v naprostém souladu s požadavky kladenými zadáním a funkční studií. Vzhledem k poměrně velkému rozsahu zdrojových textů a řešených problémů jsou zde rozebrány pouze důležité části. Na přiloženém CD jsou kromě hotových zkompilovaných aplikací také k dispozici všechny náležitě popsané zdrojové texty.

3.2.1 Zjednodušené objektové schéma aplikace, základní popis objektů

Na obrázku 3 je znázorněno zjednodušené objektové schéma aplikace MIOCS. Mimo jiné je patrné rozhraní aplikace vyznačené červeně a tvořené operačním systémem, reálnými vstupně/výstupními kanály/porty měřicí karty a připojením do sítě. Výchozím objektem aplikace je samozřejmě ***AppMIOCS*** (odvozen od ***wxApp***), který představuje instanci aplikace a zajišťuje vše potřebné pro vnitřní komunikaci s operačním systémem. Objekt aplikace vytváří instanci hlavního a zároveň jediného okna ***FormMain*** (odvozen od ***wxFrame***) poskytujícího pohodlné uživatelské prostředí, rolovací menu, panel nástrojů, stavový řádek a textové pole pro logování událostí. Vzhledem k tomu, že objekt ***ObjServer*** (odvozen od ***wxSocketServer***) je dynamicky vytvářen a mazán podle potřeby (zapnutí či vypnutí serveru), uchovává hlavní okno také informace o nastavení jednotlivých vstupních

analogových kanálů a digitálních portů včetně parametrů jejich simulace v poli proměnných strukturovaného datového typu a implementuje funkci pro výpočet jejich simulované hodnoty ve specifikovaném čase.



Obr. 3: Zjednodušené objektové schéma aplikace MIOCS (server), plné čáry znázorňují komunikaci, přerušované čáry instanční závislost objektů.

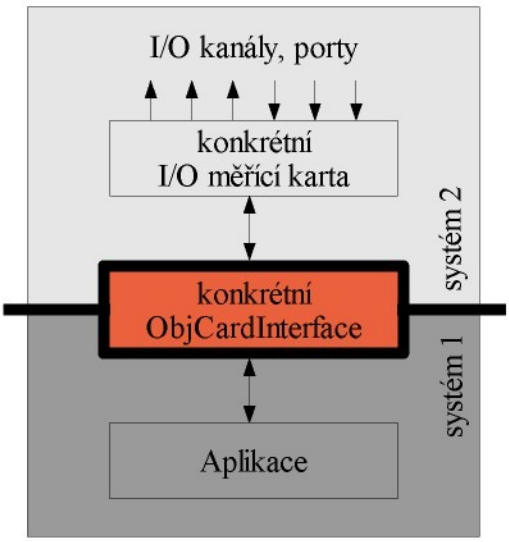
Stěžejními objekty jsou samozřejmě objekt socketového serveru ***ObjServer***, objekt univerzálního komunikačního rozhraní s měřící kartou ***ObjCardInterface*** a objekt softwarového časovače ***ObjThreadTimer***. Zdrojové texty ***ObjCardInterface*** a ***ObjThreadTimer*** jsou součástí příloh. V konstruktoru objektu hlavního okna je automaticky vytvořena instance ***ObjCardInterface***, která mimo jiné implementuje časování na bázi hardwarového čítače/časovače měřící karty a automaticky zavolanou funkci pro její nalezení v systému. Není-li měřící karta nalezena, server se automaticky přepne do simulačního režimu, kdy všechny vstupní kanály/porty jsou simulované dle zadaných nebo výchozích parametrů, a použije softwarové časování.

Objekt ***ObjThreadTimer*** představuje softwarový časovač implementovaný ve vlastním vlákně s nastavením vysoké priority, založený na odečítání systémového času (v milisekundách) ve velmi krátkých intervalech. Je automaticky použit, pokud měřící karta není vybavena hardwarovým čítačem/časovačem, nebo také pokud jej uživatel zvolí, k čemuž má dostupný nástroj v menu. Vzhledem k tomu, že oba druhy časovačů generují události v intervalech 100 ms a předávají je ke zpracování do smyčky zpráv objektu serveru, je nutné před vlastním spuštěním časování vytvořit instanci serveru, k čemuž dojde na základě příslušné akce uživatele.

Pokud se z nějakého důvodu nepodaří časování spustit, server je automaticky ukončen s ohlášením chyby. V případě obsazení nebo blokování nastaveného síťového portu (výchozí 3000) opět dojde k ohlášení problému a ukončení. Objekt serveru uchovává vlastnosti všech klientů v poli proměnných strukturovaného datového typu a přistupuje k nim při obsluze událostí časování a událostí socketu.

3.2.2 Objekt univerzálního komunikačního rozhraní s měřící kartou ObjCardInterface

Jak již bylo zmíněno, zajištění určitého jednotného schématu komunikace s měřící kartou je vzhledem k univerzální koncepci aplikace a jejímu možnému přizpůsobení pro jiné karty nutné. Jedná se o aplikování systémového přístupu na problém vstupně/výstupní komunikace s měřící kartou v rámci celé aplikace, jak znázorňuje obrázek 4. Objekt serveru, který je až na jednu malou výjimku prakticky jediným objektem využívajícím dané služby, pro tyto operace jednoduše volá univerzálně navržené metody ***ObjCardInterface*** (přehled v tabulce 5), kde jsou příslušné algoritmy implementovány tak, jak je požadováno výrobcem konkrétní měřící karty. Způsob implementace daných metod se jistě pro různé měřící karty (různé výrobce) bude lišit, neboť každý výrobce k danému problému může přistupovat jinak, používá jiné datové struktury, knihovny atd. Celý problém případného přizpůsobení aplikace pro použití s konkrétní měřící kartou tedy spočívá pouze v úpravě objektu ***ObjCardInterface*** a souboru globálních definic vlastností měřící karty, jejichž prázdné šablony jsou k dispozici.



Obr. 4: Schéma aplikování systémového přístupu na problém komunikace aplikace s měřící kartou.

Soubor globálních definic vlastností měřící karty obsahuje nejen konfigurační volby konkrétní karty, ale také makra preprocesoru pro kontrolu smysluplnosti zadání těchto voleb a případné oznámení chyby. Pro názornost je uvedena část této realizace. Jednotlivé konfigurační volby jsou také vzhledem k tomu, že příslušný hlavičkový soubor je includován ve všech objektech aplikace, automaticky použity tam, kde je to nutné. Veškeré zdrojové texty jsou vytvořeny tak, aby respektovaly globální definice a přizpůsobily výslednou podobu aplikace konkrétním požadavkům.

```
#define CARD_HAS_DI_CHANNELS 1
#define CARD_HAS_DO_CHANNELS 0
#define CARD_HAS_DI_PORTS 1
#define CARD_HAS_D0_PORTS 1
#if CARD_HAS_DI_PORTS && !CARD_HAS_DI_CHANNELS
      #error invalid CARD_HAS_DI_PORTS definition
#endif
#if CARD_HAS_DO_PORTS && !CARD_HAS_DO_CHANNELS
      #error invalid CARD_HAS_DO_PORTS definition
#endif
```

Takovéto zadání globálních definic způsobí znemožnění kompilace s oznámením popisu chyby ***„invalid CARD_HAS_DO_PORTS definition“***, protože nemá-li měřící karta specifikované žádné digitální výstupní kanály, nemůže mít specifikované ani žádné porty stejného druhu. Stejným způsobem jsou ošetřena všechna další chybná zadání.

Většina metod rozpoznává konkrétního klienta prostřednictvím ***client_index***, názvy metod začínají UNIV z důvodu jejich odlišení od funkcí, které mohou být pro konkrétní implementaci ***ObjCardInterface*** vytvořeny navíc, lze také používat vlastní objekty. Jedinou podmínkou je zachování rozhraní uvedených univerzálních metod a zajištění ošetření všech chybových stavů při komunikaci s měřící kartou s následným předáním popisu chyby a indexu klienta do ***UNIV_DCH_DoErrorActions***. Metody začínající UNIV_DCH (don't change) je zakázáno jakkoliv měnit. Detailní popis všech metod je uveden přímo v příslušných prázdných šablonách.

`void UNIV_DCH_DoErrorActions( wxString error_desc, const short client_index )` - provádí ošetření vyvstalé chyby postihující konkrétního klienta, -1 pro všechny klienty

`void UNIV_OnAnalogInputEvent( const short channel )`
`void UNIV_OnDigitalInputBitEvent( const short channel )`
- obsluha událostí změn na vstupních kanálech, pouze pro karty vybavené těmito možnostmi

`const bool UNIV_TryFindCard()` - pokouší se nalézt měřící kartu v systému, vrací příznak nalezení

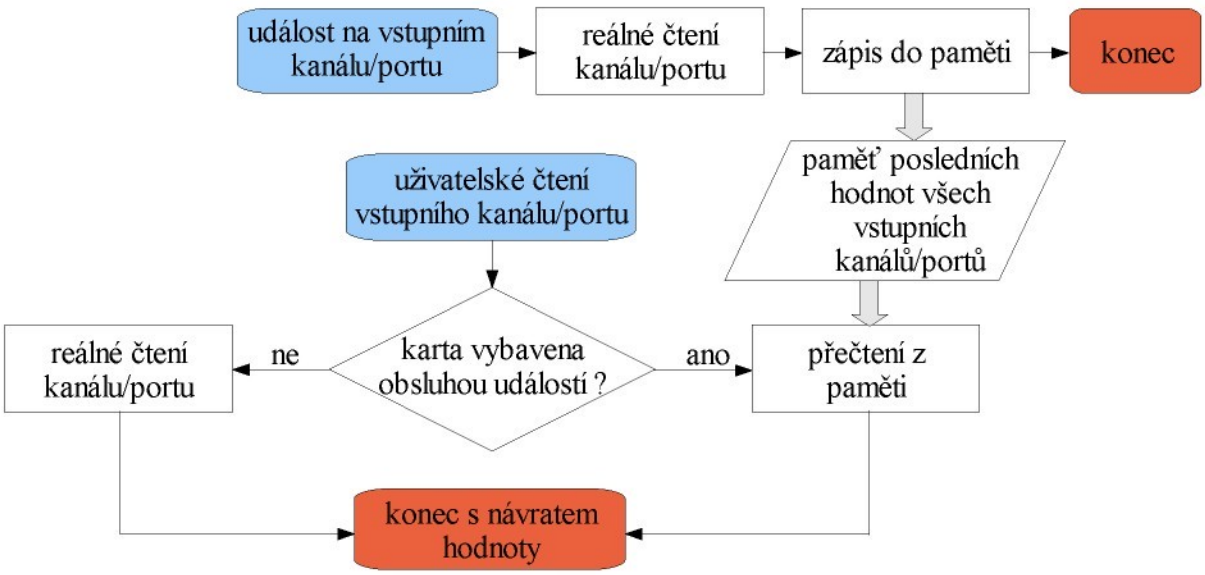
`const bool UNIV_DCH_StartInternalTimer( const int event_interval )`
`void UNIV_DCH_StopInternalTimer()`
- spuštění a zastavení algoritmu hardwarového časování

`const double UNIV_GetAnalogInputValue( const short channel, const short range_code, const short client_index )`
`const bool UNIV_GetDigitalInputValueBit( const short channel, const short client_index )`
`const short UNIV_GetDigitalInputValueByte( const short port, const short client_index )`
- čtení vstupních kanálů/portů

`void UNIV_SetAnalogOutputValue( const short channel, const double value, const short range_code, const short client_index )`
`void UNIV_SetDigitalOutputValueBit( const short channel, const bool value, const short client_index )`
`void UNIV_SetDigitalOutputValueByte( const short port, const short value, const short client_index )`
- zápis na výstupní kanály/porty

Tabulka 5: Popis univerzálních metod objektu komunikačního rozhraní s měřící kartou ObjCardInterface.

ObjCardInterface počítá s možností vybavení měřící karty funkcemi pro automatickou obsluhu (oznámení) událostí přesně tak, jak bylo navrženo funkční studií. Daný algoritmus lze znázornit následovně.



Obr. 5: Vnitřní algoritmus ObjCardInterface pro čtení vstupních kanálů/portů.

Není-li měřící karta vybavena možnostmi pro obsluhu událostí na vstupních kanálech/portech a nastane-li požadavek jejich uživatelského čtení, návratová hodnota je vždy získána prostřednictvím reálného přístupu. V libovolném jiném případě dojde k reálnému přístupu pouze po oznámení události, protože v paměti uložená hodnota se stává nepravdivou a je nutné jí zaktualizovat. Událostí se zde rozumí změna napětí na příslušném vstupním kanálu/portu, jednoznačná výhoda tohoto řešení spočívá v možném dramatickém snížení počtu reálných přístupů k měřící kartě, což může v situaci, kdy je k serveru připojeno více klientů a všichni provádějí měření s velmi krátkou periodou vzorkování, hrát určitou roli. Rozhodnutí o tom, zda měřící karta je či není vybavena zmíněnými možnostmi, je provedeno na základě podmínky, do které vstupuje hodnota příslušné globální definice ***CARD_HAS_EVENT_HANDLER***. Je zřejmé, že výsledek dané podmínky je vždy stejný, neboť globální definice je na příslušném místě zdrojového textu nahrazena konstantou. Proto je vhodné podmínku odstranit a místo ní řízení předat příslušné metodě přímo, čímž se algoritmus na obrázku 5 prakticky zjednoduší na ponechání pouze jedné větve podmínky. Koncepce přitom zůstává zachována.

3.2.3 Objekt softwarového časovače pracujícího ve vlastním vlákně ObjThreadTimer

Přesné časování je v aplikacích reálného času jedním ze základních požadavků. Splnění tohoto požadavku by se mohlo jevit jako zcela banální problém, ovšem ukazuje se, že v některých specifických případech je tomu právě naopak. Pochopitelně záleží na řádu požadované přesnosti, ale obecně je nutné si uvědomit, že vykonávání časovacího algoritmu by mělo být zcela nezávislé na ostatní činnosti aplikace, protože v opačném případě se přesnost časování odvíjí od jejího zatížení. Nejlepším řešením je potom realizace s využitím externího nezávislého zdroje, u kterého je přesnost časování zaručena, přičemž není podstatné, zda se jedná o zdroj hardwarový nebo softwarový. Opět se jedná o aplikaci systémového přístupu k problému. Jakákoliv nepřesnost ve stanovení události časování se samozřejmě zvyšuje s rostoucím časem měření, neboť dochází ke sčítání chyb. Z uvedeného důvodu není dobré zanedbávat ani časování s nižší přesností, protože i malá chyba po vyšším počtu opakování může v rámci požadavků na měření znamenat velkou nepřesnost.

V případě softwarové implementace časování je přesnost navíc ovlivněna také typem

plánování použitého operačního systému. Takzvané preemptivní plánování neustále přepíná mezi jednotlivými aplikačními procesy a přiděluje jim s ohledem na jejich prioritu určitý systémový čas, po jehož uplynutí je řízení předáno dalšímu procesu. Ukazuje se, že v případě neoddělení činnosti časovacího algoritmu od ostatní činnosti aplikace, se přesnost časování s tímto typem plánování dle předpokladu dramaticky snižuje. Naopak při nepreemptivním plánování je ve stejné situaci přesnost poměrně vysoká. V obou případech je však velmi vhodné použití tzv. vláken, která představují nezávislé aplikační procesy. Spuštěním časovacího algoritmu ve vlastním vlákně je jeho činnost zcela oddělena od ostatní činnosti aplikace a přiřazením vyšší priority pro běh tohoto vlákna lze docílit velmi vysoké přesnosti časování.

Použití vláken (***wxThread***) však na druhou stranu představuje určitou komplikaci, která na první pohled nemusí být zřejmá. Je nutné si uvědomit, že se jedná o nezávislý aplikační proces, stejně jako ostatní činnost aplikace (hlavní vlákno). Potenciálně může nastat situace, že obě vlákna se budou snažit přistoupit k proměnné nebo k jinému sdílenému prostředku současně, což v případě neošetření takového stavu může vést k neočekávanému chování nebo pádu aplikace. Použití vláken tedy sebou přináší nutnost zajistit takzvaný výhradní přístup k daným sdíleným prostředkům, což se provádí pomocí různých synchronizačních prvků, jakými jsou zámky (***wxMutex***) a kritické sekce (***wxCriticalSection***). Dále se používají tzv. semafory (***wxSemaphore***), podmínky (***wxCondition***) a další. Označení těchto prvků, tedy synchronizačních prvků, se může jevit jako paradoxní, ovšem synchronizace zde v žádném případě neznamená zajištění určitého shodného stavu, ale koordinace v přístupu k daným sdíleným prostředkům. Takové chování vláken je tedy z hlediska vnějšího pohledu označeno jako synchronní. S ohledem na celkové vytížení operačního systému a s ohledem na použitou platformu mohou být vlákna omezenými zdroji, takže není zaručeno, že se je podaří vytvořit nebo spustit. Je tedy nutné s touto situací počítat a provést odpovídající ošetření. Realizace s použitím objektu ***wxThread*** pak vypadá následovně.

```
wxThread thread = new wxThread();
if( thread -> Create() != wxTHREAD_NO_ERROR ||
    thread -> Run() != wxTHREAD_NO_ERROR )
{
    thread -> Delete();
    // oznameni uzivateli a dalsi osetreni
}
// vlastni cinnost vlakna
thread -> Delete();
```

V případě úspěšně vytvořeného a spuštěného vlákna se začne provádět činnost funkce ***Entry***, pro vlastní implementaci časovacího algoritmu je tedy nutné odvození vlastního objektu od ***wxThread*** a přetížení metody ***Entry***. Vzhledem k tomu, že činnost časovacího algoritmu je periodická a její ukončení nesmí nastat samovolně, je stěžejní část algoritmu uzavřena v nekonečné smyčce pomocí příkazu ***while***. V každém cyklu této smyčky je nutné provést test na požadavek ukončení činnosti vlákna, což zajišťuje metoda ***TestDestroy***, a pokud je příznak uvedeného požadavku pravdivý, provede se ukončení vlákna příkazem ***return*** s návratovou hodnotou ***(void *)NULL***. Po provedení cyklu je nutné vlákno dočasně uspat, aby nedocházelo ke stoprocentnímu zatížení procesoru vlivem nekonečné smyčky. Dočasné uspání vlákna zajišťuje metoda ***Sleep***, jejímž argumentem je počet milisekund.

Implementace ***ObjThreadTimerThread::Entry()*** je složitější a pokročilejší včetně využití synchronizačních zámků, ovšem pro účel vysvětlení stěžejního algoritmu zcela postačí následující zkrácený kód. Kompletní zdrojové texty jsou samozřejmě k dispozici a jsou také součástí příloh.

```
void * ObjThreadTimerThread::Entry()
{
    SetPriority( 90 );
    long event_interval = 100;
    wxLongLong event_time = wxGetLocalTimeMillis() + event_interval;
    while( 1 )
    {
        if( TestDestroy() ) return ( void * )NULL;
        if( wxGetLocalTimeMillis() >= event_time )
        {
            // vygenerovani udalosti casovani
            event_time += event_interval;
        }
        Sleep( 1 )
    }
}
```

Na začátku běhu vlákna se nastaví poměrně vysoká priorita 90, wxWidgets s ohledem na různé platformy rozlišují rozsah priorit 0-100. Proměnná ***event_interval*** drží hodnotu intervalu časování v milisekundách, takže algoritmus bude generovat události každou desetinu sekundy. ***wxGetLocalTimeMillis*** vrací aktuální systémový čas v milisekundách od 1.1. roku 1960, proměnná ***event_time*** se tedy nastaví na čas další události. Jak již bylo popsáno, v nekonečné smyčce se nejprve provede test na příznak ukončení činnosti vlákna, na konci potom jeho dočasné uspání, v tomto případě na dobu jedné milisekundy. Test na událost je realizován podmínkou porovnávající aktuální čas v milisekundách s časem další události.

`const bool Start( const long event_interval, const bool one_shot, const int thread_priority )` - spuštění časovače, kromě času pro generování událostí lze specifikovat nastavení na jednu jedinou událost a prioritu vlákna, vrací příznak úspěšnosti provedení
`void Stop()` - zastavení časovače, vlákno je ukončeno, objekt vlákna odalokován
`void Reset( const long event_interval, const short one_shot, const int thread_priority )` - reset časovače bez ukončení vlákna, lze specifikovat všechny nové vlastnosti
`void DisableEvents()` - zastavení generování událostí bez ukončení vlákna, opětovný start je nutné provést zavoláním metody Reset
`void GenerateEvent()` - generuje událost časování

Tabulka 6: Popis vybraných metod objektu softwarového časovače ObjThreadTimer.

3.2.4 Implementace hardwarového časování na bázi integrovaného čítače měřící karty

Některé měřící karty kromě vstupně/výstupních kanálů/portů poskytují pro uživatelské aplikace také určité další nástroje, jakými jsou například čítače. Jak známo čítače udržují celočíselnou hodnotu počtu nastalých událostí a s každou další událostí danou hodnotu zvýší. Dojde-li k přetečení rozsahu čítače, načítání probíhá znovu od nuly. Některé čítače poskytnuté měřícími kartami lze programovat, jiné jsou nastaveny pevně, v každém případě je generování událostí zajištěno hardwarově a zcela nezávisle na běhu operačního systému. Z tohoto důvodu se čítače s výhodou využívají pro implementaci časovacích algoritmů, neboť k nim lze pravidelně přistupovat (zjišťovat jejich stav) a s ohledem na nastavené vlastnosti usuzovat na uplynulý čas od posledního známého stavu. Některé měřící karty tyto časovací algoritmy poskytují přímo, potom lze hovořit o integrovaných časovačích. Ideální stav je takový, kdy měřící karta sama oznamuje události časovače a nástroje poskytované jejím výrobcem umožňují tato oznámení podchytit softwarově. V takovém případě je celý problém implementace značně zjednodušen. K vývoji aplikace MIOCS však byla použita měřící karta poskytující pro uživatelské potřeby jeden pevně nastavený čítač, jehož události jsou generovány s frekvencí 1 MHz (tedy milionkrát na sekundu), a proto je implementace hardwarového časování založena na výše uvedeném pravidelném odečítání stavu.

S ohledem na kapitolu 3.2.3, kde byla popsána vhodnost oddělení časovacího algoritmu

od ostatní činnosti aplikace, je i tento algoritmus implementován ve vlastním vlákně. Jeho struktura je prakticky shodná s tím rozdílem, že se nejedná o zjišťování systémového času, ale o zjišťování stavu čítače. Po spuštění vlákna je nejprve proveden reset čítače, který u této konkrétní měřící karty způsobí i jeho zastavení. Opětovný start čítače je proveden před vstupem do nekonečné smyčky, která musí následně kromě čtení stavu, usuzovaní na uplynulý čas a generování událostí časování také patřičně ošetřit moment přetečení maximální hodnoty čítače. To lze provést velmi elegantně včasným resetováním jeho stavu a opětovným startem. Některé měřící karty mohou při těchto operacích vynechat určitý počet cyklů událostí, což lze vykompenzovat odpovídajícím přepočtem stavu čítače pro novou událost časování. Zdrojový kód uvedeného algoritmu zde vzhledem k značné podobnosti s algoritmem uvedeným v kapitole 3.2.3 není.

Aplikace MIOCS umožňuje zvolit mezi použitím hardwarového a softwarového časování. Příznak přítomnosti použitelného čítače/časovače na měřící kartě je stanoven globální definicí ***CARD_HAS_COUNTER_TIMER***. Je-li tato definice nastavena na 0, aplikace MIOCS bude zkompilována bez možnosti zvolení hardwarového časování a vždy bude použito časování softwarové.

3.2.5 Objekt serveru ObjServer

Tento objekt zajišťuje veškerou vstupně/výstupní komunikaci s klienty prostřednictvím socketu a vstupně/výstupní komunikaci s měřící kartou prostřednictvím ***ObjCardInterface***. Veškeré akce měření vstupů a odesílání získaných dat klientům jsou prováděny na základě událostí časování, které nastávají pravidelně po 100 ms. Vstupní požadavky klientů server vyřizuje okamžitě, neboť se jedná o události socketu, které nastávají nezávisle na událostech časování. Pro uchování veškerých vlastností klientů včetně zadané masky použití jednotlivých kanálů/portů server používá následující strukturovaný datový typ.

```
typedef struct
{
    wxSocketBase *Socket;
    bool LocalClient;
    wxString IPAddress;
    wxString UserName;
    bool ClientSpecsAwaiting;
    bool DetailsSpecified;
    bool IsMeasuring;
    bool WaitingForLastDGramSend;
```

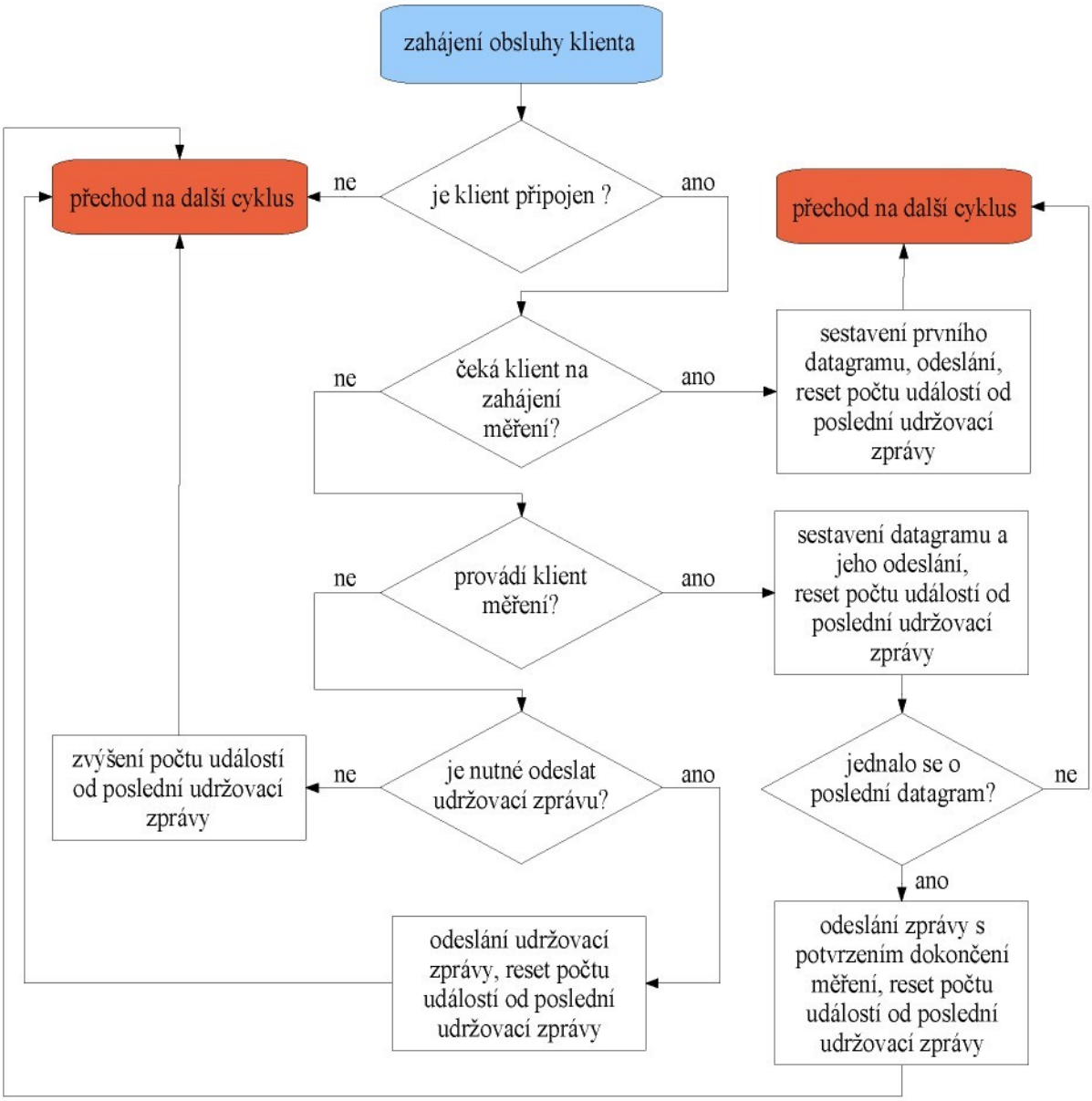
```
    double SamplingRate;
    unsigned long Samples;
    unsigned long SampleIndex;
    bool AIUsed[AI_CHANNELS];
    bool AOUsed[AO_CHANNELS];
    bool DICHUsed[DI_CHANNELS];
    bool DOCHUsed[DO_CHANNELS];
    bool DIPUsed[DI_PORTS];
    bool DOPUsed[DO_PORTS];
    short AIRangeCodes[AI_CHANNELS];
    short AORangeCodes[AO_CHANNELS];
    unsigned long TEventsFromSample;
    unsigned long TEventsBetweenSamples;
    unsigned long TEventsFromDGramSend;
    unsigned long TEventsBetweenDGramsSend;
    unsigned long TEventsFromKeepAlive;
    wxString DGrams;
}
ClientStruct;
ClientStruct m_Clients[MAX_CLIENTS];
```

Název jednotlivých proměnných vypovídá o účelu jejich použití. ***Socket** je ukazatel na objekt vlastního síťového socketu konkrétního klienta, ***ClientSpecsAwaiting*** vypovídá o tom, zda již byla provedena specifikace klienta (jméno, identifikační řetězec měřící karty atd.), ***DetailsSpecified*** poskytuje informaci o tom, zda byla provedena specifikace detailů měření a ***IsMeasuring*** o tom, zda klient měření provádí. ***SamplingRate*** a ***Samples*** drží hodnoty periody vzorkování a počtu vzorků měření. Pole booleovských příznaků ***m_AIUsed***, ***m_AOUsed***, ***m_DICHUsed***, ***m_DOCHUsed***, ***m_DIPUsed*** a ***m_DOPUsed*** představují masku použití jednotlivých kanálů/portů. Rozměry těchto polí tvoří globálně definované konstanty ***AI_CHANNELS***, ***AO_CHANNELS***, ***DI_CHANNELS***, ***DO_CHANNELS***, ***DI_PORTS*** a ***DO_PORTS***, které jsou součástí hlavičkového souboru vlastností měřící karty. Globální definice konstanty ***MAX_CLIENTS*** již je součástí hlavičkového souboru aplikace MIOCS a představuje specifikaci maximálního počtu připojených klientů.

ObjServer je odvozen od objektu ***wxSocketServer***, který není odvozen od ***wxEvtHandler***. To znamená, že nemá implementovánu vlastní smyčku zpráv, ale musí používat smyčku externí. Bylo by sice možné použít smyčku zpráv objektu hlavního okna, ale ve snaze tyto objekty systémově oddělit bylo zvoleno řešení jiné a sice odvození vlastního objektu smyčky zpráv od ***wxEvtHandler*** a jeho přímá instance v konstruktoru objektu serveru. Vytvořená instance pomocné smyčky zpráv pak neprovádí nic jiného, než že obsluhu příslušných metod předává zpět odpovídajícím metodám implementovaným v ***ObjServer***.

Obsluha události časování spouští cyklus s algoritmem obsluhujícím postupně všechny

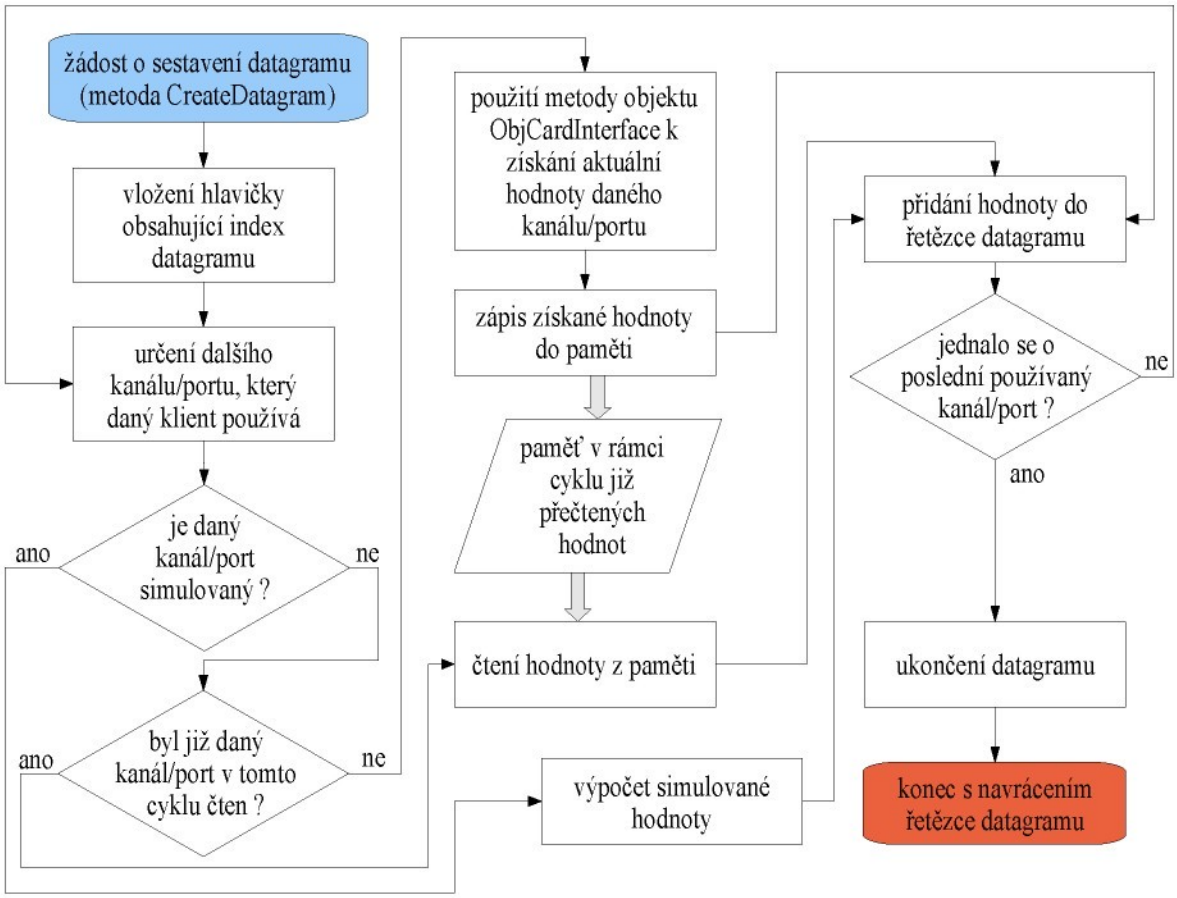
klienty. Tento algoritmus je znázorněn na obrázku 6. Počet událostí od poslední udržovací zprávy je představován proměnnou ***TEventsFromKeepAlive*** (viz. uvedená struktura), přičemž počet událostí časování nutných k odeslání udržovací zprávy se vypočítá podle nastavení globální definice aplikace ***MAX_SOCKET_SPACE_TIME***.



Obr. 6: Algoritmus cyklu obsluhy klienta.

Sestavení datagramu zajištěné metodou ***CreateDatagram*** představuje proces, kdy podle nastaveného filtru použití kanálů/portů jsou získány reálné hodnoty z měřící karty a sestaveny do validní zprávy. Pokud již v tomto cyklu některá z hodnot byla čtena, pak se pouze přečte z paměti a k zavolání příslušné metody v ***ObjCardInterface*** nedojde. Pokud je navíc některý

z kanálů/portů nastaven na simulovaný průběh, pak se místo případného čtení vstupu měřící karty zavolá funkce pro výpočet simulované hodnoty. Popsaný algoritmus je znázorněn na následujícím obrázku.



Obr. 7: Algoritmus získání řetězce s datagramem (metoda CreateDatagram).

Pokud se jedná o vzdáleného klienta, který měří s příliš krátkou periodou vzorkování, pak je celá situace o něco složitější. Server totiž s ohledem na možnou zátěž síťových prvků na trase mezi ním a klientem odesílá datagramy hromadně a to v intervalech specifikovaných globální definicí aplikace ***MAX_RCLIENT_DGRAM_FREQ***. Je-li perioda vzorkování větší nebo rovna hodnotě uvedené globální definice, pak k odesílání datagramů dochází okamžitě po jejich vytvoření. V opačném případě jsou datagramy hromaděny a odeslány jakmile to bude možné. Uvedená globální definice má ve výchozím stavu hodnotu 1000 (ms). Lokálním klientům jsou vzhledem k tomu, že lokální síťové rozhraní představuje pouze virtuální smyčku, datagramy odesílány okamžitě.

`void OnServerEvent( wxSocketEvent &event )` - obsluha události serveru, zpracovává pouze požadavek příchozího spojení
`void OnSocketEvent( wxSocketEvent &event )` - obsluha události všech socketů, reaguje na událost příchozí zprávy a ukončení spojení, příchozí zpráva je automaticky přečtena a předána funkci ServeMessage
`void OnTimerEvent()` - obsluha události časování popsaná výše (algoritmus znázorněn na obr. 6)
`void OnTimerErrorEvent()` - obsluha události chyby hardwarového časování, odesílá všem klientům informační zprávu s následným odpojením, server je poté zastaven
`void ServeMessage( const short client_index, const wxString message )` - pro klienta zadaného indexem provádí obsluhu předané příchozí zprávy, kontroluje její validitu
`const wxString CreateDatagram( const short client_index )` - pro klienta zadaného indexem vytváří validní zprávu s datagramem (algoritmus znázorněn na obr. 7)

`const bool SendMessage( wxSocketBase *socket, const wxString message )`
`const bool SendMessage( const short client_index, const wxString message )`
- odesílá zprávu socketem zadaným ukazatelem nebo klientovi zadanému indexem

`void DisconnectClient( const short client_index, const wxString err_code )` - odpojuje klienta zadaného indexem, loguje případný chybový kód
`void DisconnectAllClients()` - odpojuje všechny klienty

Tabulka 7: Popis metod objektu serveru ObjServer.

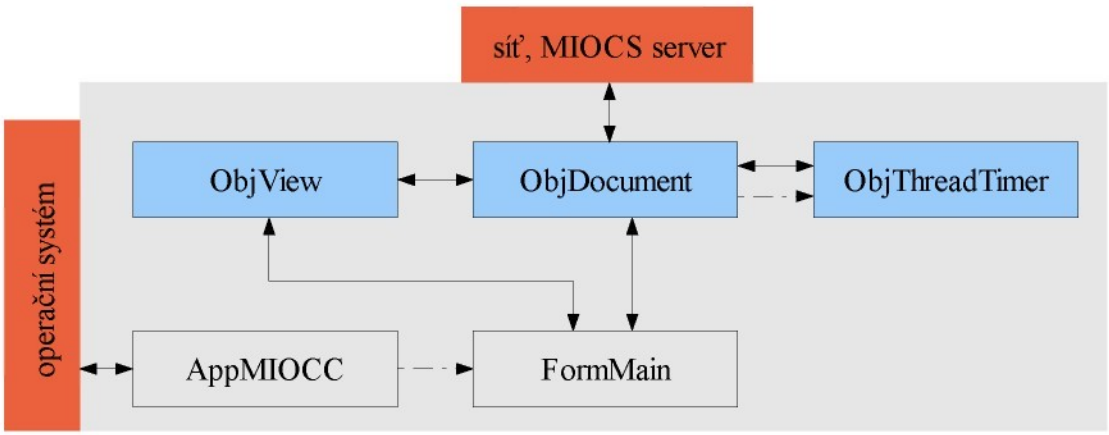
3.3 Realizace aplikace MIOCC (klient)

Realizace aplikace je provedena v naprostém souladu s požadavky kladenými zadáním a funkční studií. Vzhledem k poměrně velkému rozsahu zdrojových textů a řešených problémů jsou zde rozebrány pouze důležité části. Na přiloženém CD jsou kromě hotových zkompilovaných aplikací také k dispozici všechny náležitě popsané zdrojové texty.

3.3.1 Zjednodušené objektové schéma aplikace, základní popis objektů

Zjednodušené objektové schéma aplikace klient (MIOCC) na obrázku 8 opět kromě instančního a komunikačního vztahu objektů znázorňuje i celkové rozhraní aplikace v tomto případě tvořené pouze operačním systémem a síťovým spojením. Aplikace MIOCC vzhledem ke svému charakteru a požadavkům na ní kladeným používá standardní architekturu dokument/pohled.

Architektura dokument/pohled vznikla na základě snahy oddělit algoritmy pracující s daty dokumentu od algoritmů zajišťujících vizualizaci těchto dat, což vede k určitému. zpřehlednění struktury zdrojových textů a dalším možnostem. Například jednomu dokumentu lze definovat více druhů zobrazení, tedy více pohledů, a každý z těchto pohledů zpracuje data dokumentu jiným způsobem. Dále lze vytvářet aplikace spravující několik otevřených dokumentů současně, takové aplikace jsou označovány aplikacemi MDI (Multiple Document Interface). Pro účely aplikace MIOCC je ovšem MDI architektura nevhodná, protože některé funkce jsou založeny na práci s více otevřenými rozšiřujícími nemodálními okny a v případě více otevřených dokumentů současně by situace byla poněkud nepřehledná. Během vývoje aplikace ani nebyly zjištěny žádné konkrétní důvody, proč by v tomto případě byla architektura MDI výhodná. Aplikace MIOCC, přestože architekturu dokument/pohled používá, je tedy řešena jako SDI (Single Document Interface). Požaduje-li uživatel provádění více měření současně, může samozřejmě spustit další instanci aplikace.



Obr. 8: Zjednodušené objektové schéma aplikace MIOCC (klient), plné čáry znázorňují komunikaci, přerušované čáry instanční závislost objektů.

Z programátorského hlediska znamená použití architektury dokument/pohled velkou výhodu, protože originální objekty ***wxDocument*** a ***wxView*** již mají implementovánu celou řadu žádaných funkcionalit spojených se správou a ukládáním dat dokumentu. Objekty dokumentu a pohledu představují v rámci uvedené architektury takzvané dynamické třídy spravované centrálním manažerem dokumentů (***wxDocManager***). Vzájemné propojení dokumentu s pohledem je realizováno prostřednictvím takzvané šablony dokumentu (***wxDocTemplate***), přičemž pro každý druh pohledu je nutné definovat vlastní šablonu.

Objekt aplikace ***AppMIOCC*** vytváří instanci objektu hlavního okna ***FormMain***, který je

právě z důvodu použití uvedené architektury odvozen nikoliv od ***wxFrame***, ale od ***wxDocParentFrame***. Toto okno má definovány a připraveny všechny nástroje a jejich dostupnost je ovládána příslušnými metodami objektu pohledu. Obsluha událostí nástrojů v hlavním okně je přesměrována do příslušných metod objektu dokumentu nebo pohledu podle toho, kterého z nich se týká. Vlastní vytvoření instance dokumentu a pohledu s ním spojeného zajišťuje již zmíněný manažer dokumentu.

Objekt ***ObjThreadTimer*** představuje naprosto stejný časovač jako v aplikaci MIOCS, ovšem zde je použit pro hlídání času socketové neaktivity z důvodu monitorování správné funkce serveru a síťového spojení.

3.3.2 Objekt dokumentu ObjDocument

Objekt dokumentu představuje vzhledem k použití architektury dokument/pohled centrální objekt pro uchování všech dat a implementaci souvisejících algoritmů. V případě aplikace MIOCC se jedná o funkce pro komunikaci se serverem, zpracování příchozího datagramu, specifikace a uvolnění detailů měření, alokace a dealokace paměti, zahájení a zastavení měření, výpočet statistik, ukládání/načítání dat, export dat v textovém formátu a ovládání výstupních kanálů/portů. Pro vlastní data měření jsou vzhledem k jejich rozmanitosti a také ve snaze o úsporné využití paměti použity tři různé datové typy. Hodnoty analogových vstupních a výstupních kanálů jsou v paměti uloženy jako datový typ double, naopak je zřejmé, že pro data digitálních kanálů postačí datový typ bool. Data digitálních portů potom využívají datový typ short. Před vlastním zahájením měření musí být paměť alokována, to zajišťuje funkce ***DetailsSpecify***, která po specifikaci detailů měření na serveru tuto operaci provede. Pro uchování vlastností online kalibrace je použit následující strukturovaný datový typ proměnné.

```
typedef struct
{
   int Gain_Value;
   int Gain_Min;
   int Gain_Max;
   int Gain_Force;
   bool Gain_ImmReact;
   double Inc_Value;
   wxGenericValidator *GVldr_Gain_Value, *GVldr_Gain_Force, *GVldr_Gain_ImmReact;
}
OnlineCalStruct;
```

Gain_Value je nastavená hodnota násobící vstup (zesilovač), může být kladná i záporná, takže nastavením této hodnoty na -1 lze pouze obrátit polaritu vstupu. ***Gain_Min*** a ***Gain_Max*** představují povolené meze nastavení posuvného jezdce zesilovače v příslušném dialogu (obr. 9). Aplikace také umožňuje zvolit řád (sílu) zesílení a to na úrovni tisícin, setin, desetin, jednotek, desítek, stovek a tisíců. Příslušná nastavená hodnota je uložena v ***Gain_Force***. ***Gain_ImmReact*** je příznak nastavení okamžité reakce na změnu polohy jezdce zesilovače, je-li false, pak nastavení zvolených parametrů online kalibrace proběhne pouze stisknutím nastavovacího tlačítka. Nakonec ***Inc_Value*** je hodnota přičtená k výsledku po provedení zesílení, přičemž může být samozřejmě záporná. Struktura také obsahuje ukazatele na příslušné validátory zajišťující přenos hodnot do dialogu a zpět. Vlastní proces online kalibrace představuje velmi jednoduchou záležitost přepočtu skutečné příchozí hodnoty podle nastavených parametrů ještě před jejím zápisem do paměti.

Obr. 9: Dialog online kalibrace vstupu.

Objekt dokumentu v rámci zajištění komunikace se serverem monitoruje provoz v socketu a k určení stavu nečinnosti používá objekt časovače ***ObjThreadTimer*** implementovaného ve vlastním vlákně. Jedná se o stejný objekt, jaký používá i aplikace MIOCS. Po přijetí jakékoliv zprávy ze serveru je tento časovač resetován. Dojde-li k jeho události, pak došlo k překročení povolené doby neaktivity a klient je odpojen s ohlášením chybového kódu.

Vzhledem k možnému velkému objemu měřených dat je výsledný datový soubor vnitřně komprimován deflačním algoritmem. Toho lze docílit použitím speciálního typu toku ***wxZipInputStrem*** (čtení) a ***wxZipOutputStream*** (zápis), jak znázorňuje následující kód.

```
bool ObjDocument::OnSaveDocument( const wxString &filename )
{
    // ...

    wxFileOutputStream *file_output = new wxFileOutputStream( filename );
    if( !file_output -> Ok() )
    {
        delete file_output;
        return false;
    }

    wxZipOutputStream zip_output( *file_output );
    wxDataOutputStream data_output( zip_output );
    zip_output.PutNextEntry( "MIOCC data" );

    // zapis vlastnich polozek do toku data_output
    // ...
    file_output -> Close();
    delete file_output;

    // ...
    return true;
}
```

Po úspěšném vytvoření výstupního toku do souboru je vytvořen výstupní tok ZIP, který se nasměruje do uvedeného souborového toku. Nakonec je vytvořen obecný datový výstupní tok a nasměrován do toku ZIP. Vzájemné propojení těchto tří toků zajistí kompresi dat a jejich uložení do souboru. V případě otevírání dokumentu je situace naprosto stejná s tím rozdílem, že je nutné použít toky vstupní.

`void DetailsSpecify()` - specifikace detailů měření, odesílá požadavek na server a obsluhuje jeho reakci, alokuje potřebnou paměť, ošetřuje nedostatek volné paměti a chybu alokace
`void DetailsDrop()` - provádí uvolnění detailů měření na serveru, aby s případnými výstupními kanály/porty mohli pracovat jiní klienti
`void IncommingMessage( wxString message )` - provádí obsluhu předané zprávy přečtené ze socketu (datagram, chyba měření atd.)
`void NewDatagram( wxString datagram )` - provádí všechny operace spojené s příchodem nového datagramu, tedy po přepočtu dat v závislosti na parametrech online kalibrace zápis do paměti a zavolání příslušné metody objektu pohledu pro vkreslení nových dat do grafu

`void StartMeasuring()`
`void StopMeasuring()`
- zahájení a přerušení měření

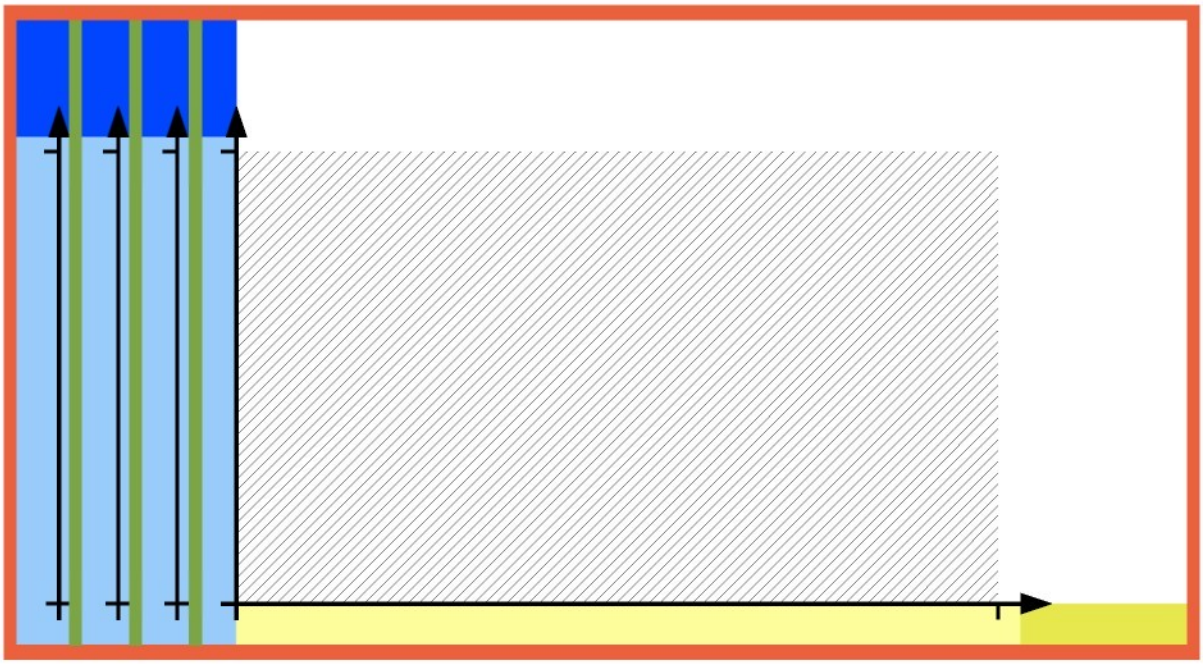
`void SetOutput( const short type, const short index, const double value )` - odesílá požadavek nastavení výstupního kanálu/portu na danou hodnotu

Tabulka 8: Popis vybraných metod objektu dokumentu ObjDocument.

3.3.3 Objekt pohledu ObjView

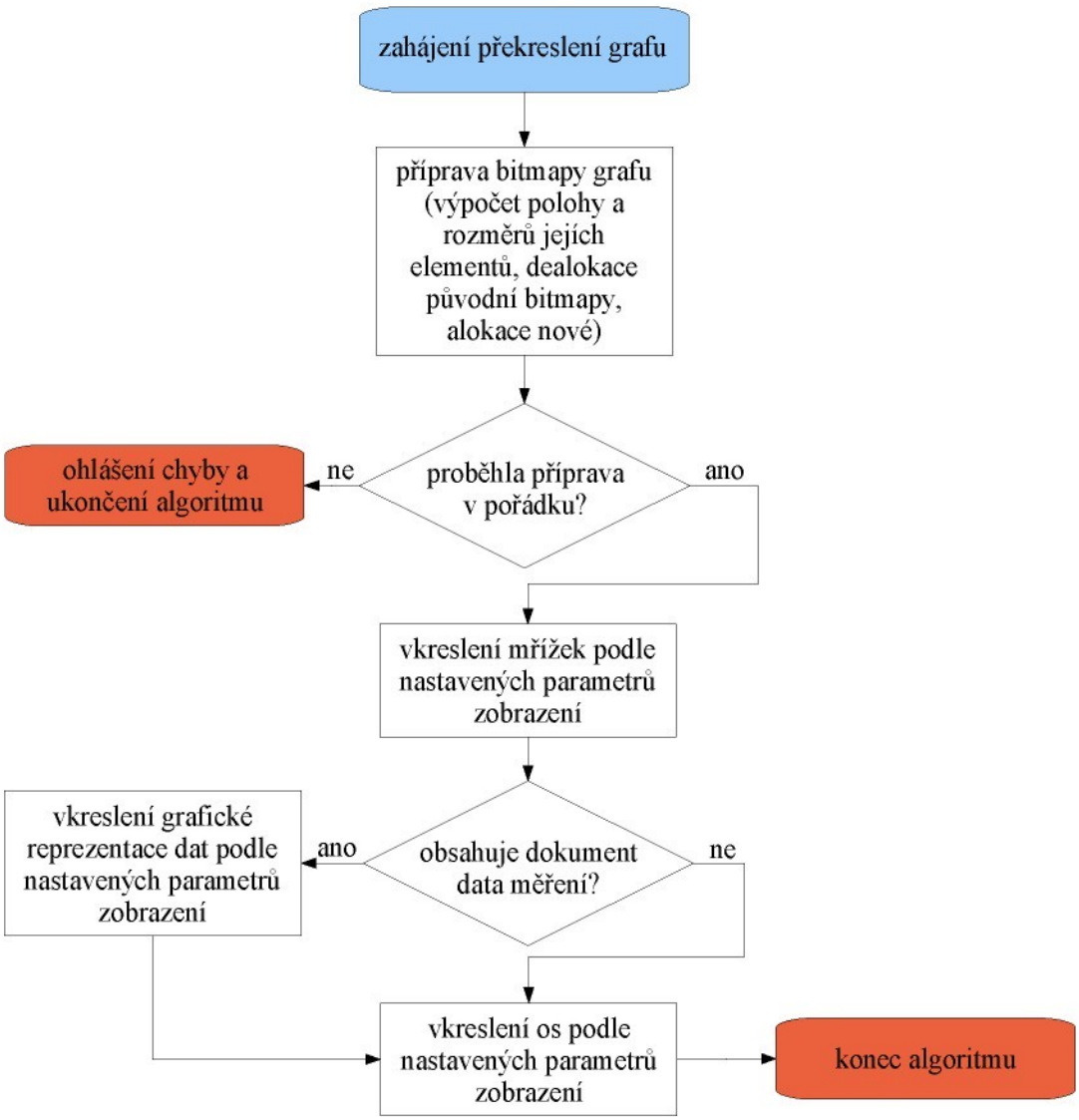
Objekt pohledu představuje vzhledem k použití architektury dokument/pohled centrální objekt pro implementaci veškerých algoritmů pohledu a vizualizace dat. V aplikaci MIOCC se konkrétně jedná o algoritmus přípravy a vykreslení grafu, změny velikosti grafu, nastavení vlastností vizualizace hlavních a rozšířených os, mřížek a vlastních datových řad, dále export grafu do bitmapových formátů. Implementuje také nástroje umožňující zobrazení pomocných oken posledních hodnot (okna informující číselně o poslední hodnotě na daném kanálu/portu), oken online kalibrace a oken ovládání výstupních kanálů/portů. Nastavení zobrazení mřížek, hlavních os, datových řad a externích os objekt pohledu uchovává ve strukturovaných datových typech proměnných.

Specifický problém představuje algoritmus přípravy a vykreslení grafu. Samotný graf je reprezentován objektem bitové mapy ***wxBitmap***, jejíž rozměry závisí na parametrech vizualizace dílčích elementů. Vzhledem k tomu, že bitová mapa (dále bitmapa) musí být před započetím vlastních kreslících algoritmů naalokována, je nutné před každým překreslením grafu přepočítat umístění a rozměry jednotlivých elementů a určit výsledný rozměr bitmapy. Objekt pohledu k tomuto účelu využívá pomocné dynamicky alokované pole struktur, kam při výpočtech zapisuje jednotlivé souřadnice počátků a rozměry dílčích elementů, při procesu vykreslování tyto informace potom používá. Nepodaří-li se případně z nějakého důvodu potřebnou bitmapu naalokovat, je tato situace posouzena jako neočekávaná chyba, uživatel je o problému informován a vyzván k uložení práce a ukončení aplikace. Tato situace však představuje spíše výjimečný stav způsobený především velmi slabými hardwarovými možnostmi konkrétního počítače vzhledem k potřebné paměti nutné pro alokaci bitmapy daných rozměrů. Pokud by nebyla provedena kontrola na úspěch alokace, došlo v případě její chyby při následném vykonávání vykreslovacích algoritmů k zásahu do nealokované paměti a pádu celé aplikace, což je z hlediska ochrany dat měření vzhledem k možným dlouhým časům měření nepřijatelné. Bitmapa grafu je rozdělena na několik oblastí, jak znázorňuje obrázek 10 na další straně.



Obr. 10: Schéma rozdělení bitmapy grafu na několik oblastí.

Šrafovaná oblast představuje datovou oblast grafu, do této oblasti je vkreslena vlastní grafická reprezentace dat. Světle žlutá ve spojení s tmavě žlutou barvou představuje oblast vodorovné osy, tmavě žlutá konkrétně prostor vyhrazený pro její popis. Obdobně světle a tmavě modré barvy znázorňují ty samé oblasti pro svislé osy. Každá svislá osa včetně svého popisu představuje samostatný element. Zóna vyznačená zeleně reprezentuje prázdný prostor oddělující jednotlivé elementy a také vzdálenost mezi textem a nejbližším prvkem (zde není vidět). Zóna vyznačená červeně ohraničuje celkovou oblast grafu, opět se jedná o prázdný prostor. Tyto zóny jsou zavedeny z důvodu docílení určité grafické formy. Šířku obou zón lze specifikovat pomocí globálních definic ***SPACER*** (výchozí hodnota 5 pixelů) a ***OVERSIZE*** (výchozí hodnota 10 pixelů). Příprava bitmapy grafu spočívá ve smazání současného pomocného pole struktur, naalokování nového, výpočtu souřadnic počátků a rozměrů jednotlivých elementů, určení celkového potřebného rozměru bitmapy, určení počátku datové oblasti grafu a alokace bitmapy. Dílčí algoritmy pro vykreslení jednotlivých oblastí potom získané informace používají. Rozměr pomocného pole struktur je dán počtem elementů k zobrazení.



Obr. 11: Algoritmus kompletního překreslení grafu.

Algoritmus znázorněný na obrázku 11 je vykonáván pouze v případě, že se jedná o kompletní překreslení grafu vyvolané uživatelskou změnou v nastavení vlastností vizualizace libovolného z elementů nebo změnou rozměru datové oblasti grafu. Během procesu měření jsou grafické reprezentace příchozích dat do grafu postupně vkreslovány jiným způsobem a sice přímým zásahem do kontextu zařízení klientské oblasti objektu posuvného okna (tento objekt se stará o zobrazení bitmapy), čímž je zamezeno nepříjemnému probliknutí celé oblasti jako v případě jejího kompletního překreslení. Zároveň je ovšem nutné naprosto stejné operace provést i s paměťovým kontextem zařízení bitmapy grafu, aby tato obsahovala aktuální podobu a také z důvodu možného nutného překreslení postižené oblasti při jejím narušení, o což se stará zmíněný objekt posuvného okna.

`const wxSize CalculateDASize( wxSize data_area_size, const bool decrease )` - vrací rozměr datové oblasti grafu po kalkulaci zadaného rozměru tak, aby vzdálenosti mezi jednotlivými mřížkami byly v obou směrech celočíselné
`const bool PrepareBitmap( const wxSize data_area_size )` - připravuje bitmapu grafu podle popsaného algoritmu, požadovaný rozměr datové oblasti grafu je předem znám, vrací příznak úspěšnosti provedení
`void DrawGrid()` - vkresluje do bitmapy grafu všechny mřížky podle parametrů jejich vizualizace
`void DrawAxes()` - vkresluje do bitmapy grafu všechny osy podle parametrů jejich vizualizace
`void DrawDatagram( const unsigned long index )` - do datové oblasti bitmapy grafu vkresluje grafickou reprezentaci dat obsažených v datagramu specifikovaném indexem podle parametrů vizualizace příslušných řad
`void DrawAllDatagrams()` - do datové oblasti bitmapy grafu vkresluje grafickou reprezentaci všech dat
`void DrawSegment( const short temp_prop_index )` - vkresluje do bitmapy grafu lineární spojnici nebo pouze bod podle parametrů vizualizace konkrétní řady a parametrů obsažených v poli dočasných struktur na zadaném indexu
`void DrawError( const unsigned long index, const short error_id )` - do datové oblasti bitmapy grafu vkresluje oznámení o chybě
`const bool RegenerateGraph()` - kompletně regeneruje zobrazení grafu, s ohledem na momentální vlastnosti dokumentu volá příslušnou sekvenci funkcí
`void ExportGraph()` - zajišťuje export bitmapy grafu do různých bitmapových grafických formátů

`void IncGDASize()`
`void DecGDASize()`
- zvětšuje a zmenšuje bitmapu grafu, jedná se kompletní vektorové překreslení s novým výchozím rozměrem datové oblasti grafu

`void UpdateTools()` - aktualizuje dostupnost nástrojů hlavního okna podle momentálního stavu dokumentu

Tabulka 9: Popis vybraných metod objektu pohledu ObjView.

3.4 Další specifická řešení spojená s realizací obou aplikací

3.4.1 Lokalizace aplikací pomocí locales mechanismu a jejich překlad s použitím nástrojů GNU gettext

Lokalizace aplikace znamená mnohem více než její pouhý překlad do příslušného jazyka. Jedná se o kompletní přizpůsobení daným národnostním zvyklostem. Tyto zvyklosti se týkají například typu desetinného oddělovače (česká lokalizace používá čárku, anglická tečku), metrického systému, měny a způsobu zápisu datumu a času. Lokalizaci aplikace

v knihovnách wxWidgets zajišťuje objekt ***wxLocale***. Tento objekt je navíc naprosto kompatibilní se standardním mechanismem GNU gettext, který poskytuje skupinu nástrojů používaných pro kompletní překlad aplikací, konkrétně označení textových řetězců ve zdrojových kódech, jejich následnou extrakci do šablon připravených k překladu (soubory s příponou po) a v konečné fázi překlad přeložených šablon do binárního katalogu zpráv (soubory s příponou mo). Podle zavedených locales je potom pro daný jazyk použit odpovídající katalog zpráv. Nelze-li daný katalog zpráv nalézt, použijí se původní řetězce. Z hlediska multiplatformní orientace aplikací přináší tento mechanismus určité výhody. Například binární katalogy zpráv jsou použitelné univerzálně bez ohledu na platformu nebo kódování příslušného zdrojového po souboru. Knihovna zajišťující čtení zpráv z příslušného binárního katalogu automaticky provádí jejich případný převod do požadované znakové sady, takže zprávy se vždy zobrazí správně. Překlad takto koncipované aplikace do nového jazyka potom představuje prakticky jen vyplnění příslušné po šablony s uvedením správné znakové sady překladu, následný převod do mo binárního katalogu zpráv pomocí ***msgfmt*** a programové zavedení příslušných locales bez jakékoliv další nutnosti zásahu do zdrojových textů. Opět se jedná o systémové řešení.

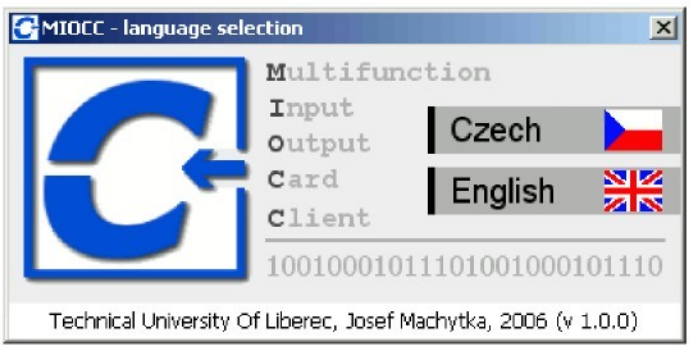
Aplikace server i klient jsou popsaným způsobem lokalizovány do češtiny a angličtiny. Programové zavedení locales probíhá při spuštění aplikace (metoda ***OnInit***) v objektu ***AppMIOCS***, resp. ***AppMIOCC*** po volbě jazyka v uvítacím dialogu. Je-li uvítací dialog zrušen bez výběru jazyka, dojde k ukončení aplikace. Příslušný zápis vypadá následovně, pro demonstraci je použit zápis aplikace MIOCC.

```
bool AppMIOCC::OnInit()
{
    // ...

    DlgLang *dlg_lang = new DlgLang();
    SetTopWindow( dlg_lang );
    dlg_lang -> Centre();
    switch( dlg_lang -> ShowModal() )
    {
        case Selected_CZ : m_Locale.Init( wxLANGUAGE_CZECH ); break;
        case Selected_EN : m_Locale.Init( wxLANGUAGE_ENGLISH ); break;
        default : dlg_lang -> Destroy(); return true; break;
    }
    dlg_lang -> Destroy();
    m_Locale.AddCatalogLookupPathPrefix( wxT("i18n") );
    m_Locale.AddCatalog( wxT("wx") );
    m_Locale.AddCatalog( wxT("miocc") );

    // ...
}
```

Objekt ***DlgLang*** představuje dialog pro výběr jazyka. Příkazem ***switch*** je rozhodnuto o akci provedené po získání návratové hodnoty. Proměnná ***m_Locale*** datového typu ***wxLocale*** je deklarována v příslušném hlavičkovém souboru a její metoda ***Init*** provede zavedení příslušných locales specifikovaných jejím argumentem. Pro češtinu tedy ***wxLANGUAGE_CZECH***, pro angličtinu ***wxLANGUAGE_ENGLISH***. Následně zavolaná metoda ***AddCatalogLookupPathPrefix*** zajistí přidání adresáře i18n do seznamu lokací, kde mechanismus hledá binární katalogy zpráv. Nakonec zavolané metody ***AddCatalog*** specifikují názvy příslušných katalogů (wx pro katalog překladu knihoven wxWidgets, miocc pro katalog vlastní aplikace).

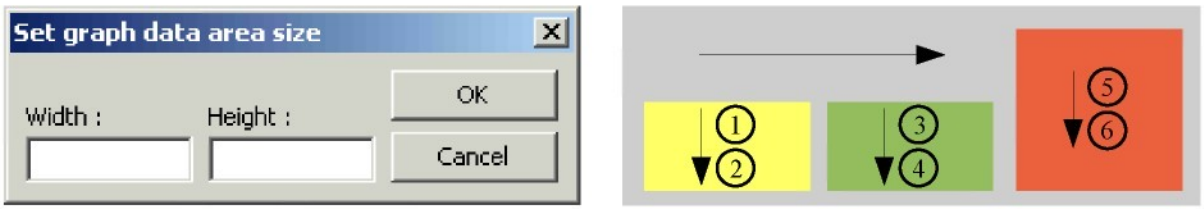


Obr. 12: Uvítací dialog aplikace MIOCC, je očekáván výběr jazyka.

3.4.2 Stanovení vnitřní geometrie dialogů pomocí tzv. sizerů

Dalo by se namítnout, že téma vnitřní geometrie dialogů není v rámci této práce podstatné, ovšem je nutné si uvědomit, že obě aplikace jsou řešeny multiplatformně a vzhledem k jejich charakteru tvoří dialogy stěžejní prvek komunikace s uživatelem. Rozmístění prvků pomocí absolutních souřadnic nemusí být vždy úplně vhodné, je nutné počítat s možnostmi různě nastavených vlastností použitého grafického prostředí a rozmístění prvků dialogu řešit pokud možno relativním pozicováním a také s ohledem na možnost změny rozměrů dialogu. Knihovny wxWidgets pro tento účel poskytují speciální objekty, tzv. sizery. Z důvodu názornosti bude uveden příklad implementace s využitím konkrétního typu sizeru ***wxBoxSizer***. Tento typ sizeru představuje z hlediska geometrie obdélník (box), do kterého jsou vsazeny jednotlivé prvky. Specifikace orientace určuje, zda se budou zařazovat od shora dolů (***wxVERTICAL***) nebo zleva doprava (***wxHORIZONTAL***). Nutnou samozřejmostí je možnost vložení dalších sizerů a to dokonce úplně jiného typu, takže lze vytvářet různě složité

struktury pozicování. Další důležitou vlastností sizerů je možnost plně specifikovat zarovnání a vzájemné odsazení jednotlivých prvků včetně samotných vsazených sizerů. Následující příklad demonstruje řešení pozicování prvků jednoho z použitých dialogů.



Obr. 13: Schéma rozmístění prvků dialogu pomocí obdélníkových sizerů.

Z obrázku 13 je patrné, že daný dialog používá čtyři obdélníkové sizery, přičemž hlavní je horizontální a obsahuje další tři sizery vertikální. V prvním vertikálním sizeru znázorněném žlutě bude umístěn statický text „Width :“ a pod ním příslušné textové pole. V druhém zeleném sizeru je situace stejná, tedy statický text „Height :“ a textové pole, ve třetím sizeru znázorněném červeně budou pod sebou umístěna dvě tlačítka. Implementace takového rozmístění je následovná.

```
// - titulek dialogu
SetTitle( _("Set graph data area size") );

// - sizery
p_BSizer_Global = new wxBoxSizer( wxHORIZONTAL );
p_BSizer_Width = new wxBoxSizer( wxVERTICAL );
p_BSizer_Height = new wxBoxSizer( wxVERTICAL );
p_BSizer_Buttons = new wxBoxSizer( wxVERTICAL );

// - viditelne prvky dialogu
p_SText_Width = new wxStaticText( ( wxWindow * )this, -1, _("Width :") );
p_SText_Height = new wxStaticText( ( wxWindow * )this, -1, _("Height :") );
p_TextCtrl_Width = new wxTextCtrl( ( wxWindow * )this, -1 );
p_TextCtrl_Height = new wxTextCtrl( ( wxWindow * )this, -1 );
p_Button_OK = new wxButton( ( wxWindow * )this, wxID_OK, _("OK") );
p_Button_Cancel = new wxButton( ( wxWindow * )this, wxID_CANCEL, _("Cancel") );

// - usporadani v sizerech
p_BSizer_Width -> Add( ( wxWindow * )p_SText_Width );
p_BSizer_Width -> Add( ( wxWindow * )p_TextCtrl_Width, 0, wxTOP, 3 );
p_BSizer_Height -> Add( ( wxWindow * )p_SText_Height );
p_BSizer_Height -> Add( ( wxWindow * )p_TextCtrl_Height, 0, wxTOP, 3 );
p_BSizer_Buttons -> Add( ( wxWindow * )p_Button_OK );
p_BSizer_Buttons -> Add( ( wxWindow * )p_Button_Cancel, 0, wxTOP, 5 );
p_BSizer_Global -> Add( ( wxSizer * )p_BSizer_Width, 0, wxALIGN_BOTTOM | wxALL, 7 );
p_BSizer_Global -> Add( ( wxSizer * )p_BSizer_Height, 0, wxALIGN_BOTTOM | wxTOP |
wxBOTTOM, 7 );
p_BSizer_Global -> Add( ( wxSizer * )p_BSizer_Buttons, 0, wxALL, 7 );

// - ridici sizer
SetSizerAndFit( ( wxSizer * )p_BSizer_Global );
```

Uvedený kód se bude nacházet v implementaci konstruktoru příslušného dialogu s tím, že použité proměnné představují členské proměnné objektu a musí být deklarovány v jeho hlavičkovém souboru. Zařazení prvku do sizeru je realizováno metodou ***Add***. První parametr určuje ukazatel na daný prvek, druhý příznak specifikuje možnost změny rozměru vsazeného prvku v hlavním směru sizeru, třetí parametr určuje kombinaci příznaků vlastností daného prvku, přičemž jednotlivé příznaky se oddělují binárním operátorem |. Jedná se o ***wxTOP*** (horní okraj), ***wxBOTTOM*** (dolní okraj), ***wxLEFT*** (levý okraj), ***wxRIGHT*** (pravý okraj), ***wxALIGN_BOTTOM*** (zarovnání dolů) atd. Čtvrtý parametr potom určuje velikost okrajů. Kromě vkládání vlastních prvků lze metodou ***AddSpacer*** vkládat pomocné mezery.

Knihovny wxWidgets nabízejí celou řadu dalších typů sizerů, např. ***wxGridSizer*** představuje uchycení do mřížky, ***wxStaticBoxSizer*** je uvedený ***wxBoxSizer*** ohraničený statickou linkou s nadpisem atd. Možnosti polohování prvků jsou ve wxWidgets hodně propracované a nabízejí také další, ovšem méně vhodné, metody. Pro základní představu možného způsobu relativního pozicování uvedený popis stačí. Bližší reference jsou k dispozici v manuálu wxWidgets.

4. Nasazení vytvořeného softwarového vybavení na reálné provozní měření

Softwarové vybavení bylo přirozeně během svého vývoje inkrementálně testováno s každou novou funkcí nebo implementací důležitého algoritmu. Ve finální podobě prokázalo svou funkčnost při krátkodobém i dlouhodobém testovacím měření a to prostřednictvím lokálního i vzdáleného serveru. Vzdálené testovací měření probíhalo na reálnou vzdálenost 150 km (není podstatné) prostřednictvím internetu při současném záměrném maximálním vytížení serveru. Případné vyvstalé problémy byly lokalizovány a odstraněny.

Dle požadavku zadání byla funkčnost vytvořeného softwarového vybavení ověřena také při reálném provozním měření. Toto měření proběhlo v laboratoři Katedry sklářských a keramických strojů Technické univerzity v Liberci, jednalo se o kontrolní měření vzájemně porovnávající okamžité teploty na dvou čelistech laboratorního lisu (uzavřených v laboratorní pícce) a teplotu vzduchu v laboratorní pícce. Cílem měření bylo stanovení okamžiku dosažení optimálních podmínek pro zahájení experimentu vyhodnocujícího reologické vlastnosti zkušebního vzorku konkrétní skloviny.

4.1 Teoretický popis daného problému

Technologie tvarování skla představuje složitý termomechanický proces, při kterém se z roztavené viskózní skloviny získává finální výrobek či polotovar následným nepřetržitým ochlazováním majícím za následek růst viskozity. Klasickým způsobem zpracování skloviny je tvarování skla lisováním.

Nejvýznamnější materiálovou vlastností skla ve vztahu k jeho deformačnímu chování je dynamická viskozita, která se v průběhu tvarování mění z hodnoty 10^1 Pa.s (tavná lázeň) po hodnoty řádově přesahující 10^{20} Pa.s (pokojová teplota). Transformační teplota odpovídá viskozitě $10^{12,3}$ Pa.s. Z fyzikálního hlediska představuje technologie lisování složitý proces, při kterém se uplatňuje mechanický a teplený děj propojený vzájemnou interakcí mezi sdílením tepla a viskózním tokem skloviny. Tvarovatelnost skloviny je ohraničena rozmezím viskozit 10^2-10^7 Pa.s.

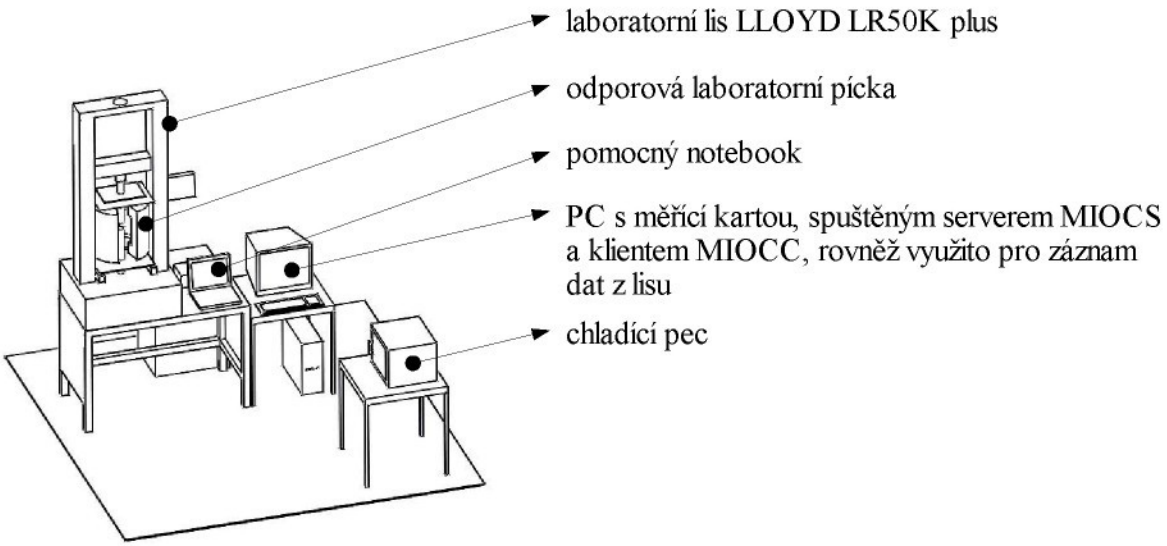
Cílem reologických měření je získat popis visko-elastického chování analyzované

skloviny na provozním zatížení v širokém teplotním rozsahu, který náleží intervalu viskozit 10^2-10^{10} Pa.s. Laboratorní experimenty jsou prováděny metodou stlačování vzorku mezi paralelními čelistmi laboratorního lisu. Princip experimentální metody spočívá ve vyhodnocování odezvy skleněného vzorku v závislosti na vnějším zatížení konstantní rychlostí. Celý tvarovací proces probíhá za izotermických podmínek, přičemž stanovená rychlost stlačení vzorku závisí na aktuální pracovní teplotě.

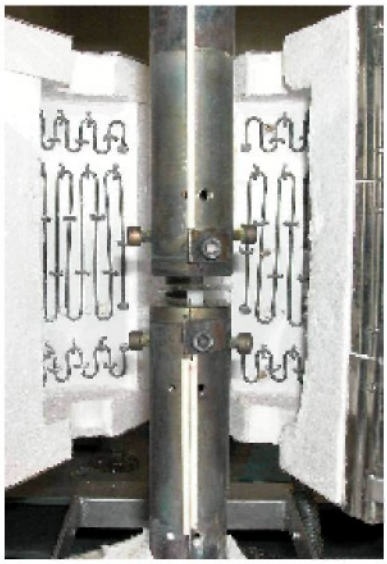
4.2 Popis a schéma měření

Měření teplot na obou čelistech laboratorního lisu (uzavřených v laboratorní pícce) i teploty vzduchu v laboratorní pícce bylo realizováno pomocí třech termočlánků typu K. Určení správné teploty na základě změřeného termoelektrického napětí předpokládá v tomto případě splnění dvou základních požadavků. Jedná se o přesnou kalibraci termočlánku i samotné měřící karty. Kalibrace měřící karty byla provedena softwarově pomocí výrobcem dodaných nástrojů, kalibrace termočlánku potom s použitím kalibrátoru. Vlastní účel měření spočíval v nutnosti stanovení okamžiku, kdy všechny tři teploty dosáhnou hodnoty 610-620 °C, přičemž jejich vzájemný rozdíl nesměl činit více než 5 °C. Jak je uvedeno v předešlé kapitole (4.1), tvarovací proces zkušebního vzorku skloviny pro splnění účelu reologického měření musí probíhat za izotermických podmínek. Všechny termočlánky byly zapojeny do svorkovnice měřící karty, konkrétně do analogových napěťových vstupů AI0, AI2 a AI4. K lokálně spuštěnému severu MIOCS byl připojen klient MIOCC, specifikoval detaily měření, tedy použití uvedených třech vstupů s napěťovým rozsahem +/- 0,1 V, perioda vzorkování činila 1 s a celkový čas měření byl pro jistotu nadsazen na 3 hodiny. Otevřená pomocná okna ukazovala hodnoty příslušných termoelektrických napětí a hodnoty vztažené na vlastní rozšířené osy s teplotní stupnicí. Konstrukce grafu probíhala současně, pro přehlednost byly jednotlivé řady nastaveny na zobrazení jiné barvy.

Obr. 14: Blokové schéma měření.



Obr. 15: Uspořádání pracoviště.

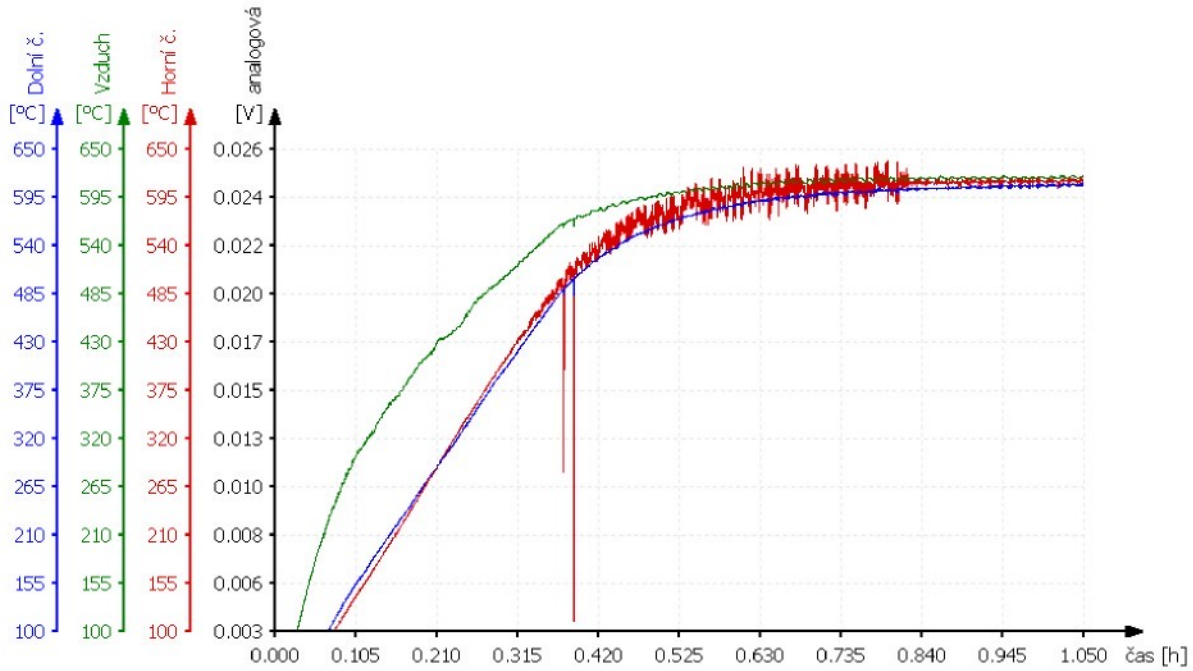


Obr. 16: Čelisti laboratorního lisu v otevřené laboratorní odporové pícce.

4.3 Výstupní grafická reprezentace dat, rozbor výsledků

Pohledem na obrázek 17, který představuje grafický výstup aplikace MIOCC, jsou patrná určitá fakta. Ohřev vzduchu (zeleně) vykazoval rychlejší náběh a s rostoucí teplotou se jeho sklon zmenšoval. Toto chování je vzhledem ke skutečnosti, že ohřev čelistí lisu probíhá vlivem ohřevu okolního vzduchu, normální. Ohřev horní (červeně) i dolní (modře) čelisti probíhal téměř shodně, což opět odpovídá předpokladu. Termočlánek horní čelisti vykazoval

v rozmezí teplot 450-600 °C kmitavý průběh výstupního napětí odpovídajícímu poměrně velkému rozpětí teplot. Tato skutečnost je pravděpodobně způsobena jeho vadou a v žádném případě neukazuje na špatnou funkci softwarového vybavení, právě naopak. K opětovnému ustálení výstupního napětí došlo po mírném pohybu horní čelistí lisu na základě vlastního zásahu experimentátora. Je pouze na něm, zda takovýto zásah a samotný fakt, že termočlánek vykazoval nestabilní chování, bude považovat za důvod výsledky experimentu nezohlednit do celkových výsledků své práce. K dosažení požadovaného stavu došlo v čase 1,05 hodiny, což s ohledem na daný experiment není podstatné. Celostránková verze grafu je k dispozici v rámci příloh.



Obr. 17: Exportovaný grafický výstup měření.

Závěr

Hlavním cílem této diplomové práce bylo s použitím programovacího jazyka C++ vytvořit softwarové vybavení umožňující komunikaci s danou měřící kartou v reálném čase, distribuci dat prostřednictvím počítačové sítě, jejich vizualizaci a další možnosti zpracování. Rozdělením na dvě nezávislé aplikace, tedy server a klient, bylo dosaženo určitého zjednodušení a zároveň rozšíření možností. Zatímco aplikace MIOCS (server) je z důvodu knihoven dodávaných výrobcem použité měřící karty k dispozici pouze pro platformu Win32, aplikaci MIOCC (klient) lze zkompilovat pro jakoukoliv platformu podporovanou projektem wxWidgets. S ohledem na možné dopravní zpoždění vlivem přenosu informací počítačovou sítí může časová nepřesnost na straně klienta interpretovaných dat vzhledem k řešení tohoto softwarového vybavení činit pouze dobu cesty požadavku zahájení měření z klienta na server zvětšenou o jednu desetinu sekundy. Obě aplikace jsou vytvořeny tak, aby byla umožněna jejich úprava pro použití s jinou měřící kartou a to poměrně pohodlným způsobem.

Práce poukázala na důležitost přesného časování v aplikacích reálného času a na vhodnost možných způsobů implementace časovacích algoritmů, kde demonstruje jedno z možných řešení. Ukazuje také vhodný způsob propojení informací získaných čtením kanálů/portů měřící karty s nutností jejich hromadné distribuce.

Funkčnost vytvořeného softwarového vybavení byla kromě inkrementálního testování během vývoje ověřena také nasazením na reálné provozní měření, které proběhlo v laboratoři Katedry sklářských a keramických strojů Technické univerzity v Liberci, kde bude systém nadále využíván a dále rozvíjen.

Použitá literatura

[1] HANSELMAN, D.-LITTLEFIELD, B.: Mastering Matlab 6. Prentice Hall, New Jersey 2001.

[2] MASÁR, I.-IVANOV, I.: Aplikácie reálneho času v programovom prostriedku MATLAB/SIMULINK. Slovenská technická univerzita, Bratislava 2001. ISBN 80-227-1601-4.

[3] Control Web 2000: Uživatelská příručka. Computer Press, Praha 1999. ISBN 80-7226-258-0.

[4] PTÁČEK, J.-JENČÍK, J. a kol.: Měření teploty v průmyslu 1. část. Dům techniky ČS VTS Praha, 1993.

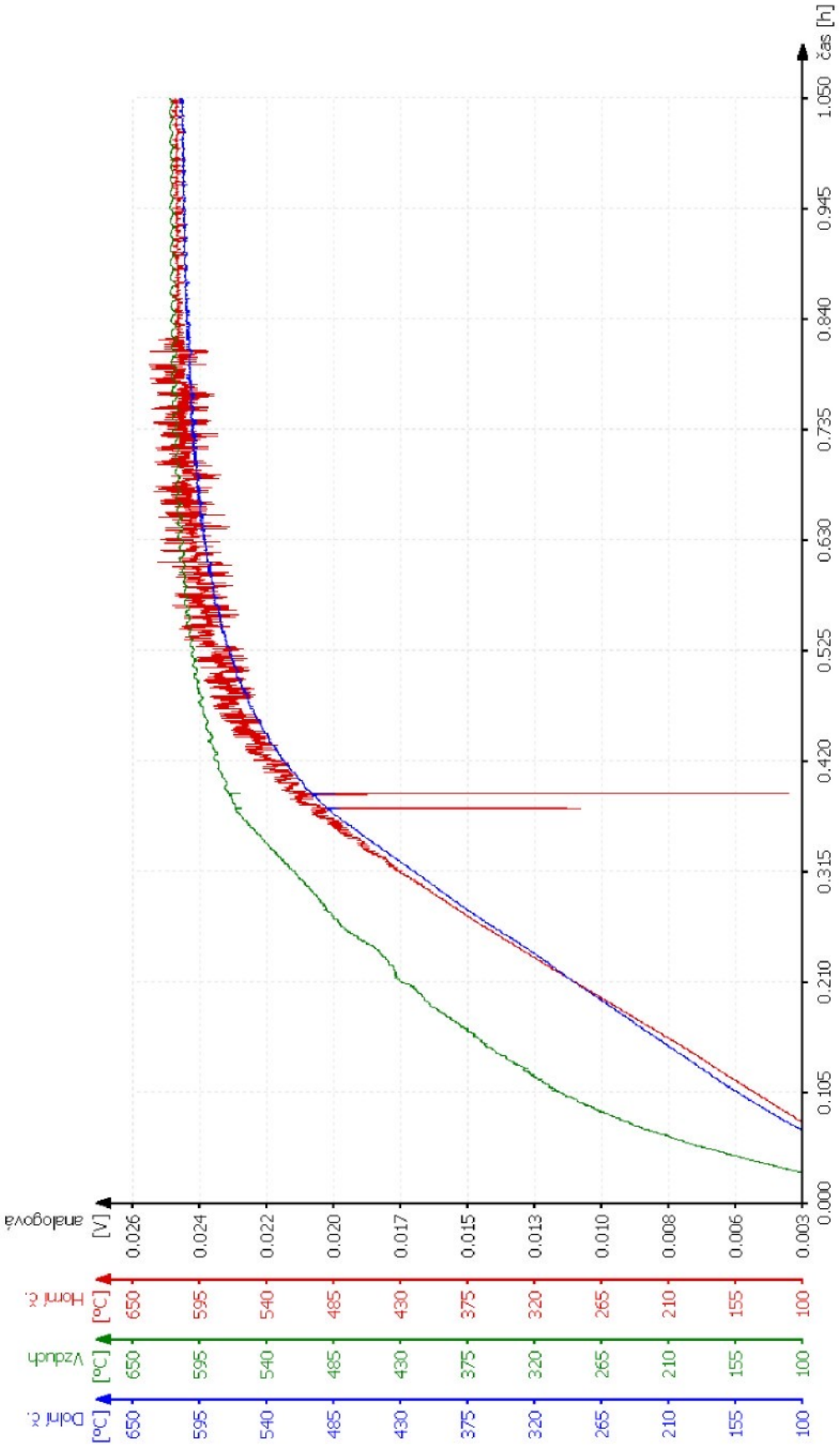
Internetové zdroje

[1] Oficiální web wxWidgets, příručka wxWidgets. http://www.wxwidgets.org

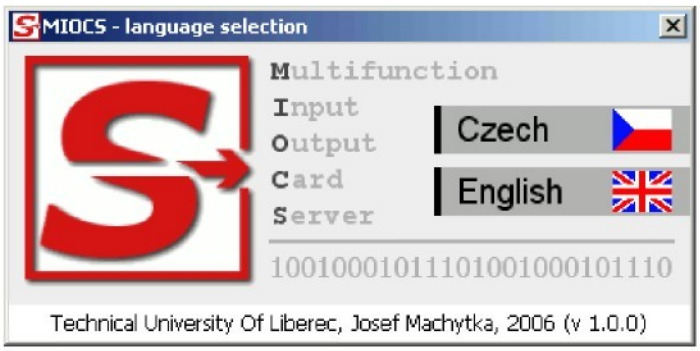
Přílohy

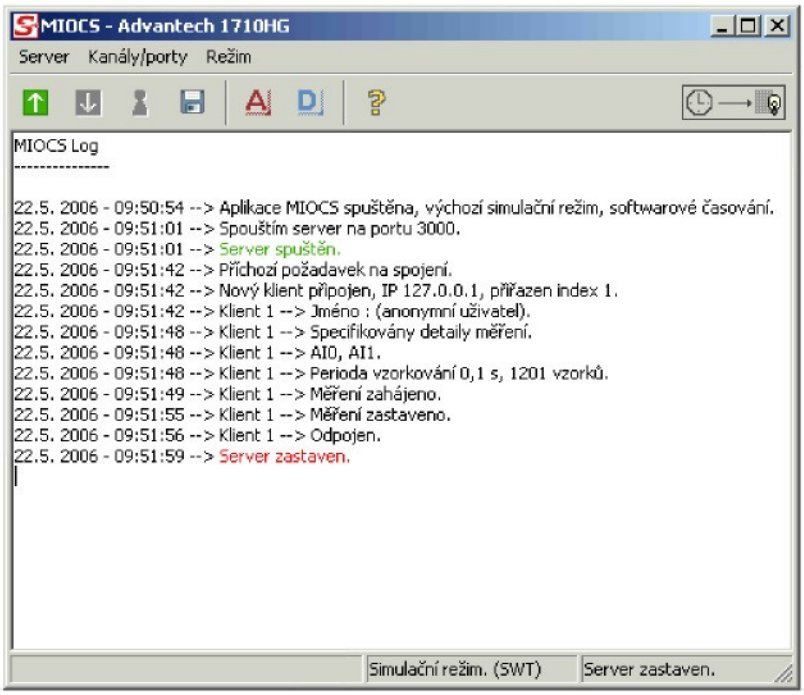
Obsah příloh

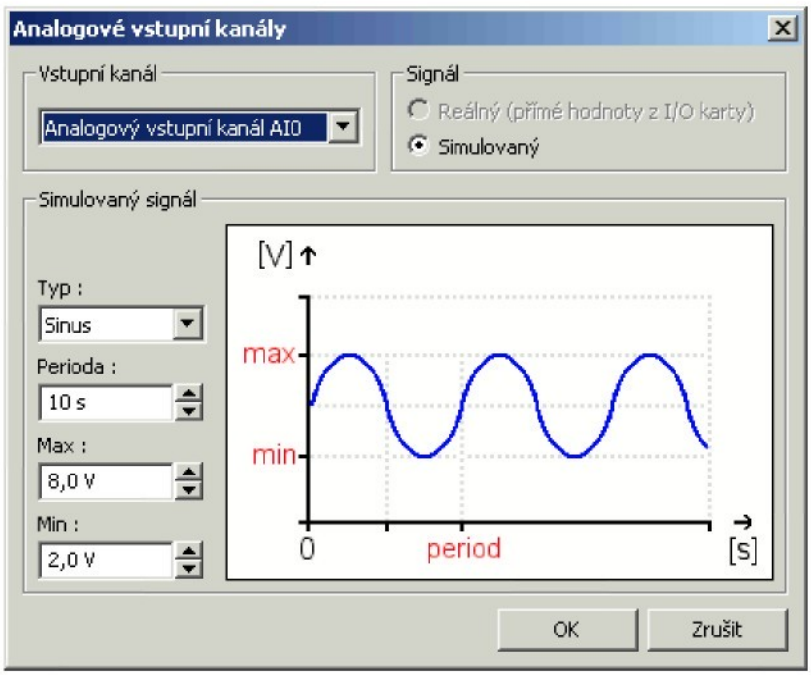
Grafický průběh měření celostránkově



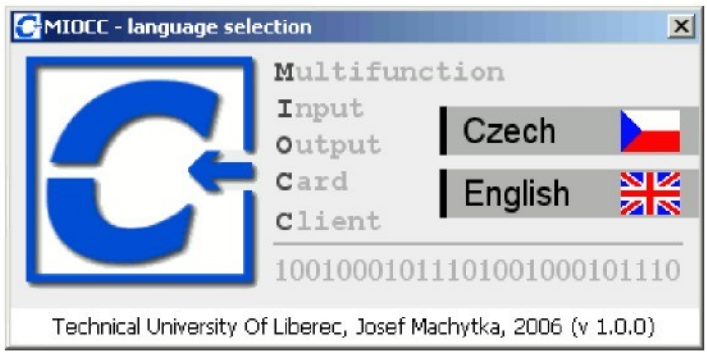
Ukázka prostředí aplikace MIOCS (server)

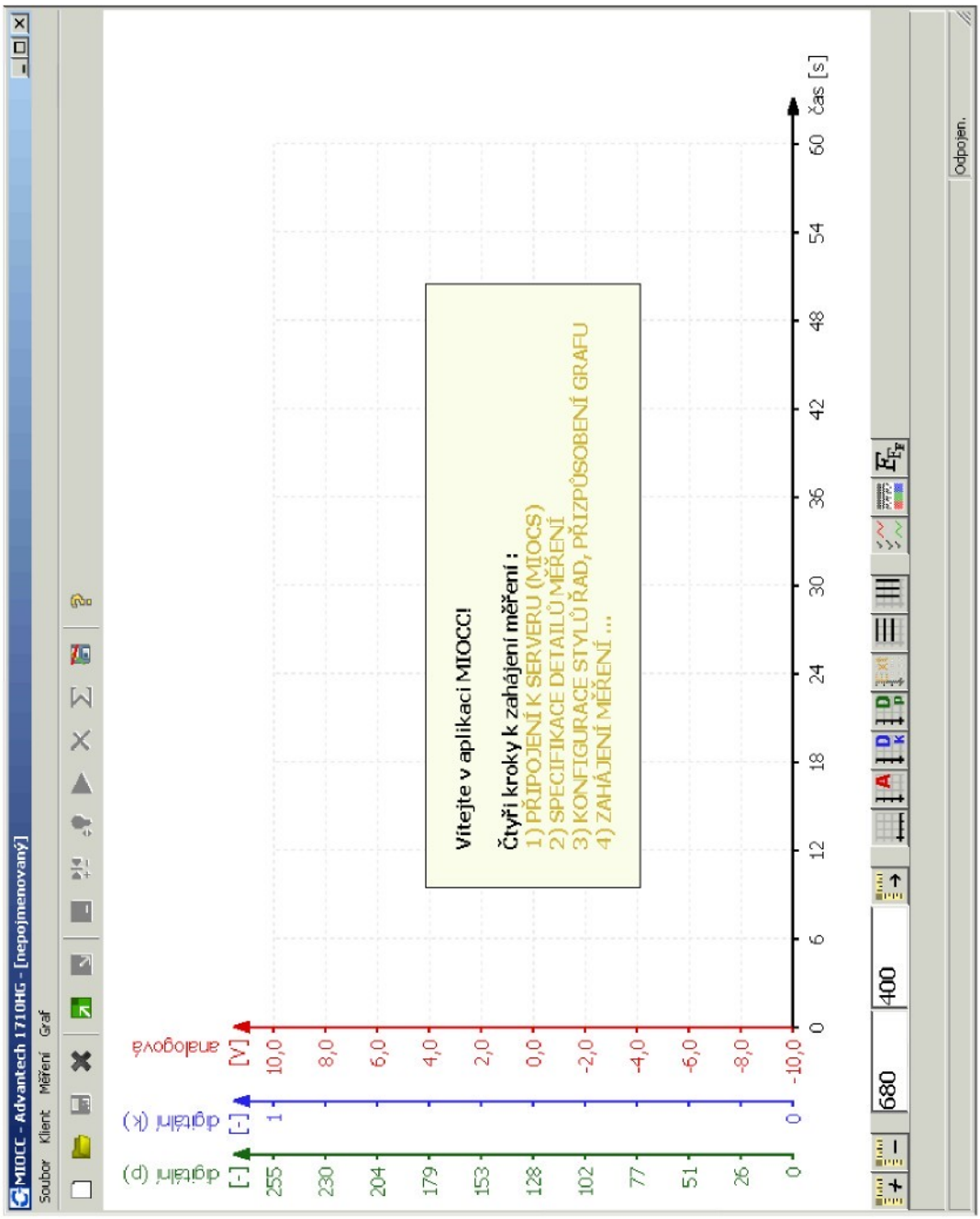




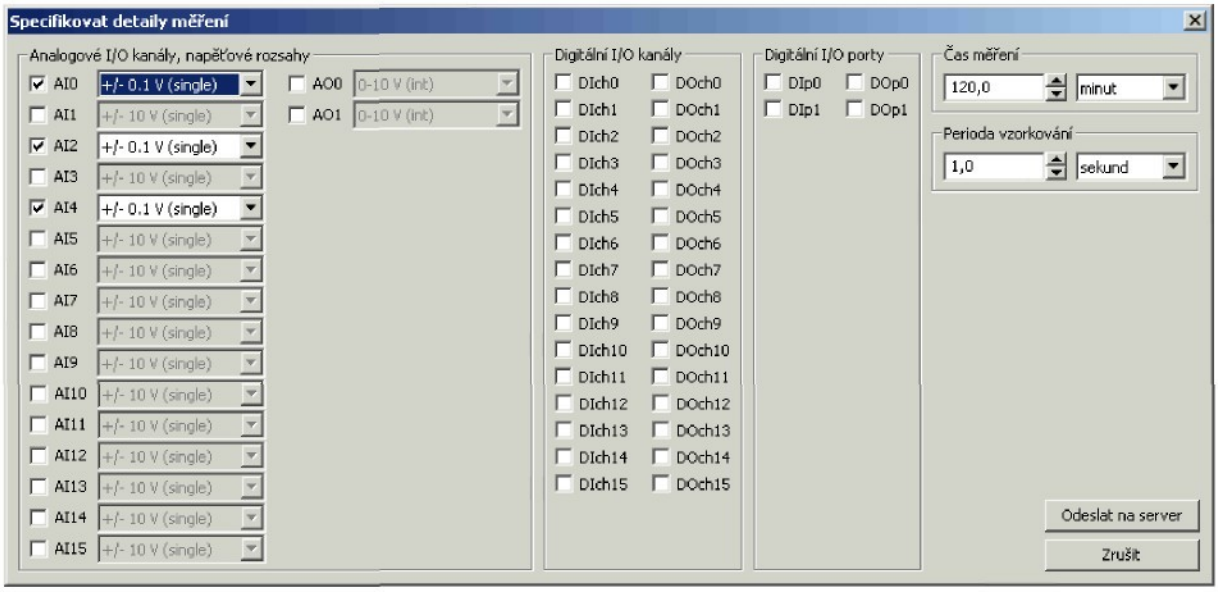


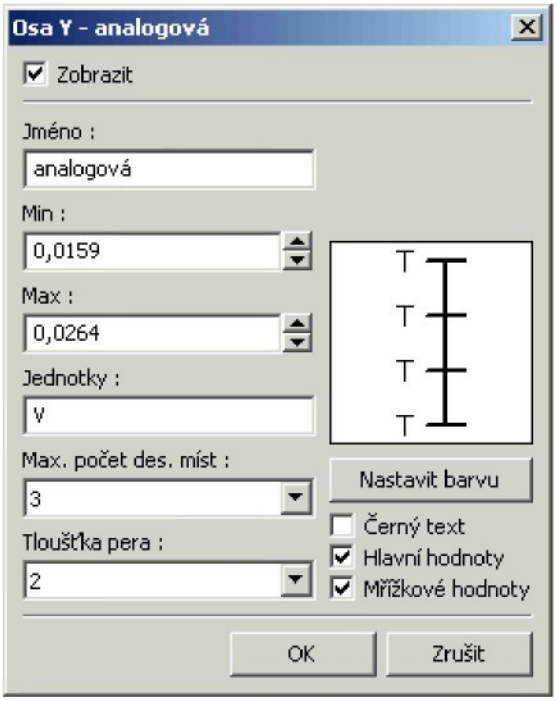
Ukázka prostředí aplikace MIOCC (klient)

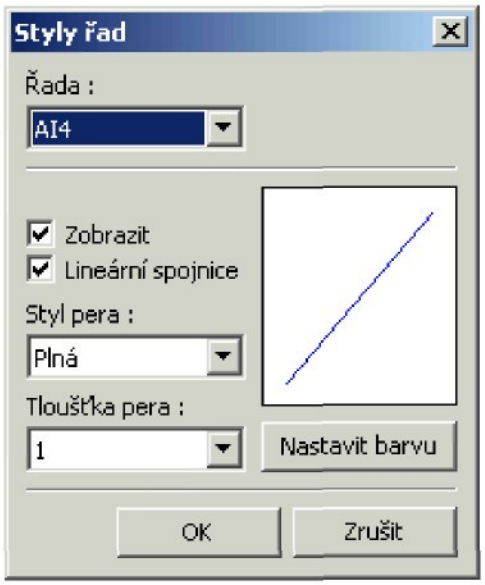




Ukázka prostředí aplikace MIOCC (klient)







Přehled zpráv komunikačního protokolu

Směr	Zpráva, popis
S-K	`CONNECTION|ACCEPTED|END` - potvrzení úspěšného připojení k serveru
S-K	`CONNECTION|DECLINED_FULL|END` - spojení odmítnuto, kapacita serveru překročena
S-K	`CONNECTION|LIVING|END` - udržovací kontrolní zpráva
K-S	`CLIENT_SPECS|typ wxString|typ wxString|typ wxString|typ bool|END` - specifikace vlastností klienta, obsahuje majoritní číslo verze, jméno uživatele, identifikační řetězec měřící karty a příznak připojení k lokálnímu serveru
S-K	`CLIENT_SPECS|OK|END` - specifikace vlastností klienta v pořádku
S-K	`CLIENT_SPECS|FALSE_VERSION|END` - majoritní verze obou aplikací neodpovídají, spojení je následně ukončeno
S-K	`CLIENT_SPECS|FALSE_CARD|END` - server byl zkompilován pro použití s jinou měřící kartou, spojení je následně ukončeno
K-S	`DETAILS_SPECIFY|typ double|typ long|pole bool|pole bool|...|pole short|END` - specifikace detailů měření, obsahuje periodu vzorkování, počet vzorků, příznaky použití jednotlivých kanálů/portů a kódy napěťových rozsahů analogových kanálů
S-K	`DETAILS_SPECIFY|OK|END` - specifikace detailů měření v pořádku
S-K	`DETAILS_SPECIFY|CONFLICT_typ wxString|END` - specifikace detailů měření bez efektu z důvodu výstupního konfliktu s jiným klientem, zpráva obsahuje řetězec se seznamem konfliktů
K-S	`DETAILS_DROP|NULL|END` - žádost o uvolnění detailů měření
S-K	`DETAILS_DROP|OK|END` - potvrzení uvolnění detailů měření
K-S	`MEASURING_START|NULL|END` - žádost o zahájení měření
K-S	`MEASURING_STOP|NULL|END` - žádost o přerušení měření
S-K	`MEASURING_STOP|OK|END` - potvrzení přerušení měření
S-K	`MEASURING_FINISHED|OK|END` - potvrzení úspěšně dokončeného měření
S-K	`MEASURING_ERROR|END` - zpráva s informací o chybě měření, spojení je následně ukončeno
S-K	`DATAGRAM|typ long|double,short,bool|double,short,bool|.....|END` - zpráva s datagramem, obsahuje index a data všech použitých kanálů/portů
K-S	`SET_OUTPUT|ANALOG,DIGITAL|CHANNEL,PORT|typ short|typ double|END` - žádost o nastavení výstupu, obsahuje specifikaci typu, index kanálu/portu a vlastní požadovanou hodnotu
S-K	`SET_OUTPUT|OK|END` - potvrzení nastavení výstupu

Komunikační chybové kódy aplikace MIOCS (server)

001 - nepodařilo se přečíst příchozí zprávu ze socketu
002 - příchozí zpráva neprošla prvotním ověřením základní validity, jedná se o hrubou chybu v komunikaci
003 - příchozí specifikace vlastností klienta není očekávána, specifikace již byla ověřena a provedena dříve
004 - příchozí specifikace vlastností klienta není validní
005 - nepodařilo se odeslat zprávu s informací o neplatné specifikaci klienta při procesu specifikace vlastností klienta
006 - nepodařilo se odeslat zprávu s informací o úspěšně provedené specifikaci vlastností klienta při procesu specifikace vlastností klienta
007 - příchozí požadavek na specifikaci detailů měření není očekáván, buď ještě neproběhla specifikace vlastností klienta, nebo právě probíhá měření
008 - příchozí zpráva s požadavkem na detaily měření není validní
009 - nepodařilo se odeslat zprávu s informací o konfliktu na výstupních kanálech/portech při procesu specifikace detailů měření
010 - nepodařilo se odeslat zprávu s informací o úspěšném nastavení detailů měření při procesu specifikace detailů měření
011 - příchozí požadavek na uvolnění detailů měření není očekáván, detaily měření nejsou nastaveny, nebo právě probíhá měření
012 - nepodařilo se odeslat zprávu s potvrzením uvolnění detailů měření při procesu uvolnění detailů měření
013 - příchozí požadavek na nastavení výstupního kanálu/portu není očekáván, měření neprobíhá
014 - příchozí zpráva s požadavkem na nastavení výstupního kanálu/portu není validní
015 - nepodařilo se odeslat zprávu s potvrzením o úspěšném nastavení výstupu podle požadavku při procesu nastavení výstupu
016 - příchozí požadavek na zahájení měření není očekáván, buď dosud nebyly nastaveny detaily měření, nebo již měření probíhá
017 - příchozí požadavek na zastavení měření není očekáván, měření neprobíhá
018 - nepodařilo se odeslat zprávu s potvrzením o zastavení měření při procesu zastavení měření
019 - příchozí zpráva obsahuje neznámý požadavek, hrubá chyba v komunikaci
020 - měření zahájeno, ale nepodařilo se odeslat zprávu s prvním datagramem
021 - nepodařilo se odeslat zprávu s datagramem nebo skupinou datagramů
022 - nepodařilo se odeslat zprávu s potvrzením o úspěšně dokončeném měření
023 - nepodařilo se odeslat zprávu s informací o korektně fungujícím a stále existujícím spojení (tzv. keep alive)

Komunikační chybové kódy aplikace MIOCC (klient)

001 - vypršel časový limit pro odpověď serveru na žádost o připojení
002 - nepodařilo se přečíst zprávu s odpovědí serveru na žádost o připojení
003 - zpráva s odpovědí na žádost o připojení není validní
004 - nepodařilo se odeslat specifikaci klienta na server
005 - vypršel časový limit pro odpověď serveru na specifikaci klienta
006 - nepodařilo se přečíst zprávu s odpovědí serveru na specifikaci klienta
007 - zpráva s odpovědí serveru na specifikaci klienta není validní
008 - nepodařilo se přečíst příchozí zprávu ze socketu
009 - vypršel časový limit pro obecnou aktivitu v socketu ze strany serveru (nedorazila ani keep alive zpráva)
010 - nepodařilo se spustit vlákna s časovači
011 - nepodařilo se odeslat zprávu s žádostí o specifikaci detailů měření
012 - vypršel časový limit pro odpověď serveru na žádost o specifikaci detailů měření
013 - nepodařilo se přečíst zprávu s odpovědí serveru na žádost o specifikaci detailů měření
014 - odpověď serveru na žádost o specifikaci detilů měření není validní
015 - nepodařilo se odeslat zprávu s žádostí o uvolnění detailů měření
016 - vypršel časový limit pro odpověď serveru na žádost o uvolnění detailů měření nebo je tato odpověď nevalidní
017 - nepodařilo se přečíst zprávu s odpovědí serveru na žádost o uvolnění detailů měření
018 - nepodařilo se odeslat zprávu s žádostí o zahájení měření
019 - nepodařilo se odeslat zprávu s žádostí o ukončení měření
020 - vypršel časový limit pro odpověď serveru na žádost o přerušení měření nebo tato odpověď není validní
021 - nepodařilo se přečíst zprávu s odpovědí serveru na žádost o přerušení měření
022 - server informoval o vyvstalé chybě měření, která postihuje tohoto klienta, měření nemá dále význam
023 - neznámá příchozí zpráva, jedná se o hrubou chybu v komunikaci
024 - příchozí datagram není validní
025 - příchozí datagram nemá očekávaný index
026 - vypršel časový limit pro potvrzení serveru o úspěšně dokončeném měření nebo toto potvrzení není validní
027 - nepodařilo se přečíst zprávu s potvrzením serveru o úspěšně dokončeném měření
028 - příchozí zpráva s datagramem nebo skupinou datagramů není očekávána, měření neprobíhá
029 - nepodařilo se odeslat zprávu s žádostí o nastavení výstupu
030 - vypršel časový limit pro odpověď serveru na žádost o nastavení výstupu nebo tato odpověď není validní
031 - nepodařilo se přečíst odpověď serveru na žádost o nastavení výstupu
032 - spojení se serverem bylo nečekaně ukončeno

Zdrojový text objektu časovače pracujícího ve vlastním vlákně ObjThreadTimer

Hlavičkový soubor obj_threadtimer.h

```
#ifndef __OBJ_THREADTIMER_H__
#define __OBJ_THREADTIMER_H__

// ------------------------------------------------------------------------------------------
// - objekt ObjThreadTimer neni odvozen od zadneho objektu
// ------------------------------------------------------------------------------------------
class ObjThreadTimerThread;
class ObjThreadTimer
{
  private:
        // - vlastnosti casovace a generovane udalosti
        int m_ID;
        wxEvtHandler *p_EvtHandler;
        void *p_ClientData;

  public:
        // - priznak spusteneho casovace, priznak nastaveni na generovani udalosti, priznak
        //   casovani na jednu jedinou udalost, interval pro generovani udalosti, zadana
        //   priorita pro beh vlakna
        // - k temto promennym lze pro potrebu zjisteni nastaveni primo pristupovat
        bool m_IsRunning;
        bool m_SetToGenerateEvents;
        bool m_OneShot;
        long m_EventInterval;
        int m_ThreadPriority;

        // - mutex pro ochranu promennych, ke kterym pristupuji obe vlakna
        wxMutex m_Mutex;

        // - ukazatel na objekt vlakna s casovacim algoritmem
        ObjThreadTimerThread *p_ThreadTimerThread;

        // - clenske metody (funkce)
        const bool Start( const long event_interval, const bool one_shot = false, const int thread_priority
= 90 );
        void Stop();
        void Reset( const long event_interval = -1, const short one_shot = -1, const int thread_priority =
-1 );
        void DisableEvents();
        void GenerateEvent();
        void SetClientData( void *client_data );
        void * GetClientData();

        // - konstruktor, destruktor
        ObjThreadTimer( const int id, wxEvtHandler *event_handler );
        ~ObjThreadTimer();
};

// ------------------------------------------------------------------------------------------
// - objekt ObjThreadTimerThread odvozen od wxThread
// ------------------------------------------------------------------------------------------
class ObjThreadTimerThread : public wxThread
{
  private:
        // - ukazatel na rodicovsky objekt
        ObjThreadTimer *p_Parent;

        // - mutex chranici promenne, ke kterym pristupuji obe vlakna
        wxMutex *p_Mutex;

        // - clenske metody (funkce)
        virtual ExitCode Entry();

  public:
        // - priznak pozadavku na provedeni restartovani casovace
        bool m_Reset;

        // - konstruktor, destruktor
        ObjThreadTimerThread( ObjThreadTimer *parent, wxMutex *mutex );
        ~ObjThreadTimerThread();
};

#endif
```

Implementační soubor obj_threadtimer.cpp

```
// ------------------------------------------------------------------------------------------------
// ------------------------------------------------------------------------------------------------
// ------------------------------------------------------------------------------------------------
// - zdrojovy soubor objektu ObjThreadTimer (presny obecne pouzitelny casovac bezici ve
//   vlastnim vlakne)
//
// - jedna se o pomerne vysoce presny obecne pouzitelny casovac bezici ve vlastnim vlakne
// - generuje udalosti EVT_TIMER s nastavenym id a predava je do specifikovane smycky zprav
//   (event handleru) ke zpracovani, lze take nastavit a cist ClientData
// - lze specifikovat priznak casovani na jednu jedinou udalost a prioritu pro beh vlakna (0 -
//   100) a to nejen pri spusteni casovace, ale i pri jeho resetu
// ------------------------------------------------------------------------------------------------
// ------------------------------------------------------------------------------------------------
// ------------------------------------------------------------------------------------------------

// ------------------------------------------------------------------------------------------------
// - include hlavickovych souboru
// ------------------------------------------------------------------------------------------------
#include "precomp.h"
#include "obj_threadtimer.h"

// ------------------------------------------------------------------------------------------------
// - konstruktor ObjThreadTimer
// ------------------------------------------------------------------------------------------------
ObjThreadTimer::ObjThreadTimer( const int id, wxEvtHandler *event_handler )
{
        m_ID = id;
        p_EvtHandler = event_handler;
        m_IsRunning = false;
        m_SetToGenerateEvents = true;
        m_OneShot = false;
        m_EventInterval = -1;
        m_ThreadPriority = -1;
        p_ThreadTimerThread = ( ObjThreadTimerThread * )NULL;
        p_ClientData = ( void * )NULL;

        // - vychozi odemceny stav mutexu
        m_Mutex.Unlock();
}

// ------------------------------------------------------------------------------------------------
// - spousti casovac implementovany ve vlastnim vlakne
// - vstupnimi parametry jsou :
//      const long event_interval - interval v ms pro generovani udalosti
//      const bool one_shot - priznak pozadavku na casovani jedne jedine udalosti (vychozi
//                            false)
//      const int thread_priority - pozadovana priorita pro beh vlakna (vychozi 90)
// ------------------------------------------------------------------------------------------------
const bool ObjThreadTimer::Start( const long event_interval, const bool one_shot, const int
thread_priority )
{
        m_Mutex.Lock();

        // - casovac spusten nebo je zadan nesmyslny interval pro generovani udalosti
        if( m_IsRunning || event_interval < 1 ) return false;

        m_EventInterval = event_interval;
        m_OneShot = one_shot;
        m_ThreadPriority = thread_priority;
        m_Mutex.Unlock();

        // - vytvoreni noveho vlakna casovace
        // - spusteni vlakna casovace
        p_ThreadTimerThread = new ObjThreadTimerThread( this, &m_Mutex );
        if( p_ThreadTimerThread -> Create() != wxTHREAD_NO_ERROR ||
                p_ThreadTimerThread -> Run() != wxTHREAD_NO_ERROR )
        {
                p_ThreadTimerThread -> Delete();
                p_ThreadTimerThread = ( ObjThreadTimerThread * )NULL;
                return false;
        }

        m_IsRunning = true;
        return true;
}

// ------------------------------------------------------------------------------------------------
// - zastavuje casovac bezici ve vlastnim vlakne
// ------------------------------------------------------------------------------------------------
void ObjThreadTimer::Stop()
{
```

```
        wxMutexLocker locker( m_Mutex );

        // - pokud casovac neni spusten
        if( !m_IsRunning ) return;

        p_ThreadTimerThread -> Delete();
        m_IsRunning = false;
}

// ------------------------------------------------------------------------------------------
// - resetuje spusteny casovac bez nutnosti preruseni behu vlakna a tim padem s minimalnim
//   zpozdenim a bez rizika, ze se jiz nepodari nove vlakno znovu vytvorit a spustit (na
//   nekterych platformach mohou byt vlakna omezene zdroje)
// - vstupnimi parametry jsou :
//      const long event_interval - novy interval v ms pro generovani udalosti (vychozi -1,
//                                  zustava stejny)
//      const short one_shot - novy priznak pozadavku na casovani jedne jedine udalosti
//                             (vychozi -1, zustava stejny)
//      const int thread_priority - novy pozadavek na prioritu vlakna (vychozi -1, zustava
//                                  stejny)
// ------------------------------------------------------------------------------------------
void ObjThreadTimer::Reset( const long event_interval, const short one_shot, const int thread_priority )
{
        wxMutexLocker locker( m_Mutex );

        // - pokud casovac neni spusten
        if( !m_IsRunning ) return;

        m_SetToGenerateEvents = false;
        if( event_interval > 0 ) m_EventInterval = event_interval;
        if( one_shot == 1 ) m_OneShot = true;
        else if( !one_shot ) m_OneShot = false;
        if( thread_priority != -1 )
        {
                m_ThreadPriority = thread_priority;
                p_ThreadTimerThread -> SetPriority( m_ThreadPriority );
        }
        p_ThreadTimerThread -> m_Reset = true;
}

// ------------------------------------------------------------------------------------------
// - zastavuje generovani udalosti, ale ponechava vlakno casovace spustene
// - pro opetovne spusteni generovani udalosti je nutne zavolat reset casovace
// - reset casovace probehne take bez nutnosti ukonceni vlakna
// ------------------------------------------------------------------------------------------
void ObjThreadTimer::DisableEvents()
{
        wxMutexLocker locker( m_Mutex );

        // - pokud casovac neni spusten
        if( !m_IsRunning ) return;

        m_SetToGenerateEvents = false;
}

// ------------------------------------------------------------------------------------------
// - generuje udalost EVT_TIMER se zadanym id a zarazuje ji do fronty prislusne smycky zprav
// ------------------------------------------------------------------------------------------
void ObjThreadTimer::GenerateEvent()
{
        // - pokud z nejakeho duvodu neexistuje specifikovana smycka zprav, jen pro jistotu
        if( !p_EvtHandler ) return;

        wxTimerEvent timer_event;
        timer_event.SetId( m_ID );
        timer_event.SetEventObject( ( wxObject * )this );
        p_EvtHandler -> AddPendingEvent( timer_event );
}

// ------------------------------------------------------------------------------------------
// - nastavuje ukazatel na ClientData tohoto objektu
// ------------------------------------------------------------------------------------------
void ObjThreadTimer::SetClientData( void *client_data )
{
        p_ClientData = client_data;
}

// ------------------------------------------------------------------------------------------
// - vraci ukazatel na ClientData tohoto objektu
// ------------------------------------------------------------------------------------------
void * ObjThreadTimer::GetClientData()
{
        return p_ClientData;
}
```

```
// ------------------------------------------------------------------------------------------
// - destruktor ObjThreadTimer
// ------------------------------------------------------------------------------------------
ObjThreadTimer::~ObjThreadTimer()
{
        if( p_ThreadTimerThread ) p_ThreadTimerThread -> Delete();
}

// ------------------------------------------------------------------------------------------
// - konstruktor ObjThreadTimerThread
// ------------------------------------------------------------------------------------------
ObjThreadTimerThread::ObjThreadTimerThread( ObjThreadTimer *parent, wxMutex *mutex ) :
    wxThread()
{
        p_Parent = parent;
        p_Mutex = mutex;
        m_Reset = false;
}

// ------------------------------------------------------------------------------------------
// - implementace casovaciho algoritmu zalozeneho na pravidelnem odecitani lokalniho casu v
//   milisekundach
// ------------------------------------------------------------------------------------------
void * ObjThreadTimerThread::Entry()
{
        // - nastaveni priority pro beh tohoto vlakna
        // - vypocet lokalniho casu v milisekundach pro prvni udalost
        p_Mutex -> Lock();
        SetPriority( p_Parent -> m_ThreadPriority );
        wxLongLong event_time = wxGetLocalTimeMillis() + p_Parent -> m_EventInterval;
        p_Mutex -> Unlock();

        // - smycka provadejici neustale cteni lokalniho casu v milisekundach
        // - porovnava ziskanou hodnotu s hodnotou odpovidajici dalsi udalosti
        while( 1 )
        {
                // - pokud je zadano ukonceni casovace
                if( TestDestroy() )
                {
                        p_Parent -> p_ThreadTimerThread = ( ObjThreadTimerThread * )NULL;
                        return ( void * )NULL;
                }

                p_Mutex -> Lock();
                if( p_Parent -> m_SetToGenerateEvents && wxGetLocalTimeMillis() >= event_time )
                {
                        p_Parent -> GenerateEvent();
                        if( p_Parent -> m_OneShot )
                        {
                                p_Parent -> m_IsRunning = false;
                                p_Mutex -> Unlock();
                                p_Parent -> p_ThreadTimerThread = ( ObjThreadTimerThread * )NULL;
                                return ( void * )NULL;
                        }
                        event_time += p_Parent -> m_EventInterval;
                }

                // - test na zadost o resetovani casovace
                if( m_Reset )
                {
                        event_time = wxGetLocalTimeMillis() + p_Parent -> m_EventInterval;
                        m_Reset = false;
                        p_Parent -> m_SetToGenerateEvents = true;
                }

                p_Mutex -> Unlock();
                Sleep( 1 );
        }
}

// ------------------------------------------------------------------------------------------
// - destruktor ObjThreadTimerThread
// ------------------------------------------------------------------------------------------
ObjThreadTimerThread::~ObjThreadTimerThread()
{
}
```

Šablona zdrojového textu objektu univerzálního komunikačního rozhraní s měřící kartou ObjCardInterface

Hlavičkový soubor _miocs___obj_cardinterface.h.blank

```
#ifndef __OBJ_CARDINTERFACE_H__
#define __OBJ_CARDINTERFACE_H__

// ------------------------------------------------------------------------------------------------
// - objekt deklarovan pro merici kartu : (doplnit)
// - autor : (doplnit)
// ------------------------------------------------------------------------------------------------

// ------------------------------------------------------------------------------------------------
// - objekt ObjCardInterface neni odvozen od zadneho objektu
// ------------------------------------------------------------------------------------------------
class ObjThreadInternalTimer;
class ObjCardInterface
{
  private:
        // - clenske metody (funkce)
        void UNIV_DCH_DoErrorActions( wxString error_desc, const short client_index );
        void UNIV_OnAnalogInputEvent( const short channel );
        void UNIV_OnDigitalInputBitEvent( const short channel );

        // ..........

  public:
        // - ukazatel na rodicovske okno
        wxWindow *UNIV_p_Parent;

        // - ukazatel na objekt pomocneho vlakna
        // - pomocne vlakno je vyuzito v implementaci casovace
        ObjThreadInternalTimer *UNIV_p_ThreadInternalTimer;

        // - posledni hodnoty na vstupech a aktualni hodnoty na vystupech
        // - pole techto dat nejsou vyuzita jen pro potreby tohoto rozhranni, ale pristupuje k nim i objekt
        //   serveru
        double UNIV_m_LastAIValue[AI_CHANNELS][AI_RANGES_COUNT];
        bool UNIV_m_LastDIValueBit[DI_CHANNELS];
        short UNIV_m_LastDIValueByte[DI_PORTS];
        double UNIV_m_CurrentAOValue[AO_CHANNELS];
        bool UNIV_m_CurrentDOValueBit[DO_CHANNELS];
        short UNIV_m_CurrentDOValueByte[DO_PORTS];

        // ..........

        // - clenske metody (funkce)
        const bool UNIV_TryFindCard();
        const bool UNIV_DCH_StartInternalTimer( const int event_interval = 100 );
        void UNIV_DCH_StopInternalTimer();
        const double UNIV_GetAnalogInputValue( const short channel, const short range_code, const short
client_index );
        const bool UNIV_GetDigitalInputValueBit( const short channel, const short client_index );
        const short UNIV_GetDigitalInputValueByte( const short port, const short client_index );
        void UNIV_SetAnalogOutputValue( const short channel, const double value, const short range_code,
const short client_index );
        void UNIV_SetDigitalOutputValueBit( const short channel, const bool value, const short client_index
);
        void UNIV_SetDigitalOutputValueByte( const short port, const short value, const short
client_index );

        // ..........

        // - konstruktor, destruktor
        ObjCardInterface( wxWindow *parent );
        ~ObjCardInterface();
};

// ------------------------------------------------------------------------------------------------
// - objekt ObjThreadInternalTimer odvozen od wxThread
// ------------------------------------------------------------------------------------------------
class ObjThreadInternalTimer : public wxThread
{
  private:
        // - ukazatel na rodicovsky objekt (rozhranni pro komunikaci s merici kartou)
        ObjCardInterface *UNIV_p_Parent;

        // - interval pro volani obsluhy udalosti
        int UNIV_m_EventInterval;
```

```
        // - clenske metody (funkce)
    virtual ExitCode Entry();

  public:
        // - konstruktor, destruktor
        ObjThreadInternalTimer( ObjCardInterface *parent, const int event_interval );
        ~ObjThreadInternalTimer();
};

#endif
```

Implementační soubor _miocs___obj_cardinterface.cpp.blank

```
// ------------------------------------------------------------------------------------------------
// - objekt implementovan pro merici kartu : (doplnit)
// - autor : (doplnit)
// ------------------------------------------------------------------------------------------------

// ------------------------------------------------------------------------------------------------
// ------------------------------------------------------------------------------------------------
// ------------------------------------------------------------------------------------------------
// - zdrojovy soubor objektu ObjCardInterface (komunikacni rozhranni mezi aplikaci a merici kartou)
// - tento objekt tvori univerzalni rozhranni mezi aplikaci a merici kartou
// - implementuje funkce pro jednotlive cteni vstupnich analogovych a digitalnich kanalu a
//   digitalnich portu, jednotlivy zapis do vystupnich analogovych a digitalnich kanalu a digitalnich
//   portu, dale funkce pro pouziti pripadneho integrovaneho citace/casovace
// - jsou take implementovany funkce pro obsluhu udalosti na vstupnich kanalech, tyto funkce jsou pouzity
//   v pripade, ze je karta vybavena moznostmi pro obsluhu udalosti
// - toto rozhranni komunikuje s merici kartou v jejim vlastnim "jazyce", pouziva k tomu dodane funkce
//   a obaluje tyto funkcionality do jednotnych nazvu a rozhrani
// ------------------------------------------------------------------------------------------------
// ------------------------------------------------------------------------------------------------
// ------------------------------------------------------------------------------------------------

// ------------------------------------------------------------------------------------------------
// - include hlavickovych souboru
// ------------------------------------------------------------------------------------------------
#include "../precomp.h"
#include "../app_miocs_setup.h"
#include "_miocs___obj_cardinterface.h"
#include "../form_main.h"
#include "../obj_server.h"

        // - hlavickovy soubor pro pristup k funkcim v dynamicky linkovane knihovne dodane k merici karte
        // - obsahuje deklaraci funkci, datovych struktur, stavovych kodu atd...
        #include "(doplnit)"

// ------------------------------------------------------------------------------------------------
// - konstruktor ObjCardInterface
// ------------------------------------------------------------------------------------------------
ObjCardInterface::ObjCardInterface( wxWindow *parent )
{
        UNIV_p_Parent = parent;
        UNIV_p_ThreadInternalTimer = ( ObjThreadInternalTimer * )NULL;

        // ..........

        for( short sh = 0; sh < AI_CHANNELS; sh++ )
                for( short sh2 = 0; sh2 < AI_RANGES_COUNT; sh2++ ) UNIV_m_LastAIValue[sh][sh2] = 0.0;
        for( sh = 0; sh < DI_CHANNELS; sh++ ) UNIV_m_LastDIValueBit[sh] = false;
        for( sh = 0; sh < DI_PORTS; sh++ ) UNIV_m_LastDIValueByte[sh] = 0;
        for( sh = 0; sh < AO_CHANNELS; sh++ ) UNIV_m_CurrentAOValue[sh] = 0.0;
        for( sh = 0; sh < DO_CHANNELS; sh++ ) UNIV_m_CurrentDOValueBit[sh] = false;
        for( sh = 0; sh < DO_PORTS; sh++ ) UNIV_m_CurrentDOValueByte[sh] = 0;

        // ..........
}

// ------------------------------------------------------------------------------------------------
// - provadi cinnosti spojene s osetrenim vznikle chyby na urovni komunikace s merici kartou
// - pokud do funkce vstupuje client_index s hodnotou -1, pak se ukonci cely server
// ------------------------------------------------------------------------------------------------
void ObjCardInterface::UNIV_DCH_DoErrorActions( wxString error_desc, const short client_index )
{
        // - pomocny ukazatel na hlavni okno a na objekt serveru
        FormMain *form_main = ( FormMain * )UNIV_p_Parent;
        ObjServer *server = form_main -> p_Server;

        if( error_desc.Last() != '.' ) error_desc << ".";

        // - log a odpojeni vsech klientu, ukonceni serveru
        if( client_index == -1 )
        {
```

```
                form_main -> Log( error_desc );
                server -> Notify( false );
                for( short sh = 0; sh < MAX_CLIENTS; sh++ )
                        if( server -> m_Clients[sh].Socket )
                        {
                                server -> SendMessage( sh, "MEASURING|ERROR|END" );
                                server -> m_Clients[sh].TEventsFromKeepAlive = 0;
                        }
                wxMilliSleep( 200 );
                server -> DisconnectAllClients();
                wxCommandEvent command_event;
                form_main -> OnServerStop( command_event );
        }

        // - odpojeni konkretniho postizeneho klienta
        else
        {
                wxString s;
                form_main -> Log( s << _("Client") << " " << ( client_index + 1 ) << " --> " << _("I/O
card") << " --> " << error_desc );
                s.Clear();
                form_main -> Log( s << _("Client") << " " << ( client_index + 1 ) << " --> " <<
_("Measuring error, disconnecting.") );
                server -> SendMessage( client_index, "MEASURING|ERROR|END" );
                server -> m_Clients[client_index].TEventsFromKeepAlive = 0;
                wxMilliSleep( 50 );
                server -> DisconnectClient( client_index );
        }
}

// ------------------------------------------------------------------------------------------
// - obsluha udalosti na merici karte, karta ohlasila zmenu analogoveho vstupniho napeti na
//   danem kanalu
// ------------------------------------------------------------------------------------------
void ObjCardInterface::UNIV_OnAnalogInputEvent( const short channel )
{
        // ..........

        // - osetreni chyby komunikace s kartou
        UNIV_DCH_DoErrorActions( "Error handling new AI event. Need to stop server.", -1 );
        return;

        // ..........

        for( short sh = 0; sh < AI_RANGES_COUNT; sh++ )
        {
                double new_voltage;

        // ..........

                UNIV_m_LastAIValue[channel][sh] = new_voltage;
        }

        // ..........
}

// ------------------------------------------------------------------------------------------
// - obsluha udalosti na merici karte, karta ohlasila zmenu digitalniho vstupniho stavu na danem kanalu
// ------------------------------------------------------------------------------------------
void ObjCardInterface::UNIV_OnDigitalInputBitEvent( const short channel )
{
        // ..........

        // - osetreni chyby komunikace s kartou
        UNIV_DCH_DoErrorActions( "Error handling new DI event. Need to stop server.", -1 );
        return;

        // ..........

        bool new_state;
        short new_value;

        // ..........

        UNIV_m_LastDIValueBit[channel] = new_state;
        UNIV_m_LastDIValueByte[port] = new_value;

        // ..........
}

// ------------------------------------------------------------------------------------------
// - pokousi se nalezt merici kartu v systemu
// - vraci true nebo false podle uspesnosti
// - reseni je na programatorovi, k dispozici je identifikacni podretezec karty CARD_ID_SUBSTRING
```

```
// ------------------------------------------------------------------------------------------
const bool ObjCardInterface::UNIV_TryFindCard()
{
        bool card_detected;

        // ..........

        return card_detected;
}

// ------------------------------------------------------------------------------------------
// - spousti casovac implementovany ve vlastnim vlakne a zalozeny na integrovanem
//   citaci/casovaci merici karty
// ------------------------------------------------------------------------------------------
const bool ObjCardInterface::UNIV_DCH_StartInternalTimer( const int event_interval )
{
        if( UNIV_p_ThreadInternalTimer ) return false;

        // - vytvoreni noveho vlakna casovace
        UNIV_p_ThreadInternalTimer = new ObjThreadInternalTimer( this, event_interval );
        if( UNIV_p_ThreadInternalTimer -> Create() != wxTHREAD_NO_ERROR )
        {
                UNIV_p_ThreadInternalTimer -> Delete();
                UNIV_p_ThreadInternalTimer = ( ObjThreadInternalTimer * )NULL;
                return false;
        }

        // - spusteni vlakna casovace
        if( UNIV_p_ThreadInternalTimer -> Run() != wxTHREAD_NO_ERROR )
        {
                UNIV_p_ThreadInternalTimer -> Delete();
                UNIV_p_ThreadInternalTimer = ( ObjThreadInternalTimer * )NULL;
                return false;
        }

        // - vsechno v poradku
        return true;
}

// ------------------------------------------------------------------------------------------
// - zastavuje naimplementovany casovac vyuzivajici vnitrni citac/casovac merici karty
// ------------------------------------------------------------------------------------------
void ObjCardInterface::UNIV_DCH_StopInternalTimer()
{
        if( !UNIV_p_ThreadInternalTimer ) return;
        UNIV_p_ThreadInternalTimer -> Delete();
}

// ------------------------------------------------------------------------------------------
// - vraci okamzitou hodnotu napeti specifikovaneho analogoveho vstupniho kanalu
// - hodnota napeti je vracena v rozsahu, ktery je klientem pozadovan
// ------------------------------------------------------------------------------------------
const double ObjCardInterface::UNIV_GetAnalogInputValue( const short channel, const short range_code, const
short client_index )
{
        // - pokud je karta vybavena funkcemi pro obsluhu udalosti, pak staci vratit posledni znamou
        //   hodnotu
        if( CARD_HAS_EVENT_HANDLER ) return UNIV_m_LastAIValue[channel][range_code];

        double voltage;

        // ..........

        // - osetreni chyby v komunikaci s kartou
        // - ziskat nebo urcit popis chyby a zapsat do error_message
        wxString error_message;
        UNIV_DCH_DoErrorActions( error_message, client_index );
        return 0.0;

        // ..........

        UNIV_m_LastAIValue[channel][range_code] = voltage;
        return UNIV_m_LastAIValue[channel][range_code];
}

// ------------------------------------------------------------------------------------------
// - vraci okamzitou hodnotu specifikovaneho digitalniho vstupniho kanalu
// ------------------------------------------------------------------------------------------
const bool ObjCardInterface::UNIV_GetDigitalInputValueBit( const short channel, const short client_index )
{
        // - pokud je karta vybavena funkcemi pro obsluhu udalosti, pak staci vratit posledni znamou
        //   hodnotu
        if( CARD_HAS_EVENT_HANDLER ) return UNIV_m_LastDIValueBit[channel];
```

```
        short bit_flag;

        // ..........

        // - osetreni chyby v komunikaci s kartou
        // - ziskat nebo urcit popis chyby a zapsat do error_message
        wxString error_message;
        UNIV_DCH_DoErrorActions( error_message, client_index );
        return false;

        // ..........

        if( bit_flag ) UNIV_m_LastDIValueBit[channel] = true;
        else UNIV_m_LastDIValueBit[channel] = false;
        return UNIV_m_LastDIValueBit[channel];
}

// ------------------------------------------------------------------------------------------
// - vraci okamzitou hodnotu specifikovaneho digitalniho vstupniho portu
// ------------------------------------------------------------------------------------------
const short ObjCardInterface::UNIV_GetDigitalInputValueByte( const short port, const short client_index )
{
        // - pokud je karta vybavena funkcemi pro obsluhu udalosti, pak staci vratit posledni znamou
        //   hodnotu
        if( CARD_HAS_EVENT_HANDLER ) return UNIV_m_LastDIValueByte[port];

        short value;

        // ..........

        // - osetreni chyby v komunikaci s kartou
        // - ziskat nebo urcit popis chyby a zapsat do error_message
        wxString error_message;
        UNIV_DCH_DoErrorActions( error_message, client_index );
        return 0;

        // ..........

        UNIV_m_LastDIValueByte[port] = value;
        return UNIV_m_LastDIValueByte[port];
}

// ------------------------------------------------------------------------------------------
// - okamzite nastavuje danou hodnotu napeti specifikovaneho analogoveho vystupniho kanalu
// ------------------------------------------------------------------------------------------
void ObjCardInterface::UNIV_SetAnalogOutputValue( const short channel, const double value, const short
range_code, const short client_index )
{
        // ..........

        // - osetreni chyby v komunikaci s kartou
        // - ziskat nebo urcit popis chyby a zapsat do error_message
        wxString error_message;
        UNIV_DCH_DoErrorActions( error_message, client_index );
        return;

        // ..........

        UNIV_m_CurrentAOValue[channel] = value;
}

// ------------------------------------------------------------------------------------------
// - okamzite nastavuje hodnotu specifikovaneho digitalniho vystupniho kanalu
// ------------------------------------------------------------------------------------------
void ObjCardInterface::UNIV_SetDigitalOutputValueBit( const short channel, const bool value, const short
client_index )
{
        // ..........

        // - osetreni chyby v komunikaci s kartou
        // - ziskat nebo urcit popis chyby a zapsat do error_message
        wxString error_message;
        UNIV_DCH_DoErrorActions( error_message, client_index );
        return;

        // ..........

        UNIV_m_CurrentDOValueBit[channel] = value;
}

// ------------------------------------------------------------------------------------------
// - okamzite nastavuje hodnotu specifikovaneho digitalniho vystupniho portu
// ------------------------------------------------------------------------------------------
void ObjCardInterface::UNIV_SetDigitalOutputValueByte( const short port, const short value, const short
```

```
client_index )
{
        // .........

        // - osetreni chyby v komunikaci s kartou
        // - ziskat nebo urcit popis chyby a zapsat do error_message
        wxString error_message;
        UNIV_DCH_DoErrorActions( error_message, client_index );
        return;

        // .........

        UNIV_m_CurrentDOValueByte[port] = value;
}

// ------------------------------------------------------------------------------------------------
// - destruktor ObjCardInterface
// ------------------------------------------------------------------------------------------------
ObjCardInterface::~ObjCardInterface()
{
        if( UNIV_p_ThreadInternalTimer ) UNIV_p_ThreadInternalTimer -> Delete();
        // .........
}

// ------------------------------------------------------------------------------------------------
// - konstruktor ObjThreadInternalTimer
// ------------------------------------------------------------------------------------------------
ObjThreadInternalTimer::ObjThreadInternalTimer( ObjCardInterface *parent, const int event_interval ) :
    wxThread()
{
        UNIV_p_Parent = parent;
        UNIV_m_EventInterval = event_interval;
}

// ------------------------------------------------------------------------------------------------
// - implementuje pouzitelny casovac zalozeny na integrovanem citaci/casovaci merici karty
// ------------------------------------------------------------------------------------------------
void * ObjThreadInternalTimer::Entry()
{
        // - vysoka priorita pro beh tohoto vlakna
        SetPriority( 90 );

        // - pomocny ukazatel na objekt serveru a jeho smycku zprav
        FormMain *form_main = ( FormMain * )( UNIV_p_Parent -> UNIV_p_Parent );
        ObjServer *server = form_main -> p_Server;
        ObjEvhServer *server_evh = server -> p_EvhServer;

        // .........

        // - osetreni chyby zabranujici spusteni casovace
        wxTimerEvent timer_event;
        timer_event.SetId( server_evh -> ID_TimerError );
        server_evh -> AddPendingEvent( timer_event );
        UNIV_p_Parent -> UNIV_p_ThreadInternalTimer = ( ObjThreadInternalTimer * )NULL;
        return ( void * )NULL;

        // .........

        // - pracovni smycka
        // - pristupovani k citaci/casovaci karty a zjistovani jeho stavu, urceni zda vygenerovat udalost a
        //   predat ji do smycky zprav serveru
        while( 1 )
        {
                // - pokud je zadano ukonceni cinnosti vlakna
                if( TestDestroy() )
                {
                        // .........

                        UNIV_p_Parent -> UNIV_p_ThreadInternalTimer = ( ObjThreadInternalTimer * )NULL;
                        return ( void * )NULL;
                }

                // .........

                // - je rozhodnuto ze nastala udalost
                // - vytvoreni udalosti casovace a jeji predani do smycky zprav serveru
                wxTimerEvent timer_event;
                timer_event.SetId( server_evh -> ID_Timer );
                server_evh -> AddPendingEvent( timer_event );

                // .........

                // - osetreni chyby ve spravne cinnosti casovace (pokud nastala chyba znemoznujici dalsi
                //   funkci casovace)
```

```
                wxTimerEvent timer_event;
                timer_event.SetId( server_evh -> ID_TimerError );
                server_evh -> AddPendingEvent( timer_event );
                UNIV_p_Parent -> UNIV_p_ThreadInternalTimer = ( ObjThreadInternalTimer * )NULL;
                return ( void * )NULL;

                // ..........

                Sleep( 1 );
        }
}

// ------------------------------------------------------------------------------------------
// - destruktor ObjThreadInternalTimer
// ------------------------------------------------------------------------------------------
ObjThreadInternalTimer::~ObjThreadInternalTimer()
{
}
```