ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická - Katedra řídicí techniky

Diplomová práce

Jiří Faust

Vzdálené řízení a monitorování technologického procesu

Vedoucí diplomové práce : **Ing. František Vacek**

PRAHA 2003

Abstrakt

The intention of this thesis is to realize the distributed control system, which controls movement of the laboratory manipulator Oscar. The system consists of control module, TCP server and TCP client. The control module is the Linux kernel module, which contains threads running in RTLinux. This module communicates with manipulator by the PCI card PLX1750. TCP server is a process running in user space on the same machine and its task is to coordinate the connected client's work. TCP client is an application developed in Java, which enables remotely, by Internet, watch and control the state of the controlling process.

Abstrakt

Cílem této diplomové práce je navrhnout a realizovat distribuovaný řídicí systém, který řídí pohyb ramen laboratorního manipulátoru Oskar. Systém se skládá z řídicího modulu, TCP serveru a TCP klienta. Řídicí modul je modul jádra Linuxu, který obsahuje vlákna (threads) spouštěná v RTLinuxu. S manipulátorem komunikuje za pomocí PCI karty PLX 1750. TCP server je proces běžící v uživatelském prostoru na stejném stroji a jeho úkolem je koordinovat činnost všech připojených klientů. TCP klient je aplikace implementovaná v Javě, která umožňuje vzdáleně, prostřednictvím sítě Internet, pozorovat a řídit vývoj stavu řízeného procesu.

Prohlášení

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

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

V Praze dne ……. …….
podpis

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

Z důvodu .....odpírám udělit souhlas s užitím tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb. , o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne ……. …….
podpis

OBSAH

1. Úvod

Distribuované řídicí systémy jsou v současné době nezbytnou součástí většiny moderních výpočetních systémů. I přes některé nevýhody znamená rozdělení výpočetní síly na více strojů výrazné zlepšení kontroly řízených procesů. Každý distribuovaný systém se dělí na několik uzlů systému s různou výpočetní silou. Tyto uzly spolu komunikují a zasílají si data k dosažení společného cíle.

Cílem této práce je navrhnout distribuovaný řídicí systém, který řídí pohyb ramen manipulátoru k dosažení žádané polohy. Manipulátor Oskar je zmenšeným modelem v průmyslové oblasti často využívaných robotů. Navržený systém má sloužit jako doplněk výuky v laboratořích.

Při implementaci systému je použit laboratorní manipulátor Oskar, jehož vnitřní elektronika je navržena pro komunikace s PLC. Pro připojení robota k PC je použita laboratorní PCI karta PLX 1750. Protože úrovně signálů obou částí jsou odlišné, je nutné provádět jejich úpravu směrem z PC i směrem do PC. Na počítači, kde běží řídicí proces, je nainstalován operační systém RTLinux s podporou TCP/IP protokolu. Aby bylo možné komunikovat se vzdálenými klienty, bylo nutné vytvořit TCP server a spouštět ho na známé IP adrese a konkrétním portu služby.

Koncoví uživatelé účastnící se komunikace v distribuovaném systému mají mít možnost sledovat a zasahovat do řídicího procesu k dosažení cílové polohy. Počet klientů má být neomezen.

Návrh systému jsem rozdělil na několik částí.

1. Pro zajištění komunikace mezi vnitřní elektronikou manipulátoru a PCI kartou PLX 1750 bylo třeba přizpůsobit úrovně signálů obou částí.
2. Dále bylo třeba vytvořit kód, pomocí něhož lze využívat připojené PCI zařízení PLX 1750.
3. Vytvoření real-time řídicího modulu umožní přesně kontrolovat pohybu manipulátoru.
4. Vytvoření TCP serveru, což je program využívající služby Internetu a koordinující činnost klientů
5. Klientská aplikace tvoří vzdálený (remote) prvek systému. Umožňuje sledovat vývoj stavu procesu v grafické 3D animaci a jednoduchým způsobem zadávat požadované pozice ramen manipulátoru.

Řízení laboratorního manipulátoru je časově kritická úloha. Je proto nutné zajistit správnou kontrolu nad řídicím procesem. Tím se zabývá problematika real-time řízení a real-time operačních systémů.

Seznámení se s PCI architekturou a podporou Linuxu pro práci s připojenými PCI kartami, umožní využívat služby karty PLX1750.

Pro zajištění vzdálené komunikace je možné využít architekturu klient-server. Jako komunikační prostředek slouží síť Internet. Téma distribuovaných systémů se zabývá architekturou a možnostmi vývoje softwaru na této síti.

Protože klientské aplikace jsou vzdálenými prvky systému, které běží na stroji s neznámou platformou, bylo třeba zvážit způsob vhodného vývoje klienta. Jako jedním z nejvýhodnějších nástrojů se ukázala Java. Je předmětem diskuse zvážit výhody a nevýhody Javy a popřípadě zmínit jiné možnosti návrhu klienta.

2. RT řízení

Tato kapitola se věnuje otázce způsobu řízení časově kritických událostí. Snaží se objasnit architekturu real-time řídicích systémů a jak se tyto systémy liší od běžných systémů. Real-time řízení je na osobních počítačích podporováno prostřednictvím real-time operačních systémů. Jedna z následujících kapitol se proto také věnuje architektuře OS RTLinux a jeho aplikačním rozhraním pro vývoj real-time řídicích systémů .

Představme si, že připojíme reproduktor na paralelní port počítače. Následně spustíme program, který na tento port posílá data zvukového záznamu. Je-li tento program jediný běžící na počítači, reproduktor vytváří čistý, ustálený zvuk podobný originálnímu. Spustíme-li na stejném stroji jinou aplikaci (např. internetový prohlížeč), vzniknou zvukové mezery a zvuk se stává nepravidelným. Jiný proces totiž ubírá strojový čas našemu procesu a ten nemůže pravidelně, s požadovanou frekvencí posílat byty na paralelní port. Zvuk se stává přerušovaným v závislosti na prioritě ostatních procesů.

Jako jiný příklad, poslouží případ s kamerou, která plní buffer obrazovými daty každou milisekundu. Jakékoliv chvilkové zpoždění v procesu, který načítá data, může způsobit ztrátu dat.

V případě robota, který je poháněn krokovými motory, je nezbytně nutné, aby byly všechny pulsy generované v přesně definovaných intervalech. V opačném případě by vznikly vibrace a nepravidelnosti v pohybu robota. Pokud navíc dojde ke krátkodobému zpoždění při generování signálu, tento puls se ztrácí a při řízení robota dochází ke kritické chybě.

V předchozích případech byly procesy náročné na přesnost přidělení strojového času procesoru. Nejde ani tak o dobu přidělenou procesu, jako o pravidelnost a přesnost přidělení. Ve víceúlohovém (multitask) prostředí, kde běží více úloh najednou, je každému procesu přidělen nesouvislý blok časových intervalů, pro které nelze přesně předurčit jejich časy.

Běžné operační systémy jsou navrženy tak, aby optimalizovaly průměrnou dobu práce všech procesů a snaží se každému procesu přidělit spravedlivé sdílení výpočetního času. To je vhodné pro všeobecné použití výpočetní techniky. Takovými operačními systémy jsou WINDOWS, Linux, apod.

Real-time systém je systém, ve kterém je přesnost časování a předvídatelnost chování systému základní podmínkou. To například znamená, že krokové motory dostanou impulsy s pravidelnou periodou, kamera bude snímat scénu v předem určených intervalech a nebo se zvukový signál dostane na výstupní zvukové zařízení se správnou frekvencí, která odpovídá originálnímu záznamu.

2.1. Typy real-time systémů

Real-time systémy se dělí podle toho, jak dodržují přesnost časování. Rozlišují se dva základní typy.

a) Hard real-time systém

Překročení mezní hodnoty v časování má nepříznivé důsledky na chod systému. Používá se v řízení časově kritických procesů, kde je přesné časování základní podmínkou. V řízení průmyslového manipulátoru krokovými motory je zajištěn pohyb po předem zjistitelné trajektorii s velkou přesností a konstantní rychlostí pohybu . V systémech sběru dat se snímají informace ze senzorů v pravidelných časových intervalech.

b) Soft real-time systém

Krajní meze při časování mohou být výjimečně překročeny a následně obnoveny. Názorným příkladem je program, který zobrazuje video signál na obrazovce. Po takovém procesu požadujeme, aby běžel rychle (tzn. rychle vykresloval pixely) a s velkou frekvencí (tzn. obnovování obrázků), což zajišťuje dobrou kvalitu obrazu. Několik milisekund zpoždění nebo naopak předstih nezpůsobí velké zkreslení.

2.2. RT Linux

Role operačního systému je poskytnout programům stejný pohled na hardware počítače. Operační systém musí navíc zajistit ochranu proti neautorizovaným přístupům ke zdrojům (resources). Toto je možné pouze v případě, kdy CPU zajistí ochranu systémového softwaru od aplikací. Každý moderní procesor podporuje toto chování pomocí implementace rozlišných operačních stupňů přímo v CPU. Stupně mají rozdílné role a některé operace nejsou povoleny v nižších vrstvách.

Unixové systémy jsou navrženy tak, že používají dvě vrstvy:

a) *Jádro* se nachází v nejvyšší vrstvě (nazývaná *supervisor mode*) , kde je ‘vše‘dovoleno. Skládá se z jednotlivých modulů, které mohou být dynamicky přidány nebo odebrány.

b) *Aplikace* jsou spouštěny jako samostatné procesy v nejnižší vrstvě (*user mode*), ve které procesor kontroluje přímý přístup k hardwaru a neautorizovaný přístup do paměti.

Unix předává řízení z uživatelského prostoru do prostoru jádra kdykoliv aplikace zapříčinila vznik systémového volání, nebo byla přerušena hardwarovým přerušením.

I když OS Linux podporuje systémová volání pro přerušení procesů na zadaný časový interval, není zaručeno, že proces bude znovu spuštěn, jakmile tento čas uplyne. V závislosti na zatížení systému může být proces spuštěn i o více než sekundu později. Navíc může být

přeplánován (preempted) v nepředvídatelném okamžiku a musí čekat na přidělení jeho času na CPU. Nastavení nejvyšší priority pro kritické úlohy problém nevyřeší. Příčinou je optimalizace plánovacího algoritmu ‘standardního‘ Linuxu na spravedlivé přidělovaní strojového času všem běžícím procesům. Máme-li real-time úlohu, pak požadujeme přidělení CPU kdykoliv je potřeba, nehledě na to, jak spravedlivé to je. Přesně to nám zaručuje real-time OS jako například RTLinux, Qnx a další.

2.2.1. Architektura RTLinuxu

RT Linux je hard real-time operační systém, který vychází z běžného typu operačního systému Linux. Na počítač není instalován jako samostatný operační systém, ale tvoří rozšíření již instalovaného OS Linuxu.

Záměrem je zkombinovat dvě skupiny obtížně mísitelných vlastností (viz. tab. 2.1):

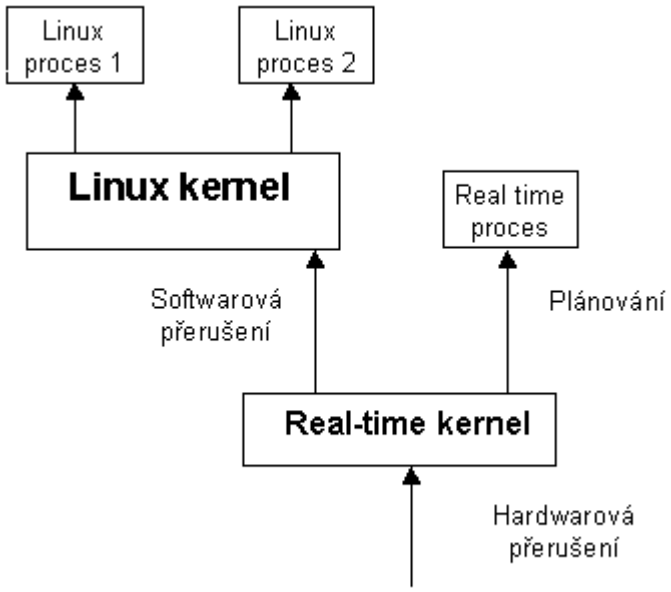
- *hard real-time služby***:** předvídatelnost, rychlost, jednoduché plánování
- *všechny služby standardního OS***:** podpora GUI (graphics user interface) aplikací, TCP/IP, NFS, kompilátory, webové servery, …

Real-Time OS	Plnohodnotný OS
Optimalizuje nejhorší případ	Optimalizuje průměrný případ
Předvídatelné plánování	Účelné plánování
Jednoduché spouštění	Široký záběr služeb
Minimalizuje prodlevy	Maximalizuje propustnost

Tab. č. 2.1: Srovnání vlastností Real-time OS a běžného OS

Kombinace těchto vlastností je zajištěna v RTLinuxu spouštěním speciálních real-time úloh a handlerů přerušení na stejném stroji, na kterém běží standardní (non real-time) Linux. Tyto úlohy a handlery se spouští kdykoliv je potřeba bez ohledu na to, co právě provádí Linux. Pokud na počítači neběží žádná real-time úloha, pak jádro naplánuje úlohu Linux. To znamená, že Linux je úloha s nejnižší prioritou real-time jádra (viz. obrázek 2.1).

Real-time programy RTLinuxu je modul jádra Linuxu, který obsahuje vlákna spouštěná v RTLinuxu. Mají pevně přidělenou paměť pro kód a data. Pokud by paměť nebyla pevně přidělená, mohlo by docházet k nepředvídatelným zpožděním v okamžiku, kdy by úloha žádala o novou paměť.



Obr. 2.1: Architektura RT Linuxu

2.2.2. Aplikační rozhraní RTLinuxu

Správně navržený systém by měl mít malou a jednoduchou real-time část a větší hlavní část, by měla být běžným Linuxovým programem. Takový systém umožňuje snazší ladění a porozumění náročnější real-time části.

Real-time úloha je vyvíjena jako speciální modul jádra Linuxu, který může být do jádra přidán v okamžiku, kdy je potřeba. Linuxový modul není nic jiného než soubor (.o) kompilovaný s parametrem *-c* kompilátoru *gcc***.** Do jádra je vložen příkazem *insmod* a z jádra je odebrán příkazem *rmmod*. Na rozdíl od běžného programu má modul jádra jeden vstupní a jeden výstupní bod v podobě následujících funkcí:

- *int init_module()* – funkce je volána, když je modul poprvé zaveden do jádra
- *void cleanup_module()* – funkce je volána, když je naopak modul z jádra odebrán

Protože každá část programu se nachází v jiném prostoru operačního systému (kernel space, user space), je nutné zajistit mechanismus umožňující komunikaci mezi oběma částmi. Nejčastěji se používají dva způsoby výměny dat:

a) *Real-time FIFO (first in first out)* jsou fronty, do kterých je možné zapisovat na jedné straně a číst na straně druhé. Protože fifo jsou pouze jednosměrné, pro obousměrnou výměnu dat je třeba vytvořit dvě takové fronty. V souborovém systému jsou reprezentovány jako zařízení */dev/rtf0 /dev/rtf1* …. V RTLinuxu se používají následující funkce pro práci s rt fifo [ref. <rtl_fifo.h>]:

- *rtf_create* – inicializuje fifo
- *rtf_destroy* – zruší fifo
- *rtf_put* – zapisuje data do fronty
- *rtf_get* – čte data z fronty

b) *Sdílená paměť* umožňuje přístup více procesům ke společnému místu v paměti. Při prvním volání funkce pro vytvoření bloku sdílené paměti pod určeným názvem, je alokováno místo v paměti. Počet odkazů na tento blok je inicializován na hodnotu 1. Pokud již blok sdílené paměti při volání funkce existuje, je navrácen pointer na tento blok paměti a zvýší se počet odkazů o 1. Při volání funkce na uvolnění paměti se sníží počet odkazů o 1. Když počet klesne na 0, je buffer uvolněn. Základní funkce pro práci se sdílenou pamětí[*] [ref.<mbuff.h>]:

- *mbuff_alloc* – alokuje blok sdílené paměti podle zadaného jména
- *mbuff_free* – uvolňuje blok přidělené sdílené paměti

Real-time program se může skládat z několika vláken (thread), která v RTLinuxu sdílejí adresový prostor jádra Linuxu. Přehled základních funkcí pro práci s POSIX thready v RTLinuxu [ref. <pthread.h>]:

- *pthread_create* – vytvoří nový real-time thread
- *pthread_attr_init* – inicializace atributů vlákna
- *pthread_attr_setschedparam* – mimo jiné nastavuje prioritu threadu
- *pthread_make_periodic_np* – umožní real-time threadu běžet periodicky
- *pthread_wait_np* – přeruší provádění threadu do uvedené doby
- *pthread_cancel* – zruší vytvořené vlákno

[*] Rozhraní pro práci se sdílenou pamětí je součástí distribuce RTLinuxu. Při jejím používání je třeba nejprve vložit do jádra modul *mbuff.o*

V RTLinuxu bylo třeba změnit způsob zpracování přerušení než jaký je v Linuxu. Existují dva typy přerušení (viz. obrázek 2.1).

a) *Soft interrupts* jsou běžná přerušení Linuxového jádra. Mají tu výhodu, že některé funkce Linuxového jádra mohou z nich být bezpečně volány. Neposkytují hard real time výhody a mohou být o nějaký čas zpožděny. Funkce pro práci se soft interrupts jsou [ref. <rtl_core.h>]:

   - *rtl_get_soft_irq* – alokuje virtuální irq číslo a instaluje pro něj handler
   - *rtl_global_pend_irq* – spouští virtuální přerušení s daným irq
   - *rtl_free_soft_irq* – uvolňuje alokované virtuální přerušení

b) *Hard interrupts* jsou real-time přerušení a mají mnohem kratší reakční čas (latency) než softwarová přerušení. Základní funkce hard interrupts [ref. <rtl_core.h>]:

   - *rtl_request_irq* – instaluje handler real-time přerušení
   - *rtl_free_irq* – odebere handler real-time přerušení
   - *rtl_hard_enable_irq* – povoluje přerušení
   - *rtl_hard_disable_irq* – zakazuje přerušení

Způsob jakým RTLinux zpracovává přerušení umožňuje jádru Linuxu zakazovat a povolovat přerušení a přitom dodatečně zpracovat nevyřízená přerušení. Real-time jádro přijímá všechna hardwarová přerušení a posílá je jádru Linuxu jako softwarová přerušení. Pokud jádro Linuxu zakázalo přerušení voláním *cli (clear interrupt bit)*, real-time jádro vytváří bitovou masku vzniklých přerušení. Po povolení přerušení voláním *sti (set interrupt bit)* jádro Linuxu čte masku a popřípadě zpracuje vzniklá přerušení.

V kapitole 5.4. je uvedena část kódu real-time řídicího programu pro RTLinux.

3. PCI (peripheral computer interface)

Pro využití PCI karty v systému nebo pro vývoj ovladače zařízení je nutná velice dobrá znalost použitého periferního zařízení a PCI architektury, která toto zařízení zpřístupňuje.

PCI je rozhraní standardizující komunikaci mezi periferními zařízeními a počítačem. Je nástupcem jednoduššího standardu ISA. PCI architektura má oproti ISA standardu tři hlavní výhody:

- výkonnější přesuny dat mezi počítačem a jeho periferními zařízeními
- vysoká platformová nezávislost
- jednoduché připojování a odpojování periferií ze systému

PCI dosahuje většího výkonu díky vyšší taktovací frekvenci než ISA a podporuje 32bitovou datovou sběrnici. Zařízení jsou automaticky konfigurována při startu systému. K úspěšné inicializaci ovladače pak pouze stačí, když jsou konfigurační registry driveru volně přístupné.

Každé PCI zařízení připojené k PC je identifikováno třemi čísly:

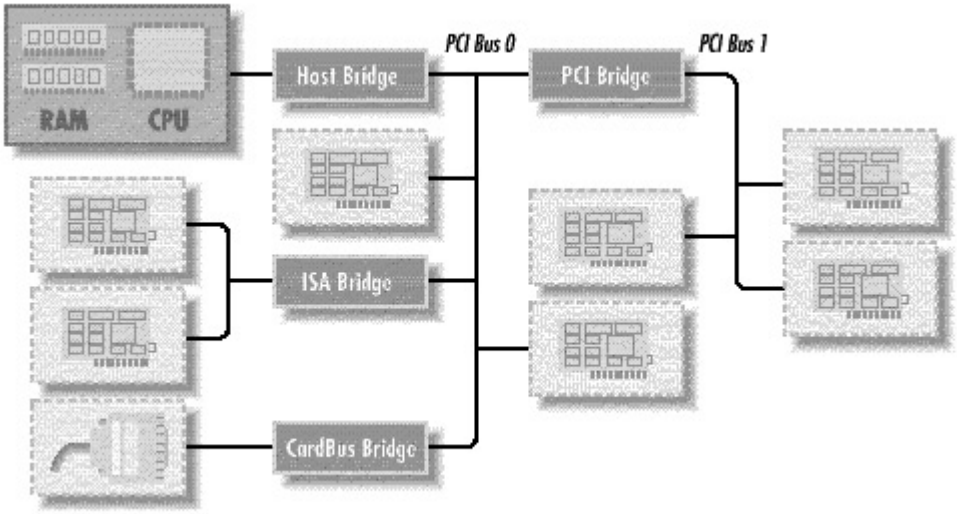
- *bus* number - číslo sběrnice (maximálně 256 sběrnic)
- *device* number – číslo zařízení (maximálně 32 hostů pro každý *bus*)
- *function* number – číslo funkce (každé zařízení má max. 8 funkcí)

Multifunkční zařízení může mít maximálně 8 funkcí. Připojování více než jedné sběrnice do jednoho systému je realizováno pomocí *mostu* (*bridge*), jehož účelem je propojovat dvě sběrnice. Celková struktura PCI systému je organizovaná jako strom, kde každá sběrnice je připojena ke sběrnici ve vyšší vrstvě. Sběrnice na počítačové kartě je také připojena do PCI systému přes bridge. Na obrázku 3.1. je naznačena typická struktura PCI systému.

V */proc/pci* a */proc/bus/pci* jsou informace o nastavení hardwarových adres. Základní adresní prostory periferních zařízení jsou adresové prostory paměti, I/O porty a konfigurační registry. První dva adresové prostory jsou sdílené všemi zařízeními na PCI sběrnici. Konfiguračními transakcemi lze naopak adresovat pouze jeden slot v daném okamžiku.

Každý PCI slot má 4 piny pro přerušení. I/O prostor používá 32-bitovou adresu sběrnice a paměť může být přístupná 32-bitovou nebo 64-bitovou adresou. Adresa musí být jedinečná v rámci každého zařízení a je na driveru (inicializaci), aby se vyvaroval mapování paměti dvou různých zařízení na stejnou adresu. Úkolem firmwaru, který provádí inicializaci při startu systému, je, aby mapoval každý region na jinou adresu. Příslušné hodnoty adres jsou zapsány do konfiguračního prostoru a ovladač zařízení je využívá k bezpečnému přístupu k hardwaru.

Konfigurační prostor, který je standardizovaný, se skládá z 256 bytů pro každou funkci zařízení. 4 byty jsou vždy obsazeny pro unikátní ID funkce. Informace z každého zařízení lze získat jako z I/O registrů.

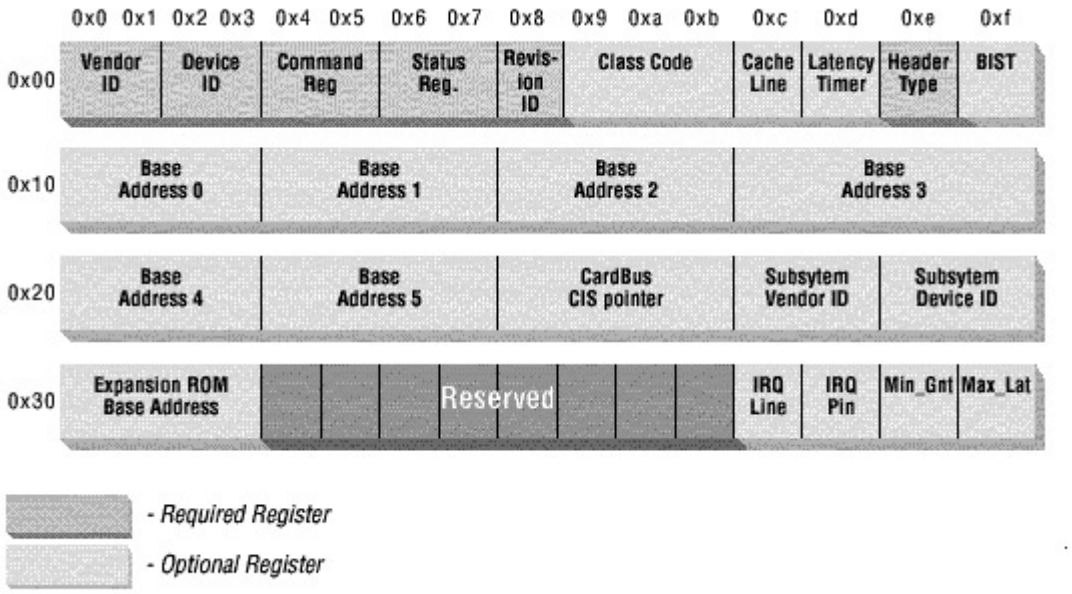


Obr. 3.1: Typická struktura PCI systému

3.1. Konfigurace PCI

Při startu systému nemá PCI mapovanou žádnou paměť ani žádné I/O porty v adresovém prostoru počítače a všechna přerušení jsou zakázána. Každá základní deska je vybavena firmwarem typu BIOS. Ten poskytuje přístup do konfiguračního adresového prostoru čtením a zápisem do registrů zařízení. Při inicializaci (bootování) firmware provádí konfigurační transakce s každým PCI zařízením, aby bez konfliktu alokoval bezpečný prostor pro každý region. Od této doby lze přistupovat k zařízení, které již má přiřazené I/O regiony do adresového prostoru procesoru. Driver tato nastavení může změnit, ale to není v žádném případě třeba. Konfigurační registry jsou zobrazeny v textové formě v */proc/bus/pci/devices* .

Všechna zřízení mají stejnou strukturu konfiguračního prostoru. Prvních 64 bytů je standardizovaných a zbytek je specifický pro každé zařízení viz. obrázek 3.2.



Obr. 3.2: Struktura konfiguračních registrů PCI zařízení

Tři základní čísla jsou vždy použita. Jsou to read-only registry a je možné je použít při hledání zařízení[*]:

- *vendorID* – 16-bitové číslo identifikující výrobce hardwaru.
- *deviceID* – 16 bitový registr, přiřazené výrobcem
- *class* – každé periferní zařízení přísluší do nějaké třídy.

[*] Tato čísla se liší od čísel *bus, device a function* v tom, že jsou zadávána jednou provždy výrobcem hardwaru a jsou unikátní v rámci všech vyráběných zařízení. Kdežto čísla *bus, device a function* identifikují, jakým způsobem je zařízení připojeno do systému.

3.2. Linuxové API pro práci s PCI kartou

Při programování PCI karty potřebuje program vědět, zda je programová PCI podpora dostupná z modulu v jádře systému. Proto je třeba zahrnout hlavičkový soubor *<linux/config.h>*, pomocí něhož ovladač získá přístup k makru *CONFIG_* nebo *CONFIG_PCI*. Není-li toto makro definované, každé volání PCI funkce vrací chybu. Přehled základních funkcí pro práci s PCI zařízením v linuxu [ref. *<linux/pci.h>*]:

- *pci_present* – zjišťuje, zda je přítomná PCI podpora a kontroluje přítomnost nějakého PCI zařízení
- *struct pci_dev* – datová struktura reprezentující PCI zařízení
- *pci_find_device* – prohlíží seznam instalovaných zařízení
- *pci_find_class* – podobná předchozí funkci, ale hledá zařízení náležející příslušné třídě
- *pci_enable_device* – probouzí zařízení. Nutné před každým přístupem k I/O regionům
- *pci_find_slot* – vrací strukturu PCI zařízení podle dvojice sběrnice/zařízení

Po úspěšné detekci zařízení je potřeba číst nebo zapisovat do konfiguračních registrů. Jedná se o adresové prostory paměti, portů a konfiguračních registrů.

Ke konfiguračním registrům nelze přistupovat přímo. Linux k jejich přístupu poskytuje 8-bitové, 16-bitové a 32-bitové funkce [ref. <linux/pci.h>]:

- *pci_read_config_byte, pci_write_config_byte*
- *pci_read_config_word, pci_write_config_word*
- *pci_read_config_dword, pci_write_config_dword*

Ty čtou/zapisují byty z/do konfiguračního prostoru dané struktury *pci_dev* se zadaným offsetem. V *<linux/pci.h>* jsou definované konstanty, které usnadňují zadávání offsetu v konfiguračním registru. Prohlížet konfigurační prostor je také možné v */proc/bus/pci*[*].

Při práci s PCI zařízením je nutné přistupovat k I/O regionům dané karty. To jsou buď I/O prostory nebo paměť na zařízení. PCI rozhraní implementuje až 6 adresových I/O regionů. Ne všechna zařízení mapují své registry na I/O porty sběrnice. Většina zařízení mapují své I/O registry do paměťových prostor z důvodu lepšího využití CPU v přístupu do paměti.

[*] Alternativou je příkaz :
dd bs=256 skip=$PORAD_CISLO count=1 if=/proc/pcidata | ./pcidump
Kde pořadové číslo je stejné jako je pořadí zařízení v */proc/bus/pci/devices*. *pcidata* a *pcidump* je součástí balíku *pciutils*.

Zařízení informuje o svých regionech v 32-bitových konfiguračních registrech *PCI_BASE_ADDRESS_0* až *PCI_BASE_ADDRESS_5* (obr. 3.2). Linux poskytuje rozhraní pro čtení těchto registrů:

- *pci_resource_start* – vrací první počáteční příslušející jednomu ze šesti regionů
- *pci_resource_end* – vrací poslední adresu části I/O regionu
- *pci_resource_flags* – vrací atributy příslušející zdroji [ref. <linux/ioport.h>]

V mnoha případech je nutné zpracovávat přerušení generované zařízením. K tomu je třeba znát číslo přerušení, které je uloženo v konfiguračním registru *PCI_INTERRUPT_LINE* (1byte) (obr. 3.2). Při startu systému, tzv. firmware prohlíží PCI zařízení a nastavuje tento registr každého zařízení podle toho, jak je připojen pin přerušení na PCI slot. Takže hodnota čísla přerušení je nastavena firmwarem, protože jedině ten ví, jak základní deska připojuje různé piny přerušení k procesoru. Ovladač zařízení tento registr využívá jen pro čtení.

V kapitole 5.3 je uveden jednoduchý příklad jak lze využít služby připojené PCI karty.

4. Distribuované systémy

V současné době se v různých odvětvích používá celá řada informačních a výpočetních systémů s distribuovanou architekturou. Předávání údajů ve formě dat a z nich vytváření informací tvoří základ měřících a řídicích systémů. Tyto většinou lokální systémy uživatelé potřebují implementovat po stránce informačního toku do takzvaných otevřených systémů, jejichž součástí jsou většinou systémy více typů z hlediska funkce, stupně řízení, způsobu přístupu i různé komunikační filosofie.

Za *distribuovaný systém* lze považovat množinu autonomních zpracovatelských prvků, které jsou propojeny počítačovou sítí a které spolupracují při provádění jim přiřazených úloh. Prvkem se rozumí počítač, který samostatně řeší požadovanou úlohu.

Při návrhu systému je vždy třeba zvážit výhody a nevýhody distribuovaných systémů a při implementaci případně zahrnout jistá omezení.

Mezi nevýhody lze zařadit potřebu složitější komunikace po propojené síti, problém integrity a ochrany dat a použití různorodých technických a programových prostředků.

Výhody jsou specifické pro každý distribuovaný systém. Obecně lze však zmínit například zkrácení odezvy systému, otevřenost systému, možnost sdílení zdrojů a periferií, vyšší spolehlivost a provozuschopnost nebo přímá účast koncových uživatelů.

Důležitým atributem činnosti celého systému je způsob výměny údajů nebo dat mezi jednotlivými procesory. To v podstatě specifikuje těsnost vazby procesorů. Nejtěsnější vazba se vyskytuje u systémů, které sdílejí společnou paměť. Naopak nejvolnější vazbu představují například systémy LAN (Local area network), jejichž architektura je založena na výměně zpráv. Jednotlivé stanice pomocí komunikačních procesorů předávají a přebírají zprávy v dvoubodových nebo vícebodových komunikačních kanálech.

Z hlediska propojení jednotlivých stanic na síti se používají tyto základní struktury topologie propojení sítí:

- sběrnicová – nejvíce se používá pro měřicí a řídicí systémy i lokální sítě
- hvězdnicová – modifikace předchozí, ale stanice jsou propojeny přímo a spojovací cesta od každé stanice je vedena do centrálního bloku
- kruhová – založena na dvoubodovém spojení sousedních stanic
- stromová – vznikla rozšířením sběrnicové struktury

4.1. Komunikační protokoly

Pro splnění podmínek pro vzájemnou komunikaci dvou stanic přes obecnou propojovací síť potřebujeme určit pravidla komunikace. Doporučení pro vzájemnou komunikaci mezi stanicemi až do úrovně specifikace přenosu dat nebo programů vydala organizace

International Standard Organization (ISO) a označila jako *Open System Interconnection* (ISO/OSI). Protokol se člení do sedmi vrstev (viz. obr. 4.1).

Fyzická vrstva

Definuje přenosové medium (optické vlákno, koaxiální kabel, kroucená dvoulinka, …)

Linková vrstva

Určuje způsob předávání dat zpráv v síti. Zprávy jsou přenášeny ve formě dat v pevně definovaných rámcích. Data musí být chráněna proti poruchám, proto se zajišťují detekčními cyklickými kódy. Zprávy většinou nelze přenést v jednom rámci (přenášeném bloku), proto se přenášená data rozdělí do několika částí kterým se říká *pakety*.

Paket je část přenášeného datového souboru, kterou lze přenést najednou. Doplněním paketu o záhlaví, adresaci, zabezpečení cyklickým kódem, případně dalšími informačními znaky se získá rámec přenosového protokolu.

Síťová vrstva

Definuje způsob, jakým se pakety pohybují v síti. U mnohobodové decentralizované sítě musíme určit způsob předávání rámce řetězem stanic umístěných mezi koncovými uživateli. Těmto stanicím, které propojují jednotlivé linky, se říká propojovací uzly. Podle způsobu, jakým propojovací síť zajišťuje přenos dat, se rozlišují sítě pracující na principu:

- *Přepojování kanálů* – Obdoba telefonní sítě, kdy linky sítě jsou vyhrazeny pro spojení koncových účastníků.
- *Přepojování zpráv* – Obdoba poštovního styku, kdy každý dopis má svého adresáta, případně odesílatele
- *Přepojování paketů* – Data přenášená mezi koncovými účastníky jsou rozdělena do paketů a přenášena postupně mezi koncovými uzly komunikačního spoje. Struktura přenášených rámců je schopna přizpůsobovat se topologii a vytížení sítě.

Transportní vrstva

Definuje adresaci počítačů a aplikačních programů v síti, zajišťuje vytváření dočasných komunikačních spojení mezi nimi a rozklad zpráv do paketů a jejich opětné složení na přijímací straně.

Relační vrstva

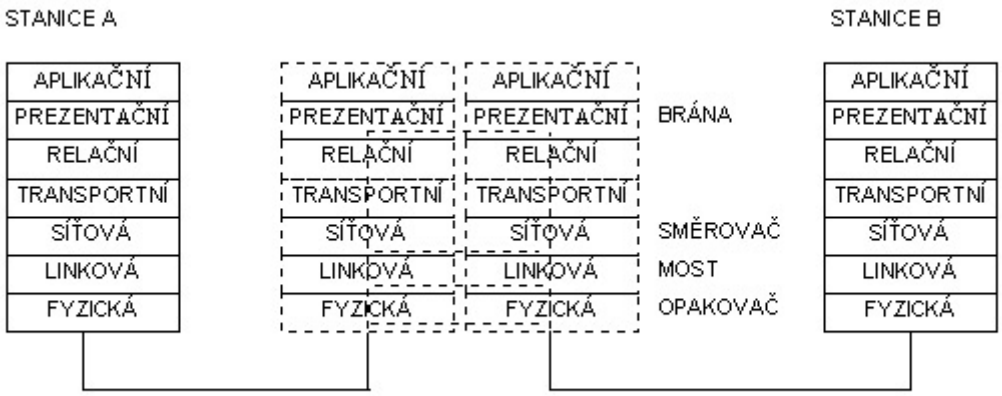
Vytváří logické rozhraní pro aplikační programy, které používají služeb sítě. Identifikuje uživatele, ověřuje přístupová práva, eviduje provoz a v případě komerčních sítí stanovuje a účtuje poplatky za provoz na síti.

Prezentační vrstva

Transformuje data do formy vhodné pro přenos, provádí převody kódů a formátů dat pro nekompatibilní počítače, kompresi dat pro lepší využití komunikačního kanálu, případně šifrování dat pro požadované utajování.

Aplikační vrstva

Je oblastí aplikačních programů, které se v síti využívají.



Obr. 4.1: Síťový protokol OSI

4.2. Komunikace mezi lokálními sítěmi

Lokální sítě jsou vymezeny jedním nebo skupinou uživatelů sítě, kteří se nalézají v budově nebo komplexu budov. Přenosový výkon sítě je omezen kapacitou media, počtem připojených stanic a délkou nejvzdálenější stanice. To jsou tři základní omezení pro funkci sítě.

Při vyšších nárocích nezbývá než stanice zapojit do menších sítí, které lze mezi sebou propojit. Lokální sítě se propojují pomocí speciálních stanic *mostů* (*bridges*) a *směrovačů* (*routers*), připojených ke dvěma nebo více propojovaným sítím. Pro označení několika propojených lokálních sítí se používá výraz *internetwork*. Mosty a směrovače se od sebe liší rozsahem, kterým zasahují do přenosových zpráv.

Most (bridge)

Je to nejjednodušší modul určený pro vzájemné propojování lokálních sítí. Podmínkou pro propojení dvou lokálních sítí je jejich identický protokol. Mosty se opírají pouze o adresační pole přenášeného rámce. Působí tedy pouze jako spojovací prvek mezi dvěma sítěmi na úrovni *fyzické* a *linkové* vrstvy.

Mosty přijímají všechny rámce ze všech připojených sítí a analyzují adresu příjemce. V případě že je zpráva určená do vedlejší sítě, most provede odeslání.

Směrovač (router)

Pro složitější propojování sítí už mosty nestačí plnit funkci bezproblémového propojení sítí. Směrovače analyzují přenášený protokol do síťové vrstvy. Typický příklad takové sítě je síť *Internet*. Filozofie Internetu je založena na využití stávajících lokálních sítí s různými protokoly. Síť Internet prakticky propojuje směrovače jednotlivých sítí. Propojené směrovače tvoří nadstavbovou síť nad lokálními sítěmi. Přenášený rámec se zabaluje do nového paketu a na začátek je přidána patřičná hlavička.

Brána (gateway)

Při propojení dvou sítí s jinými operačními systémy je možno použít směrovač a síť Internet jen v případě zasílání informací, souborů dat nebo programů. Pokud je vyžadována užší spolupráce, je potřeba komunikovat se stanicí s odlišným OS v jiné lokální síti a pak je nutné propojení těchto sítí pomocí *brány*.

Programové vybavení pro bránu provádí odpovídající konverzi protokolů. V případě spojení dvou rozdílných systémů budou zpravidla obě spojované sítě používat odlišné protokoly ve všech sedmi vrstvách protokolu OSI. Samotná brána nemůže zajistit konverzi aplikační vrstvy, provádí pouze konverzi na nižších vrstvách protokolu ISO.

4.3. Internet

Propojování heterogenních sítí je velmi důležitou součástí počítačové komunikace. Internet je mezi počítačovými sítěmi určitou anomálií. Je jediným typem sítě, která nemá vlastní fyzickou vrstvu. Aby taková síť mohla být funkční, využívá fyzickou vrstvu ostatních stávajících sítí. Protože tyto sítě jsou postaveny na různých základech, s různými přenosovými protokoly, musí být protokol *TCP/IP* spojovacím článkem mezi různými typy protokolů.

Propojování a transport dat řídí soubor síťových protokolů TCP/IP. Tyto protokoly využívají pouze čtyři vrstvy* protokolu ISO viz. obrázek 4.2:

- linkovou
- síťovou
- transportní
- aplikační

Obr. 4.2: Vrstvy Internetového protokolu TCP/IP

Oproti klasickému sedmivrstvému protokolu ISO jsou v síti Internet tyto odlišnosti:

- spolehlivost sítě je problém zdroje a cíle zprávy
- detekce chyb a zotavení je součástí transportní vrstvy
- v síťové vrstvě IP je zajištěna doprava datagramů z jednoho uzlu do druhého bez ohledu na cílový počítač
- transportní vrstva zajišťuje přenos několika procesů najednou a používá virtuální adresování.
- aplikační vrstva je rozhraním mezi aplikačním programem a spolehlivým transportním protokolem TCP

Každý počítač v síti Internet je identifikován jednou, nebo více numerickými adresami délky 32 bitů, tzv. IP adresa. Brána Internetu nerozlišuje jednotlivé počítače ale celé sítě. IP adresa má proto hierarchický charakter. Z adresy počítače musí být směrovač schopen odvodit příslušnou síť, na kterou rámec směřuje.

Adresy jednotlivých počítačů pro síť musí přidělovat správce lokální sítě a nesmí být přidělovány centralizovaně (to je jedna ze zásad internetu). Jeden počítač může mít i více IP adres podle připojení do více různých sítí.

* Někdy se protokol TCP/IP označuje jako třívrstvý bez linkové vrstvy

Při komunikaci na internetu je možné využít dva základní typy protokolů.

a) *Protokol IP* zajišťuje přenos datragramů mezi koncovými účastníky, ale nezajišťuje potvrzování došlých zpráv. Spoléhá na služby sítě v nižších úrovních. Takový způsob by mohl být nedokonalý, proto jsou v Internetu služby *UDP* (User Datagram Protocol) a TCP (Transmission Control Protocol). Protokol IP tvoří základ pro vyšší přenosové protokoly a pro služební protokoly Internetu.

b) *Protokol TCP* umožňuje zabezpečený přenos dat včetně potvrzování na rozdíl od protokolu UDP. Spojení stanic pomocí TCP vytváří dva vzájemně nezávislé kanály pro oba směry spojení.

4.4. API pro tvorbu TCP/IP aplikací

Aplikace v síti Internet se opírají o filosofii spolupráce označovanou jako *klient-server*. Server je proces, který poskytuje nějakou službu neznámému (anonymnímu) klientovi. Klient o tuto službu žádá zasíláním zprávy serveru. Server provede požadovanou činnost a odpoví zprávou.

Co se týká konkrétní implementace v operačním systému UNIX (ale i v jiných operačních systémech), aplikace komunikují prostřednictvím tvz. *socketů*. Mechanismus byl pro UNIX navržen na univerzitě v Berkley a proto je označován jako *Berkley socket*.

Základním prvkem rozhraní je *socket*, který dovoluje jednomu aplikačnímu programu spojit se prostřednictvím sítě s jiným aplikačním programem. Po jeho otevření používají aplikační programy stejná volání jako pro práci se soubory a periferiemi. Tedy pro odesílání je to *write* a pro čtení *read*. Do programu klienta i serveru je třeba začlenit hlavičkové soubory <sys/socket.h> <sys/types.h>.

Socket na straně serveru

Vytváření serverové aplikace se skládá z několika kroků (příklad viz. kapitola 5.5.):

a) Nejdříve je třeba vytvořit socket, který reprezentuje proces na serveru. To zajistí systémové volání *socket*. Po jeho vytvoření nemůže být využit jinými procesy.
b) Síťový socket je přiřazen na příslušný port, který společně s internetovou adresou přesně identifikuje službu na Internetu, kterou může klient žádat . Vazba socketu na lokální adresu provede systémové volání *bind*.
c) Proces čeká, až se na socket připojí nějaký klient. Systémové volání *listen* vytvoří frontu přicházejících připojení.
d) Nyní je již možné přijímat klienty pomocí volání *accept*. Po tomto volání se vytvoří nový socket, který je odlišný od původního na příslušném portu serverové služby.

Nový socket je využíván pouze pro komunikaci s tímto příslušným klientem. Původní socket je ponechán pro další připojení s dalšími klienty.

e) Socket se uzavírá voláním *close*, avšak v mnoha případech je spojení ukončeno ze strany klienta.

Socket na straně klienta

Klientská část postavená na architektuře socketu je o něco jednodušší. Následující kroky jsou velmi podobné pro většinu klientských internetových aplikací (příklad klienta je uveden v kapitole 5.6.):

a) Voláním *socket* se vytváří nový socket, kterému nepřiřazujeme žádné číslo portu.
b) Je možné volat *connect* s příslušnou adresou a portem požadované serverové služby.
c) Pokud již klient nepotřebuje se serverem kumunikovat, uzavírá spojení voláním *close*.

Komunikace mezi serverem a klientem může probíhat dvěma způsoby:

a) S*ynchronní* výměna zpráv při níž musí vysílač počkat do okamžiku, kdy je k převzetí zprávy připraven i přijímač. Samozřejmě pokud je přijímač k převzetí zprávy připraven dříve, musí počkat na vysílač.
b) *Asynchronní* výměna zpráv umožňuje vysílači pokračovat ve výpočtu, zatímco zpráva je dočasně uložena komunikačním kanálem a předána přijímači až v okamžiku, kdy je k převzetí zprávy připraven. Přijímač však musí na zprávu v kanále počkat.

5. Návrh distribuovaného řídicího systému s manipulátorem Oscar

Laboratorní manipulátor Oscar je modelem skutečného průmyslového manipulátoru používaného v různých podobách v řadě výrobních procesů. Navržený systém může sloužit jako školní pomůcka při výkladu řízení časově kritických událostí a distribuovaného řízení.

Distribuovaný systém má umožnit více klientům vzdáleně kontrolovat pohybový proces manipulátoru. Pojmem vzdáleně (remote) se myslí možnost využití libovolného počítače připojeného k síti (zde Internet), ke kterému robot není přímo připojen. Počet účastníků komunikace není omezen.

5.1. Rozdělení systému na jednotlivé uzly

Protože řízení pohybu ramen manipulátoru je řízení časově kritické úlohy, je nutné implementovat řídicí část systému na OS reálného času. Manipulátor Oskar je proto připojen k počítači s operačním systémem RTLinux.

Nejtěsnější vazbou je s řízeným procesem spojen řídicí modul v jádře RTLinuxu. Jeho úkolem je generovat přesné pulsy podle zadané periody pro krokové motory manipulátoru a hlídat krajní polohy pohybu všech jeho ramen. Komunikace probíhá prostřednictvím PCI karty PLX 1750. Z pohledu modulu se jedná o zapisování/čtení řídicích/stavových bytů na porty nebo z portů. Bity jednotlivých bytů reprezentují signály pro krokové motory a při čtení z portů signalizují stavy koncových spínačů. O samotné generování signálů pro vinutí krokových motorů se stará elektronika manipulátoru.

Dalším uzlem systému je server TCP spojení. Server je přítomný na stejném počítači jako řídicí modul. Běží v uživatelském prostoru jako jeden z procesů Linuxu.

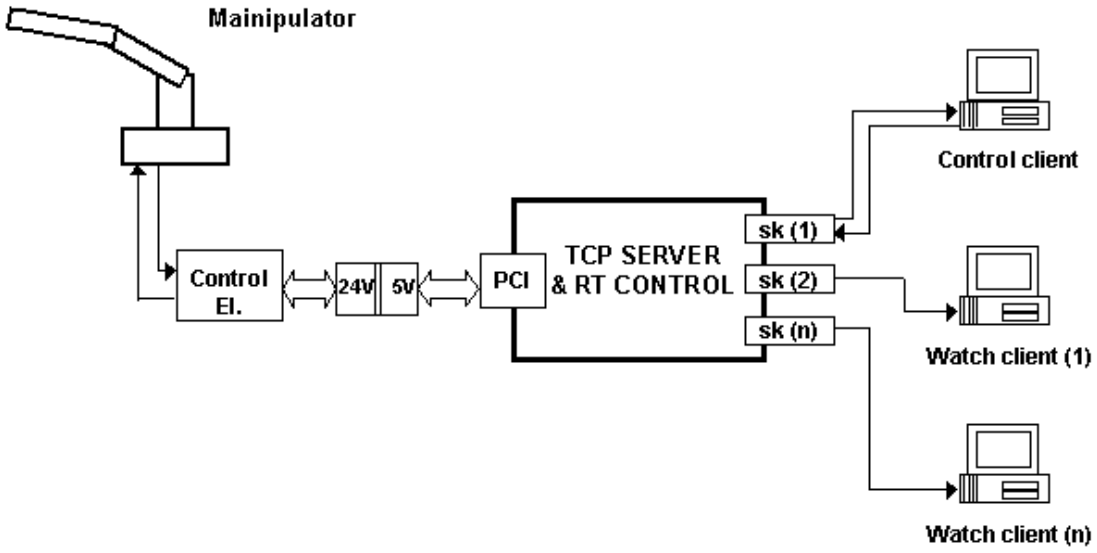
Protokol TCP byl zvolen jako standard v oblasti internetu a navíc Linux poskytuje rozhraní pro vývoj aplikací podporující architekturu socketu. Jak bylo popsáno v dřívějších kapitolách, tato architektura umožňuje vývoj aplikací typu klient/server. TCP server má koordinovat činnost všech vzdálených klientů systému. Slouží také jako zprostředkovatel předávající řídicímu modulu požadavky na cílovou polohu ramen manipulátoru a rozesílá všem klientům informace o stavu řízeného procesu.

Vazba mezi modulem v jádře a TCP serverem je volnější než mezi manipulátorem a řídicím modulem. Komunikace probíhá posíláním zpráv na jedné straně komunikačního kanálu a přijímáním zpráv na straně druhé. Z pohledu návrhu softwaru se jedná o postup podobný zapisování a čtení do souboru. Aby bylo možné rozumně řídit manipulátor, je třeba

zajistit výlučné přijímání požadavků od klientů. To znamená, že je povoleno přijímat požadavky na řízení pouze od jediného klienta. Je úkolem serveru takto koordinovat stavy a činnost klientů.

V koncových uzlech systému se nacházejí výpočetní centra TCP spojení označovaná v použité architektuře jako klienti. Počet klientů je ve spojení se serverem neomezen. Je však úkolem TCP serveru zajistit koordinaci jejich činnosti při řízení manipulátoru. Vazba mezi oběma stranami TCP spojení je nejvolnější ze všech. Komunikace probíhá za pomocí zasílání zpráv mezi serverem a klientem přes prostředníka komunikace síť Internet. Protože je to různorodá síť, není předem zřejmé, jaká bude cílová platforma použitého stroje. Nejvhodnější vývojový nástroj aplikace klient v těchto a podobných internetových aplikacích je bezesporu jazyk Java.

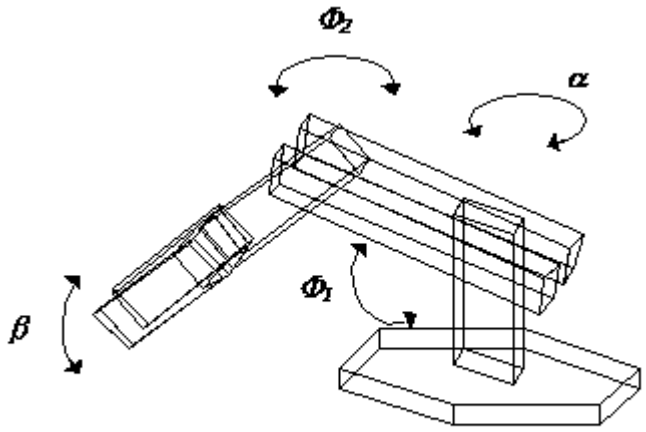
Na obrázku 5.1 je naznačena struktura navrhovaného distribuovaného systému řízení laboratorního manipulátoru Oskar.



Obr. 5.1: Blokové schéma distribuovaného řídicího systému s manipulátorem Oskar

5.2. Laboratorní manipulátor Oscar

Manipulátor se skládá ze dvou rotačních ramen otočných o úhly ϕ_1 a ϕ_2, z jednoho koncového chapadla svírajícího úhel β a ze základny otáčivé kolem vlastní osy o úhel α viz. obrázek 5.2.

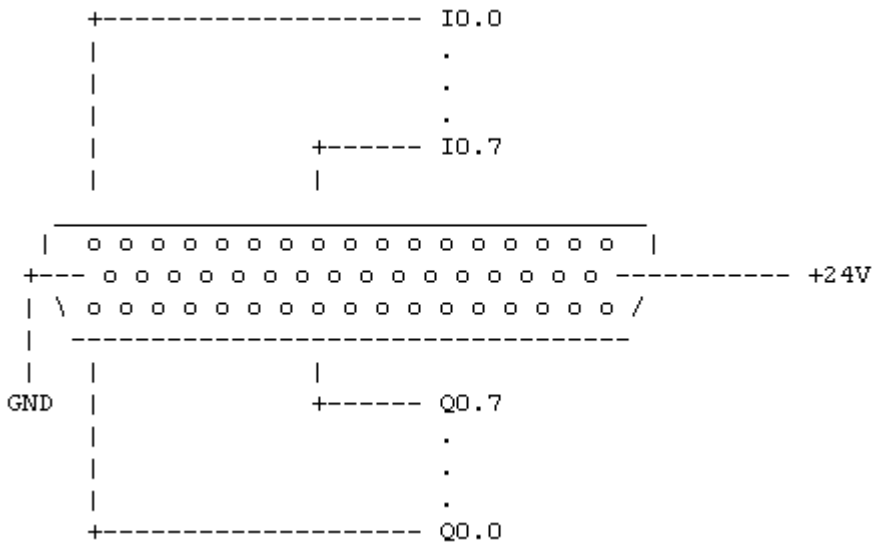


Obr. 5.2: Použitý manipulátoru Oscar s vyznačením rotací jednotlivých kloubů

Pohony kloubů jsou krokové motory napájené 12V. Signály pro vinutí motorů jsou generovány vnitřní elektronikou manipulátoru na základě pulsního signálu generovaného programem. Pohyb vpřed a vzad každého motoru je zajištěn přivedením impulsů na jeden ze dvou bitů připadajících každému krokovému motoru. Tyto bity jsou vstupy reverzibilního čítače, který řídí rozdělování impulsů do jednotlivých vinutí krokových motorů. Bit, na kterém není pulsní signál, musí být v logické jedničce, jinak se motor netočí. Význam bitů konektoru manipulátoru je naznačen na obrázku 5.3 a v tabulce 5.1.

Výstupy z PC			Vstupy do PC		
Negace	**Funkce**	**Výstup**	**Negace**	**Funkce**	**Vstup**
-	základna směr vpravo	Q0.0	-	Konc. snímač chapadla	I0.0
-	základna směr vpravo	Q0.1	-	Konc. snímač vedl. ramena	I0.1
-	hlavní rameno nahoru	Q0.2	-	Konc. snímač hl. ramena	I0.2
-	hlavní rameno dolů	Q0.3	-	Konc. snímač základny	I0.3
-	vedlejší rameno dolů	Q0.4			
-	vedlejší rameno nahoru	Q0.5			
-	chapadlo otevřít	Q0.6			
-	chapadlo zavřít	Q0.7			

Tab. 5.1: Význam bitů konektoru robota



Obr. 5.3: Konektor robota (pohled zepředu)

Koncová poloha každého ramene je detekována koncovým snímačem. Druhou krajní mez je třeba hlídat počítáním generovaných pulsů pro krokové motory. Nejdříve je však nutné tuto mez experimentálně zjistit a uložit jí jako konstantu parametrů robota.

Řídicí elektronika Oskara je napájena 24V a tedy i úrovně vstupních a výstupních signálů jsou v logické jedničce 24V[*]. Manipulátor má komunikovat s PC připojením na PCI laboratorní kartu PLX 1750, proto jsem musel zajistit úpravu úrovní signálů z 5V na 24V směrem z PC a 24V na 5V směrem do PC.

K úpravě signálů z PC jsem použil zapojení s obvodem L272. L272 je monolitický integrovaný obvod určený k použití jako výkonový operační zesilovač s maximálním napájecím napětím 28V. Pro převod z 5V na 24V je zapojen jako komparátor s komparační úrovní 2.5V.

Signál do PC jsem přizpůsobil využitím obvodu ULN2802A, který obsahuje 8 darlingtonových zesilovačů v jednom pouzdře. Na výstupu je tranzistor s otevřeným kolektorem a na vstup je možné přivést signál o hodnotách 14V-25V.

5.3. PCI laboratorní karta PLX 1750

Ne v každém případě, kdy se v programu využívá nějaké připojené zařízení (PCI karta), je nutné vytvářet ovladač zařízení.

Ovladač zařízení (device driver) je speciální část linuxového jádra. Je to černá skříňka, která převádí specifické chování hardwaru na přesně známé a definované programátorské rozhraní. Zcela zakrývá detaily skutečné práce zařízení. Uživatelské aktivity jsou prováděny množinou standardizovaných volání, nezávislých na specifikovaném ovladači. Úkol ovladače je mapovat tyto funkce na určené operace toho kterého zařízení.

V mém programu využívám pouze služby čtení a zápisu bytů z příslušných regionů PCI karty. Není ani potřeba provádět registraci zařízení tak, aby mohla být karta přístupná z uživatelského prostoru. To mi umožnilo ‘obejít’ vývoj ovladače zařízení pro kartu PLX 1750 a problémy s tím spojené. Přesto bylo nutné využít aplikační rozhraní Linuxu pro práci s PCI zařízením.

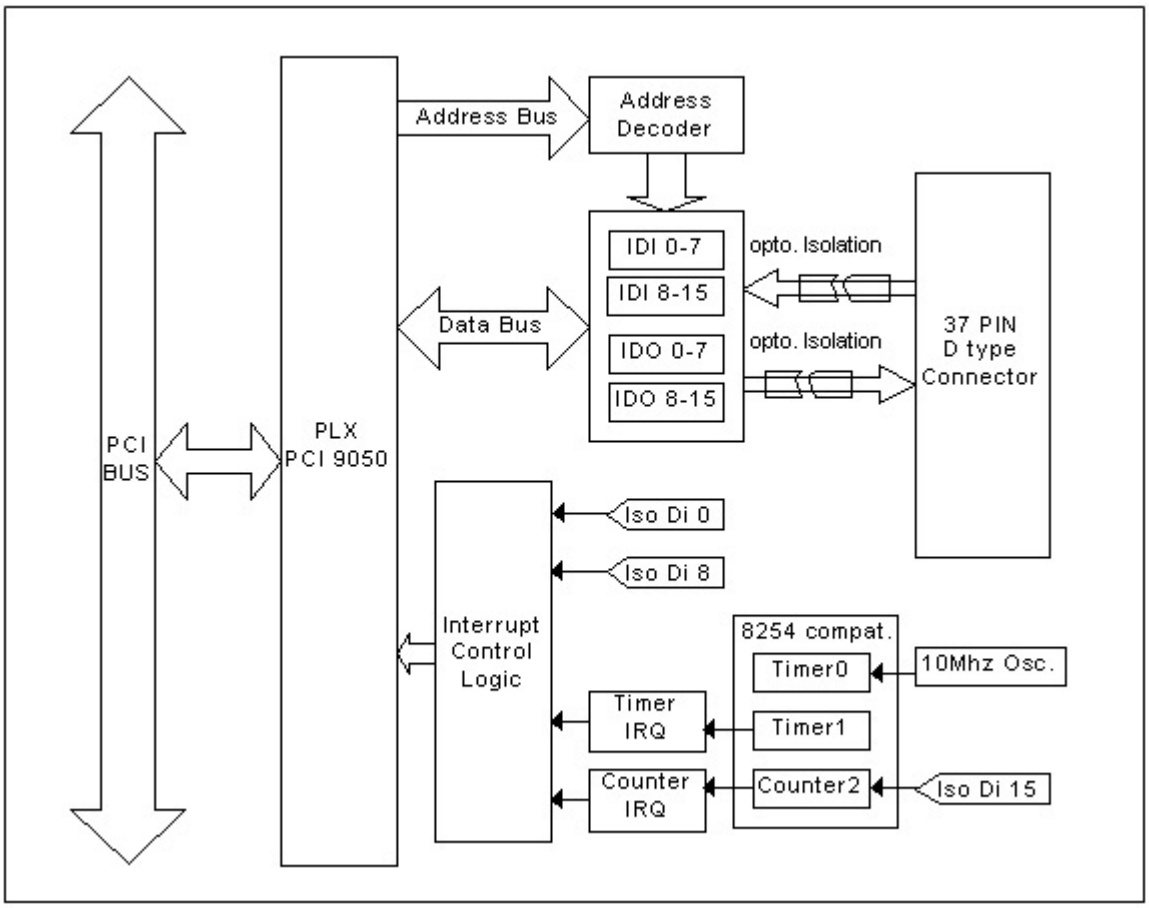
PCI karta PLX-1750 poskytuje 16 izolovaných digitálních vstupních kanálů, 16 izolovaných digitálních výstupních kanálů, jeden izolovaný čítač a jeden časovač. Izolační ochrana je 2500 V_{DC} a karta je tedy ideální pro průmyslové aplikace s vysokonapěťovou ochranou.

[*] Manipulátor je navržen pro komunikaci s PLC. Proto je úroveň signálů v logické jedničce 24V.

PLX 1750 také poskytuje možnost duálního zpracování přerušení, umožňující větší flexibilitu v používání čítače, časovače, digitálních vstupů, nebo kombinaci pro generování přerušení pro PC.

PLX 1750 používá PCI řadič pro připojení karty k PCI sběrnici. Tento řadič plně implementuje specifikaci PCI sběrnice. Všechny konfigurace související se sběrnicí (jako např. nastavování bázové adresy a přiřazení přerušení) jsou automaticky softwarově řízeny. Pro konfiguraci nejsou potřeba žádné „jumpery“ ani „DIP switche“.

Na obrázku 5.4 je zobrazeno blokové schéma laboratorní PCI karty PLX 1750. Formát registrů je v tabulce 5.3.



Obr. 5.4: Blokové schéma PCI laboratorní karty PLX 1750

Base Address + (Decimal)	Function	
	Read	Write
0	IDI 0-7	IDO 0-7
1	IDI 8-15	IDO 8-15
2 – 23	Reserve	Reserve
24	8254 Counter 0	8254 Counter 0
25	8254 Counter 1	8254 Counter 1
26	8254 Counter 2	8254 Counter 2
27		8254 Control Register
28	Reserved	Reserved
29	Reserved	Reserved
30	Reserved	Reserved
31	Reserved	Reserved
32	Interrupt Status Register	Interrupt Control Register

Tab. 5.3: Formát registrů karty - PLX 1750

Použití krokových motorů jako pohonů jednotlivých ramen umožňuje řízení pohybu bez nutnosti generovat a zpracovávat přerušení. Zjišťování polohy totiž může probíhat synchronně s generováním řídicích pulsů. Proto také odpadá nutnost použít generování přerušení na PCI kartě při dosažení koncových spínačů. Služby čtení (read) a zápis (write) karty PLX 1750 provádějí přesuny jednoho bytu dat, jehož bity mají význam řídicích pulsů jednotlivých krokových motorů.

Následující výpis z kódu zobrazuje postup, jakým využívám služeb PCI karty PLX 1750 z prostředí jádra Linuxu:

1. Nejdříve je třeba začlenit nezbytné hlavičkové soubory umožňující využít podporu systému pro práci s PCI kartou.

```
#include <sys/pci.h>
#include <linux/errno.h>
```

2. Za pomocí systémového rozhraní je třeba nalézt příslušné zařízení.

```
int plx1750_find_device(  )
{
   if ( pcibios_present() ) {
      if(( dev = pci_find_device( PCI_VENDOR_ID_PLX,
           PCI_DEVICE_ID_PLX_1750, dev )) != NULL ) {
         rtl_printf("PLX 1750 : Device Found : %s \n", dev->name);
```

3. Rezervace I/O regionů.

```
         if( check_mem_region( dev->resource[2].start, 2 )) {
            printk( "PLX 1750 ERROR : Memory allready in use\n" );
            return -EBUSY;
          }
         request_mem_region( dev->resource[2].start, 2, MOD_NAME );
         base_addres = dev->resource[2].start;
         return 1;
      }
   }
  return 0;
}
```

4. Jednoduchým voláním je možné zapisovat 1B na kartu (podobné volání je i pro čtení z karty).

```
void dev_write( unsigned char c )
{
   outb( c, base_addres );
}
```

5. Při ukončení práce s kartou je nutné uvolnit přidělený I/O region.

```
int plx1750_close_device(  )
{
   release_mem_region( dev->resource[2].start, 2 )
}
```

V uvedeném kódu je třeba nejprve nalézt PCI kartu PLX1750 resp. strukturu *pci_dev*, která v Linuxu reprezentuje PCI zařízení. Zařízení umožňuje nalézt volání funkce *pci_find_device* s parametry jednoznačnými pro každé připojené zařízení, tedy s čísly *DEVICE_ID* a *VENDOR_ID*. Některé konstanty pro zařízení se nacházejí v hlavičkovém souboru <pci_ids.h>. Pro kartu PLX1750 jsem však nalezl pouze číslo výrobce. Druhý parametr může být *PCI_ANY_ID*, pokud je zřejmé, že se na daném počítači vyskytuje pouze jediná karta od tohoto výrobce. Druhou možností je postupně v cyklu procházet všechny karty a vybrat si tu správnou, a nebo nalézt přesné číslo zařízení. Zde uvádím konkrétní příklad vyhledávání čísla karty PLX 1750 na mém počítači:

1. *lspci | cut -d: -f1-3*
2. *cat /proc/bus/pci/devices | cut -f2*

V prvním kroku jsem vypsal všechna zařízení a podle pořadí řádku hledané karty jsem našel čísla ve stejném řádku druhého volání.

Z tabulky 5.3. je možné zjistit, že vstupní a výstupní porty se nacházejí na adrese *BASE_ADDRESS + 0* až *BASE_ADDRESS + 1*. K portům je možné přistupovat pomocí pole *resources* struktury *pci_dev_t*. Adresa portů, které využívám se nachází ve třetím prvku pole *resources*.

Po úspěšném nalezení karty zkontroluji, zda již porty nejsou rezervovány někým jiným. Pokud nejsou, žádám o jejich přidělení. Pak již stačí jen implementovat funkce čtení a zápisu 1 bytu na příslušné porty.

Po ukončení práce s kartou je třeba uvolnit přiřazené paměťové prostory voláním *release_mem_region*.

5.4. Řídicí modul RTLinuxu

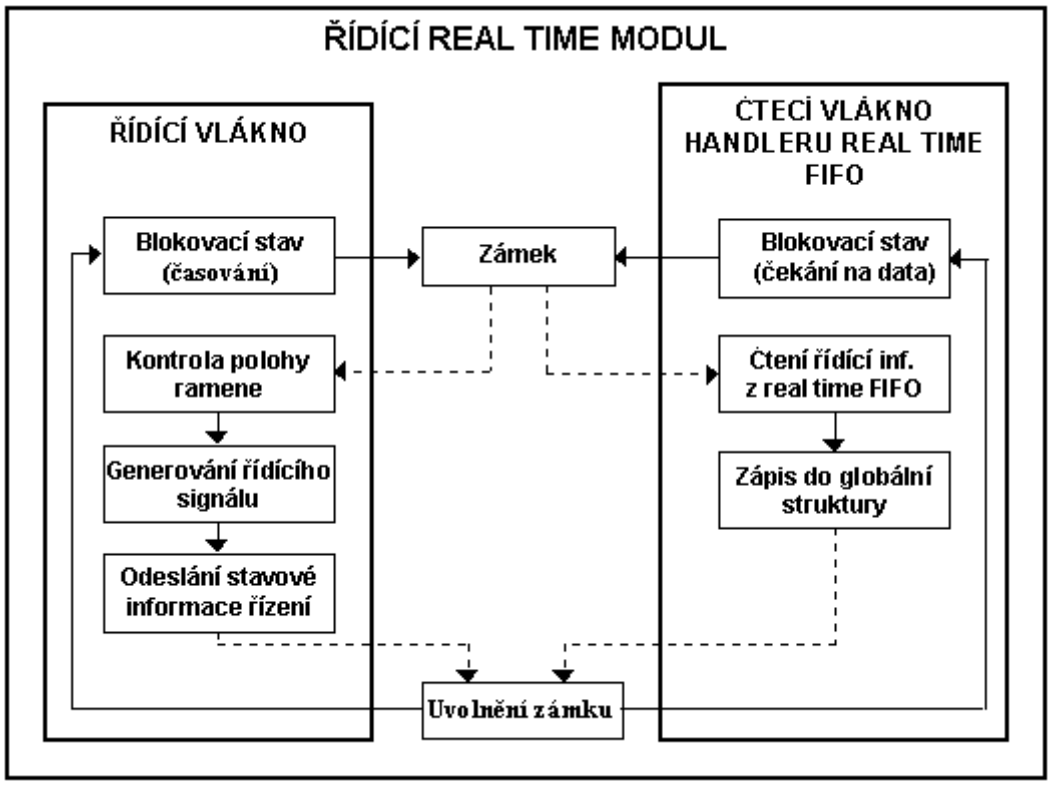
Řídicí modul je modul jádra Linuxu, který obsahuje vlákna spouštěná v RTLinuxu. Protože použité pohony jsou krokové motory a pro zjištění aktuální polohy se nepoužívají žádné senzory (kromě koncové krajní polohy), je nutné, aby generované řídicí signály krokových motorů byly vytvářeny s požadovanou přesností (frekvencí) a aby nedocházelo ke ztrátě těchto signálů. Při splnění této podmínky je možné zjišťovat přesnou polohu z počtu vygenerovaných pulsů pro krokový motor ve směru dopředu i zpět. Generovaný signál nesmí mít příliš vysokou frekvenci, aby nedocházelo ke ztrátě kroku krokových motorů (určení polohy by bylo chybné).

5.4.1. Struktura řídicího modulu

Struktura řídicího modulu se skládá ze dvou vláken viz. obrázek 5.4. Vlákno řízení a generování signálu a vlákno čtení požadavků na řízení z real time fronty (fifo) z uživatelského prostoru.

Tato vlákna používají globální strukturu (globální v rámci modulu) reprezentující požadavky na cíl řízení. Vlákno řízení ji používá pro čtení a vlákno handleru real time fifo do ní zapisuje. I když zapisování do real-time fifo reprezentuje zásah uživatele, je málo pravděpodobné, že obě dvě vlákna (čtecí i zapisovací) budou přistupovat ke globálním datům v jednom okamžiku. Přesto jsem se rozhodl obě vlákna synchronizovat tak, aby byl zajištěn výlučný přístup do společného místa. K tomu využívám kruhové blokování *spinlock*. Vlákno, které se pokouší vstoupit do kritické sekce, která je již blokována jiným vláknem, bude čekat v cyklu, dokud nebude zámek odstraněn.

Vazba mezi řídicím modulem a manipulátorem je reprezentována čtením a zapisováním na porty PCI karty, která tvoří rozhraní komunikace.



Obr. 5.6: Blokové schéma řídicího modulu v jádře RT Linuxu

5.4.2. Vlákno řízení

Vlákno modulu provádí skutečnou práci řízení a generování řídicích signálů. Začíná inicializací smyčky řídicího vlákna a zadáním periody opakování řídicího cyklu a tedy i periody generovaného signálu pro krokové motory. Vlákno se skládá ze tří synchronních kroků které na sebe postupně navazují.

Kontrola polohy ramene je první krok v řídící smyčce. Zjišťují se dvě krajní polohy každého ramene a testuje se, zda je možné dosáhnout požadované polohy. Pokud je zadaná pozice mimo rozsah, řídicí smyčka indikuje chybu v zadání polohy.

Druhým krokem je generování jednoho bytu řízení, jehož bity mají význam pulsních signálů pro jednotlivé krokové motory podle požadavků na cílový stav. Tento vygenerovaný byte je odeslán na příslušný port PCI karty.

Posledním krokem je odeslání stavu řízení a polohy jednotlivých ramen prostřednictvím real-time fifo, která byla alokovaná pro zápis, do uživatelského prostoru. Protože řídicí smyčka je volána řádově ve stovkách Hz, neprobíhá odesílání stavu řízení v každém cyklu. Zapisování do real-time fifo je realizováno řádově v jednotkách Hz.

V následujícím výpisu kódu uvádím nejdůležitější části řídicího vlákna:

1. Inicializace modulu.

```
int init_module( void )
{
  int input_f, output_f;
  pthread_attr_t attr ;
  struct sched_param sched_p;
  int ret;

   rtl_printf(" init module\n");
```

2. Pro kontrolu uzavírám rt-fifo s příslušnými minoritními čísly a alokuji si nové fronty příslušných velikostí.

```
  rtf_destroy( STATUS_SEND_FIFO );
  rtf_destroy( CONTROL_READ_FIFO );
  input_f = rtf_create( STATUS_SEND_FIFO, FIFO_SIZE );
  output_f = rtf_create( CONTROL_READ_FIFO, FIFO_SIZE );
```

3. Vytvářím hlavní řídicí vlákno s požadovanými atributy a prioritou.

```
   pthread_attr_init( &attr );
   sched_p.sched_priority = 2;
   pthread_attr_setschedparam( &attr, &sched_p );
   ret = pthread_create( &thr[0], &attr, fcn_control, ( void* )STATUS_SEND_FIFO );
   if( ret != 0 ) {
      rtl_printf( "ERROR : creating thread\n " );
      return 1;
   }
```

4. Vytvoření vlákna čtení z rt-fifo jako real-time fifo handlenu.

```
    rtf_create_handler( CONTROL_READ_FIFO, &read_handler );
    return 0;
}
```

5. Při odstranění modulu z jádra je třeba zrušit vytvořená fifo a ‘nenásilně’ ukončit řídicí vlákno.

```
void cleanup_module( void )
{
    rtf_destroy( CONTROL_READ_FIFO );
    rtf_destroy( STATUS_SEND_FIFO );
    pthread_cancel( thr[0] );
    pthread_join( thr[0], NULL );
    rtl_printf("PLX 1750 : cleanup module \n");
}
```

K předchozímu kódu je snad jen třeba dodat, že rt fifo je Linuxové znakové zařízení (character device) s majoritním číslem 150. Při jeho alokování se zadává minoritní číslo od 0 až do 63 a velikost fronty v bytech. V mém případě vytvářím dvě fifa s minoritními čísly *CONTROL_READ_FIFO* a *STATUS_SEND_FIFO* o velikosti *FIFO_SIZE*.

Další důležitou částí programu je samotné řídicí vlákno jehož zkrácený výpis následuje:

1. Funkce *fcn_control* je předávána jako parametr při volání na vytvoření vlákna.

```
void *fcn_control( void *t )
{
    int fifo = ( int )t;
    int ret;
```

2. Je třeba zadat periodu, s kterou se bude vlákno pravidelně spouštět.

```
    pthread_make_periodic_np( pthread_self(), gethrtime(), CONTROL_PERIOD );
```

3. Nekonečná smyčka vlákna.

```
    while( 1 ) {
```

4. Přerušení provádění smyčky na dobu *CONTROL_PERIOD*.

```
        ret = pthread_wait_np();
```

5. Zamčení části přistupující ke globálním datům.

```
spin_lock( &lock );
...
```

6. Odeslání stavové informace do uživatelského prostoru (TCP serveru).

```
rtf_put( fifo, &buf, sizeof( buf ));
```

7. Odemčení kritické sekce.

```
spin_unlock( &lock );
  }
}
```

Zámek je definovaný jako globální proměnná modulu [ref. <asm/spinlock.h>].

```
spinlock_t lock = SPIN_LOCK_UNLOCKED;
```

5.4.3. Čtecí vlákno handleru real time FIFO

Aby mohl řídicí modul přijímat požadavky z uživatelského prostoru na cílovou polohu ramen, vytvořil jsem real-time čtecí FIFO. Pro fifo je možné vytvořit handler, který běží jako samostatné vlákno. Při zapisování dat do globální struktury je zajištěno vyloučení vzájemnému přístupu dvou vláknem ke společným datům (čtecí vlákno handleru rt-fifo a řídící vlákno).

Handler bude číst data od uživatele a pokud ve frontě žádná nebudou, bude čekat v blokovacím stavu.

Následuje výpis z kódu čtecího handlenu rt-fifo:

1. Funkce, která se předává při vytváření handleru jako parametr

```
int read_handler( unsigned int fifo )
{
    int err = 0;
```

2. Nekonečné čtení rt-fifo (pokud nejsou data, funkce blokována)

```
   while(( err = rtf_get( fifo, &buf, sizeof( buf ))) == sizeof( buf )) {
```

3. Uzamčení kritické části

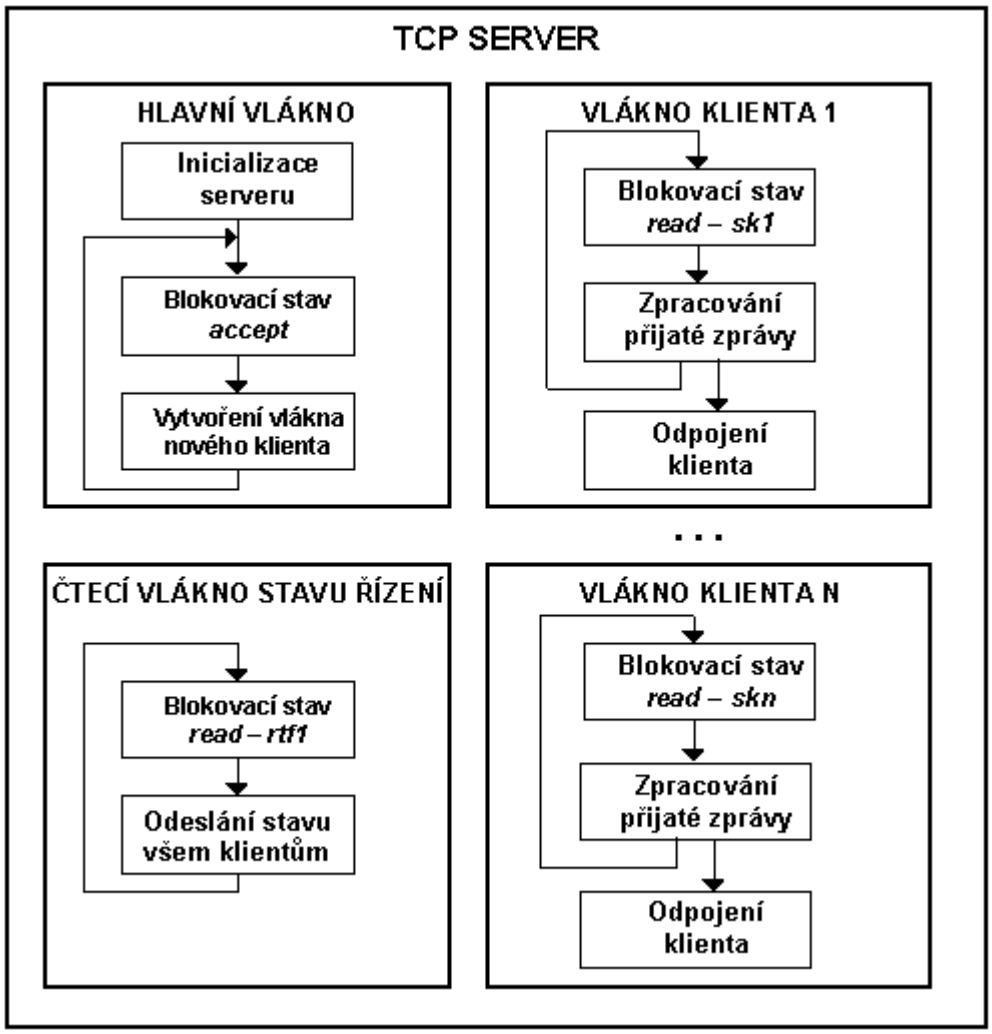
```
spin_lock( &lock );
...                    (zápis  do globální struktury)
```

4. Odemčení kritické sekce

```
        spin_unlock( &lock );
    }
    if ( err != 0 ) {
        return -EINVAL;
     }
     return 0;
}
```

5.5. TCP server

TCP server běží na stejném stroji jako řídicí modul, ale v uživatelském prostoru systému Linux. Na obrázku 5.7 je naznačena struktura programu TCP serveru.



Obr. 5.7: Struktura programu TCP serveru

5.5.1. Hlavní vlákno

Ve fázi inicializace serveru se provádějí nutná nastavení a inicializace komunikačních služeb. Pokud alespoň jedna z těchto služeb selže, server není nastartován. Inicializační kroky jsou následující:

- otevření dvou real-time FIFO pro čtení/zápis z/do uživatelského prostoru
- nastavení a vytvoření čtecího vlákna z modulu řízení *rtf1*
- vytvoření serverového TCP socketu, přiřazení služby příslušnému portu a naslouchání na tomto portu

Server umožňuje spojení s několika klientskými aplikacemi. Protože řídicí klient musí být pouze jeden, je úkol serveru evidovat seznam svých klientů. Po příchodu požadavku na řízení je zkontrolován stav aktivních klientů a zjišťuje se, zda již mezi nimi není řídicí klient. Pokud ano, server zakáže připojujícímu se klientovi kontrolu procesu a povolí mu stát se pozorovacím klientem.

Způsob připojování klientů vychází z architektury socketu. Inicializace serveru obsahuje několik často používaných kroků, proto zde uvádím část inicializačního kódu:

1. Server je standardním linuxovým programem v C se vstupním bodem *main*.

```
int main()
{
  int  ssock, i;
  pthread_attr_t attr;
  struct sched_param sched_p;
  struct sockaddr_in addr;
  connection_t* con = NULL;
```

2. Inicializace atributů vlákna čtení z rt-fifo.

```
  pthread_attr_init( &attr );
  sched_p.sched_priority = 3;
  pthread_attr_setschedparam( &attr, &sched_p );
```

3. Vytvoření rt-fifo pro zápis.

```
  if(( fwrite1 = open( "/dev/rtf2", O_WRONLY )) < 0 ) {
    fprintf( stderr, "Error open write fifo\n" );
    return( 1 );
  }
```

4. Vytvoření rt-fifo pro čtení.

```
if(( fread1 = open( "/dev/rtf1", O_RDONLY )) < 0 ) {
   fprintf( stderr, "Error open read fifo\n" );
   return( 1 );
}
```

5. Vytvoření vlákna čtení z fifo s pointerem na příslušnou funkci.

```
pthread_create( &thr, &attr, thr_client_read, ( void* )NULL );
```

6. Vytvoření serverového TCP Internet societu.

```
if(( ssock = socket( AF_INET, SOCK_STREAM, 0 )) < 0 ) {
   fprintf( stderr, "ERROR creating server socket \n");
   return 0;
}
```

7. Přiřazení socketu na příslušný port (adresa zůstává nevyplněná).

```
addr.sin_family = AF_INET;
addr.sin_port = htons( SER_PORT );
addr.sin_addr.s_addr = htonl( 0L );
i = sizeof( addr );
if( bind( ssock, ( struct sockaddr* )&addr, i ) < 0 ) {
   fprintf( stderr, "ERROR bind server socket \n" );
   return 0;
}
```

8. Vytvoření fronty pro připojení a čekání na klienty.

```
listen( ssock, 5 );
printf( "SERVER listening \n" );
```

9. Spuštění nekonečné smyčky přijímající připojení.

```
while( 1 ) {
   if(( csock = accept( ssock, (struct sockaddr* )&addr, &i )) < 0 ) {
        fprintf( stderr, "ERROR  client accepting \n" );
        continue;
   }
```

10. Přijatého klienta, který je reprezentován nově vytvořeným socketem, je třeba přidat do evidence (pole *clients*). Funkce *cl_add_client* je lokální funkce serveru, která přidává nový záznam do pole evidence klientů.

```
if( !( con = cl_add_client( clients, csock ))) {
    fprintf( stderr, "ERROR client adding \n" );
    continue;
}
printf( "Server accepted client \n" );
```

11. Pro každého klienta je vytvořeno samostatné vlákno

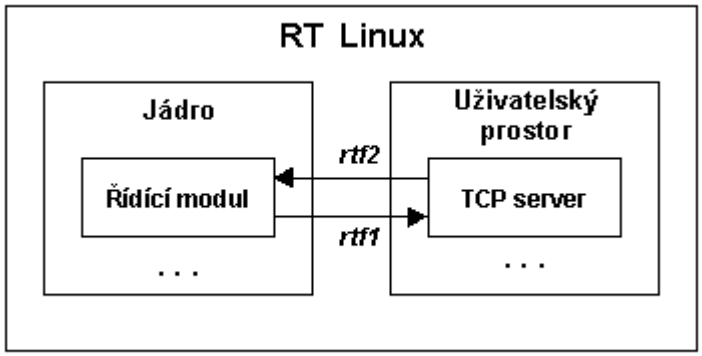
```
        pthread_create( &thr, &attr, thr_cli_proc, con );
      }
    }
```

Při zveřejnění služby serveru je nutné vybrat port, na kterém server poběží. Protože čísla portů menších než 1024 jsou v obvykle vyhrazena pro systémové služby unixu, zvolil jsem číslo portu 2003. To je třeba brát v úvahu při spouštění serveru. Jestliže jiná služba běží na tomto portu, server nebude spuštěn.

5.5.2. Komunikace mezi serverem a řídicím modulem

Server otevírá dvě rt-fifa pro komunikaci s řídicím modulem. *rtf2* je určeno pro zápis a *rtf1* je pro čtení. S těmito frontami se v Linuxu pracuje podobně jako se soubory.

Úkolem handleru čtecího fifa je získávat data o stavu řízeného procesu a rozesílat je všem připojeným klientům. Na obr. 5.8 je naznačena komunikace mezi prostorem jádra a uživatelským prostorem.



Obr. 5.8: Komunikace a výměna dat mezi prostorem jádra (řídicím modulem) a uživatelským prostorem (TCP serverem)

Následuje zkrácený výpis kódu čtecího vlákna:

1. Funkce provádějící čtení z fifa je spuštěna jako vlákno.

```
void *thr_client_read( void* t )
{
  ...
```

2. V nekonečné smyčce probíhá čtení z fifa.

```
  while( 1 ) {
    read( fread1, &buf, sizeof( buf ));
    ...
```

3. Rozeslání informace všem aktivním klientům. Funkce *cl_write_all* je lokální funkce serveru a rozesílá buffer všem klientům v poli připojených klientů.

```
    cl_write_all( clients, ( char* )&buf, sizeof( buf ));
  }
}
```

5.5.3. Vlákno klienta

Počet možných klientů je omezen nastavením serveru. Počet možných aktivních klientů je dán velikostí pole, které eviduje klienty. Vždy se však vyskytuje maximálně 1 vlákno řídicího klienta a několik vláken pozorovacích klientů, kteří pouze přijímají stavové informace od serveru.

Poté, co server přijme klienta a vytvoří mu vlastní vlákno, je možné provádět obousměrné zasílání zpráv. Komunikace probíhá formou dotaz odpověď. Každá zpráva se skládá z hlavičky (pro všechny zprávy má stejnou velikost), která definuje typ zprávy a obsahuje kontrolní data pro zajištění správnosti přijaté zprávy[*]. Za hlavičkou následuje zbytek zprávy, jehož struktura a velikost je závislá na typu zprávy. V tabulce 5.4 jsou uvedeny možné zprávy, které lze serveru zaslat.

Zpráva	Odpověď	Význam
CM_INIT_CONTROL	CM_CONTROL_ACCEPT CM_CONTROL_DENIED	Klient žádá o přidělení kontroly nad řízením
CM_POST_CONTROL	-	Klient posílá požadavek na řízení
CM_INIT_READ	CM_READ_ACCEPT	Klient se připojuje jako pozorovací (watch)
CM_CONTROL_CLOSE	-	Klient oznamuje konec své řídicí činnosti
CM_POST_STAT	-	Server posílá stavové informace

Tab. 5.4: Přehled možných zpráv komunikace mezi serverem a klientem

[*] Komunikace může být narušena z řady důvodů. Např. porušení scénáře komunikace, nebo čtení jiného typu zprávy než byla zaslána (což se mi při vývoji několikrát stalo). Kontrolní značka zajišťuje, že přijímaná data z fronty v TCP kanálu opravdu začínají novou zprávou

Až do ukončení spojení se v nekonečné smyčce zpracovávají zprávy a podle jejich typu se odesílají klientům příslušné odpovědi. TCP server se velmi často vyskytuje v různých podobách při programování aplikací na Internetu, proto uvádím podrobnější výpis:

1. Jako parametr spouštěného vlákna se předává ukazatel do pole evidence klientů

```
void* thr_cli_proc( void* p )
{
```

2. Proměnná *con* reprezentuje pointer do pole evidence klientů (strukturu *connection_t* definuje serverová aplikace). Struktura *msg* reprezentuje hlavičku zprávy

```
  connection_t* con = (connection_t*)p;
  int i;
  head_t msg;
```

3. V nekonečné smyčce se zpracovávají požadavky od klienta (dokud se klient neodpojí, nebo nenastane chyba při čtení zprávy)

```
  while( 1 ) {
    if(( i = read( con->socket, &msg, sizeof( head_t ))) != sizeof( head_t )) {
      cl_remove_client( con );
      return NULL;
    }
```

4. Test správnosti přijaté zprávy

```
  if( msg.magic != MAGIC ) {
    printf( "ERROR message format. Wrong magic number \n" );
    cl_remove_client( con );
    return NULL;
  }
```

5. Zpracování zpráv

```
  switch( msg.command ) {
    case CM_INIT_CONTROL:
    ...
  }
 }
}
```

5.6. TCP klient

TCP klient je proces běžící na libovolném počítači připojeném k síti Internet. Se serverem distribuovaného systému komunikuje za pomocí TCP protokolu, který se také nazývá Internet protokol. Při vývoji klientské aplikace jsem musel zvolit vhodný vývojový nástroj, který vyhovuje všem nárokům kladeným na program.

5.6.1. Výběr vývojového nástroje

Pro vývoj klienta jsem zvolil poměrně mladý programovací jazyk Java, který patří v současné době mezi nejoblíbenější a nejpoužívanější v různých typech aplikací. Vychází z populární programovacího jazyka C++, kterému se velmi podobá a má i podobnou syntaxi.

V následující části zdůvodňuji, proč jsem zvolil právě Javu. Při jejím studiu jsem zaznamenal tyto základní vlastnosti, které splňují požadavky kladené na vývoj klientské aplikace.

1. *Jednoduchost*

Java je o něco jednodušší než populární objektově orientovaný jazyk C++. Vícenásobnou dědičnost v C++ rozšiřuje jednoduchým způsobem za pomocí tzv. *interface*. Dalším výrazným rozdílem je eliminace používání pointerů.

Usnadňuje práci s pamětí zavedením automatického uvolňování (garbage collection) paměti.

2. *Objektově orientovaný návrh*

Objektově orientované programování modeluje svět v elementech nazvaných *objekty*. Java je objektově orientovaná, protože programování v ní se soustřeďuje na vytváření objektů, manipulování s objekty a vzájemnou spolupráci mezi nimi.

3. *Podpora distribuovaných systémů*

Distribuované výpočetní systémy umožňují vzájemnou spolupráci počítačů propojených sítí. Java je navržena tak, že umožňuje jednoduše vytvářet distribuované výpočetní systémy. Programování síťových programů je převedeno na problém podobný čtení/zápisu z/do souboru.

4. *Interpretovaný vývojový nástroj*

Pro spuštění programu v Javě je třeba použít interpretr. Programy jsou kompilované do JVM (Java Virtual Machine) kódu nazývaném *byte kód*. Tento kód je nezávislý na stroji, na kterém běží a může být spuštěn kdekoliv, kde je přítomný interpretr javy bez ohledu na cílovou platformu. Interpretr javy překládá byte kód do jazyka cílového počítače.

5. Robustní program

Robustní se zde myslí spolehlivý a stabilní. Java eliminuje některé k chybám náchylné programátorské techniky jazyka C/C++. Například neposkytuje možnost práce s pointery kvůli nebezpečí přepsání paměťového prostoru a poškození dat. K robustnosti také přispívá možnost odchytávání a ošetřování výjimek za běhu programu.

6. Bezpečný kód

Každá internetová aplikace musí zajišťovat jistou míru zabezpečení. Java, jako internetový programovací jazyk, je používaná v distribuovaném a síťovém prostředí. Proto implementuje řadu zabezpečovacích mechanismů k ochraně systému před napadením a poškozením jinými podezřelými programy.

7. Nezávislost na architektuře

To se také označuje jako *platformová nezávislost*. Díky JVM (Java Virtual Machine), jednou napsaný kód je spustitelný kdekoliv bez ohledu na použitou platformu.

8. Přenositelnost

Program v Javě může být spuštěn na jakékoliv platformě bez nutnosti překompilování, neexistují zde žádné na platformě závislé znaky. Prostředí Javy je přenositelné na nový hardware a operační systém. To je také způsobené implementací samotného překladače v Javě.

9. Podpora více vláken (multithreaded)

Podpora více vláken je schopnost programu poskytnout paralelní zpracování více úloh (zároveň). Tato schopnost je v Javě jednoduše zabudovaná. Podpora více vláken je vhodná zejména pro programování animací nebo síťových aplikací.

10. Dynamický program

Java je navržená tak, že se přizpůsobuje prostředí, které se vyvíjí. Je možné volně přidat nové metody a vlastnosti do třídy. Za běhu programu Java nahrává třídy podle toho, jak je jich potřeba.

11. Přehledná a úplná dokumentace

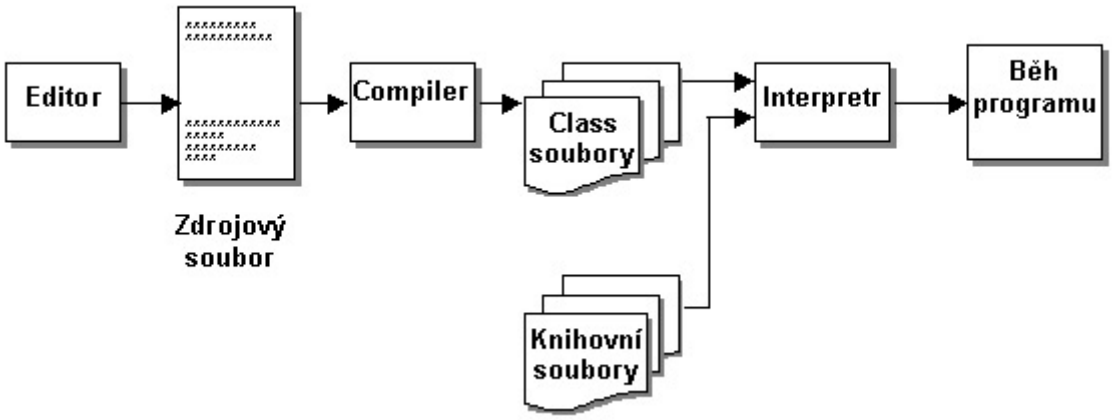
Při programování je potřeba mít vždy po ruce přehlednou dokumentaci k používaným třídám. Dokumentace Javy je opravdu velice přehledná a úplná a lze jí nalézt na zdroji [8]. Co se týká dokumentace vyvíjených programů, Java poskytuje nástroj *javadoc*, který ze zdrojových souborů vytvoří html soubory dokumentace ve stylu klasické dokumentace Javy.

Klientskou aplikaci jsem mohl vytvořit i v klasickém jazyce C. Vše co lze provést v Javě je možné i v jazyce C (myšleno při vývoji mé klientské aplikace). Do problémů bych se ale dostal ve chvíli, kdybych se snažil splnit požadavek platformové nezávislosti. Kód v jazyce C by musel být pro každý PC s jiným operačním systémem samostatně přeložen. Největší potíže by nebyly v části TCP spojení, ale při vývoji GUI části, protože v každém operačním systému se používají odlišná API pro podporu uživatelského rozhraní. I když existuje řada platformově nezávislých knihoven (např. *fltk*), při vývoji aplikace na Internetu by to znamenalo zavedení požadavků na cílový (neznámý) stroj. Java se dá v oblasti Internetu považovat za standard a je přítomná téměř na každém stroji.

Původním požadavkem bylo vytvořit klienta jako *Applet*. To je speciální typ javovského programu, který je spouštěn přímo z internetového prohlížeče s podporou Javy. Pro ochranu dat a systému, na který je applet při prohlížení Webové stránky stažen, byla zahrnuta některá omezení appletů:

1. není dovoleno čtení/zápis systémových souborů stroje, na němž je applet spuštěn
2. z appletu není dovoleno spouštět jakékoliv programy na klientském počítači
3. není dovoleno vytvořit spojení mezi PC uživatele a jiným počítačem s výjimkou vytvoření spojení se serverem, ze kterého byl applet stažen.

Díky těmto omezením jsem musel zvolit vývoj běžné aplikace, která běží jako samostatný program spouštěný z příkazové řádky příkazem *java*. Každá taková samostatná aplikace má podobný vstupní bod programu jako všechny programy napsané v C/C++ funkci *public static void main(String[ ] args)*. Aplikace se skládá ze zdrojových souborů (.java) pojmenovaných podle *public* třídy, kterou obsahuje. Kompiler javy převádí zdrojový kód na byte kód (.class). Pokud je zdrojových souborů více, je možné je uspořádat do tzv. balíků (packages) a zapouzdřit je do jediného do archivu javy (.jar). Java také umožňuje přidávání referencí na třídy volně dostupné na Internetu, o jejichž stažení se za běhu programu postará samotná Java. Na obrázku 5.9 je zobrazen postup kompilace.



Obr. 5.9: Kompilace v jazyce Java

5.6.2. Struktura programu klient

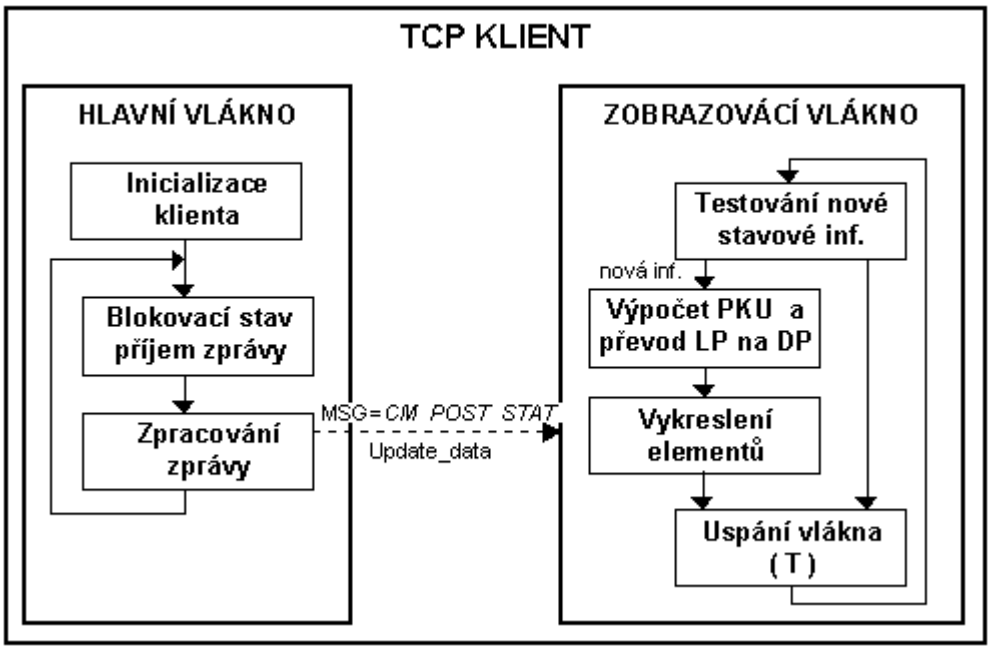
Podpora síťových aplikací a protokolu TCP/IP se nachází v balíku *java.net*. Java podporuje architekturu socketu jako jednoho konce dvoubodového spojení na síti. Třídy *java.net.Socket* a *java.net.ServerSocket* poskytují systémově nezávislý komunikační kanál používající TCP protokol.

Java rozlišuje mezi dvěma typy socketů. *Socket* je objekt reprezentující klientskou stranu dvoubodového spojení mezi javovským programem a jiným programem na síti. *Server socket* je objekt implementující stranu serveru obousměrné linky.

Komunikační schéma je podobné komunikačnímu schématu vyvíjeném v C/C++.

1. Otevření socketu
2. Otevření vstupního a výstupního streamu na socketu
3. Čtení/zápis z/do streamu v souladu s aplikací serveru
4. Zavření streamů

Struktura TCP klienta je na obrázku 5.10.



Obr. 5.10: Struktura programu TCP klienta

5.6.2.1. Hlavní vlákno

V hlavním vláknu běží *main* funkce aplikace Javy. Při inicializaci je zde vytvořeno okno aplikace (frame) a vytvořeno vlákno animace stavu procesu. Následuje vytvoření klientského socketu a navázání spojení s TCP serverem. Pro toto spojení je nutné znát IP adresu serveru a port, na kterém služba běží. Každý klient je inicializován jako pozorovací klient a při prvním spojení přijímá úvodní informaci o stavu řídicího procesu.

Po inicializaci probíhá nekonečná smyčka provádějící zasílání a příjem zpráv z komunikace s TCP serverem. Ve stavu blokování je smyčka zastavena a čeká na data zaslaná serverem. Po jejich přijetí pokračuje ve zpracování zprávy podle jejího typu viz. tabulka 5.2.

Při příjmu nové stavové informace jsou poslána data (obsahující stavovou informaci) vláknu, které provádí zobrazování stavové informace.

TCP část klientské aplikace:

1. Import potřebných knihoven a vytvoření hlavní třídy.

```
import java.net.*;
import java.io.*;

public class TcpClient {
   InetAddress adr;
```

```
int port;
Socket sk;
OutputStream oStream;
InputStream iStream;
MyBuf buf;

...

public boolean ClientInit( ) {
```

2. Vytvoření socketu pro komunikaci se serverem (*adr* je IP adresa serveru, *port* je port služby serveru).

```
try {
   sk = new Socket( adr, port );
}
catch( UnknownHostException e ) {
   System.out.println( "Unknown host" );
   return false;
}
catch ( IOException e ) {
   System.out.println( "Host on the port "+port+" is not active" );
   return false;
}
```

3. Získání vstupního a výstupního streamu.

```
try {
   iStream = sock.getInputStream();
   oStream = sock.getOutputStream();
}
catch( IOException e ) {
   Systém.out.println( "Creating Input / Output stream" );
   return false;
}
return true;
}
```

4. Čtení zprávy = čtení ze vstupního streamu.

```
public void  ReadMsg ( )
{
   try {
      is.read( buf, 0, buf.size );
   }
   catch( IOException e ) {
      System.out.println( "ERROR reading from socket" );
   }
}
```

5. Zápis zprávy = zápis do výstupního streamu.

```
public void SendMsg( )
{
   try {
      os.write( buf, 0, buf.size );
   }
   catch( IOException e ) {
      System.out.println("ERROR writing to socket");
   }
  }
}
```

5.6.2.2. Zobrazovací vlákno

Každá plnohodnotná aplikace komunikující s uživatelem by měla mít příjemné, přehledné a na ovládání jednoduché grafické rozhraní (pokud to typ aplikace umožňuje). Pro sledování řídicího procesu ovládání vzdáleného manipulátoru je třeba mít názornou představu o poloze jednotlivých ramen. Tato aplikace vytváří v zobrazovacím cyklu 3D animaci pohybu manipulátoru.

Velké omezení pro vývoj 3D grafiky bylo v použití ‘pouze‘ standardních balíků Javy*, která obsahuje jen 2D grafické knihovny. Bylo proto nutné jednoduše vytvořit 3D podporu.

Protože 3D animace je poměrně náročná na výpočetní čas procesoru, je nutné ji spouštět v samostatném vlákně.

* Zde se myslí bez podpory 3D grafických knihoven jako OpenGL nebo Java3D, i když pro Javu není problém stáhnout potřebné balíky ze zadané URL.

Vlákno porovnává v každém svém cyklu předchozí zobrazená data s aktuální stavovou informací. Pokud nastala změna stavu, probíhá výpočet PKU (přímá kinematická úloha), jejímž výsledkem jsou souřadnice bodů 3D modelu v LP (logické souřadnice) po provedení všech geometrických transformací. Následuje transformace bodů z 3D prostoru do 2D roviny a nakonec projekce do DP (zobrazovací souřadnice zařízení). Provedením těchto transformací v popsaném pořadí získám výsledné body, které lze již přímo vykreslit na grafický canvas.

Pro minimalizaci blikání animace jsem se pokoušel využít tzv. *offscreen buffer*, který nejprve vytvoří obrázek v paměti (mimo prostoru obrazové paměti = *offscreen*) a následně ho najednou vloží přímo na canvas (kreslící plátno okna). Tento postup však blikání animace nezamezil a proto jsem musel použít jiný způsob vykreslování. Aktivní objekty na canvasu nejdříve překreslím stejnými objekty s barvou pozadí a poté vykreslím nové objekty.

Model manipulátoru jsem rozdělil na grafické elementy typu kvádru. Transformace otočení a posunutí kvádru v prostoru provádím přepočtem souřadnic 8 krajních bodů kvádru a následném propojení bodů hranami. Tyto transformace jsou reprezentovány maticemi v homogenních souřadnicích (4x4). Na obrázku 5.11 je naznačen přechod mezi světovým souřadnicovým systémem a jednotlivými kloubovými souřadnicovými systémy manipulátoru. Po zjištění transformačních matic je již jednoduché vypočítat body grafických elementů, které patří do souřadnicového systému příslušného kloubu:

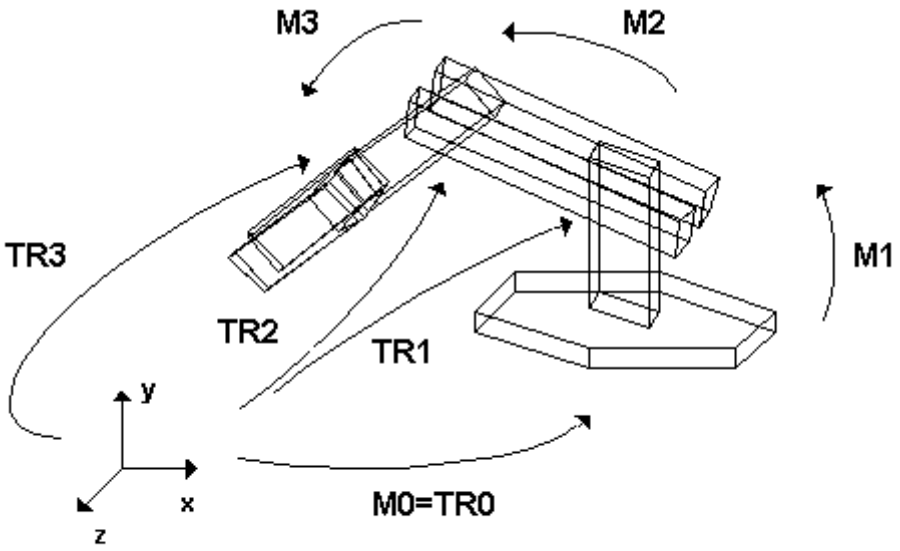
$$\mathbf{x = M * x_0}$$

x - je nový bod po geometrických transformacích
M – transformační matice v homogenních souřadnicích
x_0 – původní bod (před geometrickými transformacemi)

Význam a výpočet matic přechodu mezi jednotlivými souřadnicovými systémy ramen z obrázku 5.11:

- *M0* – matice přechodu mezi světovým ss. a ss. základny **M0=T0**
- *M1* – matice přechodu ss. základny a ss. hl. ramene **M1=R1*T1**
- *M2* – matice přechodu ss. hl. ramene a vedl. ramene **M2=R2*T2**
- *M3* – matice přechodu ss. hl. ramene a vedl. ramene **M3=R3*T3**

Matice **T** jsou translační matice a **R** jsou rotační matice v homogenních souřadnicích. V tomto případě reprezentují rotační matice natočení ramen řízeného manipulátoru a translační matice reprezentují rozměry manipulátoru.



Obr. 5.11: Znázornění postupu výpočtu PKU. Význam transformačních matic a souvislost mezi souřadnicovými systémy kloubů manipulátoru a světovými souřadnicemi

Transformační matice TR obsahují všechny informace o rozměrech modelu manipulátoru. To znamená, že pokud provádíme transformaci počátku v souřadnicovém systému libovolného ramene (bod [0,0,0]) příslušnou transformační maticí, získáme souřadnice tohoto bodu ve světových souřadnicích.

Vztahy pro transformační matice souřadnicových systémů kloubů a světovým souřadnicovým systémem:

- transformační matice pro určení bodů základny **TR0=T0**
- transformační matice pro určení bodů hl. ramene **TR1=TR0*M1**
- transformační matice pro určení bodů vedl. ramene **TR2=TR1*M2**
- transformační matice pro určení bodů chapadla **TR3=TR2*M3**

Rotační a translační matice jsou standardními transformačními maticemi (záleží na volbě orientace jednotlivých os). Jejich tvar lze nalézt například v [6].

6. Překlad, spuštění a ovládání

Řídicí modul tvoří zdrojový soubor *contr_mod.c*, hlavičkový soubor *contr_mod.h* a hlavičkový soubor popisující rozhraní mezi TCP serverem *interface.h*. Pro překlad jsem vytvořil soubor *Makefile*, který lze spustit programem *make*. Struktura souboru *makefile* a program *make* jsou popsány např. v [1]. Tento program je třeba spustit v adresáři, kde se nachází příslušný soubor *Makefile*. Po překladu se vytvoří soubor *contr_mod.o*. Je-li na počítači nainstalován systém RTLinux a je připojená PCI karta PLX 1750, je možné modul vložit do jádra Linuxu příkazem *insmod contr_mod.o* nebo *rtlinux start contr_mod.o*. Úspěšnost inicializace modulu lze prohlédnout ve výpisu na konzoli příkazem *dmesg* a poté je již možno komunikovat s modulem.

Modul je také možné přeložit s definovaným makrem *__DEMO__*, které umožňuje vložit modul do jádra bez přítomnosti PCI karty PLX 1750.

Odstranění modulu z jádra se provede příkazem *rmmod contr_mod.o* nebo *rtlinux_stop contr_mod.o*.

Zdrojové soubory TCP serveru jsou *tcp_ser.c*, *tcp_ser.h* a *interface.h*. Ve stejném souboru *Makefile* jako v případě modulu jádra, je také příkaz pro přeložení serveru. Server se spouští příkazem *./tcp_ser* a přímo na výpisu konzole je vidět, zda byl server spuštěn bez chyb. Parametry spuštění serveru jsou dva:

-h - zobrazí nápovědu
-p[port] - umožňuje spustit službu serveru na zadaném portu. Bez udání tohoto parametru je server spuštěn na portu 2003.

Protože zdrojových souborů klientské aplikace je více, vytvořil jsem jednu knihovnu (package) obsahující několik tříd. Tuto knihovnu využívá aplikace *CliFrm* a applet *DemoApl*. Applet zobrazuje pouze ukázkovou animaci pohybu manipulátoru. K přeložení celého projektu slouží další *Makefile*. Před překladem je nutné nastavit cestu k místu, kde se budou nacházet přeložené třídy. Následují příklady pro nastavení cest v Linuxu a Windows:

a) Př. Linux

CLASSPATH=/root/java_proj/classes
export CLASSPATH

b) Př. Windows

set CLASSPATH=c:\java_proj\classes

Spuštěním příkazu *make help* se zobrazí na konsoli možnosti překladu aplikace. Ze zdrojových souborů je možné vygenerovat nápovědu ve standardním formátu Javy.

Aby byla orientace v knihovně jednodušší, uvádím přehled vytvořených balíků:

c*lient.comm.buffers* – definuje rozhraní pro komunikaci serverem (konstanty, struktury, …)
client.comm – provádí komunikaci se serverem a umožňuje serveru zasílat požadavky na řízení
client.grUtil – jednoduše vytváří 3D grafickou podporu aplikace
client.robElements – sestavuje 3D model manipulátoru z jednotlivých elementů
client.robModel – umožňuje zobrazit pohyb ramen na obrazovce

Spouštět aplikaci je možné spuštěním třídy *CliFrm.class*. Parametry příkazové řádky jsou:

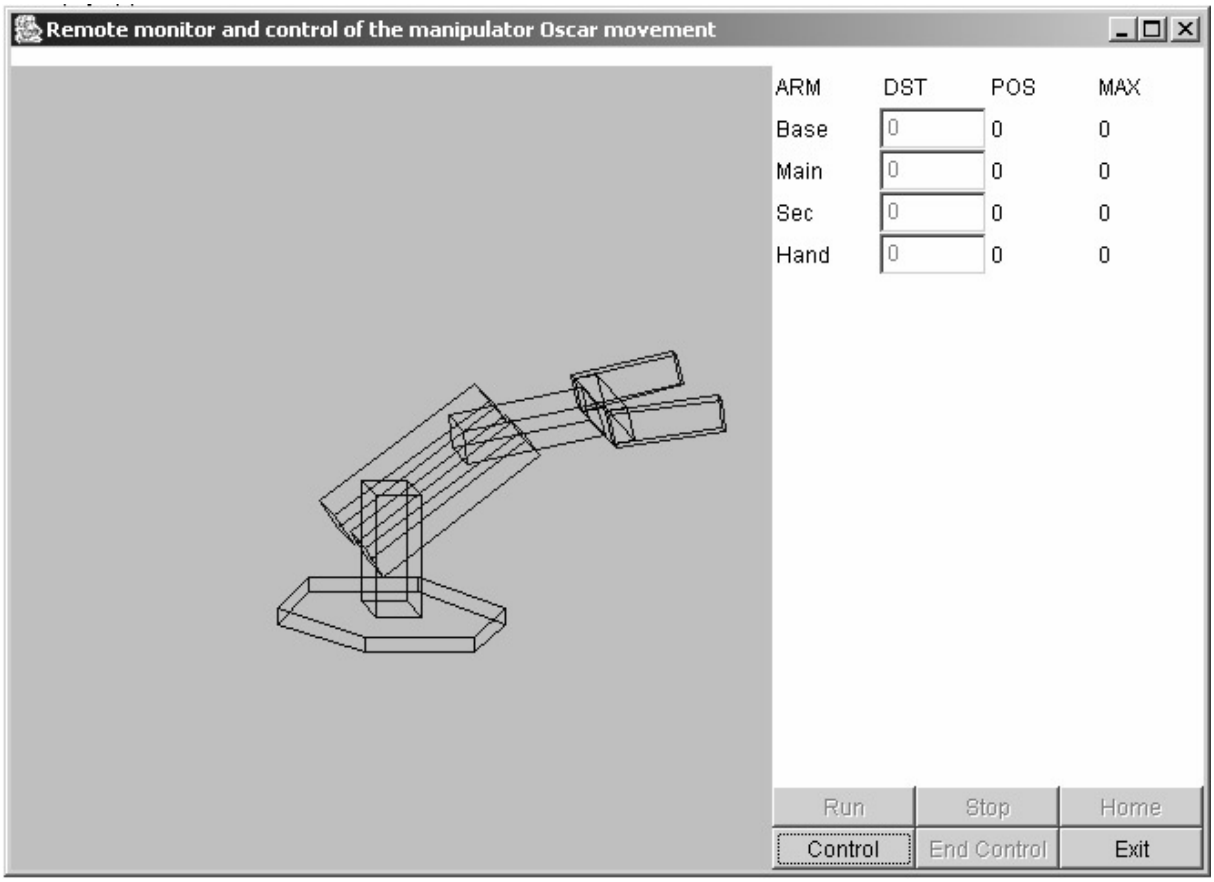
-h	- zobrazí nápovědu
-p[port]	- zadání portu, na kterém běží služba serveru (defaultně je *2003*)
-a[address]	- IP adresa TCP serveru (defaultně *localhost*)

Z přeložených tříd lze také vytvořit jeden javovský archiv *cli.jar* a ten lze spustit příkazem *java –jar cli.jar*. Pokud vše proběhne jak má, výpis na konsoli neobsahuje žádná chybová hlášení a zobrazí se okno aplikace jako na obrázku 6.1.

Aplikace umožňuje jednoduchým způsobem zadávat požadavky na řízení pohybu manipulátoru. V levé části je graficky zobrazen stav a pozice ramen manipulátoru. Tato část je stejná pro pozorovacího i řídicího klienta. Pravá část je závislá na tom, zda se jedná o řídicího nebo pozorovacího klienta.

Na obrázku 6.1 je zobrazeno okno pozorovacího klienta. Ten má povoleno tlačítko *Control*, jehož stisknutím lze přejít do stavu řízení (pokud se jiný klient v tomto stavu již nenachází). Tlačítkem *Exit* se ukončuje aplikace. Ostatní tlačítka jsou zakázána. V horní části se zobrazují hodnoty *DST* (cíl pohybu), *POS* (aktuální pozice) a *MAX* (rozsah pohybu) všech ramen v jednotkách počtu kroků krokových motorů. *Base* je základna, *Main* je hlavní rameno, *Sec* je vedlejší rameno a *Hand* je chapadlo. Pozorovací klient nemá přístup k polím pro zadávání cílové polohy a jejich hodnoty jsou cílové hodnoty pohybu, které zadává řídicí klient.

Řídící klient má povolena tlačítka *Run, Stop, Home* a *End Control*. Jsou povolena editovatelná pole, do kterých lze zadávat hodnoty poloh ramen od 0 až do *MAX*. Tlačítkem *Home* se uvede manipulátor do výchozí polohy. Tlačítko *Run* slouží pro spuštění řídicího procesu pro uvedení do zadané pozice. *Stop* zastaví pohyb a *End Control* odpojí klienta od řízení procesu a ten se stane pozorovacím klientem.



Obr. 6.1: Okno klientské aplikace

Aby bylo možné přesně řídit polohu manipulátoru, je nutné dovést všechna ramena do počáteční polohy. To musí zajistit první klient, který se přihlásí jako řídicí (první klient od startu serveru). Dokud se tak nestane, nelze řídit pohyb manipulátoru a tlačítko *Run* s editovatelnými poli nejsou povolena.

7. Závěr

Podařilo se mi úspěšně vytvořit distribuovaný řídící systém s laboratorním manipulátorem Oskar. Systém splňuje všechny požadavky na něj kladené.

Karta na převod úrovní signálů mezi vnitřní elektronikou manipulátoru a PC umožňuje spolupráci počítače s manipulátorem. Jednoduché komunikační rozhraní zajišťuje laboratorní karta PCI PLX 1750.

Správu a koordinaci klientů provádí TCP server. Počet klientských připojení je omezen nastavením serveru na 100 klientů. Síť Internet je propojovacím článkem systému a pomocí protokolu TCP/IP aplikace mohou zasílat zprávy a využívat služby serveru.

Při implementaci klientské části se ukázala být nejvhodnějším nástrojem Java. Především požadavek na platformovou nezávislost bych obtížně splnil použitím jiného vývojového nástroje.

Návrh celého distribuovaného systému pokrývá poměrně široký záběr z oblasti řídící a výpočetní techniky. Některá témata by zasloužila více pozornosti jako například vývoj ovladačů zařízení, která umožňují efektivně pracovat s HW počítače.

Jako vylepšení a rozšíření systému by bylo možné, doplnit model na straně klienta mřížkou. Pro řízení pohybu manipulátoru by pak zadávaly pouze souřadnice polohy chapadla. K tomu by bylo nutné vyřešit IKU (inversní kinematická úloha) a ze všech řešení vybrat to nejvýhodnější (například s ohledem na pravděpodobnost dalšího vývoje pohybu manipulátoru).

Dalším rozšířením systému by bylo přidání překážek, které by musel manipulátor obcházet. Na serveru by bylo zajištěno zadávání parametrů a pozic překážek. Tímto rozšířením by se prostředí manipulátoru přiblížilo reálnému pracovnímu prostředí.

Zdroje, ze kterých jsem čerpal, uvádím v příloze. V příloze na CD jsou zdrojové soubory vytvořeného systému.

Velký význam při návrhu systému přikládám výběru důvěryhodných zdrojů, věnujících se příslušné problematice. Na závěr bych chtěl poděkovat Ing. Františku Vackovi za poskytnutí cenných rad a nasměrování do studia zadané problematiky.

Literatura

[1] N. Mathew, W. Stones : Linux začínáme programovat : Computer Press Praha 2000

[2] N. P. Simons : Getting Started with RTLinux : 2001

[3] Kocourek P.: Přenos informace: Vydavatelství ČVUT, Praha 1994

[4] Janeček J.:Distribuované systémy. Vydavatelství ČVUT, Praha 1997

[5] Daněček J. : JAVA. Vydavatelství ČVUT 2000

[6] B. Hudec : Základy počítačové grafiky : Vydavatelství ČVUT Praha 2001

Odkazy na www zdroje

RtLinux

[7] http://www.fsmlabs.com/

Java

[8] http://www.java.sun.com

Přílohy

Na přiloženém CD jsou zdrojové soubory vytvořeného systému. Zdrojové soubory jsou rozděleny do tří částí:

1. řídící modul jádra Linuxu (adresář *contr_mod*)
2. TCP server (adresář *server*)
3. TCP klient (adresář *client*).