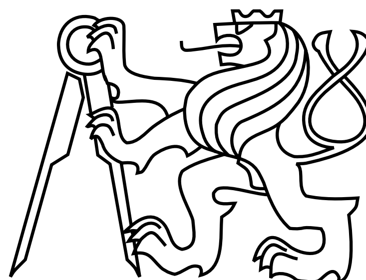
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky



Diplomová práce

Prostředí pro výuku vývoje PCI ovladačů do operačního systému GNU/Linux

Bc. Rostislav Lisový

Vedoucí práce: Ing. Pavel Píša, Ph.D.

Studijní program: Otevřená informatika, Navazující magisterský

Obor: Počítačové inženýrství

23. května 2011

Poděkování

Rád bych poděkoval Ing. Pavlu Píšovi, Ph.D., za věcné rady a připomínky.

Prohlášení

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

V Praze dne 12. 4. 2011 ....

Abstract

The goal of this article is to describe the main principles of PCI driver development for the GNU/Linux operating system. Apart from the general PCI driver development, implementation of UIO and Comedi drivers is also described.

Humusoft MF624 and MF614 cards were chosen as exemplary devices. Basic functions (D/A, A/D converters, digital inputs/outputs) of Humusoft MF624 card were implemented into the Qemu emulator, so that it is possible to try out all of the described procedures without physical access to the card.

Abstrakt

Cílem této práce je popsat základní principy implementace ovladačů PCI zařízení pro operační systém GNU/Linux. Kromě obecných principů je popsána implementace ovladačů typu UIO a Comedi.

Jako ukázková zařízení byly zvoleny karty Humusoft MF624 a MF614. Základní funkce (D/A, A/D převodníky, digitální vstupy a výstupy) karty Humusoft MF624 byly implementovány do emulátoru Qemu tak, aby bylo možné popsané postupy vyzkoušet bez fyzického přístupu ke kartě.

Obsah

1 Úvod 1

1.1 Motivace, cíl . . . 1
1.2 Dostupné materiály . . . 2

2 Hardware 3

2.1 Základní principy komunikace s hardwarem . . . 3
2.2 PCI sběrnice . . . 6
2.2.1 Historie . . . 6
2.2.2 Konektory . . . 6
2.2.3 Dynamická konfigurace a konfigurační adresní prostor . . . 6
2.2.4 Přerušení . . . 9
2.2.5 Budoucnost . . . 9
2.3 Humusoft MF624 . . . 10
2.3.1 Komunikace s kartou . . . 11
2.3.2 Digitální vstupy a výstupy . . . 12
2.3.3 A/D převodníky . . . 13
2.3.4 D/A převodníky . . . 14
2.4 Humusoft MF614 . . . 15
2.4.1 Komunikace s kartou . . . 16
2.4.2 Digitální vstupy a výstupy . . . 17
2.4.3 A/D převodníky . . . 17
2.4.4 D/A převodníky . . . 18

3 Implementace ovladačů 19

3.1 Operační systém GNU/Linux . . . 19
3.1.1 Práce s PCI zařízeními z uživatelského prostoru . . . 19
3.1.2 Základní jaderný modul . . . 21
3.2 Základní funkce v prostředí jádra Linux . . . 25
3.2.1 Funkce `printk()` pro vypisování ladících zpráv . . . 26
3.2.2 Funkce `kzalloc()` pro alokaci paměti . . . 27
3.2.3 Funkce `kfree()` pro uvolňování alokované paměti . . . 27
3.3 Ovladače PCI zařízení . . . 27
3.3.1 Struktura `struct pci_device_id` . . . 28
3.3.2 Struktura `struct pci_driver` . . . 29
3.3.3 Funkce `probe()` . . . 30

3.3.4 Přístup ke zdrojům karty . . . 30
3.3.5 Funkce `remove()` . . . 31
3.4 Přístup k paměti zařízení . . . 31
3.4.1 Vstupně-výstupní adresní prostor . . . 31
3.4.2 Paměťový adresní prostor . . . 32
3.5 UIO ovladač . . . 34
3.5.1 Jaderný modul . . . 34
3.5.2 Program v uživatelském prostoru . . . 37
3.6 Comedi ovladač . . . 40
3.6.1 Registrace ovladače . . . 40
3.6.2 Struktura `struct comedi_driver` . . . 40
3.6.3 Funkce `attach` . . . 41
3.6.4 Struktura `struct comedi_subdevice` . . . 41
3.6.5 Funkce pro čtení a zápis z/do podzařízení . . . 42
3.6.6 Funkce `detach` . . . 43
3.6.7 Přístup z uživatelského prostoru . . . 43

4 Implementace karty Humusoft MF624 v Qemu **45**
4.1 Qemu . . . 45
4.1.1 Kompilace, instalace . . . 46
4.1.2 Kompilace virtuální karty Humusoft MF624 . . . 46
4.1.3 Použití . . . 46
4.2 Qt grafiké rozhraní . . . 47
4.2.1 Kompilace, použití . . . 47

5 Testování **49**
5.1 UIO ovladač, Comedi ovladač . . . 49
5.2 Qemu virtuální hardware, Qt grafické rozhraní . . . 50

6 Závěr **51**

A Obsah přiloženého CD **53**

Literatura **55**

Seznam obrázků

1.1 *Vlevo*: Kniha Lukáše Jelínka (v českém jazyce). *Vpravo*: kniha od autorů Jonathan Corbet, Alessandro Rubini a Greg Kroah-Hartman (v anglickém jazyce) . . . 2

2.1 Registr odpovídající GPIO pinům. Změnou hodnoty tohoto registru je možné měnit chování nebo stav GPIO pinů . . . 4
2.2 Paměťový a vstupně-výstupní adresní prostor u architektury IA-32 . . . 5
2.3 *Vlevo*: Schéma znázorňující rozdíly mezi konektory pro karty s napájením 3,3 V a 5 V. *Vpravo*: Reálná fotografie PCI konektorů . . . 7
2.4 Obsah 256 bajtů konfiguračního prostoru PCI karty (zvýrazněny jsou nejdůležitější registry) . . . 8
2.5 Měřící karta Humusoft MF624 . . . 10
2.6 Měřící karta Humusoft MF614 . . . 15

3.1 Diagram znázorňující funkci UIO ovladače . . . 34

4.1 Diagram znázorňující princip funkce implementované karty MF624 v Qemu . 45
4.2 Vzhled grafické aplikace pro ovládání vstupů a výstupů virtuální karty MF624 48

5.1 Svorkovnice TB620 . . . 49

Seznam tabulek

2.1 Paměťové regiony, které využívá karta MF624 11
2.2 Registry karty MF624 obsažené v regionu BAR1 12
2.3 DIN – Digital Input Register Format 13
2.4 DOUT – Digital Output Register Format 13
2.5 Kódování vstupních hodnot A/D převodníku 14
2.6 Kódování vstupních hodnot D/A převodníku 14
2.7 Paměťové a vstupně-výstupní regiony, které využívá karta MF614 16
2.8 Registry karty MF614 náležící digitálním vstupům/výstupům a analogovým vstupům/výstupům 16
2.9 Funkce jednotlivých bitů registru ADCTRL 17
2.10 Volba rozsahu A/D převodníku 18
2.11 Kódování vstupních hodnot D/A převodníku 18

Kapitola 1

Úvod

1.1 Motivace, cíl

Tato práce vnikla na základě potřeby připravit, zdokumentovat a otestovat prostředí vhodné pro výuku hardwarově zaměřených předmětů. Jedná se především o seznámení s nízkoúrovňovým přístupem k hardwaru a vývoj ovladačů. Práce předkládá návod a vlastní řešení určené pro PCI karty a operační systém GNU/Linux. Vytvořená emulace hardwaru jedné z karet umožňuje vytváření a testování vlastních ovladačů bez nutnosti fyzického přístupu k dané kartě a je přímo použitelná i při vývoji ovladačů pro jiné operační systémy.

Zvolené vstupně výstupní karty jsou na Katedře řídicí techniky využívané i v mnoha dalších předmětech k propojení počítačů s řízenými modely fyzikálních soustav. Navržené ovladače a otestovaný přístup z operačního systému GNU/Linux tedy umožňuje použít i variantu plně preemptivního jádra Linux pro řízení modelů s využitím i rozsáhlého matematického aparátu v reálném čase. Reálné nasazení pro přizpůsobený GNU/Linux umožní řízení s reálnými maximálními latencemi menšími než 200 mikrosekund. Robustnost řešení umožňuje další vývoj řídicích algoritmů a v budoucnu po delším testování i využití v reálných průmyslových aplikacích.

Text popisuje základní aspekty práce s PCI zařízeními v jádře Linux a uvádí dva konkrétní způsoby implementace ovladače zařízení PCI – pro přístup k hardwaru z uživatelského prostoru s minimální nutnou podporou z jádra (UIO ovladač) a plnohodnotný ovladač na úrovni jádra operačního systému začleněný do subsystému určeného pro měřící a řídicí vstupně-výstupní zařízení (Comedi).

Text obsahuje pouze nezbytné množství teorie, která je podložena četnými příklady pro snadnější pochopení. Pro čtenáře neznalého psaní programů těsně svázaných s hardwarem, jsou názorně vysvětleny základní principy a úskalí tohoto druhu programování.

Jako ukázková zařízení na sběrnici PCI byly zvoleny karty Humusoft MF624 a MF614. Podrobně je popsána jejich funkce, včetně způsobu obsluhy ovladačem. Tyto karty byly zvoleny také z důvodu snadno pochopitelného způsobu obsluhy.

Výsledkem práce jsou, kromě popisu vývoje PCI ovladačů, i ovladače typu UIO a Comedi podporující základní funkce (A/D, D/A převodníky a digitální vstupy a výstupy) karet Humusoft MF614 a MF624, které slouží jako jednoduché ukázkové ovladače.

Pro maximální možné zhodnocení návodů, je cílem práce implementovat některé funkce karty Humusoft MF624 do emulačního programu Qemu tak, aby bylo možné popsané postupy implementace ovladačů vyzkoušet i bez fyzického přístupu ke kartě.

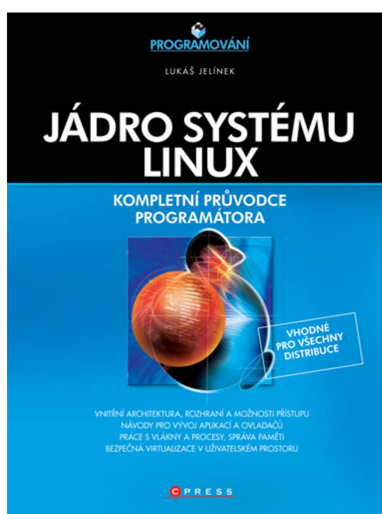
1.2 Dostupné materiály

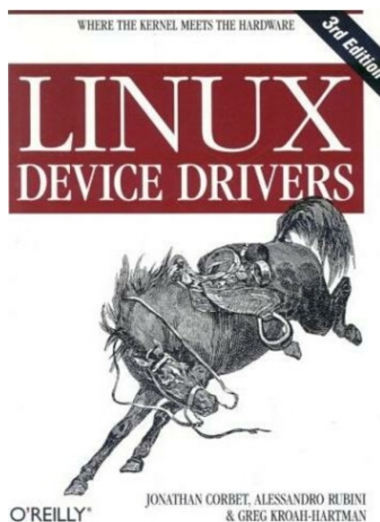
V českém jazyce dosud vyšla pouze jedna tištěná kniha, která se zabývá problematikou programování v prostředí jádra Linux. Jedná se o knihu Jádro systému Linux [6] od Lukáše Jelínka. Je dělena do 3 základních částí: *Vnější rozhraní jádra*, *Vývoj ovladačů*, *Pohled dovnitř jádra*.

Jednotlivá témata jsou popsána pouze stručně (kniha je koncipována spíše jako příručka než jako učebnice) a pro čtenáře, neznalého vývoje ovladačů zařízení, nemá příliš velký přínos.

Za nejpřínosnější knihu, zabývající se problematikou jaderného programování, považuji anglicky psanou knihu Linux Device Drivers [7] od autorů Jonathan Corbet, Alessandro Rubini a Greg Kroah-Hartman. Tato kniha podrobně vysvětluje jak obecné principy a funkce používané u jaderných ovladačů, tak i způsob implementace ovladačů zařízení konkrétních typů.

Knihu je možné stáhnout zdarma ve formátu PDF.[1]





Obrázek 1.1: *Vlevo*: Kniha Lukáše Jelínka (v českém jazyce). *Vpravo*: kniha od autorů Jonathan Corbet, Alessandro Rubini a Greg Kroah-Hartman (v anglickém jazyce)

[1] http://lwn.net/Kernel/LDD3/

Kapitola 2

Hardware

2.1 Základní principy komunikace s hardwarem

Komunikace s periferiemi je v nejjednodušších případech založena na čtení a zápisu obsahu registrů mapovaných do adresního prostoru procesoru. V případě jednočipového počítače (mikrokontroléru) již žádné další softwarové a systémové vrstvy do hry nevstupují a princip lze snadno popsat.

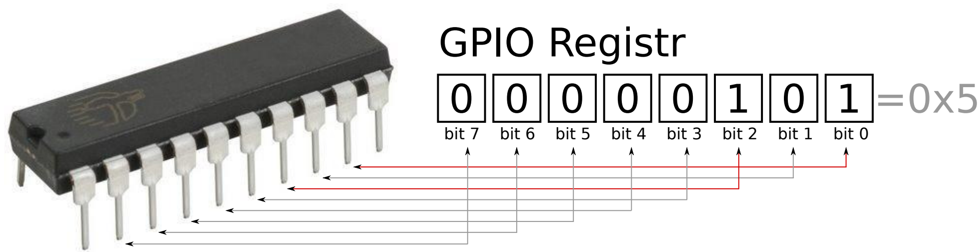
Mikrokontrolér, neboli jednočipový počítač, má velikost pouze jednoho čipu. Obsahuje přitom procesor, paměť, vstupně-výstupní zařízení a jiné. Je obvyklé, aby mikrokontrolér obsahoval tzv. GPIO piny.

GPIO piny (General Purpose Input/Output) – u těchto pinů je možné nastavit, zda má být jejich hodnota čtena (slouží jako piny pro vstup informace) nebo zda má být jejich hodnota nastavována (tj. výstupní piny).

Prvním způsobem, jak změnit stav (obecného) GPIO pinu (ať už nastavení, zda se má jednat o vstupní/výstupní pin nebo jakou hodnotu má mít v případě, že je výstupní) je provedení operace zápisu na určitou adresu v paměťovém adresním prostoru (ta je pro konkrétní typ součástky – nebo celou rodinu příbuzných typů – pevně daná). Tato adresa odpovídá **registru**[1] GPIO pinu. Adresa zápisu je přivedena na adresový dekodér, který zjistí, do které oblasti (vnitřní paměť dat, programu, oblast periferních registrů) adresa náleží. Pokud se jedná o oblast periferií, provede další podrobnější určení, které periferii připojené ke sběrnici data náleží a pověří obvody vybrané periferie dalším zpracováním zapisovaných dat. Zapsaná hodnota se tedy projeví změnou stavu GPIO pinu. Tato možnost je nejjednodušší a je možná v případě, že jsou hardwarové periferie mapovány do určité části tzv. **paměťového adresního prostoru**.[2]

[1]Registr může být pro zjednodušení považován za malou paměťovou buňku. Změna její hodnoty přímo ovlivňuje stav hardware. V dokumentaci ke konkrétnímu mikrokontroléru/mikroprocesoru/programovatelnému integrovanému obvodu je uvedeno, jakou funkci mají jednotlivé bity registru.

[2]Také označováno jako MMIO – *Memory-mapped input/output*



Obrázek 2.1: Registr odpovídající GPIO pinům. Změnou hodnoty tohoto registru je možné měnit chování nebo stav GPIO pinů

U některých procesorových architektur se změna hodnoty registru provede jiným způsobem – použitím, při zápisu do registru, jiné instrukce než která se používá pro paměťové operace – tj. místo zápisu na adresu v paměťovém prostoru vyhrazenou pro GPIO registr, se provede zápis do tzv. **vstupně-výstupního adresního prostoru**[3] na adresu (v tomto případě označovanou jako **port**) odpovídající registru GPIO pinů. Adresy paměťového a vstupně-výstupního adresního prostoru jsou nezávislé. V případě zápisu a čtení do/z portu I/O adresního prostoru je potřeba z dokumentace **přesně vědět** jak široká (kolikabitová) slova je možné zapisovat/číst.

V případě architektury IA-32 (označované také jako x86) máme k dispozici paměťový a vstupně-výstupní adresní prostor. Adresy vstupně výstupního adresního prostoru jsou pouze 16bitové, zatímco paměťového jsou (*pro zjednodušení není brán ohled na PAE – Physical Address Extension*) 32bitové. Toto rozdělení přetrvává z historických důvodů – i přesto je již možné některá zařízení mapovat do paměťového prostoru. (Znázorněno na obrázku 2.2.)

Informace o obsazenosti paměťového a vstupně-výstupního adresního prostoru je možné v GNU/Linuxu zjistit čtením souboru `/proc/iomem` a `/proc/ioports`.

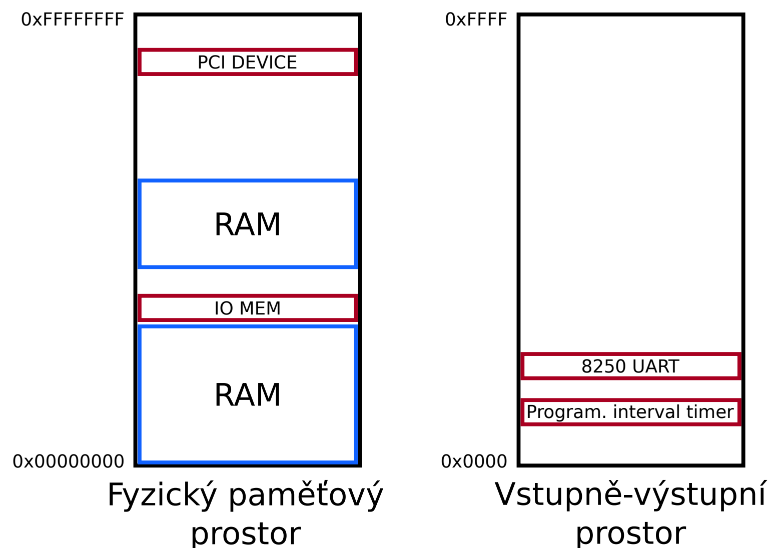
Hlavní rozdíly mezi chováním paměťové buňky a registru zařízení jsou:

- Změnou hodnoty registru je možné měnit stav zařízení/periferie odpovídající danému registru.
- V případě zápisu do registru a jeho okamžitém čtení, nemusí být přečtená hodnota shodná se zapisovanou – v tom případě byla hodnota registru změněna hardwarem.
- V případě čtení z registru může být spuštěn tzv. **side effect**, kdy hardware na toto čtení reaguje změnou stavu, podobně jako by byl proveden zápis do registru (Příklad:

[3] Také označován zkratkou PIO – *Programmed input/output* nebo jako I/O adresní prostor

ihned po vyčtení hodnoty registru A/D převodníku se spustí nový převod a původní hodnota se přepíše novou). Side effects mohou nastat i při zápisu do registru.

- Při zápisu a čtení do/z registru je nutné přesně rozlišovat, kolika-bitové operace zápisu/čtení smějí být použity (8-, 16-, 32bitové).
- Při přístupu k registrům mapovaným do paměťového adresního prostoru (z jádra operačního systému nebo z uživatelského programu), je nutné o této skutečnosti *upozornit* jak překladač, tak samotný procesor. Důvodem jsou optimalizace, které mohou být provedeny – které při přístupu do obyčejné paměti mohou program zrychlit, ale při přístupu do registru mohou způsobit nesprávnou funkci programu. Podrobněji je tento problém popsán v kapitole 3.4.



Obrázek 2.2: Paměťový a vstupně-výstupní adresní prostor u architektury IA-32

2.2 PCI sběrnice

PCI (*Peripheral Component Interconnect*) je standard paralelní sběrnice využívaný v počítačích různých architektur. Šířka paralelně přenášených dat je 32 nebo v modernější, méně často používané verzi, 64 bitů. Sběrnice je orientována na přenos zpráv oproti přímé komunikaci mezi zařízeními.[4]

Komunikace mezi zařízeními připojenými na sběrnici a procesorem zajišťuje tzv. *PCI most* (PCI bridge). Propojení více nezávislých sběrnic v jednom počítači jsou zajištěny také PCI mosty.

2.2.1 Historie

V roce 1990 začala práce na specifikaci PCI v laboratořích firmy Intel. První specifikace definující jak komunikační protokol, tak vzhled konektoru a slotu, byla zveřejněna 30. dubna 1993 (jedná se o PCI 2.0). PCI sběrnice se poté začala objevovat v počítačích architektury IBM PC a PowerPC.

V pozdějších letech se původní standard dočkal vylepšení – zvýšení šířky paralelní sběrnice z 32 bitů na 64 bitů a zrychlení z 33 MHz na 66 Mhz a výše. Tyto pokročilejší verze se však příliš neujaly.

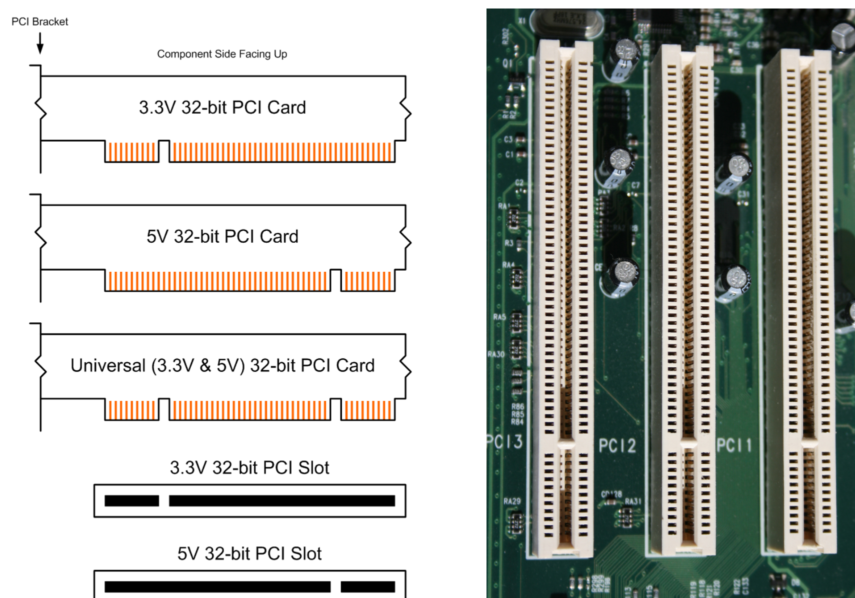
2.2.2 Konektory

Pro spojení mezi kartou a sběrnicí je potřeba pouze konektor na straně sběrnice – tzv. slot. V závislosti na napájecím napětí (3,3 V nebo 5 V) jsou na kartách *klíčovací* zářezy – tyto zářezy znemožňují zasunutí *napěťově nekompatibilní* karty do slotu. Existují však univerzální karty, které mají tyto zářezy oba, díky čemuž mohou být použity v libovolném slotu (obr. 2.3).

2.2.3 Dynamická konfigurace a konfigurační adresní prostor

Mezi hlavní výhody PCI sběrnice (oproti její předchůdkyni – sběrnici ISA) patří dynamická konfigurace připojených zařízení: Ve většině případů probíhá komunikace mezi hostitelským systémem a připojenou (a nakonfigurovanou) PCI kartou zápisem/čtením do určité paměťové (nebo vstupně-výstupní) oblasti. U starší sběrnice ISA bylo při návrhu karty nebo propojkami na kartě, při jejím zapojení do PC, pevně určeno, kam se její část paměti namapuje – v takovém případě mohl nastat problém, že více než jedna karta mapovala svoji paměť na stejnou adresu (nebo se jednotlivá mapování překrývala). PCI sběrnice tomuto problému předchází takovým způsobem, že každá z karet nese informaci o tom, kolik jak velkých paměťových nebo I/O regionů potřebuje namapovat – o samotné mapování se poté dynamicky postará BIOS počítače nebo PCI subsystém operačního systému.

[4]Tj. místo toho, aby PCI most přistupoval přímo k paměti jednotlivých zařízení, vyšle se na sběrnici zpráva s požadavkem. V případě, že je některé zařízení schopno požadavek obsloužit, umístí na datovou sběrnici požadovaná data. (Takto probíhá komunikace na úrovni sběrnice – ze strany procesoru se jedná pouze o zápis/čtení paměťového/vstupně-výstupního adresního prostoru.)



Obrázek 2.3: *Vlevo*: Schéma znázorňující rozdíly mezi konektory pro karty s napájením 3,3 V a 5 V. *Vpravo*: Reálná fotografie PCI konektorů

Informaci o tom, kolik (a jaké) paměti karta bude potřebovat má před nakonfigurováním uloženu v tzv. **Base Address Registrech** – BAR0–BAR5[5]. Poté co se (při konfiguraci) podaří tuto hodnotu přečíst a požadovanou paměť alokovat, zapíše se zpět do daného registru adresa, na které se alokovaná paměť nachází. Tu si poté pro potřeby komunikace vyčte ovladač zařízení, který je součástí operačního systému.

Kromě výše zmíněných 6 BAR registrů, obsahují PCI zařízení i následující registry:

Vendor ID
Obsahuje unikátní 16bitové číslo identifikující výrobce zařízení. Za poplatek je udělováno PCI-SIG (*PCI Special Interest Group*) organizací.[6]

Device ID
Obsahuje 16bitové číslo identifikující model zařízení. Hodnotu tohoto identifikátoru si volí sám výrobce zařízení.

Class code
Označuje (ve 24 bitech) druh zařízení – zda se jedná např. o grafickou kartu, zvukovou kartu nebo kartu zpracovávající signál.

[5]Informace o velikosti požadované oblasti je v registru uložena takovým způsobem, že je pouze jeho část určena k zápisu a zbytek je pouze pro čtení. PCI most se pokusí do registru zapsat hodnotu 0xFFFFFFFF, poté je hodnota zpět vyčtena – z bitů náležejících do zapisovatelné části registru, je přečtena 1_2, zbývající část obsahuje hodnoty 0_2.

[6]V Debianu, po nainstalování balíčku `hwdata`, se seznam těchto identifikátorů nachází v souboru `/usr/share/hwdata/pci.ids`

Subsystem Vendor ID

Podobá se Vendor ID. V případě, že karta využívá PCI řadič třetí strany, jako Vendor ID se zobrazí ID výrobce tohoto řadiče. Aby bylo možné zařízení odlišit od jiného, které využívá stejný řadič, skutečné ID zařízení bude uloženo v tomto registru.

Subsystem ID

Opět se jedná o údaj podobný Device ID sloužící k rozlišení karet postavených na univerzálním řadiči.

Registry Vendor ID, Device ID (příp. ještě Subsystem Vendor ID a Subsystem ID) slouží operačnímu systému k jednoznačné identifikaci zařízení, při volbě správného ovladače.

31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Lat. Timer	Cache Line S.	0Ch
Base Address Registers				10h 14h 18h 1Ch 20h 24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			Cap. Pointer	34h
Reserved				38h
Max Lat.	Min Gnt.	Interrupt Pin	Interrupt Line	3Ch

Obrázek 2.4: Obsah 256 bajtů konfiguračního prostoru PCI karty (zvýrazněny jsou nejdůležitější registry)

Výše popsané registry (spolu s ostatními, které zde nebyly popsány) se nacházejí v 256bitovém tzv. **konfiguračním adresním prostoru** karty (obr. 2.4).[7] Pro *konfigurační* adresní prostor není hardwarová podpora v téměř žádné procesorové architektuře – přístup do něj je například na architektuře IA-32 možný pomocí zapsání adresy (*kam má být v konfiguračním prostoru zapisováno*) a dat (*která mají být do konfiguračního prostoru zapsána*) do dvou speciálních I/O portů, které jsou pro tuto operaci vyhrazeny.[8]

[7] Po paměťovém a vstupně-výstupním adresním prostoru je zde třetí – konfigurační – adresní prostor.

[8] Toto je možné považovat za *klasický* způsob přístupu. Na moderních procesorech architektury IA-32 je již možné konfigurační adresní prostor namapovat do paměťového adresního prostoru.

2.2.4 Přerušení

Sběrnice PCI obsahuje čtyři linky přerušení a všechny z nich jsou dostupné každému zařízení. Přerušení mohou být sdílená, tudíž o jedno přerušení se může dělit více zařízení. Pro snazší sdílení, jsou přerušení úrovňově spouštěná (oproti hranovému spouštění nedochází k promeškání přerušení).

V pozdějších revizích PCI specifikací je přidána podpora pro přerušení signalizované zprávou. V tomto případě zařízení oznamuje svůj požadavek na obsloužení zápisem do paměti PCI mostu – ten poté tento požadavek směruje dále k procesoru.

2.2.5 Budoucnost

V posledních letech je na poli osobních počítačů PCI sběrnice nahrazována její nástupkyní – sběrnicí PCIe (*PCI Express*). Ta je na rozdíl od PCI sériová a dosahuje rychlostí až 16 GiB/s. I přesto je sběrnice PCI stále využívána mnohými zařízeními – převážně v průmyslu.

PCI Express zařízení využívají, podobně jako PCI, konfigurační prostor (o velikosti 4 KiB). Prvních 256 bajtů je totožných s konfiguračním prostorem PCI zařízení (registry jako Vendor ID, BAR, apod. jsou totožné). PCIe sběrnice umožňuje nejen připojení jiné PCI sběrnice pomocí PCIe–PCI mostu, ale také připojení zařízení na sběrnici PCIe, jehož logická funkce je totožná s PCI zařízením. Postupy týkající se PCI sběrnice, popsané v této práci, jsou tedy přímo aplikovatelné i na zařízení typu PCIe.

2.3 Humusoft MF624

Obrázek 2.5: Měřící karta Humusoft MF624

Měřící karta Humusoft MF624 (obr. 2.5), připojitelná k počítači pomocí PCI sběrnice, má pro účely výkladu psaní ovladačů několik nesporných výhod:

- Komunikace (na úrovni ovladače) s kartou probíhá snadno pochopitelným, přímočarým způsobem, kdy je pouze zapisováno (nebo čteno) do registrů karty (bude vysvětleno dále).
- Je možné si ověřit správnou funkci napsaného ovladače – např. připojením LED diody k digitálnímu výstupu nebo měřením napětí na výstupu D/A převodníku.

Karta MF624 najde své uplatnění hlavně v laboratorním prostředí – v případech, kdy je potřeba zpřístupnit měřené hodnoty senzorů. V případě analogových, resp. digitálních signálů jsou použity A/D převodníky, resp. digitální vstupy. Kartu je možné použít i pro řízení akčního členu/zařízení – k dispozici jsou D/A převodníky a digitální výstupy.

Karta disponuje následujícími funkcemi (v popisu implementace ovladačů se omezím pouze na A/D, D/A převodníky a digitální vstupy/výstupy):

- 8 digitálních vstupů (TTL kompatibilní logické úrovně)
- 8 digitálních výstupů (TTL kompatibilní logické úrovně)
- 8 14bitových A/D převodníků (rozsah ±10 V)
- 8 14bitových D/A převodníků (rozsah ±10 V)
- 4 časovače/čítače
- 4 vstupy inkrementálních snímačů

2.3.1 Komunikace s kartou

Komunikace s kartou není nijak složitá – zjednodušeně by se dala popsat následovně:

- V případě čtení hodnoty digitálních vstupů, přečte se hodnota registru určeného právě digitálním vstupům – v případě zápisu na digitální výstupy, se zapíše do registru určeného digitálním výstupům.
- V případě čtení hodnoty A/D převodníku, se nejprve zapíše do konfiguračního registru A/D převodníku hodnota odpovídající požadované konfiguraci. Poté se již z registru náležícího A/D převodníku vyčte požadovaná hodnota.

Které registry karta obsahuje, jakou mají funkci a na jakých adresách jsou umístěny je možné zjistit z oficiálního manuálu ke kartě – ten je možné stáhnout z internetových stránek výrobce: `http://www2.humusoft.cz/www/datacq/manuals/mf624um.pdf`.

Po jeho otevření je na straně 11 k vidění první důležitá tabulka (zde tab. 2.1):

Region	**Function**	**Size (bytes)**	**Width (bytes)**
BADR0 (memory mapped)	PCI chipset, interrupts, status bits, special functions	32	32
BADR1 (memory mapped)	A/D, D/A, digital I/O	128	16/32
BADR2 (memory mapped)	Counter/timer chip	128	32

Tabulka 2.1: Paměťové regiony, které využívá karta MF624

Z ní je patrné, že karta využívá 3 regiony[9] mapované do paměťového adresního prostoru – o velikostech 32, 128 a 128 bajtů.
Pro čtení/zápis z/do nich je potřeba používat 32-, 16- a 32bitové operace.[10]

2.3.2 Digitální vstupy a výstupy

Z tabulky 2.1 lze vyčíst informaci, že registry ovládající digitální vstupy a výstupy leží v regionu BAR1 (sloupec 2). Dále je potřeba se podívat na přehled registrů náležejících tomuto paměťovému regionu – tomu odpovídá tabulka (s menšími úpravami) 2.2.

Address (BADR1 offset)	**Read**	**Write**
0x00	**ADDATA** – A/D data	ADCTRL – A/D control
0x02	**ADDATA** – A/D data mirror	
0x04	**ADDATA** – A/D data mirror	
0x06	**ADDATA** – A/D data mirror	
0x08	**ADDATA** – A/D data mirror	
0x0A	**ADDATA** – A/D data mirror	
0x0C	**ADDATA** – A/D data mirror	
0x0E	**ADDATA** – A/D data mirror	
0x10	**DIN** – Digital input	**DOUT** – Digital output
0x20	**ADSTART** – A/D SW trigger	**DA0** – D/A 0 data
0x22		**DA1** – D/A 1 data
0x24		**DA2** – D/A 2 data
0x26		**DA3** – D/A 3 data
0x28		**DA4** – D/A 4 data
0x2A		**DA5** – D/A 5 data
0x2C		**DA6** – D/A 6 data
0x2E		**DA7** – D/A 7 data

Tabulka 2.2: Registry karty MF624 obsažené v regionu BAR1

Na devátém řádku jsou zmíněny `DIN` (Digital input) a `DOUT` (Digital output) registry. Z této tabulky je zřejmá pozice těchto registrů v paměťovém prostoru (tj. offset v bytech vůči adrese BAR1).

Jak jsou data v registrech reprezentována, je možné si přečíst (v oficiálním manuálu) na straně 16, kde jsou tyto dva registry podrobně popsány (zde tabulka 2.3 a 2.4). První

[9] V manuálu je uvedeno, že se jedná o regiony odpovídající BAR0, BAR1 a BAR2 registrům – na počítačích s procesory rodiny IA-32 a s operačním systémem GNU/Linux však karta využívá BAR0, BAR2 a BAR4. Důvod rozdílu mezi skutečností a manuálem není jasný. Skutečné použití BAR registrů musí být před implementací ovladače zkontrolováno na konkrétním systému.

[10] V manuálu je uvedeno, že za určitých podmínek je možné k BAR1 přistupovat i pomocí 32bitových operací. V této práci bych se tomuto složitějšímu přístupu raději vyhnul. Částečná implementace karty MF624 do emulátoru Qemu (popsaná v kapitole 4) umožňuje **pouze** 16bitový přístup do BAR1 paměťového regionu.

sloupec určuje, kterých bitů se daný řádek týká. V druhém sloupci je informace o funkci. Třetí sloupec udává výchozí hodnotu. Z toho, co je v tabulkách uvedeno, plyne, že pro čtení 8bitového digitálního vstupu stačí přečíst spodních 8 bitů DIN registru, horních 8 bitů je potřeba ignorovat. Stejně tak pro nastavení 8bitového digitálního výstupu se zapíše požadovaná hodnota do spodních 8 bitů registru DOUT, horních 8 bitů je potřeba ignorovat.

Bit	Description	Default
7:0	**Digital input 7:0.** Reads digital input port.	1
15:8	Reserved	N/A

Tabulka 2.3: DIN – Digital Input Register Format

Bit	Description	Default
7:0	**Digital output 7:0.** Writes to digital output port.	0
15:8	Reserved	N/A

Tabulka 2.4: DOUT – Digital Output Register Format

2.3.3 A/D převodníky

Karta MF624 obsahuje osm 14bitových A/D převodníků s pevně stanoveným měřeným rozsahem ±10 V. Jejich vyčtení může probíhat následujícím způsobem:

- Nejprve se v registru ADCTRL zvolí, které A/D převodníky mají být čteny. Každý z A/D převodníků je reprezentován jedním bitem. Zápisem 1 do daného bitu se nastaví, že bude daný A/D převodník aktivní – 0 ho deaktivuje. Je možné zvolit více než jeden A/D převodník.
- Čtením registru ADSTART se spustí převod na zvolených A/D převodnících. Přečtená hodnota se dále nepoužívá.
- V případě, že se úspěšně provedl převod na všech zvolených A/D převodnících, je EOLC bit (17. bit) GPIOC registru nastaven na 0 (jinak je v 1).
- Výslednou hodnotu je možné přečíst z registru ADDATA, který je typu FIFO. To znamená, že opětovným čtením jednoho registru jsou vyčítány jednotlivé naměřené hodnoty z měřených A/D převodníků v pořadí od 0 do 7.

  Jinou možností je místo čtení registru ADDATA číst některý z jeho *zrcadlených registrů* (celkem je jich 7, v manuálu jsou označeny jako *BADR1 + 0x02* až *BADR1 + 0x0E*). Tyto registry se chovají **zcela stejně** jako registr ADDATA, pouze leží na jiných adresách. Příklad: pokud byly aktivovány první čtyři A/D převodníky, po převodu je možné výslednou hodnotu vyčíst opakovaným čtením registru ADDATA nebo čtením registru ADDATA, ADDATA1, ADDATA2, ADDATA3 přesně v tomto pořadí. Čtení z registrů v jiném pořadí bude stále vracet hodnoty převodníků 0–4.

Hodnota vyčtená z A/D převodníků je ve formátu dvojkového doplňku – příklad konkrétních hodnot je v tabulce 2.5.

2.3.4 D/A převodníky

Karta MF624 obsahuje také osm 14bitových D/A převodníků s rozsahem ±10 V.

Nastavení výstupních hodnot D/A převodníků může probíhat následujícím způsobem:

- Hodnota v aditivním kódu (tabulka 2.6) se zapíše do jednoho z osmi registrů DA0–DA7 odpovídajícího D/A převodníku, který má být nastaven.
- Bit DACEN (26. bit) registru GPIOC je potřeba nastavit na 1, jinak jsou výstupy D/A převodníků připojeny na *zem*.
- Bit LDAC (23. bit) registru GPIOC je potřeba nastavit na 0, aby byl spuštěn samotný převod D/A převodníků (jinak zůstane zapsaná hodnota pouze v registru, výstupní hodnota D/A převodníku zůstane nezměněna).

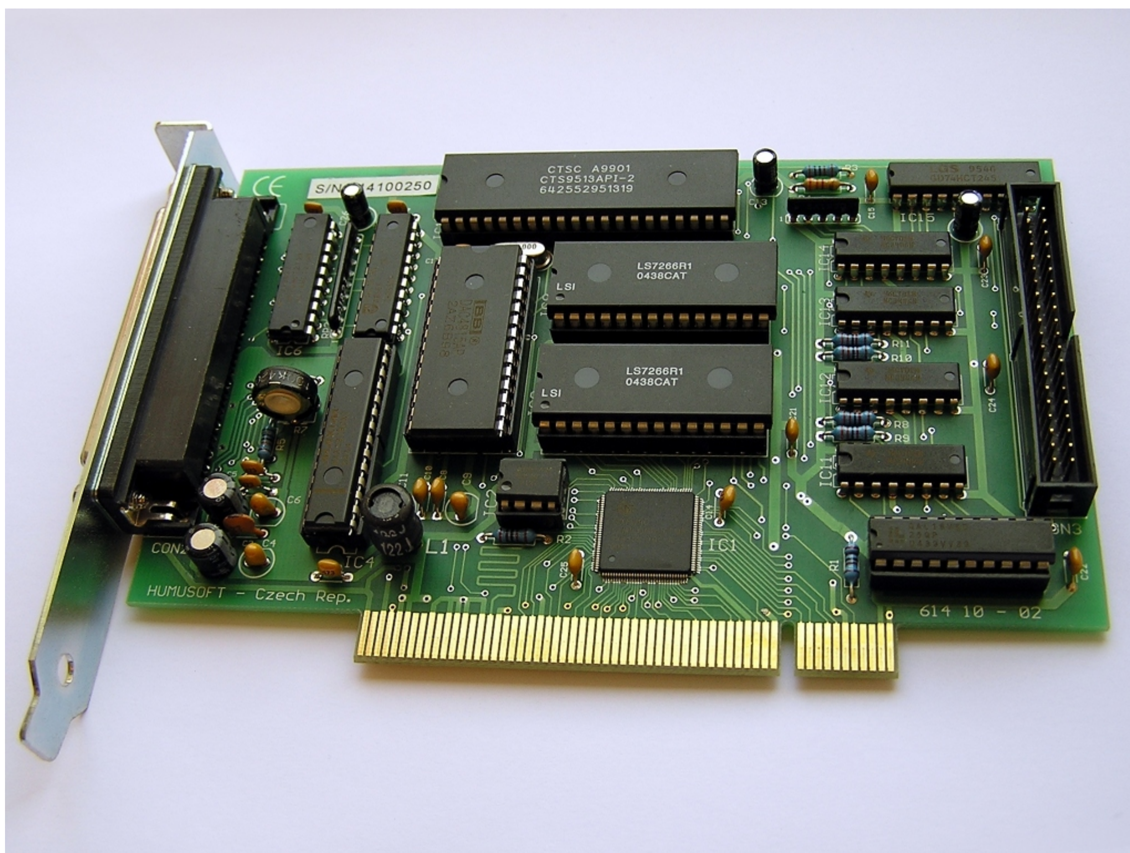
Digitální hodnota	**Analogová hodnota**
0x3FFF	-0.0012 V
0x2000	-10.0000 V
0x1FFF	9.9988 V
0x0000	0.0000 V

Tabulka 2.5: Kódování vstupních hodnot A/D převodníku

Digitální hodnota	**Analogová hodnota**
0x3FFF	9.9988 V
0x2000	0.0000 V
0x1FFF	-0.0012 V
0x0000	-10.0000 V

Tabulka 2.6: Kódování vstupních hodnot D/A převodníku

2.4 Humusoft MF614



Obrázek 2.6: Měřící karta Humusoft MF614

Karta Humusoft MF614 má podobné funkce a využití jako karta MF624. Ve skutečnosti se jedná o její předchůdkyni.

Karta disponuje následujícími funkcemi:

- 8 digitálních vstupů (TTL kompatibilní logické úrovně)
- 8 digitálních výstupů (TTL kompatibilní logické úrovně)
- 8 12bitových A/D převodníků (volitelné rozsahy ±10 V, ±5 V, 0–5 V, 0–10 V)
- 4 12bitových D/A převodníků (rozsah ±10 V)
- 4 časovače/čítače
- 4 vstupy inkrementálních snímačů

2.4.1 Komunikace s kartou

Způsob komunikace s kartou MF614 se mírně liší od MF624.

Po nahlédnutí do manuálu (dostupný ze stránek výrobce: `http://www2.humusoft.cz/www/datacq/manuals/mf614um.pdf`) je z tabulky 9 (zde tab. 2.7) patrné, že karta využívá více regionů než MF624, přičemž některé jsou mapovány do paměti, jiné do vstupně-výstupního adresního prostoru. Po prohlédnutí tabulky popisující rozložení registrů (zde tab. 2.8) je zřejmé, že pro přístup k digitálním vstupům/výstupům a analogovým vstupům/výstupům jsou použity vstupně-výstupní regiony BAR0, BAR2.

Region	**Function**	**Size (bytes)**
BADR0 (I/O mapped)	Board programming registers	32
BADR1 (I/O mapped)	Reserved	4
BADR2 (I/O mapped)	OX9162 local configuration registers	32
BADR3 (memory mapped)	OX9162 local configuration registers	4096
BADR4 (memory mapped)	Board programming registers	4096

Tabulka 2.7: Paměťové a vstupně-výstupní regiony, které využívá karta MF614

Address	**Read**	**Write**
BADR0 + 0x0	**ADLO** – A/D data low	**ADCTRL** – A/D control
BADR0 + 0x1	**ADHI** – A/D data high	
BADR0 + 0x2	**9513A** – Data read	**9513A** – Data write
BADR0 + 0x3	**9513A** – Command read	**9513A** – Command write
BADR0 + 0x4		
BADR0 + 0x5		
BADR0 + 0x6	**DIN** – Digital input	**DOUT** – Digital output
BADR0 + 0x7		
BADR0 + 0x8	**DALE** – D/A latch enable	**DA0LO** – D/A 0 data low byte
BADR0 + 0x9		**DA0HI** – D/A 0 data high byte
BADR0 + 0xA		**DA1LO** – D/A 1 data low byte
BADR0 + 0xB		**DA1HI** – D/A 1 data high byte
BADR0 + 0xC		**DA2LO** – D/A 2 data low byte
BADR0 + 0xD		**DA2HI** – D/A 2 data high byte
BADR0 + 0xE		**DA3LO** – D/A 3 data low byte
BADR0 + 0xF		**DA3HI** – D/A 3 data high byte
...	...	...
BADR2 + 0x10	**STATUS** – Status register	

Tabulka 2.8: Registry karty MF614 náležící digitálním vstupům/výstupům a analogovým vstupům/výstupům

Jednotlivé registry v těchto regionech jsou 8bitové, proto je potřeba při čtení/zápisu používat pouze 8bitové funkce. 16bitové hodnoty jsou rozděleny do dvou 8bitových registrů

– v takovém případě, obsahuje-li registr ve svém názvu písmena **LO**, jedná se o spodní bajt, zatímco **HI** značí horní bajt. Výsledná 16bitová hodnota se získá složením dvou 8bitových:

```
u8 regAHI, regALO;
u16 regA;

regA = regALO | (regAHI << 8);
```

2.4.2 Digitální vstupy a výstupy

Pro nastavení hodnoty digitálních výstupů se zapíše požadovaná hodnota do registru DOUT, kde jeden bit odpovídá jednomu digitálnímu výstupu. Pro čtení digitálních vstupů je potřeba přečíst hodnotu registru DIN.

2.4.3 A/D převodníky

Čtení A/D převodníků je u karty MF614 oproti MF624 trochu složitější, hlavně díky tomu, že je u převodníků potřeba nastavit, v jakém rozsahu bude provedeno měření. Je možné vybírat mezi rozsahy -10–10 V, -5–5 V, 0–10 V, 0–5 V.

K nastavení vlastností A/D převodníků slouží registr ADCTRL (přeložená tab. 2.9). Bity 2:0 slouží k volbě jednoho z osmi A/D převodníků, které budou při příštím měření použity. Dekadická hodnota určující pořadí A/D převodníku je uložena ve třech bitech jako binární číslo (tj. $0_{10} = 000_2$, $1_{10} = 001_2$, $2_{10} = 010_2$, $3_{10} = 011_2$, $4_{10} = 100_2$, ...).

Bity 3 a 4 slouží k nastavení použitého rozsahu (způsob nastavení viz tabulka 2.10).

Bity 5, 6 a 7 nemají žádnou funkci a musí bát nastaveny na 0, 1, 0.

Bit	Jméno	Popis
7		Musí být nastaveno na 0
6		Musí být nastaveno na 1
5		Musí být nastaveno na 0
4	RNG	Nastavení měřeného rozsahu A/D převodníku (tab. 2.10)
3	BIP	Nastavení, zda bude měřený rozsah *bipolární* (tab. 2.10)
2, 1, 0	A2, A1, A0	Výběr A/D převodníku pro příští měření

Tabulka 2.9: Funkce jednotlivých bitů registru ADCTRL

Vyčtení hodnoty A/D převodníku může probíhat následujícím způsobem:

- Nejprve se v registru ADCRTL zvolí, který A/D převodník bude čten a který měřící rozsah bude použit.
- Zápis do registru ADCTRL automaticky spouští převod.
- Je-li CC bit (2. bit) registru STATUS nastaven na 0, převod již byl ukončen.

RNG	BIP	Vstupní rozsah [V]
0	0	0–5 V
1	0	0–10 V
0	1	-5–5 V
1	1	-10–10 V

Tabulka 2.10: Volba rozsahu A/D převodníku

- Data je poté možné přečíst z registru ADLO a ADHI – jedná se o 8bitové registry, které je potřeba pro získání 12 bitové výsledné hodnoty *složit* dohromady. Je-li nastaven unipolární rozsah měření (tj. 0–5 V nebo 0–10 V) je měřená hodnota kódována jako binární číslo. V případě bipolárního rozsahu je hodnota kódována pomocí dvojkového doplňku.

2.4.4 D/A převodníky

Karta MF614 obsahuje 4 D/A převodníky. Ty mají pevně nastavený výstupní rozsah -10–10 V a nevyžadují žádnou konfiguraci.

Nastavení výstupu D/A převodníků může probíhat následujícím způsobem:

- Do registru DAxLO a DAxHI (kde x může nabývat hodnot 0, 1, 2, 3 a určuje, ke kterému D/A převodníku registr patří) se zapíše hodnota k převodu. 12bitová hodnota je do 8bitových registrů rozdělena takovým způsobem, že 8 LSB je zapsáno do DAxLO a zbývající čtyři bity jsou zapsány do DAxHI na 4 nejnižší bity, nepoužité 4 MSB registru DAxHI jsou vyplněny nulami.

  Hodnota je zapsána v aditivním kódu (tab. 2.11).

- Čtením registru DALE se spustí převod všech D/A převodníků.

Digitální hodnota	Analogová hodnota
0xFFF	9.9951 V
0x800	0.0000 V
0x7FF	-0.0049 V
0x000	-10.0000 V

Tabulka 2.11: Kódování vstupních hodnot D/A převodníku

MSB (Most Significant Bit) je označení pro bit s nejvyšší hodnotou v binárním vyjádření čísla. V obvyklém dvojkovém zápisu jde o bit nejvíce vlevo.
LSB (Least Significant Bit) je bit s nejnižší hodnotou. Jde o bit nejvíce vpravo.

```
 1 0 1 0 1 0 1 0
MSB           LSB
```

Kapitola 3

Implementace ovladačů

3.1 Operační systém GNU/Linux

Jako cílový operační systém, na kterém bude vysvětlena implementace základních ovladačů, byl zvolen GNU/Linux[1]. Hlavním důvodem je svobodné šíření zdrojových kódů, velké množství kvalitní dokumentace, rozšířenost a vysoká kvalita. *Distribucí* použitou při vývoji je Debian GNU/Linux (verze jádra Linux 2.6.35) – popsané postupy by však měly fungovat i pro jiné distribuce.

3.1.1 Práce s PCI zařízeními z uživatelského prostoru

Pro výpis všech zařízení připojených pomocí sběrnice PCI k počítači slouží program `lspci`. Po jeho spuštění bez udání parametrů bude vypsán základní seznam PCI zařízení.

Mezi důležité parametry patří:

`-t` Zobrazí diagram znázorňující jednotlivé PCI sběrnice a mosty.

`-v, -vv, -vvv` Umožňuje vypisování podrobných informací o zařízeních. (Postupně od *střední podrobnosti* k *vysoké podrobnosti*).

`-nn` Zobrazí Vendor ID a Device ID v číselné a zároveň i textové podobě

`-d [<vendor>]:[<device>]` Zobrazí informace pouze o zařízeních odpovídajících Vendor ID, případně i Device ID

Příklad, jak takový výpis může vypadat:

```
$ lspci -nn -d 186c:0624 -vvv
01:0b.0 Signal processing controller [1180]: Humusoft, s.r.o. MF624
             Multifunction I/O Card [186c:0624]
        Subsystem: Humusoft, s.r.o. MF624 Multifunction I/O Card [186c:0624]
```

[1]Operační systém sestávající z GNU nástrojů a jádra Linux je označován jako GNU/Linux.

```
Control: I/O+ Mem+ BusMaster- SpecCycle- MemWINV- VGASnoop- ParErr-
     Stepping- SERR- FastB2B- DisINTx-
Status: Cap- 66MHz- UDF- FastB2B+ ParErr- DEVSEL=medium >TAbort-
     <TAbort- <MAbort- >SERR- <PERR- INTx-
Interrupt: pin A routed to IRQ 22
Region 0: Memory at d2dffc00 (32-bit, non-prefetchable) [size=128]
Region 1: I/O ports at b800 [size=128]
Region 2: Memory at d2dff800 (32-bit, non-prefetchable) [size=128]
Region 4: Memory at d2dff400 (32-bit, non-prefetchable) [size=128]
Kernel driver in use: mf624
```

Jinou možností, jak zjistit informace o PCI zařízení, je nahlédnutí do souborového systému *sysfs*, kde jsou pro jednotlivá zařízení (nejen na PCI sběrnici) soubory[2], které obsahují informace o zařízeních.

Fyzická adresa PCI zařízení je tvořena adresou *sběrnice*, adresou *zařízení* a adresou *logického zařízení*. PCI specifikace umožňuje, aby jeden systém obsahoval až 256 sběrnic. Každá sběrnice může obsahovat až 32 zařízení. Jedno fyzické zařízení může obsahovat až 8 logických.

Informace o PCI zařízeních se nacházejí ve složce `/sys/bus/pci/devices/` – jednotlivá zařízení jsou reprezentována podsložkou, jejíž název je tvořen fyzickou adresou PCI zařízení. Mezi nejdůležitější soubory, které tato podsložka obsahuje patří:

`vendor` – Obsahuje Vendor ID zařízení.

`device` – Obsahuje Device ID zařízení.

`class` – Obsahuje 24bitový identifikátor třídy zařízení.

`subsystem_vendor` – Obsahuje Subsystem Vendor ID.

`subsystem_device` – Obsahuje Subsystem ID.

`resource` – Soubor obsahuje popis jednotlivých regionů využívaných zařízením (také označováno jako *obsah* BAR registrů).

Struktura souboru `resource` může vypadat následovně:

```
0x00000000d2dffc00 0x00000000d2dffc7f 0x0000000000020200
0x000000000000b800 0x000000000000b87f 0x0000000000020101
0x00000000d2dff800 0x00000000d2dff87f 0x0000000000020200
0x0000000000000000 0x0000000000000000 0x0000000000000000
0x00000000d2dff400 0x00000000d2dff47f 0x0000000000020200
0x0000000000000000 0x0000000000000000 0x0000000000000000
```

[2]Tyto soubory ve skutečnosti nejsou uloženy nikde na disku, ale jsou dynamicky vytvářeny operačním systémem.

První sloupec označuje adresu začátku regionu, druhý jeho konec. Třetí sloupec obsahuje příznaky daného regionu. Díky nim je možné zjistit, zda se např. jedná o paměťový nebo I/O region. Tyto příznaky jsou popsány v souboru `include/linux/ioport.h` (ve zdrojových souborech jádra Linux).

3.1.2 Základní jaderný modul

Jádro operačního systému GNU/Linux je monolitické – to znamená, že po zkompilování a slinkování je tvořeno jedním kusem kódu. Tento druh jádra je léty prověřen a mezi výhody patří hlavně jeho snadná implementace a stabilita. Aby běžící jádro nemuselo obsahovat veškeré dostupné ovladače zařízení (nebo aby v případě potřeby přidat do jádra ovladač pro nový hardware nebylo nutné celé jádro znovu kompilovat), existuje mechanismus načítání jaderných modulů za běhu, tzv. LKM – *Loadable Kernel Module*. V praxi to vypadá tak, že jsou v běžícím jádře zakompilovány pouze nejnutnější ovladače, všechny ostatní si může systém nebo uživatel za běhu do jádra načíst – v případě, že již nejsou potřeba, je možné je z jádra uvolnit.

Jak se takový jaderný modul může vypadat, je nejlepší si ukázat na příkladu:

```
 1 | #include <linux/init.h>
 2 | #include <linux/module.h>
 3 |
 4 | static int hello_init(void)
 5 | {
 6 |     printk("Hello, world!\n");
 7 |     return 0;
 8 | }
 9 |
10 | static void hello_exit(void)
11 | {
12 |     printk("Goodbye, cruel world!\n");
13 | }
14 |
15 | module_init(hello_init);
16 | module_exit(hello_exit);
17 |
18 | MODULE_LICENSE("Dual BSD/GPL");
```

Z příkladu je patrné, že je modul napsán v programovacím jazyce C. To platí pro většinu jaderných modulů (stejně jako zdrojových kódů jádra samotného). Ve skutečnosti se jedná o mírně modifikovaný standard ANSI C90.

Z čeho se modul skládá:

Řádky 1 a 2 obsahují vložení hlavičkových souborů – obsahují prototypy volaných funkcí a jsou nutné pro tvorbu jaderného modulu.

Na řádcích 4–8 je funkce, která bude spuštěna ihned po zavedení modulu do jádra. Ta obsahuje pouze volání funkce `printk()`.

Pro jednoduchost je možné s funkcí `printk()` pracovat jako s, jistě známou, funkcí `printf()` dostupnou v uživatelském prostoru – na rozdíl od standardního výstupu se však text vypsaný funkcí `printk()` zapíše do *logu* jádra. Jedním ze způsobů, jak ho zobrazit je pomocí programu `dmesg`.
O to, že se tato funkce vykoná ihned po zavedení modulu do jádra, se postará příkaz na →

řádku 15 – ten obsahuje makro `module_init()`, kterému je sděleno právě to, která funkce se má po načtení spustit.

Řádek 16 obsahuje naopak makro, které udává, která funkce se má zavolat v případě, že se bude modul uvolňovat z jádra. V tomto případě je to funkce na →

řádcích 10–13. Tato funkce nemá na starost pouze výpis krátkého textu do logu jádra.

Na řádku 18 je použito makro udávající licenci definující práva a povinnosti pro šíření/používání zdrojových kódů daného modulu. Uvedení licence je důležité z toho důvodu, že jaderné moduly nevyužívající některou z opensource licencí nemají dostupná všechna jaderná volání (to platí i pro případ, že není uvedena žádná licence).

3.1.2.1 Kompilace modulu

Poté co je vytvořen zdrojový kód modulu, je třeba jej přeložit[3]. K tomu poslouží následující `Makefile`:

```
1 |  KERNEL_VER=‘uname -r‘
2 |  obj-m += hello.o
3 |
4 |  all:
5 |      make -C /lib/modules/$(KERNEL_VER)/build M=$(PWD) modules
6 |  clean:
7 |      make -C /lib/modules/$(KERNEL_VER)/build M=$(PWD) clean
```

Linux využívá při kompilaci systému `KBUILD`. Ten je tvořen větším množstvím samostatných Makefile souborů a jeho smyslem je umožnit uživateli snadnou konfiguraci před kompilací – určující, které části se do jádra zakompilují a které nikoliv. Výše uvedený (základní) Makefile soubor je tvořen následovně:

Na prvním řádku se do proměnné `KERNEL_VER` přiřadí verze aktuálně běžícího jádra (po zavolání příkazu `uname -r`, který tuto informaci vrací).

Druhý řádek říká, že modul bude vytvářen ze zdrojového souboru `hello.c` (tj. modul popisovaný v kapitole 3.1.2).

[3] Před samotným překladem modulu je potřeba mít k dispozici zdrojové kódy jádra. Ty je možné stáhnout z `http://kernel.org/` nebo v distribuci Debian nainstalovat pomocí příkazu `apt-get install linux-source`.

Na pátém řádku (uvozeném tabelátorem) se volá (pomocí přepínače `-C`) Makefile ze systému `KBUILD`, který se nachází v adresáři spolu se zdrojovými kódy jádra. Parametr `M` určuje, které moduly mají být vytvořeny – v tomto případě jsou to ty, které jsou uvedeny v Makefile, nacházejícím se v aktuálním adresáři (tj. `PWD`).

V případě, že se v adresáři, ve kterém se nachází zdrojový soubor modulu `hello.c` a výše popsaný soubor `Makefile`, spustí příkaz `make`, měl by proběhnout samotný překlad:

```
$ make
make -C /lib/modules/‘uname -r‘/build M=/tmp/kernel_module_example modules
make[1]: Entering directory ‘/usr/src/linux-headers-2.6.35-28-generic’
  CC [M]  /tmp/kernel_module_example/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /tmp/hello.mod.o
  LD [M]  /tmp/hello.ko
make[1]: Leaving directory ‘/usr/src/linux-headers-2.6.35-28-generic’
```

V aktuálním adresáři by se měl nacházet kromě různých souborů, které vzniky při překladu, i potřebný `hello.ko` – tj. zkompilovaný jaderný modul připravený na zavedení do jádra.

```
$ ls
hello.c  hello.ko  hello.mod.c  hello.mod.o  hello.o
Makefile  modules.order  Module.symvers
```

3.1.2.2 Zavedení modulu

Po úspěšném zkompilování jaderného modulu již pouze zbývá ho zavést do jádra. To se provede programem `insmod` – ten musí být spouštěn se superuživatelským oprávněním:

```
$ sudo insmod ./hello.ko
```

V případě, že vše proběhlo správně, měl by být v logu jádra text vypisovaný modulem po jeho zavedení. To je možné ověřit:

```
$ dmesg | tail -1
[ 9245.757491] Hello, world!
```

Pro plné otestování funkčnosti ukázkového modulu, je potřeba ho ještě z jádra uvolnit. K tomu slouží program `rmmod` (opět je potřeba spouštět se superuživatelskými privilegii).

```
$ sudo rmmod hello

$ dmesg | tail -1
[ 9612.256929] Goodbye, cruel world!
```

V logu se opět nachází text vypisovaný modulem při uvolňování z jádra.

> V případě, že má být do jádra zaveden modul, jehož funkčnost a stabilita není jistá, je vhodné si veškerou práci uložit (případně zálohovat) a před zavedením/uvolněním modulu do/z jádra spustit program `sync`, který uloží obsah diskových bufferů na disky.

3.1.2.3 Na co si dávat pozor

Při psaní základního modulu pro jádro Linux nejsou patrné větší rozdíly oproti psaní programů pro uživatelský prostor. I přesto, že tyto rozdíly nejsou vidět, stále tady jsou. Mezi ty nejdůležitější, kterých si má být programátor vědom, patří:

Žádná ochrana paměti
Libovolný jaderný modul má přístup k veškeré paměti počítače. V případě, že se chybně pokusí zapsat do paměti, do které by zapisovat neměl, není zde žádný mechanismus, který by mu v tom zabránil.

Uvolňování paměti
Stejně jako pro programy psané v uživatelském prostoru platí, že nepotřebná dynamicky alokovaná paměť by měla být dealokována. V případě neuvolňování paměti programem v uživatelském prostoru je zde stále operační systém, který po skončení programu veškerou paměť uvolní. Nic takového však v jádře operačního systému nefunguje – po uvolnění modulu z jádra není nic, co by se postaralo o alokovanou paměť.

Přímý přístup k hardwaru
Základní jaderný modul psaný například nezkušeným studentem má zcela stejné možnosti přístupu k hardware jako subsystémy jádra, které se starají o správnou funkci jednotlivých ovladačů. V lepším případě může špatný ovladač způsobit pád systému, v horším např. zničení dat na disku nebo dokonce zničení hardware[4].

Globální proměnné
Každý ovladač může být spuštěn ve více instancích, proto by v kódu neměly být globální proměnné. Proměnné, které je potřeba zpřístupnit z více míst ovladače se vloží do jedné struktury, která je poté přístupná skrze ukazatel na *privátní data* ovladače. Struktura reprezentující daný ovladače většinou obsahuje ukazatel s názvem `private` nebo `priv`, který slouží k tomuto účelu.

V případě ukončení funkce ovladače musí být tato paměť uvolněna.

[4]Například velmi těžko opravitelné poškození firmware síťových karet Intel e1000e:
`http://www.abclinuxu.cz/clanky/jaderne-noviny/jaderne-noviny-22.-10.-2008#pricina-chyby-poskozujici-e1000e`

3.1.2.4 Příkaz GOTO

Obecně je doporučováno příkaz `goto` nepoužívat. Najdou se ale případy, kdy jeho použití usnadní práci a i přesto neznepřehlední kód. V jádře Linux se tento příkaz používá při postupném uvolňování zdrojů zařízení.

Příklad pro lepší názornost:

```
 1 |  int mf614_attach(...)
 2 |  {
 3 |      if(pci_enable_device(devpriv->pci_dev))
 4 |          goto out_exit;
 5 |
 6 |      if(pci_request_regions(devpriv->pci_dev, "mf614"))
 7 |          goto out_disable;
 8 |
 9 |      if (!pci_iomap(devpriv->pci_dev, 0, 0))
10 |          goto out_release;
11 |
12 |  out_release:
13 |      pci_release_regions(devpriv->pci_dev);
14 |  out_disable:
15 |      pci_disable_device(devpriv->pci_dev);
16 |  out_exit:
17 |      return -ENODEV;
18 |  }
```

Na řádcích 3, 6 a 9 jsou volány funkce, které mají za následek alokaci zdrojů zařízení. Po skončení funkce ovladače je potřeba zavolat jiné funkce, které tyto zdroje uvolní.

V případě, že by volání na řádku 6 skončilo neúspěchem, musela by být zavolána funkce `pci_disable_device()`, která deaktivuje zařízení (aktivované příkazem na řádku 3). V případě, že by poslední volání proběhlo neúspěšně, musela by být zavolána kromě funkce `pci_disable_device()` ještě funkce `pci_release_regions()`. V případě, že by tyto funkce byly volány z více míst, došlo by k duplikaci kódu – ta vede k nepřehlednosti a může způsobovat chyby (v případě, že se omylem místo všech výskytů dealokační sekvence opraví pouze některé).

Za pomoci volání `goto` je výše popsaný problém elegantně vyřešen.

3.2 Základní funkce v prostředí jádra Linux

V jádře Linux je kromě funkcí specifických pro práci se zařízením určitého typu, také sada obecných funkcí používaných napříč všemi ovladači. Mezi ně patří například funkce pro vypisování ladících zpráv, funkce pro alokaci a uvolňování paměti.

3.2.1 Funkce `printk()` pro vypisování ladících zpráv

```
int printk(const char *s, ...);
```

V kapitole 3.1.2 již byla zmíněna funkce `printk()` v základní verzi, přirovnaná k funkci `printf()` z uživatelského prostoru. Kromě *obyčejného* vypisování textu do logu jádra podporuje tato funkce navíc *nastavení úrovně důležitosti* zprávy a speciální *formátovací řetězce*.

Nastavení úrovně důležitosti zprávy se provede vložením *nastavovacího* makra **před** samotný řetězec obklopený uvozovkami. Možné druhy zpráv jsou (od nejkritičtější po nejméně důležitou):

`KERN_EMERG`
: Zpráva nejvyšší důležitosti. Většinou předchází neodvratnému pádu jádra.

`KERN_ERR`
: Informace o vzniklé chybě (např. při informování o špatné funkci hardware).

`KERN_WARNING`
: Upozornění o nezávažné chybě.

`KERN_INFO`
: Informační zpráva (např. od ovladače zařízení o úspěšném spuštění).

`KERN_DEBUG`
: Obyčejná ladící zpráva.

Formátovací řetězce fungují podobně jako u funkce `printf()`. Kromě známých, `%s`, `%u`, `%d` a `%x` je zde navíc `%p`, který slouží k výpisu hodnoty ukazatele.

Možné způsoby použití jsou:

`%pF` Pro ukazatel na funkci vypíše název dané funkce.

`%pf` Pro ukazatel na funkci vypíše název dané funkce včetně offsetu.

`%pR` Pro ukazatel na strukturu vypíše adresy paměti příslušející dané struktuře, včetně příznaků.

`%pr` Pro ukazatel na strukturu vypíše adresy paměti příslušející dané struktuře, bez příznaků.

Příklad nastavení typu zprávy a použití formátovacího řetězce:

```
printk(KERN_DEBUG "Hodnota ukazatele ptr je %p\n", ptr);
```

3.2.2 Funkce `kzalloc()` pro alokaci paměti

Problematika alokace paměti v prostředí jádra Linux je velice rozsáhlá. Pomocí speciálních funkcí je možné alokovat fyzickou paměť, velké bloky virtuální paměti nebo celé paměťové stránky.

```
void *kzalloc(size_t size, gfp_t flags);
```

Základní funkce pro alokaci malé paměťové oblasti (např. pro strukturu obsahující privátní data ovladače) je `kzalloc()`. Prvním parametrem je velikost alokované paměti (maximálně však 128 KB), druhým je příznak určující o jaký druh alokace se jedná. Nejuniverzálnější možností je `GFP_KERNEL`.

Nově alokovaná paměť je vždy vynulována.

Příklad alokace a uvolnění paměti (včetně ošetření chybových stavů):

```
1 |  struct uio_info *info;
2 |  info = kzalloc(sizeof(struct uio_info), GFP_KERNEL);
3 |  if (!info) {
4 |      return -ENOMEM;
5 |  }
6 |  /* práce s pamětí */
7 |
8 |  kfree(info);
```

3.2.3 Funkce `kfree()` pro uvolňování alokované paměti

```
void kfree(void *obj);
```

Když již alokovaná paměť není potřeba, je nutné ji voláním `kfree()` uvolnit.

3.3 Ovladače PCI zařízení

Jako nejlepší reference jednotlivých funkcí slouží zdrojové kódy jádra. Pro snadné procházení je vhodné využít nástroje vytvářející křížové odkazy (mezi voláním a definicí funkce, použitím a deklarací proměnné apod.).

Nejpohodlnějším způsobem je použití online nástroje *The Linux Cross Reference* – `http://lxr.linux.no/linux/`.

Jinou možností je přímé čtení a procházení zdrojových kódů jádra. Pro vytváření indexu křížových odkazů poslouží programy `ctags` a `cscope`. Samotné procházení je poté možné například pomocí programů `vim` a `Kscope`.

Ovladače PCI zařízení jsou ve většině případů kompilovány jako jaderné moduly, dynamicky načítané za běhu jádra. Takový modul je možné buď načíst ručně, pomocí příkazu `insmod` (se zadanou absolutní cestou) nebo, nachází-li se v adresáři `/lib/modules/$(uname -r)/` a je součástí seznamu `modules.dep`[5] (v témže adresáři), je možné ho načíst pomocí příkazu `modprobe` (kde se jako parametr předá pouze název modulu bez koncovky `.ko`) – ten také zajistí i načtení modulů potřebných pro splnění případných závislostí načítaného modulu. Druhá varianta se týká všech ovladačů standardně zkompilovaných s jádrem.

V případě, že se v systému objeví nové PCI zařízení, je jádrem informován subsystém v uživatelském prostoru, který má na starosti správu *hotplug* zařízení (např. *udev*). Tento subsystém poté na základě získaných informací, jako je Vendor ID a Device ID, rozhodne, který ovladač má být pro dané zařízení načten. Seznam, dle kterého je ovladač vybírán, je v souboru `/lib/modules/$(uname -r)/modules.pcimap`.

Proto, aby mohl být ovladač součástí výše popsaného seznamu, musí ve struktuře `struct pci_device_id` obsahovat informaci o tom, pro které zařízení je určen.

3.3.1 Struktura `struct pci_device_id`

Struktura `struct pci_device_id` slouží k identifikaci, pro která zařízení je ovladač určen. Mezi hlavní položky struktury patří `vendor`, `device`, `subvendor`, `subdevice` (typu `__u32`) – jejichž hodnota odpovídá hodnotě stejnojmenných registrů v konfiguračním prostoru daného PCI zařízení. Jelikož může být ovladač napsán pro více zařízení, je tato struktura inicializována jako prvek pole, které je vždy zakončeno prázdným prvkem. Různé způsoby inicializace mohou vypadat následovně:

```
 1 |  #define PCI_VENDOR_ID_HUMUSOFT          0x186c
 2 |  #define PCI_DEVICE_ID_MF624             0x0624
 3 |  #define PCI_DEVICE_ID_MF614             0x0614
 4 |  #define PCI_SUBVENDOR_ID_HUMUSOFT       0x186c
 5 |  #define PCI_SUBDEVICE_MF624             0x0624
 6 |
 7 |  static struct pci_device_id mf624_pci_id[] = {
 8 |      {
 9 |          .vendor = PCI_VENDOR_ID_HUMUSOFT,
10 |          .device = PCI_DEVICE_ID_MF624,
11 |          .subvendor = PCI_SUBVENDOR_ID_HUMUSOFT,
12 |          .subdevice = PCI_SUBDEVICE_MF624,
13 |      },
14 |
15 |      { PCI_VENDOR_ID_HUMUSOFT, PCI_DEVICE_ID_MF614,
16 |          PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0 },
17 |
18 |      { 0, } /* seznam je vždy zakončen prázdným prvkem */
19 |  };
```

[5]Tento seznam je aktualizován pomocí příkazu `depmod`.

V případě, že je u zařízení rozhodující Vendor ID, ale na Subvendor ID nezáleží, je možné použít makro `PCI_ANY_ID` (to platí i pro Subdevice ID).

```
MODULE_DEVICE_TABLE(type, struct pci_device_id* name);
```

Tato struktura se – pro nástroje v uživatelském prostoru vytvářející seznamy ovladačů – exportuje pomocí makra `MODULE_DEVICE_TABLE(pci, mf624_pci_id)`, kde první parametr určuje typ zařízení a druhý je ukazatel na seznam typu `struct pci_device_id`.

3.3.2 Struktura `struct pci_driver`

Pro to, aby se mohl ovladač PCI zařízení stát součástí jaderného PCI subsystému, je potřeba ho do něj zaregistrovat. To se provede voláním funkce `pci_register_driver()`, které se jako parametr předá ukazatel na strukturu `struct pci_driver`.

Tato struktura obsahuje základní informace o ovladači. Mezi hlavní položky patří:

`const char name*`
: Název ovladače. Tento název by měl být unikátní mezi všemi ovladači PCI zařízení. Většinou je totožný s názvem modulu.

`const struct pci_device_id *id_table`
: Pole struktur popisujících, pro která zařízení je ovladač vytvořen (viz kap. 3.3.1).

`int (*probe) (struct pci_dev *dev, const struct pci_device_id *id)`
: Ukazatel na funkci, která je volána PCI subsystémem, v případě, že je přítomno zařízení, pro které je tento ovladač vytvořen.

`void (*remove) (struct pci_dev *dev)`
: Ukazatel na funkci, která je volána poté, co je tento ovladač odstraňován ze seznamu ovladačů aktuálně používaných PCI subsystémem nebo v případě, že dochází k uvolnění modulu.

Příklad, jak může být struktura `pci_driver` inicializována a následně zaregistrována:

```
1 |  static struct pci_driver mf624_pci_driver = {
2 |      .name = "mf624",
3 |      .id_table = mf624_pci_id,
4 |      .probe = mf624_pci_probe,
5 |      .remove = mf624_pci_remove,
6 |  };
7 |  pci_register_driver(&mf624_pci_driver);
```

3.3.3 Funkce `probe()`

```
int (*probe) (struct pci_dev *dev, const struct pci_device_id *id);
```

Funkce `probe()` náležící danému ovladači zařízení je volána poté, co jaderný subsystém PCI zařízení zjistí, že se v systému nachází zařízení, pro které je tento ovladač určen. Tato funkce má na starosti inicializaci zařízení.

Prvním parametrem funkce předává PCI subsystém ukazatel na strukturu `struct pci_dev`, která reprezentuje fyzické zařízení. V druhém parametru je předán ukazatel na strukturu, na základě které byl zvolen daný ovladač (viz kap. 3.3.1).

```
pci_enable_device(struct pci_dev *dev);
```

V rámci inicializace ovladače je nejprve potřeba zavolat funkci `pci_enable_device()` – ta se postará o inicializaci karty na úrovni hardware – např. přiřazení linky přerušení, zresetování registrů karty a její probuzení. Poté je již možné začít přistupovat ke zdrojům zařízení.

3.3.4 Přístup ke zdrojům karty

Jak bylo popsáno v kapitole 2.2.3, PCI zařízení může využívat až 6 paměťových nebo vstupně-výstupních regionů (označovaných jako *zdroje* karty). Jejich alokace do paměťového nebo I/O prostoru počítače je zajištěna dynamicky PCI mostem. Pro přístup do regionů si musí ovladač zařízení zjistit jejich adresu a vyžádat si u operačního systému *výlučný přístup*.

```
int pci_request_regions(struct pci_dev *pdev, const char *res_name);
```

Nejprve je potřeba operační systém požádat o výlučný přístup ke zdrojům zařízení. To se provede voláním funkce `pci_request_regions()`. Je-li návratová hodnota zavolané funkce negativní, není ovladači umožněn přístup (jiný ovladač přistupuje ke stejné kartě nebo po jeho odstranění nedošlo k uvolnění zdrojů karty). V takovém případě by ovladač měl korektním způsobem ukončit svoji funkci a nesnažit se k zařízení přistupovat.

```
unsigned long pci_resource_start(struct pci_dev *dev, int bar);
```

V případě, že volání `pci_request_regions()` proběhlo úspěšně, je již možné získat přístup přímo k jednotlivým regionům karty. Fyzickou adresu jednotlivých regionů lze zjistit voláním funkce `pci_resource_start()`, kde se jako druhý parametr uvede číslo BAR registru určujícího region (tj. 0–5).

```
unsigned long pci_resource_len(struct pci_dev *dev, int bar);
```

V případě, že je potřeba zjistit velikost daného paměťového nebo I/O regionu, slouží k tomu funkce `pci_resource_len()`.

```
void __iomem *pci_ioremap_bar(struct pci_dev *pdev, int bar);
```

S ukazatelem, který vrátí funkce `pci_request_regions()` však není možné přímo pracovat – je to totiž **fyzická adresa** daného regionu, ke které neumí procesor přímo přistupovat. Aby tato fyzická adresa byla přemapována na adresu **virtuální**, je potřeba zavolat funkci `pci_ioremap_bar()`.

K ukazateli, který vrátí volání `pci_ioremap_bar()` je již možné pomocí speciálních funkcí (popsány v kap. 3.4) přistupovat.

3.3.5 Funkce `remove()`

```
void remove(struct pci_dev *dev);
```

Funkce je volána, když PCI subsystém ze svého seznamu odstraňuje strukturu `struct pci_dev` reprezentující dané zařízení, nebo v případě, že dochází k uvolnění modulu.

Tato funkce by se měla postarat o úklid všech naalokovaných prostředků. Měla by obsahovat volání:

`iounmap()`
: Uvolnění virtuální paměti namapované voláním `pci_ioremap_bar()`.

`pci_release_regions()`
: Uvolnění zdrojů karty, které byly zarezervovány voláním `pci_request_regions()`.

`pci_disable_device()`
: Opak k volání `pci_enable_device()`.

3.4 Přístup k paměti zařízení

Poté co se ovladači podařilo získat přístup ke zdrojům zařízení, je nutné (důvody jsou popsány dále v textu) využít speciálních volání pro zápis/čtení do/z těchto zdrojů.

3.4.1 Vstupně-výstupní adresní prostor

Stejně jako program v jazyku symbolických instrukcí využívá pro přístup k vstupně-výstupnímu adresnímu prostoru (tj. I/O portům) zvláštní instrukce, je nutné využít speciální funkce v programech psaných ve *vyšších* programovacích jazycích. V případě čtení jsou v jádře k dispozici tři volání:

```
unsigned inb(unsigned port);
```

```
unsigned inw(unsigned port);
```

```
unsigned inl(unsigned port);
```

Třetí písmeno značí o *kolika-bitové* čtení se jedná: b = 8 b, w = 16 b, l = 32 b.

Pro zápis je možné využít volání:

```
void outb(unsigned char byte, unsigned port);
```

```
void outw(unsigned char byte, unsigned port);
```

```
void outl(unsigned char byte, unsigned port);
```

Třetí písmeno, stejně jako u funkcí pro čtení, značí o kolika-bitový přístup se jedná.

Funkce se stejným *prototypem* jsou k dispozici i v uživatelském prostoru (potřebný hlavičkový soubor je `<sys/io.h>`).

3.4.2 Paměťový adresní prostor

I přesto, že se k přístupu k paměti zařízení mapované do paměťového adresního prostoru používá virtuální adresa (stejně jako v případě přístupu do operační paměti), není možné k paměti zařízení přistupovat přímo *přes ukazatel*. Důvodem je to, že buď překladač (při kompilaci) nebo procesor (za běhu) zoptimalizují[6] sekvenci zápisů/čtení do/z paměti zařízení takovým způsobem, že se výsledek může lišit od toho, jak to bylo v programu zamýšleno.

Těmto optimalizacím lze nejsnáze zabránit použitím volání pro čtení:

```
unsigned int ioread8(void *addr);
```

```
unsigned int ioread16(void *addr);
```

```
unsigned int ioread32(void *addr);
```

[6]Tyto optimalizace, v případě přístupu k operační paměti, urychlují vykonávání programu, aniž by negativně ovlivnily jeho funkci. V případě zápisu/čtení do/z registrů, u kterých mohou tyto operace vyvolávat tzv. *side effects*, již může dojít k nesprávné funkci programu.

Příklad optimalizace: V programu se do jedné paměťové buňky ihned po sobě zapíší dvě různé hodnoty, poté se výsledná hodnota přečte – optimalizace možná u klasického programu je taková, že se ve skutečnosti provede pouze druhý zápis, protože ten první nemá žádný efekt (hodnota je ihned přepsána druhým zápisem). V případě přístupu do registru zařízení může zápis například spouštět převod A/D převodníků – po optimalizaci se však provede pouze jednou, nikoliv dvakrát.

a pro zápis:

```
void iowrite8(u8 value, void *addr);
```

```
void iowrite16(u16 value, void *addr);
```

```
void iowrite32(u32 value, void *addr);
```

Číslo na konci funkce označuje o kolika-bitový přístup se jedná.

V případě, že se na paměť ve vstupně-výstupním adresním prostoru zavolá funkce

```
void *ioport_map(unsigned long port, unsigned int count);
```

nebo v případě PCI zařízení funkce

```
void *pci_iomap(struct pci_dev *dev, int bar, unsigned long maxlen);
```

se rozsah I/O portů chová jakoby byl součástí paměťového adresního prostoru. Pro přístup je poté nutné používat volání popsaná v této kapitole, která zakryjí rozdílný charakter přístupu do vstupně-výstupního adresního prostoru.

Na procesorových architekturách využívajících reorganizaci pořadí přístupu k operandům na úrovni CPU je nutné definovat i vlastní sadu operací pro tvorbu paměťových bariér. Architektury IA-32 se tyto potíže netýkají, při portaci na architektury jiné bude třeba tyto funkce podle požadavků příslušné architektury doplnit. Bohužel přenositelné hlavičkové soubory nejsou ve standardních GNU/Linuxových distribucích založených na GNU LibC pro uživatelský prostor (na rozdíl od jádra) k dispozici.

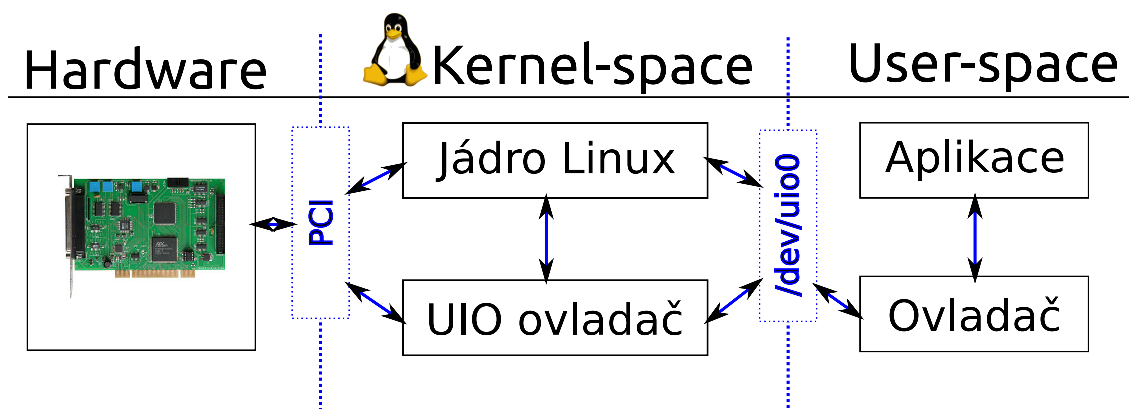
Příklad jak takové funkce (používané na architektuře IA-32) mohou vypadat:

```
1 |  static inline void mf624_write32(uint32_t val, uint32_t *ptr)
2 |  {
3 |      *(volatile uint32_t*) ptr = val;
4 |  }
5 |
6 |  static inline uint32_t mf624_read32(uint32_t *ptr)
7 |  {
8 |      return (volatile uint32_t) *ptr;
9 |  }
```

3.5 UIO ovladač

V případě, že je vytvářen ovladač pro linuxové jádro, mělo by být rozhodnuto, kterého subsystému se stane součástí – např. zda jde o jednoduché znakové zařízení, síťovou kartu nebo USB zařízení. Tato volba určí, kterou sadu pomocných funkcí bude moci ovladač používat a jakým způsobem bude zařízení zpřístupněno do uživatelského prostoru.

V případě, že zařízení nelze snadno zařadit do žádné kategorie (jedná-li se například o neobvyklou průmyslovou PCI kartu), je možné vytvořit tzv. UIO (*Userspace I/O*) ovladač. Tento ovladač se skládá ze dvou částí: jednoduchého jaderného modulu a aplikace v uživatelském prostoru (viz diagram na obrázku 3.1).



Obrázek 3.1: Diagram znázorňující funkci UIO ovladače

Mezi jeho hlavní výhody patří to, že v jádře je obsažena pouze malá obecná část, která zpřístupňuje zdroje zařízení do uživatelského prostoru. Druhou částí je aplikace v uživatelském prostoru, která přistupuje k jednotlivým zdrojům karty a tvoří hlavní logiku ovladače. Většina vývoje tedy probíhá v uživatelském prostoru, čímž klesá riziko narušení stability jádra.

3.5.1 Jaderný modul

Jaderný modul UIO ovladače PCI zařízení by měl obsahovat:

- Funkci volanou PCI subsystémem při registraci ovladače
  - Volání funkcí pro namapování regionů zařízení
  - Inicializaci struktury `struct uio_info` a registraci do UIO subsystému
- Funkce pro *úklid* a uvolnění regionů karty

Většina z těchto úkonů již byla popsána v kapitole 3.3 a jsou zcela standardní pro jakýkoliv ovladač PCI zařízení. Co nebylo dosud popsáno je pouze úkon *registrace do UIO subsystému*.

```
int uio_register_device(struct device *parent, struct uio_info *info);
```

Registrace UIO ovladače PCI zařízení se provede zavoláním funkce `uio_register_device()`, které se jako první parametr předá ukazatel na strukturu obecného zařízení `dev` vnořenou do struktury `struct pci_dev`. Důvod je ten, že ovladač typu UIO může být vytvořen i pro jiná zařízení než ta na sběrnici PCI. Druhý parametr předá ukazatel na strukturu `struct uio_info`.

3.5.1.1 Struktura `struct uio_info`

Jedná se o strukturu vyplněnou informacemi o zařízení, která je předána při registraci UIO subsystému. Mezi její hlavní položky patří:

`const char *name`
: Název ovladače. Většinou se shoduje s názvem modulu.

`const char *version`
: Verze ovladače v textové podobě.

`struct uio_mem mem[MAX_UIO_MAPS]`
: Pole struktur obsahujících informace o regionech zařízení mapovaných do paměťového prostoru (bude vysvětleno dále).

`struct uio_port port[MAX_UIO_PORT_REGIONS]`
: Pole struktur obsahujících informace o regionech zařízení mapovaných do vstupně-výstupního prostoru (bude vysvětleno dále).

3.5.1.2 Struktura `struct uio_mem` a `struct uio_port`

Tyto struktury obsahují informace o regionech zařízení. Které (a kolik) z těchto dvou struktur budou inicializovány záleží na tom, zda zařízení mapuje regiony do paměťového nebo vstupně-výstupního prostoru.

Struktura `struct uio_mem` obsahuje položky:

`const char *name`
: Textový popis daného regionu (viditelný z uživatelského prostoru).

`unsigned long addr`
: Fyzická adresa regionu. V případě PCI zařízení získána voláním `pci_resource_start()`.

`unsigned long size`
: Délka regionu. V případě PCI zařízení nejsnáze získána voláním `pci_resource_len()`.

`int memtype`
: Typ paměti. Pro fyzickou paměť na zařízení se použije `UIO_MEM_PHYS`.

`void __iomem *internal_addr`
: Virtuální adresa. V případě PCI zařízení získána voláním `pci_ioremap_bar()`.

Struktura `struct uio_port` obsahuje položky:

`const char *name`
: Textový popis daného regionu (viditelný z uživatelského prostoru).

`unsigned long start`
: Fyzická adresa regionu. V případě PCI zařízení získána voláním `pci_resource_start()`.

`unsigned long size`
: Délka regionu. V případě PCI zařízení nejsnáze získána voláním `pci_resource_len()`.

`int porttype`
: Typ portu (tj. vstupně-výstupní paměti). Pro porty na architektuře IA-32 se použije `UIO_PORT_X86`.

Příklad, jak taková jednoduchá inicializace struktury `struct uio_info` včetně registrace může vypadat (bez ošetření chybových stavů):

```
 1 |  /* struct pci_dev *dev */
 2 |  struct uio_info *info;
 3 |  info = kzalloc(sizeof(struct uio_info), GFP_KERNEL);
 4 |
 5 |  info->name = "mf624";
 6 |  info->version = "0.0.1";
 7 |
 8 |  info->mem[0].name = "PCI chipset, ...";
 9 |  info->mem[0].addr = pci_resource_start(dev, 0);
10 |  info->mem[0].size = pci_resource_len(dev, 0);
11 |  info->mem[0].memtype = UIO_MEM_PHYS;
12 |  info->mem[0].internal_addr = pci_ioremap_bar(dev, 0);
13 |
14 |  info->port[0].name = "Board programming registers";
15 |  info->port[0].porttype = UIO_PORT_X86;
16 |  info->port[0].start = pci_resource_start(dev, 1);
17 |  info->port[0].size = pci_resource_len(dev, 1);
18 |
19 |  uio_register_device(&dev->dev, info);
20 |  pci_set_drvdata(dev, info);
```

```
void pci_set_drvdata(struct pci_dev *pdev, void *data);
```

Na posledním řádku je, dosud nepopsané, volání `pci_set_drvdata()`. To (v tomto případě) zajistí, že struktura `struct uio_info` se stane součástí struktury reprezentující zařízení (`struct pci_dev`) – což umožní pozdější přístup ke struktuře `struct uio_info` z funkcí jako je například `remove()`, která jako parametr získá ukazatel na strukturu `struct pci_dev`.

```
static inline void *pci_get_drvdata(struct pci_dev *pdev);
```

Funkce `pci_get_drvdata()` slouží k *získání* dat uložených do struktury `struct pci_dev` pomocí volání `pci_set_drvdata()`.

Příklad použití:

```
1 |  static void mf624_pci_remove(struct pci_dev *dev)
2 |  {
3 |      struct uio_info *info = pci_get_drvdata(dev);
4 |      /* ... */
5 |  }
```

3.5.2 Program v uživatelském prostoru

Poté, co je jaderná část UIO ovladače úspěšně zkompilována a zavedena do systému, ve kterém se nachází požadované zařízení, je rozhraní mezi tímto modulem a uživatelským prostorem tvořeno:

- souborem `/dev/uio0`[7].
- složkou `/sys/class/uio/uio0`, která obsahuje informace o regionech, které jsou zpřístupněny skrze UIO modul v jádře.

3.5.2.1 Obsah složky `/sys/class/uio/uio0`

Tato složka obsahuje soubory převážně pouze pro čtení. Obsahuje podsložku `maps`, ve které se nachází pro každý region zařízení mapovaný do paměti (zpřístupněný jaderným ovladačem) složka obsahující soubory popisující tyto regiony (Soubor `addr` obsahuje fyzickou adresu regionu; `name` slovní pojmenování; `size` velikost regionu).

V případě, že jsou zpřístupněny regiony zařízení, které jsou mapovány do vstupně-výstupního adresního prostoru, nacházejí se jednotlivé podsložky a soubory popisující regiony ve složce `portio`.

[7] Pro názornost je v textu uvedeno konkrétní zařízení `uio0`. V případě, že systém obsahuje více aktivních UIO ovladačů, jsou postupně číslovány od 0 výše.

3.5.2.2 Soubor `/dev/uio0`

Tento soubor tvoří rozhraní mezi jaderným subsystémem UIO a uživatelským prostorem. Skrze něj je přistupováno k regionům zařízení. K souboru se přistupuje pomocí funkce `mmap()`.

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd,
off_t offset);
```

Tato funkce slouží k *namapování* souboru nebo zařízení do operační paměti. V případě, že je funkce zavolána na soubor, proběhne-li vše správně, návratová hodnota bude obsahovat ukazatel do paměti, kam je možné přistupovat k obsahu souboru pomocí ukazatelové aritmetiky – stejně, jako by to byla paměť.

Popis jednotlivých parametrů:

`addr`
: V případě, že není nulový, určí na jakou adresu ve virtuálním adresním prostoru aplikace by měla být paměť mapována. Není-li adresa určena, volnou oblast vybere podpůrná C knihovna (LibC).

`length`
: Udává velikost mapované paměti v násobcích velikosti paměťové stránky.

`prot`
: Obsahuje příznaky definující, zda bude mapovaná paměť pro čtení/zápis, apod.

`flags`
: Pomocí příznaků určuje, zda se mají změny zapisovat pouze do *lokální kopie* (příznak `MAP_PRIVATE`) nebo zda mají být zapisovány do původního souboru/zařízení (příznak `MAP_SHARED`).

`fd`
: Obsahuje *filedescriptor* na zařízení, které má být namapováno (v tomto případě filedescriptor vrácený voláním `open("/dev/uio0", ... );`).

`offset`
: Určuje, zda se daný soubor/zařízení začne mapovat od posunuté adresy. V případě UIO ovladače je možné jako offset používat násobky velikosti paměťové stránky – tento offset určí, který z regionů zpřístupněných jadernou částí ovladače má být namapován.

Příklad, jak takové volání může vypadat (bez ošetření chybových stavů):

```
1 |  #define BAR2_offset      (1 * sysconf(_SC_PAGESIZE))
2 |
3 |  void* mf624_BAR2 = NULL;
4 |  int device_fd = open("/dev/uio0", O_RDWR);
5 |
6 |  mf624_BAR2 = mmap(0, 1 * sysconf(_SC_PAGESIZE),
7 |                    PROT_READ | PROT_WRITE, MAP_SHARED,
8 |                    device_fd, BAR2_offset);
```

S adresou vrácenou voláním `mmap()` však není možné vždy ihned pracovat. Může se stát, že mapovaný region zařízení (reprezentovaný zařízením `/dev/uio0`, na které je `mmap()` volán) je menší než je velikost celé stránky, `mmap()` však vrací ukazatel zarovnaný na velikost stránky. Je tedy potřeba se v rámci této stránky posunout na tu část, která odpovídá požadovanému regionu.

Jak velký je potřeba udělat *posun* pomůže zjistit soubor `/sys/class/uio/uio0/maps/map1/addr`[8] – ten obsahuje fyzickou adresu požadovaného regionu. Z té je možné následujícím trikem získat ukazatel, se kterým je již možné pracovat (nejnižší bity totiž budou zachovány z fyzické adresy):

```
mf624_BAR2 += (BAR2_phys_addr & (sysconf(_SC_PAGESIZE) - 1));
  |                \-- Fyzická adresa
   \-- Ukazatel vrácený voláním mmap()
```

3.5.2.3 Přístup k paměti zařízení

Jelikož se jedná o paměť zařízení, je potřeba i v uživatelském prostoru k této paměti přistupovat pomocí speciálních funkcí. Ty jsou popsány v kapitole 3.4.

[8]Pro názornost je uvedena konkrétní cesta – jedná se tedy o *druhý* paměťový region zařízení *uio0*.

3.6 Comedi ovladač

Kromě popsaných výhod UIO ovladače jsou zde i nevýhody. Jednou z nich je pomalejší odezva než v případě plnohodnotného jaderného ovladače. Další nevýhodou je neexistence knihovny v uživatelském prostoru, která by poskytovala jednotné API pro přístup k zařízení (v případě UIO ovladače je potřeba vytvořit pro každé zařízení specifický program).

Pro ovladače měřících karet existuje v Linuxu subsystém – tzv. Comedi (*Control and Measurement Device Interface*).

Comedi se skládá ze tří částí:

Comedi – jsou jednotlivé nízkoúrovňové ovladače zařízení, včetně hlavního ovladače `comedi`, který poskytuje základní funkce.

Comedilib – je knihovnou v uživatelském prostoru, která poskytuje jednotné rozhraní pro ovládání jednotlivých zařízení.

Kcomedilib – je jaderný modul, který poskytuje stejné rozhraní jako Comedilib v uživatelském prostoru. Používá se v případě potřeby ovládat zařízení v reálném čase.

3.6.1 Registrace ovladače

Pro správnou funkci je potřeba, aby byl ovladač ihned po načtení modulu (tj. v *init* funkci) zaregistrován – jak do PCI subsystému, tak do subsystému Comedi. Registrace do PCI subsystému je popsána v kapitole 3.3.2. Registrace mezi Comedi ovladače se provede voláním `comedi_driver_register()`, kde jako parametr se předá ukazatel na strukturu `struct comedi_driver`.

3.6.2 Struktura `struct comedi_driver`

Jednotlivé položky struktury `struct comedi_driver` popisují daný ovladač. Mezi nejdůležitější položky patří:

`const char *driver_name;`
Obsahuje textový název ovladače.

`struct module *module;`
Ukazatel na modul, kterému tato struktura náleží. Inicializuje se makrem `THIS_MODULE`.

`int (*attach) (struct comedi_device *, struct comedi_devconfig *);`
Ukazatel na funkci, která má být zavolána při aktivaci ovladače.

`int (*detach) (struct comedi_device *);`
Ukazatel na funkci, která má být zavolána při deaktivaci ovladače.

Na rozdíl od předchozích příkladů je v tomto případě tou hlavní *inicializační* funkcí nikoliv funkce `probe()` volaná PCI subsystémem v případě, že se v systému nachází hardware, který umí ovladač obsloužit, ale funkce `attach()`, která je volána Comedi subsystémem v závislosti na tom, zda má být ovladač použit nebo ne.

3.6.3 Funkce `attach`

Funkce `attach` je volána v případě aktivace Comedi ovladače. Dříve než dojde na popis inicializačních kroků je nutné vysvětlit názvosloví, které je u Comedi ovladačů používáno.

Board označuje konkrétní zařízení – měřící kartu. Některé ovladače podporují celou sadu zařízení (např. od stejného výrobce).

Subdevice (*podzažízení*) je jedna z mnoha funkcí zařízení. V případě ovladače karty Humusoft MF614 budou implementovány 4 podzařízení: digitální vstupy, digitální výstupy, analogové vstupy, analogové výstupy. Každé z těchto podzařízení bude schopno obsluhovat více *kanálů*.

Kromě obvyklých operací, jako je *aktivace zařízení*, žádos o *výhradní přístup* ke zdrojům zařízení a *mapování* paměťových nebo I/O regionů, popsaných v kapitole 3.3, je nutné alokovat a inicializovat struktury `struct comedi_subdevice` odpovídající jednotlivým podzařízením.

3.6.4 Struktura `struct comedi_subdevice`

Každé podporované funkci zařízení (tj. měřící karty) by měla odpovídat jedna struktura `struct comedi_subdevice`. Hlavní položky, které struktura obsahuje jsou:

`int type`
: Označuje druh *podzařízení*. Na výběr jsou např. možnosti: `COMEDI_SUBD_AI` (analogový vstup), `COMEDI_SUBD_AO` (analogový výstup), `COMEDI_SUBD_DI` (digitální vstup), `COMEDI_SUBD_DO` (digitální výstup).

`int subdev_flags`
: Označuje základní vlastnost podzařízení. Nejpoužívanější hodnoty: `SDF_READABLE` (z podzařízení může být čteno), `SDF_WRITABLE` (do podzařízení může být zapisováno).

`int n_chan`
: Počet kanálů podzařízení (např. pro 8 digitálních vstupů bude tato hodnota 8).

`unsigned int maxdata`
: Maximální hodnota, která může být do podzařízení zapsána/čtena.

`const struct comedi_lrange *range_table`
: Označuje rozsah, ve kterém dané podzařízení měří (např. u A/D převodníku 0–10 V). K dispozici jsou definované struktury (stačí pouze předat ukazatel na některou z nich):
`range_digital`,
`range_bipolar10`,
`range_bipolar5`,
`range_unipolar10`,
`range_unipolar5`.
Jejich názvy jsou samovysvětlující.

`int (*insn_read) ( ... );`
: Ukazatel na funkci, která má na starosti čtení z podzařízení (většinou se používá pro A/D převodníky).

`int (*insn_write) ( ... );`
: Ukazatel na funkci, která má na starosti zápis do zařízení (většinou se používá pro D/A převodníky).

`int (*insn_bits) ( ... );`
: Ukazatel na funkci použitou pro zápis a čtení digitálních výstupů a vstupů,

`int (*insn_config) ( ... );`
: Ukazatel na funkci, která má na starosti konfiguraci podzařízení.

Poslední čtyři funkce mají parametry:
`(struct comedi_device *dev, struct comedi_subdevice *s, struct comedi_insn *insn, unsigned int *data)`. První z nich je ukazatel na strukturu popisující Comedi ovladač. Druhý je ukazatelem na strukturu odpovídající podzařízení. Třetí obsahuje ukazatel na strukturu popisující *instrukci*, která má být provedena. Poslední obsahuje ukazatel na proměnnou, ze které je vyčtena zapisovaná hodnota nebo je do ni čtená hodnota zapsána.

```
int alloc_subdevices(struct comedi_device *dev, unsigned int
num_subdevices);
```

Alokace paměti pro struktury se provede voláním `alloc_subdevices()`, které je poskytováno Comedi subsystémem. Prvním parametrem je předán ukazatel na strukturu `struct comedi_device`, pro kterou má být alokace provedena. Alokovaná paměť je přístupná skrze proměnnou `subdevices` náležící struktuře `struct comedi_device`.

V případě dealokace zdrojů ovladače není potřeba tuto paměť dealokovat – o uvolnění paměti se postará Comedi subsystém.

3.6.5 Funkce pro čtení a zápis z/do podzařízení

Funkce pro čtení, zápis a konfiguraci A/D, D/A převodníků a digitálních vstupů a výstupů mají stejné parametry. Jsou to: `(struct comedi_device *dev, struct comedi_subdevice *s, struct comedi_insn *insn, unsigned int *data)`.

V prvním parametru je předán ukazatel na strukturu reprezentující Comedi zařízení. Díky tomu je možné prostřednictvím její proměnné `private` získat ukazatel na strukturu obsahující privátní data ovladače.

Druhý parametr je ukazatel na strukturu reprezentující podzařízení. Tato struktura obsahuje, kromě položek inicializovaných ve funkci *attach* i proměnnou `state`. Tato proměnná popisuje *stav zařízení* a používá se především pro zjištění stavu digitálních výstupů (stav digitálních výstupů většinou není možné ze zařízení přečíst, pro změnu pouze jednoho bitu je tedy potřeba znát stav ostatních).

Třetí parametr obsahuje ukazatel na strukturu popisující danou *instrukci*, která má být provedena. Důležité položky, které tato struktura obsahuje:

`unsigned int n`
: Udává počet instrukcí, které mají být provedeny.

`unsigned int chanspec`
: Obsahuje informace o kanálu podzařízení, na kterém má být operace provedena. V jedné proměnné typu `unsigned int` je obsaženo více údajů, proto je potřeba ke čtení používat speciální makro `CR_CHAN()`, které vrací číslo zvoleného kanálu.

Čtvrtý parametr obsahuje ukazatel na pole zapisovaných/čtených položek. Počet prvků pole odpovídá proměnné `n` struktury `struct comedi insn`. V případě čtení A/D převodníku obsahuje opakovaně čtené položky. Podobně to platí pro nastavování hodnoty D/A převodníku. V případě čtení/zápisu digitálních vstupů/výstupů má toto pole pouze dva prvky. Čtenou/zapisovanou hodnotu obsahuje položka `data[1]`. Položka `data[0]` obsahuje jako binární masku zadané kanály čtených/ zapisovaných digitálních vstupů/výstupů.

Příklad, jak může být implementováno čtení digitálních vstupů:

```
 1 |  static int mf614_di_insn_bits(struct comedi_device *dev,
 2 |                                struct comedi_subdevice *s,
 3 |                                struct comedi_insn *insn,
 4 |                                unsigned int *data)
 5 |  {
 6 |      if(insn->n != 2) {
 7 |          return -EINVAL;
 8 |      }
 9 |
10 |      data[1] = ioread8(devpriv->BAR0_io + DIN_reg);
11 |
12 |      return 2;
13 |  }
```

3.6.6 Funkce `detach`

Tato funkce je volána, jak v případě ukončení funkce ovladače, tak v případě, že funkce *attach* neproběhla v pořádku. Proto je potřeba rozlišit, které zdroje ovladače již byly úspěšně naalokovány a mají být uvolněny.

Odregistrování ovladače z PCI a Comedi subsystému by mělo být voláno v *úklidové funkci* modulu. O samotné odregistrování se starají funkce: `pci_unregister_driver()` a `comedi_driver_unregister()`, kterým se jako parametr předá ukazatel na strukturu použitou při registraci.

3.6.7 Přístup z uživatelského prostoru

Pro správnou funkci konkrétního Comedi ovladače je nejprve potřeba načíst modul Comedi (`modprobe comedi`). Poté je již možné načíst ovladač zařízení (v případě ručně kompilovaného ovladače, pomocí příkazu `insmod`, jinak opět pomocí `modprobe`).

V případě, že proběhlo načtení modulu a spuštění funkce *attach* bez problémů, měl by se ve složce `/dev` objevit nový soubor odpovídající načtenému ovladači zařízení – `comedi0`[9] K tomuto souboru je poté možné pomocí knihovních funkcí Comedilib přistupovat.

Pro přístup k zařízení je potřeba zavolat na soubor `/dev/comedi0` funkci `comedi_open()`. Ta vrací ukazatel datového typu `comedi_t`, reprezentující dané zařízení. K němu je možné přistupovat pomocí funkcí: `comedi_data_read()`, `comedi_data_write()`, `comedi_dio_read()`, `comedi_dio_write()` a jiných.

První dvě slouží pro zápis/čtení A/D a D/A převodníků, zatímco poslední dvě slouží pro přístup k digitálním vstupům/výstupům. Prvním parametrem všech funkcí je ukazatel na `comedi_t` odpovídající danému zařízení. Druhým je číslo *podzařízení*. Třetí parametr určuje kanál (tj. např. který z osmi A/D převodníků má být čten). Posledním parametrem je ukazatel na proměnnou, kam mají být zapsána přečtena data nebo hodnota, která má být zapsána.

Ukázka jednoduchého userspace programu:

```
 1 |  #include <stdio.h>
 2 |  #include <comedilib.h>
 3 |  #define MF614_DO_SUBDEV      1 /* Je potřeba vědět, jak je
 4 |                                   implementováno v ovladači */
 5 |
 6 |  int main(int argc, char* argv[])
 7 |  {
 8 |      comedi_t* comedi_dev;
 9 |
10 |      comedi_dev = comedi_open("/dev/comedi0");
11 |      if (comedi_dev == NULL) {
12 |          comedi_perror("comedi_open");
13 |          return 1;
14 |      }
15 |
16 |      /* Zápis 1 na 0. kanál digitálního výstupu */
17 |      comedi_dio_write(comedi_dev, MF614_DO_SUBDEV, 0, 1);
18 |      sleep(1);
19 |      comedi_dio_write(comedi_dev, MF614_DO_SUBDEV, 0, 0);
20 |      sleep(1);
21 |      comedi_dio_write(comedi_dev, MF614_DO_SUBDEV, 0, 1);
22 |
23 |      return 0;
24 |  }
```

Při kompilaci je potřeba použít parametry `-lcomedi -lm`.

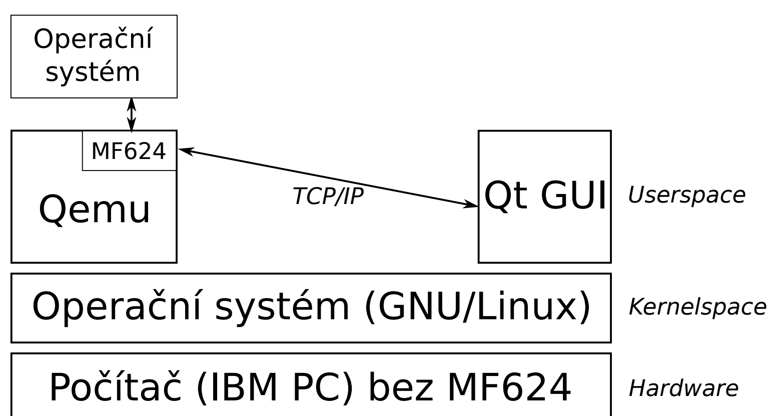
[9] Pro názornost je uveden konkrétní příklad, v případě načtení více ovladačů budou odpovídající soubory číslovány od 0 výše.

Kapitola 4

Implementace karty Humusoft MF624 v Qemu

Měřící karta Humusoft MF624 je hardware vhodný pro výklad principů implementace ovladačů PCI zařízení. Nevýhodou může být její cena. Pro účely výuky proto byly do emulačního software Qemu implementovány základní funkce této měřící karty – konkrétně se jedná o A/D převodníky, D/A převodníky a digitální vstupy a výstupy. Takto modifikovaná verze qemu může pří implementaci základního ovladače plně nahradit původní kartu.

Kromě částečné implementace karty MF624 do Qemu je součástí tohoto *virtuálního hardware* grafická aplikace, která má na starosti nastavování vstupních hodnot a zobrazování výstupních hodnot do/z karty (obr. 4.1).



Obrázek 4.1: Diagram znázorňující princip funkce implementované karty MF624 v Qemu

4.1 Qemu

Qemu je emulátor různých procesorových architektur. Od klasických virtualizačních nástrojů se odlišuje tím, že podporuje kromě IA-32 architektury také např. ARM, SPARC,

PowerPC, MIPS, m68k. Qemu umožňuje kromě *plné emulace* (kdy je spuštěn celý operační systém) tzv. *uživatelskou emulaci*, kdy je v uživatelském prostoru spuštěn program zkompilovaný pro jinou architekturu. Uživatelská emulace je možná pouze pro operační systém GNU/Linux.

4.1.1 Kompilace, instalace

Po stažení a rozbalení zdrojových kódů některé ze stabilních verzí emulátoru Qemu je potřeba spustit příkaz (pro emulaci architektury IA-32):

```
$ ./configure --enable-system --target-list=i386-softmmu
```

Neohlásí-li spuštěný skript žádné chybějící knihovny, je možné spustit kompilaci:

```
$ make
```

4.1.2 Kompilace virtuální karty Humusoft MF624

V případě, že je potřeba zkompilovat virtuální kartu MF624, je nejprve potřeba překopírovat zdrojový soubor implementující zařízení do složky `/hw` a do souboru `Makefile.objs` (nachází se v kořenovém adresáři se zdrojovými kódy) přidat řádek:

```
hw-obj-$(CONFIG_PCI) += mf624.o
```

Poté je již možné spustit příkaz:

```
$ make
```

4.1.3 Použití

Zkompilovaný binární soubor se nachází v adresáři `i386-softmmu`. Nejnutnější parametr při spuštění je `-hda`, který uvádí cestu k souboru reprezentujícím *obraz* spouštěného systému.

V případě správně zkompilované virtuální karty MF624, je možné ji spustit zadáním parametru `-device mf624`. Po spuštění je v příkazové řádce vypsáno číslo TCP/IP portu, na kterém virtuální karta MF624 naslouchá. Tento port slouží k připojení klientského programu, který má na starosti vykreslování výstupních a nastavování vstupních hodnot karty (možná implementace je popsána v kapitole 4.2). V případě neexistence klientského software je možné se k virtuální kartě připojit pomocí programu `telnet`.

Příklad spuštění:

```
$ ./qemu -device mf624 -hda ../os_images/debian.qcow --boot c
MF624 Loaded.
Waiting on port 55555 for MF624 client to connect
Client connected
```

Příklad ovládání vstupů a zobrazování výstupů karty pomocí aplikace `telnet`:

```
$ telnet localhost 55555
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
DA1=9.998779
DOUT=255.000000
DOUT=0.000000
DA1=5.000000
^]
telnet> Connection closed.
```

4.2 Qt grafiké rozhraní

Pro komunikaci s virtuální kartou MF624 bylo implementováno jednoduché grafické rozhraní, které má na starosti vykreslování hodnot výstupů karty (nastavovaných ovladačem běžícím v operačním systému virtualizovaném Qemu) a posílání nastavovaných vstupních hodnot zpět virtuální kartě.

Komunikace mezi virtuální kartou a grafickou aplikací probíhá pomocí TCP/IP protokolu. Přenášené informace jsou textového charakteru, ve formátu `REGISTR=HODNOTA`.

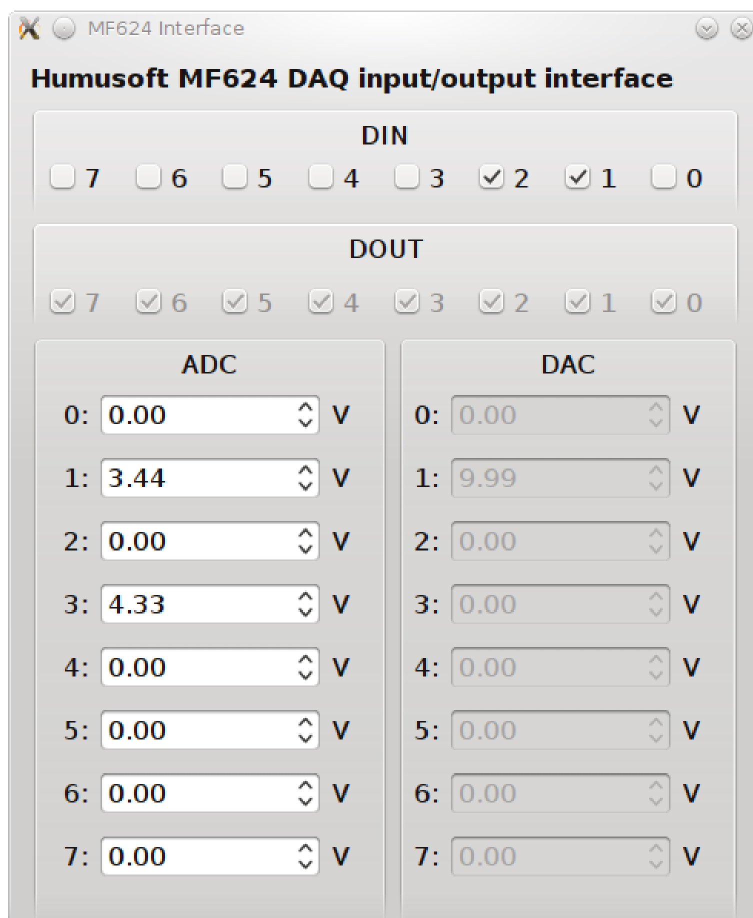
Na obrázku 4.2 je snímek obrazovky implementované grafické aplikace.

4.2.1 Kompilace, použití

Grafická aplikace je vytvořena za pomoci knihovny Qt. V případě, že je v systému nainstalována vývojářská verze Qt knihovny, včetně potřebných vývojářských nástrojů, stačí pro kompilaci spustit:

```
$ qmake
$ make
```

Jako parametr při spuštění je nutné zadat číslo portu, na kterém naslouchá virtuální karta MF624. Použití aplikace by mělo být intuitivní. Položky, u kterých není možné měnit jejich hodnotu, jsou záměrně pouze pro čtení (zobrazují výstupní hodnoty).



Obrázek 4.2: Vzhled grafické aplikace pro ovládání vstupů a výstupů virtuální karty MF624

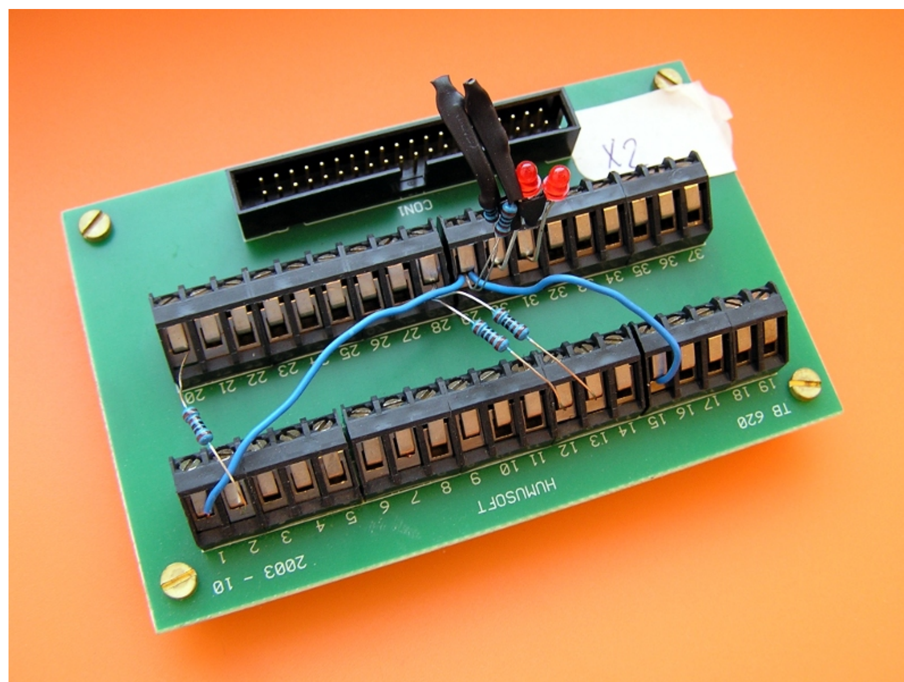
Kapitola 5

Testování

5.1 UIO ovladač, Comedi ovladač

Jednotlivé ovladače jsou tvořeny samostatnými jadernými moduly, které pouze využívají funkce jednotlivých subsystémů – neexportují žádné *symboly* ani nemění globální proměnné. V tomto případě nebylo nutné provádět regresní testování.

Testování správnosti funkce ovladačů probíhala přímo na hardware, za pomoci *univerzální svorkovnice TB620* (obrázek 5.1).



Obrázek 5.1: Svorkovnice TB620

Základní propojení na svorkovnici, které se osvědčilo, bylo:

- 2 × LED pro nejnižší a nejvyšší bit DOUT
- 2 × 1kΩ rezistory mezi 5 V a nejnižším a nejvyšším bitem DIN
- 2. bit DIN dynamicky spojován s GND nebo pomocí 1kΩ rezistoru s 5 V
- Měření multimetrem výstupní hodnoty z DAC (většinou DAC0 nebo DAC1)
- ADC0 spojen s GND, ADC1 spojen pomocí 1kΩ rezistoru s DAC0

Konzistence jádra byla testována opětovným načítáním a uvolňováním jednotlivých ovladačů.

5.2 Qemu virtuální hardware, Qt grafické rozhraní

Správnost implementace virtuálního hardware byla testována spouštěním ovladačů (testovaných na skutečném hardware) v systému virtualizovaném v Qemu. Zároveň byla ověřena funkčnost grafického rozhraní, reprezentujícího vstupy a výstupy do/z virtuální karty.

Kapitola 6

Závěr

V této práci se mi podařilo vysvětlit základní aspekty psaní ovladačů PCI zařízení pro operační systém GNU/Linux – jak na obecné úrovni, tak u konkrétních ovladačů typu UIO a Comedi.

Součástí práce jsou základní (pokrývající pouze A/D, D/A převodníky a digitální vstupy/výstupy) ovladače pro karty Humusoft MF624 a Humusoft MF614. V budoucnu by tyto ovladače mohly být rozšířeny tak, aby pokrývaly všechny funkce těchto karet.

Pro potřeby výuky byly implementovány základní funkce karty Humusoft MF614 do emulátoru Qemu. Tato implementace by mohla být v budoucnu rozšířena, případně by mohla posloužit jako příklad pro tvorbu jiných jednoduchých PCI zařízení v Qemu, sloužících pro výuku implementace PCI ovladačů. I když tak nebylo původně zamýšleno, mohla by částečná implementace karty MF614 do Qemu posloužit i při výuce psaní ovladačů pro jiné operační systémy, jako například systémy rodiny Microsoft Windows.

Příloha A

Obsah přiloženého CD

```
.
|-- doc
|   |-- diploma_thesis  Text diplomové práce
|   `-- hw              Informace k použitým kartám MF614 a MF624
`-- src
    |-- comedi          Zdrojové kódy Comedi ovladačů pro karty MF614 a MF624
    |-- qemu            Implementace funkcí karty MF624 do Qemu
    |-- qemu_qt_gui     Grafické rozhraní pro virtuální kartu MF624 v Qemu
    `-- uio             Zdrojové kódy UIO ovladačů karet MF614 a MF624
```

Literatura

[1] Fabrice Bellard a různí autoři. Qemu emulátor (zdrojové kódy), ?–2011. git://git.qemu.org/qemu.git.

[2] Linus Torvalds a různí autoři. Jádro Linux (zdrojové kódy), 1991–2011. http://kernel.org/.

[3] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, 3rd Edition*. O'Reilly Media, 2005.

[4] David Schleef, Frank Hess, and Herman Bruyninckx. The Control and Measurement Device Interface handbook, 1998–2003. http://www.comedi.org/doc/.

[5] Greg Kroah-Hartman. *Linux Kernel in a Nutshell*. O'Reilly Media, 2006.

[6] Lukáš Jelínek. *Jádro systému Linux*. Computer Press, 2008.

[7] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, 2005.

[8] Hans-Jürgen Koch. The Userspace I/O HOWTO, 2006–2009. http://www.kernel.org/doc/htmldocs/uio-howto.html.

[9] François Poulain. Humusoft MF624 Comedi driver, 2007. http://polux.rootard.free.fr/driver_comedi_07-05-08.tar.gz.