VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
CENTRUM VÝPOČETNÍ TECHNIKY

FACULTY OF INFORMATION TECHNOLOGY
THE COMPUTER CENTER

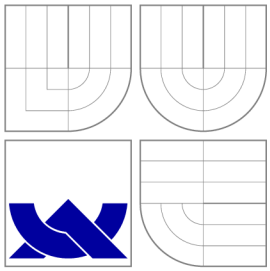
PŘENESENÍ NETFLOW/COMBO6 SONDY Z LINUXU DO FREEBSD

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

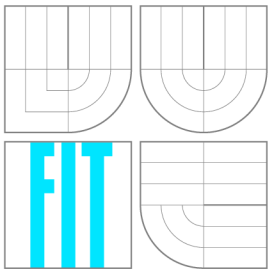
AUTOR PRÁCE
AUTHOR

PAVOL GREŠŠA

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
CENTRUM VÝPOČETNÍ TECHNIKY

FACULTY OF INFORMATION TECHNOLOGY
THE COMPUTER CENTER

PŘENESENÍ NETFLOW/COMBO6 SONDY Z LINUXU DO FREEBSD

PORTING NETFLOW/COMBO6 PROBE FROM LINUXU TO FREEBSD

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PAVOL GREŠŠA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RUDOLF ČEJKA

BRNO 2008

Abstrakt

Práce se zaobírá a řeší problém vývoje ovladačů na operační systém *FreeBSD* a rozebírá jednotlivé kroky nutné k jejich implementaci. Dále popisuje modularitu ovladačů a základní technologie používané v tomto operačním systému. Mimo jiné si klade za cíl vytvořit ovladač pro stěžejní kartu projetku *Liberouter Combo6x*. Výstupem práce by měl být kompletní ovladač s podporou DMA přenosů a přerušení.

Abstract

The thesis deals with the problem of developing device drivers for FreeBSD operation system and explains particular steps necessary for their implementation. Furthermore, it describes modularity of drivers and basic technologies used in this operation system. Its purpose is to create a device driver for the principal card *Combo6x* of the *Liberouter* project. The output of the thesis should be a complete device driver supporting DMA transfers ant interrupts.

Klíčová slova

Liberouter, ovladač zařízení, protokol NetFlow, FlowMon, FreeBSD, modul jádra, DMA, Combo6x

Keywords

Liberouter, device driver, protocol NetFlow, FlowMon, FreeBSD, kernel modul, DMA, Combo6x

Citace

Pavol Grešša: Přenesení Netflow/Combo6 sondy z Linuxu do FreeBSD, bakalářská práce, Brno, FIT VUT v Brně, 2008

Přenesení Netflow/Combo6 sondy z Linuxu do FreeBSD

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Rudolfa Čejku. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavol Grešša
19. května 2009

Poděkování

Na tomto mieste by som sa chcel poďakovať Mgr. Jiřímu Slabému za podporu a smerovanie v získavaní znalostí o projektu Liberouter. Vďaka patrí kaktiež vedúcemu práce, Ing. Rudolfovi Čejkovi za trpezlivosť.

Obsah

Kapitola 1

Úvod

1.1 Úvod do problematiky

V súčasnosti má prístup k internetu podľa serveru *http://www.internetworldstats.com* 1,57 miliard ľudí, čo tvorí zhruba štvrtinu celkovej populácie na zemi. Za týmto číslom sa skrývajú stovky gigabajtov dát, ktoré sa prenesú medzi užívateľmi internetu. Tak ako tieto čísla každý deň rastú, tak vznikajú nové nároky na rýchlu, presnú a v neposlednom rade prehľadnú analýzu stavu siete. Tá umožní predpovedať potenciálne problémy, podozrivé aktivity účastníkov a odhalenia mnohých ďalších turbulencií, ktoré ohrozujú stabilitu siete.

Jedným z takýchto nástrojov, ktorý spĺňa všetky uvedené vlastnosti efektívneho analyzátoru sieťového provozu, je séria Combo6 kariet s prídavným Netflow modulom. Tento nástroj vznikol pod záštitou projektu Liberouter[1], ktorý je zameraný na vývoj programovateľných hardwérovo akcelerovaných kariet s otvoreným návrhom (designom), ako aj firmvérom a softvérom.

Kartu Combo6 a jej prídavný modul Netflow je v tomto čase možné používať len pod operačným systémom Linux. Preto hlavným cieľom práce je preniesť ovládač a aplikačný softwér tohto nástroja z operačného systému Linux do FreeBSD. Taktiež má práca slúžiť ako jednoduchý a prostý manuál na tvorbu ovládačov zariadenia na tento operačný systém. Tento cieľ vznikol hlavne z dôvodu nedostatku informácií potrebných na vývoj vlastných ovládačov zariadení.

1.2 Štruktúra dokumentu

Úvod tohto dokumentu sa zaoberá základnými pojmami a vlastnosťami operačných systémov založených na báze UNIXu. Obsahuje tiež stručný popis technológií používaných ku komunikácii medzi hardvérovou a softvérovou časťou. Nasledujúca kapitola detailne opisuje štruktúru a jednotlivé komponenty, z ktorých sa skladá ovládač zariadenia. V ďalšej časti sa nachádza popis modulov ovládača pre Combo6x kartu a designu FlowMon. Jedna z posledných kapitol stručne zhŕňa postup, akým som portáciu vykonal. Záver dokumentu je venovaný testovaniu a ukážke použitia aplikácií, ktoré sú potrebné k práci s NetFlow sondou.

Kapitola 2

Základné pojmy

Táto kapitola poskytuje informácie o základných pojmoch používaných v oblasti operačných systémov (ďalej len OS), pričom je zameraná hlavne na OS FreeBSD a Linux. Ďalej bude čitateľ oboznámený s projektom Liberouter a jednotlivými generáciami Combo6 kariet. Nakoľko detailnejší popis kľúčových technológií nie je predmetom tejto práce, nižšie popísané oblasti OS majú len informatívny charakter. Hlbšie pochopenie problematiky je možné nájsť v [3],[6].

2.1 Úrovne privilegovanosti procesov

Vo všeobecnosti operačné systémy rozlišujú medzi niekoľkými časťami systému, do ktorých majú prístup len procesy s potrebnou úrovňou privilegovanosti. Zväčša sa používajú dve úrovne privilegovanosti, a to *USER* mód a *KERNEL* mód. Existujú aj OS, ktoré obsahujú viac úrovní[1]. Na procesy spustené v užívateľskom móde sa vzťahuje najviac obmedzení v prístupe do pamäti, nemôžu vykonávať privilegované inštrukcie, ktoré by mohli ohroziť stabilitu systému a nemajú priamy prístup na I/O zariadenie. Na rozdiel od užívateľského módu, procesy spustené v kernel móde nie sú nijakým spôsobom obmedzované.

Moderné OS obsahujú nástroje, ktorými medzi sebou tieto dva priestory komunikujú. Z *KERNEL* módu sa do *USER* módu prepína pomocou privilegovanej inštrukcie. Naopak procesy bežiace v *USER* móde komunikujú pomocou prerušení generovanými zariadeniami. OS založené na Unixovej architektúre obsahujú nástroj **ioctl**, ktorý sprostredkováva most medzi užívateľskými procesmi a I/O zariadeniami, pričom tento nástroj je popísaný v ďalších kapitolách.

2.2 Direct Memory Access(DMA)

Direct Memory Access je spôsob, akým OS umožňuje hardvérovým subsystémom, ku príkladu sieťovému adaptéru, pristupovať do operačnej pamäte takmer bez účasti procesora. V architektúre DMA jestvuje prvok *bus master*, ktorý iniciuje prenos dát a oznamuje procesoru koniec prenosu. Práve *bus master* riadi prenos, takže procesor sa môže venovať iným procesom. V prípade zbernice PCI, zariadenie, ktoré chce využívať priamy prístup do pamäte, požiada radič zbernice o jej kontrolu a samo sa stáva *bus master*-om. Naopak v prípade zbernice ISA neriadi prenos karta, ale *bus master*-om je radič zbernice.

[1] VMS má štyri: *KERNEL*, *EXECUTIVE*, *SUPERVISOR* a *USER*. Multics mal až osem úrovní

2.3 Prerušenia (IRQ)

Prerušenie je signál, ktorým zariadenie žiada procesor o obsluhu. Prerušenie vzniká asynchrónne na základe definovanej udalosti na zariadení. Ako príklad je možné uviesť stisk klávesy. Zariadenie obsahuje tzv. radič prerušení, PIC (Programmable Interrupt Controller), ktorý zabezpečuje správnu obsluhu daného prerušenia.

2.4 FreeBSD

FreeBSD[5] je slobodný Unix-like[2] operačný systém, ktorý vznikol z BSD verzie Unixu vyvinutého na Kalifornskej Univerzite v Berkeley. Kľúčové vlastností systému sú stabilita, rýchlosť a efektivita obsluhy danej úlohy. FreeBSD možno viac pokladať za serverový operačný systém než desktopový.

Nakoľko projekt Liberouter spadá do oblasti sieťových riešení, systém FreeBSD ako jeden z najróbustnejších sieťových OS by mal obsahovať ovládače a nástroje ná prácu s produktmi tohto projektu.

2.5 Projekt Liberouter

Pod záštitou združenia CESNET[2] vznikol projekt Liberouter, ktorého hlavným cieľom je rozvoj a rozšírenie moderných komunikačných technológií s podporou najnovších monitorovacích protokolov za účasti akademickej sféry. Ako hlavný produkt možno považovať sériu Combo6 kariet a ich designové moduly, ktoré svojím výkonom a flexibilitou ďaleko prevyšujú softvérové a mnohé hardvérové produkty podobného účelu. Základným kameňom projektu Combo6 bolo vytvoriť kartu, ktorá by bola otvorená v rámci svojej funkčnosti a konfigurácie, a tak poskytnúť vývojárom podobné prostredie, ako tomu je u otvoreného softvéru.

Karty Combo6 obsahujú vlastné procesory, niekoľko FPGA[3] polí, operačnú pamäť a rozhranie pre prídavné dosky. Obyčajne Combo6 karta predstavuje terminál, na ktorom sa nachádza výpočetná sila zariadenia, a prídavné moduly obsahujú rozhranie k pripojeniu periférií sieťového charakteru. Combo6 karta môže bez prítomnosti rozširujúcej dosky zastávať pozíciu koprocesora.

Samostatné dosky môžu taktiež obsahovať programovateľné polia či vlastný mikroprocesor. Následkom toho je možné dosiahnuť odlišné chovanie terminálu v závislosti na prídavnom module. Ku príkladu *COMBO-4SFPRO* obsahuje 4 SFP konektory pre GE alebo OC48, ďalej modulom *COMBO-4MTX* získáme 4 ethernetové rozhrania s konektormi metalického vedenia, a podobne. Detailnejší popis prídavných modulov je možné nájsť na [1].

Výsledná funkčnosť celého systému Combo6 karty a jej prídavných dosiek je daná vlastnosťami tzv. designu, ktorý sa nahráva do programovateľných polí. Design, možno ho nazývať aj firmvér, je preložený program v jazyku VHDL. Vďaka technoglógii programovateľných FPGA čipov a možnosti prídavných dosiek je Combo6 karta flexibilným systémom umožňujúcim vytvárať nástroje podľa aktuálnej potreby.

[2] Nie je priamym derivátom Unixu, ale pracuje ako Unix a jeho API je prispôsobené Unixovým štandardom
[3] Field Programmable Gate Array - programovateľné hradlové pole

2.5.1 Combo6

Prvá generácia karty je charakteristická využívaním zbernice PCI (32/33)[4]. Karta je osadená Xilinx FPGA (Virtex II) čipom, DRAM konektorom pre PC DDR rozšíriteľného až na 2GB. Komunikáciu medzi PCI zbernicou a kartou zabezpečuje PLX čip PCI Bus Master I/O Accelerator s kódovým označením PCI9054. Nevýhodou PLX čipu je, že má presne daný mikrokód.

2.5.2 Combo6x

Druhej generácii karty sa dostalo úpravy v podobe zámeny PLX čipu, ktorý disponoval pevne daným mikrokódom, za Virtex II s PCI core (nazývaná tiež PCI-X), obsahujúci PowerPc procesor. Touto úpravou získala karta až zdvojnásobenie maximálnej frekvencie na 133MHz, rozšírila sa dátová zbernica na 64 bitov, čím sa zvýšila dátová priepustnosť. Nemenej významnou zmenou je rozšírenie hlavného FPGA čipu o novú generáciu Virtex II PRO - XC2VP50, ktorý obsahuje ďalši procesor.

2.6 NetFlow

NetFlow je otvorený protokol vyvinutý spoločnosťou Cisco Systems. Jeho hlavným účelom je monitorovanie sieťového provozu na základe IP tokov. NetFlow architektúra pozostáva z niekoľkých NetFlow exportérov a jedného NetFlow kolektoru. NetFlow exportér je pripojený k monitorovanej linke a analyzuje prechádzajúce pakety. NetFlow kolektor zbiera dáta od exportérov a generuje štatistiky. Pomocou týchto dát je možné sledovať aktuálne problémy v sieti, detekovať útoky a podobne. V súčasnosti existuje niekoľko verzií, z ktorých sa prvou masovejšie využívanou stala práve verzia NetFlow v5. Postupne sa začína hojne využívať verzia NetFlow v9.

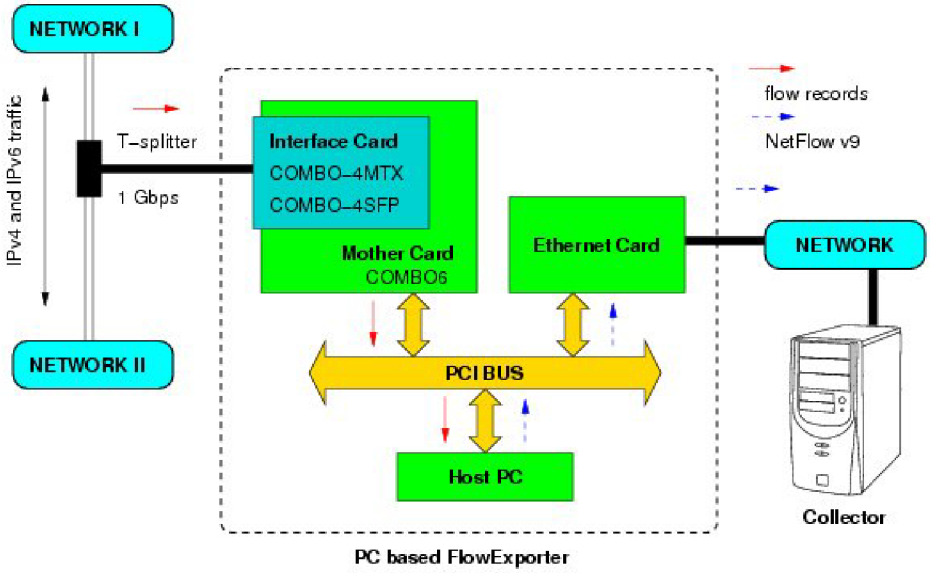
2.7 FlowMon

Flowmon[4] je projekt vyvíjaný pod hlavičkou Liberouteru, ktorý spája flexibilitu a rýchlosť kariet Combo6 s protokolom NetFlow. Výsledkom je vytvorený nový design (firmware) a súbor vlastných aplikácií k nemu. Design je navrhnutý v jazyku VHDL a po inicializácii karty sa nahraje do FPGA hradla. Na obrázku 2.1 je vidieť architektúra komponent a tok dát od vstupno-výstupných rozhraní cez kartu, PCI rozhranie až do pamäte operačného systému, kde ich spracuje aplikácia.

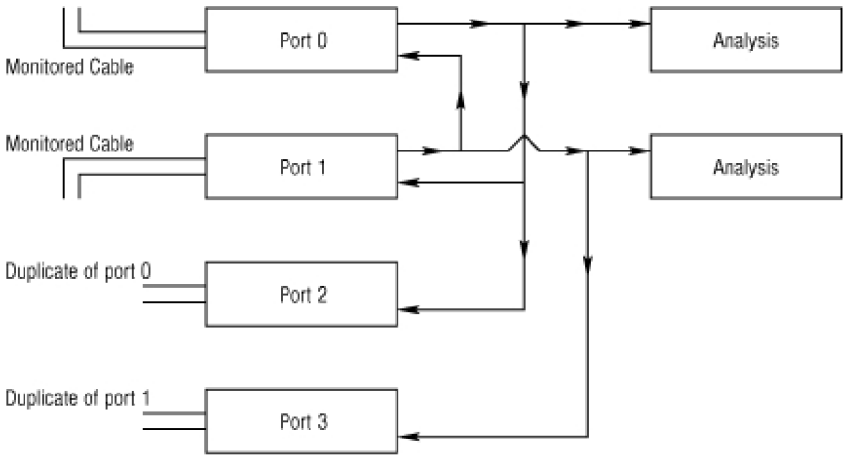
Design je prioritne navrhnutý pre dcérsku kartu so 4 vstupno-výstupnými sieťovými rozhraniami pre Gigabitový Ethernet (4-MTX a 4-SPF). Z hľadiska sieťovej architektúry vystupuje sonda ako opakovač a splitter (2.2). Dátovy tok prichádzajúci na port 0 prepína na výstupný port 1 a 2 a preposiela ďalej k analýze. Vstupný tok prichádzajúci na port 1 prepína naspäť na port 0 a tiež duplikuje na port 3 a preposiela k analýze. Z toho vyplýva, že vo firmware sa nachádzajú dva veľkokapacitné buffery, z ktorých sa generuje výsledná analýza v požadovanom NetFlow formáte, ale to už je na strane aplikáčného softvéru.

Projekt FlowMon v súčasnosti podporuje dve najpoužívanješie verzie protokolu NetFlow a to v5 a v9.

[4] bit/MHz



Obrázek 2.1: Architektúra FlowMon



Obrázek 2.2: Zapojenie sondy ako opakovača a splittera

Kapitola 3

Ovládač zariadenia v OS FreeBSD

Ovládač je zložitá softwarová komponenta, ktorá kontroluje špecifické zariadenie. Ovládač môže predstavovať jeden modul, ale môže sa skladať aj z viacerých znovu použiteľných modulov, kde každý z nich môže byť zameraný na určitú technolgógiu alebo verziu hardvéru. Táto kapitola popisuje v čo najširšom možnom zábere všetky komponenty, ktorých znalosť je potrebná k úspešnej implementácii ovládača zariadenia do operačného systému FreeBSD.

Najskôr bude čitateľ zoznámený s nástrojom KLD umožnujúcim dynamické nahrávanie modulov do jadra systému. Následne objasním význam uzlov zariadení a nástroje, pomocou ktorých aplikácie spustené v užívateľskom móde pracujú so zariadeniami. Ďalej bude popísaná časť, ktorá zodpovedá za identifikáciu hardvéru a jeho korektného zavedenia do systému. Koniec kapitoly bude zameraný na srdce samotného ovládača, ktoré má na starosti obsluhu prerušení, prácu s DMA pamäťou, interakciu medzi užívateľským a kernel priestorom.

V prílohe A je možné nájsť základnú kostru pre ovládač zariadenia.

3.1 KLD (Dynamic Kernel Linker Facility)

Nástroj KLD disponuje rozhraním umožnujúcim za behu systému dynamicky nahrávať a odoberať moduly. Moduly môžu byť rôzneho charakteru. Táto vlastnosť zabezpečuje plnú modularitu systému od základného modulu jadra až k ovládačom či vlastným modulom.

Každý modul môže byť taktiež priamo zakompilovaný do jadra operačného systému. V tom prípade sa už nespúšťa pomocou nástroja KLD, ale je priamo obsiahnutý v jadre. Ak chceme vytvoriť modul, musíme použiť makrá z knižnice <`sys/module.h`>.

Systémový príkaz na nahranie modulu do systému je *kldload* <*module*>. Analogicky je možné modul zo systému odstrániť volaním *kldunload* <*module*>, avšak za predpokladu, že modul nie je súčasťou závislosti iného modulu. V tom prípade bude zo systému odstránený, až keď bude odstránený nadradený modul. Pomocou volania *kldstats* je možné vidieť všetky moduly, ktoré sú práve nahrané do operačného systému.

3.1.1 Všeobecný modul

Základné makro deklarujúce modul `DECLARE_MODULE ( name, moduledata_t, sub, order )` pozostáva z parametra `name`, ktoré prideľuje modulu unikátny názov v rámci celého systému. Definovanie funkcie typu `modeventhand_t` slúži ako vstupný bod pre správu modulu. Má na starosti ošetriť základné stavy modulu, ktoré sú do funkcie vložené pomocou argumentu. Základné stavy sú `MOD_LOAD` – nahranie modulu do systému; `MOD_UNLOAD` – odobranie

modulu zo systému; `MOD_SHUTDOWN` – ukončenie behu systému. Parametrom `sub` definujeme zavádzací typ, ktorý radí modul do kategórie, ktorá pevne stanovuje, v akom bode bude modul zavedený pri štartovacom procese celého systému (napríklad `SI_SUB_DRIVERS` pre modul ovládača zariadenia). S ním súvisí aj parameter `order`, ktorý definuje poradie v rámci kategórie (napríklad `SI_ORDER_ANY`, `SI_ORDER_FIRST`).

Prirodzene môžeme modulom nastavovať ich verziu. K tomu sa využíva makro `MODULE_VERSION(name, version)`. Na základe verzie môžeme medzi modulmi vytvárať závislosti vzťahujúce sa na názov a verziu.

Závislosť modulu sa deklaruje makrom `MODULE_DEPEND(name, moddepend, minversion, prefversion, maxversion)`. V prípade, že má modul závislosti, nemôže byť nahraný do jadra systému skôr než moduly, na ktorých je závislý.

Základné makro je použité v špecifickejších makrách ako `DRIVER_MODULE` , `DEV_MODULE` a `SYSCALL_MODULE`, ktoré pevne nastavujú druhý a tretí parameter makra `DECLARE_MODULE`.

```
static int foo_modevent(module_t mod, int type, void *data){
    switch (type) {
    case MOD_LOAD:
 /* init module */
 break;
    case MOD_UNLOAD:
    case MOD_SHUTDOWN:
     /* exit module */
 break;
   ...
return (error);
}
/* module declaration macro */
static moduledata_t sc6pcr_mod = { "foo", foo_modevent, 0 };
DECLARE_MODULE(foo, foo_mod, SI_SUB_EXEC, SI_ORDER_ANY);
```

3.1.2 Modul ovládača zariadenia (Device Driver Module)

Modul ovládača zariadenia sa deklaruje makrom `DEV_MODULE`. Takýto modul má často na starosti základnú správu zariadenia pomocou funkcií `make_dev()` a `destroy_dev()`, ktoré vytvoria resp. zmažú špeciálny súbor v adresári `/dev`. Nakoľko tento modul nemusí byť priamo naviazaný na hardvér, je možné vytvárať pseudo-zariadenia, teda virtuálne zariadenia, ktoré emulujú hardvér (napríklad virtuálne mechaniky a podobne).

3.1.3 Modul ovládača (Kernel Driver Module)

Makro `DRIVER_MODULE` zabezpečuje vytvorenie modulu, ktorý je priamo zviazaný s hardvérom. Tomuto typu modulu bude venovaná kapitola 3.2 a v kapitole 3.3 bude podrobne popísaný proces autokonfigurácie, ktorý pomocou vlastností tohto typu modulu identifikuje zariadenia.

3.2 Modul ovládača pre zbernicu PCI

Tento modul zodpovedá za správnu identifikáciu ovládača k príslušnému zariadeniu, za jeho inicializáciu do jadra systému, respektíve za jeho odstránenie zo systému. Tento modul

zabezpečuje reálne namapovanie hardvéru do jadra systému.

Makro `DRIVER_MODULE(name, busname, driver_t, devclass_t , ...)`, ktoré deklaruje tento typ modulu, sa skladá z parametrov, ktoré jasne identifikujú zariadenie, pre ktoré je modul implementovaný. Argument `name` je použitý v systéme ako prefix pre zariadenia (foo0, foo1) a jeho funkcie. Parameter `busname` popisuje, ktorú zbernicu zariadenie využíva. Každý ovládač zariadenia v jadre systému je popísaný štruktúrou `driver_t`. Tá obsahuje názov zariadenia, odkaz na štruktúru zoznamu funkcií(`device_method_t`), ktoré definujú druh zariadenia, pre ktorý je modul použitý, a veľkosť pamäte, ktorá sa musí alokovať pre privátne dáta. Trieda `devclass_t` obsahuje informácie, ktoré jadro používa pre internú identifikáciu zariadenia.

Zoznam funkcií `device_method_t`, ktorý je súčasťou hlavného objektu ovládača, zabezpečuje ovládaču nástroj, akým je v procese autokonfigurácie priradený k zariadeniu. Tento proces je popísaný v ďalšej kapitole.

```
/* Device interface */
static device_method_t foo_methods[] = {
    DEVMETHOD(device_probe,     foo_probe),
    DEVMETHOD(device_attach,    foo_attach),
    DEVMETHOD(device_detach,    foo_detach),
    ...
    { 0, 0 }
};

/* Device class for foo */
static devclass_t foo_devclass;

/* Device driver structure declaration */
static driver_t foo_driver = { "foo", foo_methods,sizeof(struct foo_sc) };
DRIVER_MODULE(foo, pci, foo_driver, foo_devclass, 0, 0);
```

Po zaregistrovaní systém pridá modul do zoznamu ovládačov spadajúcich pod triedu rodičovskej zbernice. To znamená, že všetky ovládače pre zbernicu PCI sú obsiahnuté v triede `pci` a ovládač pre zbernicu ISA zasa v triede `isa`. Týmto prístupom je možné v systéme registrovať ovládače s rovnakým identifikačným menom pre odlišné typy zbernice.

3.3 Autokonfiguračný proces

Autokonfigurácia je proces zabezpečený jadrom OS, slúžiaci k identifikovaniu a spusteniu zariadenia do stavu, kedy bude pripravené k používaniu. Automaticky je spustený pri štarte operačného systému.

Autokonfigurácia sa počas behu systému môže opakovať, a to buď v prípade výskytu nového zariadenia, alebo modulu zariadenia. Proces systematicky sonduje všetky zbernice a kontroléry, ktoré sa v počítači nachádzajú.

Zbernice sú v systéme reprezentované v stromovej štruktúre, kde najvyšší rodič je uzol `root0`. Proces rekurzívne prechádza stromom zberníc a žiada o identifikáciu zariadení. V prípade, že je zariadenie ohlásené, interpretuje sa a podľa typu sa spúšťa fáza sondovania. Sondovanie sa vykonáva voči zaregistrovaným modulom zariadení volaním funkcie

`device_probe()`. V prípade úspešnosti sondy nájdením ovládača sa automaticky spúšťa proces pripojenia zariadenia do systému volaním funkcie `device_attach()`.

3.4 Súborový systém zariadení a znakové zariadenia

Zariadenia sú v Unixovo založených OS sprístupňované pomocou uzlov zariadení, taktiež nazývaných špeciálnymi súbormi. Tieto uzly sa obyčajne nachádzajú v súborovej hierarchii v adresári `/dev`.

Za vytvorenie tohto uzlu zodpovedá vo väčšine prípadov práve ovládač zariadenia po úspešnej fáze autokonfigurácie. Súbor sa vytvorí pomocou volania funkcie `make_dev()`. Funkcia vytvorí nové zariadenie, ktoré dostane svoje jednoznačné číslo v rámci celého systému, ktoré je zložené z majoritného a minoritného čísla zariadenia. Ak OS obsahuje súborový systém zariadení(označovaný skratkou DEVFS), oznámi tomuto systému výskyt nového zariadenia. Dôležitým parametrom funkcie `make_dev()` je jednoznačný názov, ktorý identifikuje zariadenie v súborovom systéme DEVFS. Nemenej dôležitým parametrom je štruktúra systémových volaní `cdevsw`, na ktoré vie zariadenie reagovať. Táto štruktúra predstavuje znakové zariadenie.

Takmer všetky periférie v systéme, okrem sieťových zariadení, môžeme považovať za znakové zariadenia. Takéto zariadenia obyčajne mapujú hardvérové rozhranie do byte–ového prúdu, podobne ako u súborových systémov.

Znakové zariadenie je popísané množinou funkcií práve v štruktúre `cdevsw`. Tieto funkcie predstavujú vstupné body k manipulácii so zariadením z užívateľského priestoru do priestoru jadra. Môžeme povedať, že táto štruktúra popisuje najbližšiu vrstvu medzi ovládačom a užívateľským softvérom. Základné vlastnosti štruktúry `cdevsw`:

3.4.1 Open

Otvorenie zariadenia k príprave na I/O operáciu. Tento vstupný bod je volaný vždy pre každé systémové volanie nad špeciálnym súborom. Obyčajne táto procedúra obsahuje verifikáciu správnej počiatočnej inicializácie a pripravenosti k použitiu.

3.4.2 Close

Zatvorenie zariadenia. Táto rutina je volaná vždy po vykonaní a ukončení systémového volania. V prípade, že zariadenie môže používať v ten istý čas len obmedzený počet klientov, musí rutina *close()* zabezpečiť uvoľnenie zariadenia.

3.4.3 Read a Write

Čítanie dát, respektíve vpisovanie do/zo zariadenia. Typicky to znamená, že sa vypisujú dáta na štandardný výstup.

3.4.4 Ioctl

Vykoná operáciu odlišnú od bežného *read()* a *write()*. Táto operácia je identifikovateľná príkazom zadefinovaným v spoločnom rozhraní jedným z makier `_IOR, _IOW, _IOWR, _IO`. Každý príkaz má priestor, v ktorom môže zo sebou niesť dáta. Toto systémové volanie poskytuje mechanizmus, ako ovládať zariadenie len pomocou týchto príkazov.

```
#define FOO_IOC_BOOT   _IOWR(c, 1, struct foo_boot)
...
foo_ioctl(struct cdev *dev, u_long cmd, ... ){
   switch(cmd){
      case FOO_IOC_BOOT: ...; break;
      ...
   }
}
```

3.4.5 Poll

Zkontroluje zariadenie, či sú dáta pripravené k čítaniu, čiže preneseniu z pamäťového priestoru jadra do užívateľského, alebo či je priestor na vpísanie nových dát. Rutina *poll()* sa používa v spojení so systémovými volaniami *select()* a *poll()*, ktoré sú vykonávané nad špeciálnymi súbormi.

3.4.6 Mmap

Toto systémové volanie zabezpečí namapovanie pamäti priestoru jadra do užívateľského priestoru. Týmto spôsobom môže následne proces spracovávať dáta priamo vo svojom aplikačnom kóde a nemusí využívať podstatne pomalšie volania ako *ioctl()*

```
foo_mmap(struct cdev *dev, vm_offset_t offset, vm_paddr_t *paddr, ... ){
   *paddr = vtophys( foo_virtual_address + offset);
}
```

3.5 Alokácia pamäte pre DMA a jej synchronizácia

Úvodná kapitola stručne popísala teoretický základ technológie priameho prístupu do pamäte. V tejto časti bude popísaná architektúra spracovania DMA pamäte pre FreeBSD.

Na popis a alokáciu sa používajú dve štruktúry `bus_dma_tag_t` a `bus_dmamap_t`. Tag popisuje vlastnosti, ktoré má mať požadovaná DMA pamäť. Mapa reprezentuje pamäťový blok alokovaný vo vzťahu k týmto vlastnostiam. K jednému tagu je možné priradiť viac máp. Tagy sú organizované v stromovej štruktúre s dedičnosťou ich vlastností. Potomok tagu automaticky dedí všetky vlastnosti od rodičovského tagu a môže ich ďalej upravovať, ale len na viac obmedzujúcejšie.

Všeobecne existuje v module jeden tag, ktorý nemá žiadneho rodiča, a ten je považovaný za rodičovský pre celý modul. Všetky požiadavky na DMA pamäť s odlišnými vlastnosťami používajú tento rodičovský tag. Tagy môžu byť použité na vytvorenie máp dvomi spôsobmi.

Prvý spôsob je jednoduchá alokácia kontinuálneho priestoru pamäte s vlastnosťami vloženého tagu a jej neskoršie uvoľnenie. Toto je prirodzený spôsob alokácie pre dlho žijúce oblasti pamäte, určenej ku komunikácii so zariadením. Alokácia prebehne jednoduchým namapovaním fyzickej a virtuálnej pamäťe do mapy.

Druhý spôsob predstavuje alokáciu ľubovolnej oblasti virtuálnej pamäte, ktorá bude zviazaná s mapou. Každá takáto oblasť bude podrobená testu, či zodpovedá nastaveniu mapy. Ak áno, dáta sa budú nachádzať na svojej originálnej adrese. V prípade, že oblasť nejakým spôsobom nezodpovedá nastaveniu mapy, bude vytvorená nová prispôsobená oblasť. Potom v prípade vpisovania dát do nezodpovedajúcej oblasti budú tieto dáta prekopírované do jej prispôsobenej oblasti a potom prenesené na zariadenie. V prípade čítania dát z takejto

oblasti sa najskôr dáta načítajú do prispôsobenej oblasti a potom prekopírujú na pôvodnú adresu. Tento proces kopírovania a prispôsobovania sa nazýva synchronizácia. Je to typické použite pre dočasné buffery, kde pre každý buffer sa načíta, prenesie a uvoľní.

Scenár jednoduchej alokácie vyzerá následovne: vytvorí sa tag s nastavenými vlastnosťami požadovanej pamäte. V ďalšom kroku sa vytvorí mapa a odkaz, na ktorom sa bude nachádzať virtuálna prispôsobená pamäť. V poslednom kroku sa táto mapa inicializuje a synchronizovateľná pamäť nahraje do odkazu. Súčasťou posledného kroku je aj vloženie argumentu, ktorý je odkaz na callback funkciu, v ktorej môžu prebehnúť overenia alokavanej pamäte, prípadne ošetrenie chýb.

```
/* DMA simple callback declaration */
foo_callback(void *arg, bus_dma_segment_t *segs, int nseg, int error)
{
  *(bus_addr_t *)arg = seg[0].ds_addr;
}

/* DMA base properties */
struct foo_data{
  ...
};
bus_dma_addr_t foo_phys;
bus_dma_tag_t foo_tag;
bus_dmamap_t foo_map;



/* Simple DMA allocation */
bus_dma_tag_create(parent_tag,..., &foo_tag);

/* Creates a map internally. */
bus_dmamem_alloc(foo_tag, &foo_data, /* flags */ , &foo_map);

/* Load map */
bus_dmamap_load(foo_tag, foo_map, (void *)foo_data, sizeof(struct foo_data),
                  foo_callback,&foo_phys,/* flags */);
```

Keď sa prenesie pamäť v smere zo alebo do zariadenia, aplikačný kód musí zabezpečiť, že dáta sa v poriadku preniesli a nijakým spôsobom nedošlo k ich modifikácii, či už aplikačným kódom alebo zariadením. K tomu slúži procedúra `bus_dmamap_sync`, ktorá synchronizuje vloženú mapu zo zariadenia k ovládaču (BUS_DMASYNC_POSTREAD) a opačne (BUS_DMASYNC_PREWRITE).

```
/* Synchronization from device to host memory */
bus_dmamap_sync(foo_dtag,foo_map,BUS_DMASYNC_POSTREAD);
```

3.6 Mapovanie adresných registrov

K tomu aby mohol ovládač pracovať s PCI zariadením je potrebné, aby mohol zapisovať a čítať v jeho registroch. Systém abstrahuje túto formu ako jeden typ zdroja informácii do štruktúry `resource`.

Každý adresný register na zbernici má svoje identifikačné číslo, na základe ktorého ho PCI kontrolér môže sprístupniť operačnému systému, a tým aj ovládaču zariadenia. Namapovaniu adresného registra predchádza získanie adresného identifikátora pomocou `PCI_BAR`. Konečné namapovanie do štruktúry `resource` sa vykoná pomocou funkcie `bus_alloc_resource_any`. K tomu, aby systémová funkcia vedela, odkiaľ má namapovať adresný register, sa vloží argument popisujúci zariadenie `device_t`. Mapovanie adresných registrov sa zväčša vykonáva vo funkcii `device_attach`, ktorej jediným argumentom je práve deskriptor zariadenia `device_t`. Druhým argumentom do mapovacej funkcie je typ zdroja informácií. Ako bolo už raz spomenuté, systém abstrahuje niekoľko týchto typov a jedným z nich je aj adresný register, ktorého typ je `SYS_RES_MEMORY`. Ako tretí argument sa vkladá adresný identifikátor. Posledným argumentom sa nastavujú príznaky zdroja ako zdieľanie zdroja alebo jeho aktivácia. Obyčajne sa používa príznak `RF_ACTIVE | RF_SHAREABLE`.

```
/* Adress register id */
int foo_rid = PCIR_BAR(0);
/* Allocate and activate resource */
struct resource *foo_res = bus_alloc_resource_any(device,SYS_RES_MEMORY,
                  &foo_rid, RF_SHAREABLE | RF_ACTIVE);
...
/* Read  line */
bus_read_4(foo_res, register_address);
/* Write line */
bus_write_4(foo_res, register_address,value);
```

3.7 Inicializácia prerušenia a jeho obsluha

Prerušenia sú generované zariadeniami, aby informovali nadriadený systém o nejakej udalosti. Aplikačný kód musí najskôr zistiť, či prerušenie vyvolalo zariadenie, pre ktoré je ovládač implementovaný. Prerušenia sú zdieľané, a preto môže nastať situácia, kedy prerušenie vyvolá iné zariadenie. Preto zariadenia obsahujú stavový register, voči ktorému ovládač overuje odôvodnenosť vyvolania obslužnej rutiny. K tomu slúžia adresné registry 3.6. Na základe príznaku prerušenia sa buď vykoná obslužná rutina, alebo sa prerušenie ignoruje.

Prerušenie sa spracováva v procedúre, ktorá sa musí zaregistrovať do operačného systému. Registrácia sa vykonáva podobne ako u alokácií adresných registrov pomocou abstrakcie zdroja informácii štruktúry `resource`. Rozdielom je vloženie typu informácie do alokačnej funkcie `SYS_RES_IRQ`. Po alokovaní prerušenia sa funkciou `bus_setup_intr` zaregistruje handler, ktorý bude v prípade prerušenia zavolaný systémom. Ako argumenty sa vložia alokovaný zdroj, príznaky vlastnosti handleru, handler a argument, ktorý sa má do neho vložiť.

```
/* Allocate resource for IRQ */
struct resource *foo_intr = bus_alloc_resource(device, SYS_RES_IRQ, 0x0,
                  0, ~0, 1, RF_SHAREABLE | RF_ACTIVE);
```

```
/* Setup intterrupt handler */
bus_setup_intr(device, foo_intr, INTR_TYPE_TTY | INTR_MPSAFE, NULL,
                  foo_intr_handler, foo_sc, &foo_cookie);
...
/* Interrupt handler */
foo_intr_handler(void *arg){
  foo_sc *sc = (struct foo_sc*) arg;
  flags = bus_read_4(sc->foo_res, FOO_INTR_STAT);
  if(flags & INTR_DMA){
    ...
  }
...
}
```

Kapitola 4

Štruktúra ovládača pre Combo6x/Netflow

V tejto kapitole budú predstavené jednotlivé moduly ovládača pre Combo6x/NetFlow, ktoré sú štruktúrovane rozdelené na logické celky. Toto rozdelenie je z dôvodu opakovaného použitia kódu, nakoľko projekt Liberouter má v tomto čase tri verzie matičných kariet (Combo6, Combo6x, Combo6E) a mnoho designových projektov.

Za základný modul je možné považovať generačne nezávislý **combo6core**. Tento modul musí byť nahraný do jadra systému ako prvý. Následuje modul, ktorý zabezpečuje identifikáciu konkrétnej generácie matičnej karty a jej úspešné pripojenie do operačného systému. V tomto prípade sa jedná už o hardvérovo závislý modul.

Tento tandem tvorí základ pre nasledujúce designové moduly. Tie už nie sú toľko závislé na matičnej doske. Avšak niektoré designové moduly potrebujú k svojej funkčnosti dátový **szedata** modul. Jedným z takýchto designov je aj FlowMon. To znamená, že po nahraní modulu **szedata** sa nahraje do systému modul **sc6pcr**, ktorý je posledný prvok uzatvárajúci moduly požadované k spusteniu NetFlow sondy do provozu.

Každý modul bude teraz popísaný z logickej a funkčnej stránky. Projekt Liberouter samozrejme obsahuje ďalšie množstvo modulov, ale tie nie sú predmetom tejto práce. Bližšie informácie o nich je možné nájsť na [1].

4.1 Bázový modul combo6core

Ako bolo spomenuté v úvode kapitoly, modul **combo6core** je považovaný za základný. Obsahuje implementáciu spoločného rozhrania k vytvoreniu uzlu zariadenia a jeho systémové volania pre všetky matičné karty. Uzol zariadenia je vytvorený v zložke `/dev/combosix/0-N` kde N je poradové číslo karty v systéme.

Pre systémové volanie `ioctl` sú definované príkazy informatívneho a inicializačného charakteru, ktorých časť je implementovaná v tomto module. Tieto známe príkazy nie sú spracované, ale sú preposlané do matičného modulu. Ten pri registrácii nastaví vlastný odkaz na funkciu, ktorá volanie spracuje. Potom už len záleží na matičnom module, či využije stávajúce implementácie v tomto module, alebo využije iné. Jedným z dôležitých `ioctl` volaní je pripojenie známych designov a ich uvedenie do aktívneho stavu, kedy sú pripravené k spusteniu.

Aplikačný proces používajúci systémové volanie `mmap` získa do pamäťového priestoru namapovaný adresný priestor FPGA hradla.

V module sa nachádza mechanizmus zabezpečujúci registráciu podporovaných designov a naopak ich odregistrovanie. Tento mechanizmus je využívaný hlavne designovým modulom a zabezpečuje, že všetky matičné karty, ktoré sa v OS nachádzajú, budu mať pripojené podporujúce designové moduly.

4.2 Matičný modul combo6x

Tento modul je zaregistrovaný do operačného systému ako modul ovládača3.2. To znamená, že je súčasťou autokonfiguračného procesu a implementuje rozhranie na sondovanie hardvéru a jeho pripojenie do operačného systému. Tento modul je už hardvérovo závislý na matičnej karte.

V procese autokonfigurácie je pripojenie ovládača podmienené úspešnosťou sondovacieho mechanizmu, ktorý porovnáva majoritné a minoritné číslo hardvéru. Spoločnosti CESNET bolo pridelené majoritné číslo `0x18ec`, ktorá pridelila Combo6x karte výrobné číslo `0xc058`. Ak sa nájde zhoda, prechádza sa k štádiu pripojenia ovládača a karty do operačného systému. Týmto ovládač automaticky prevoláva službu modulu **combo6core** na inicializáciu a vytvorenie uzlu zariadenia. Postupne inicializuje súkromné dáta ovládača reprezentované štruktúrou `combo6`. Súčasťou inicializácie je namapovanie všetkých adresných priestorov. Combo6x karta obsahuje tri priestory, ako je možné vidieť na obrázku 4.1:

- Adresný priestor PowerPC, ktorý je súčasťou PCI(X) s identifikátorom 0
- Indetifikátorom 1 je adresovaný konfiguračný priestor zbernice
- V treťom adresnom priestore sa nachádzajú FPGA registry

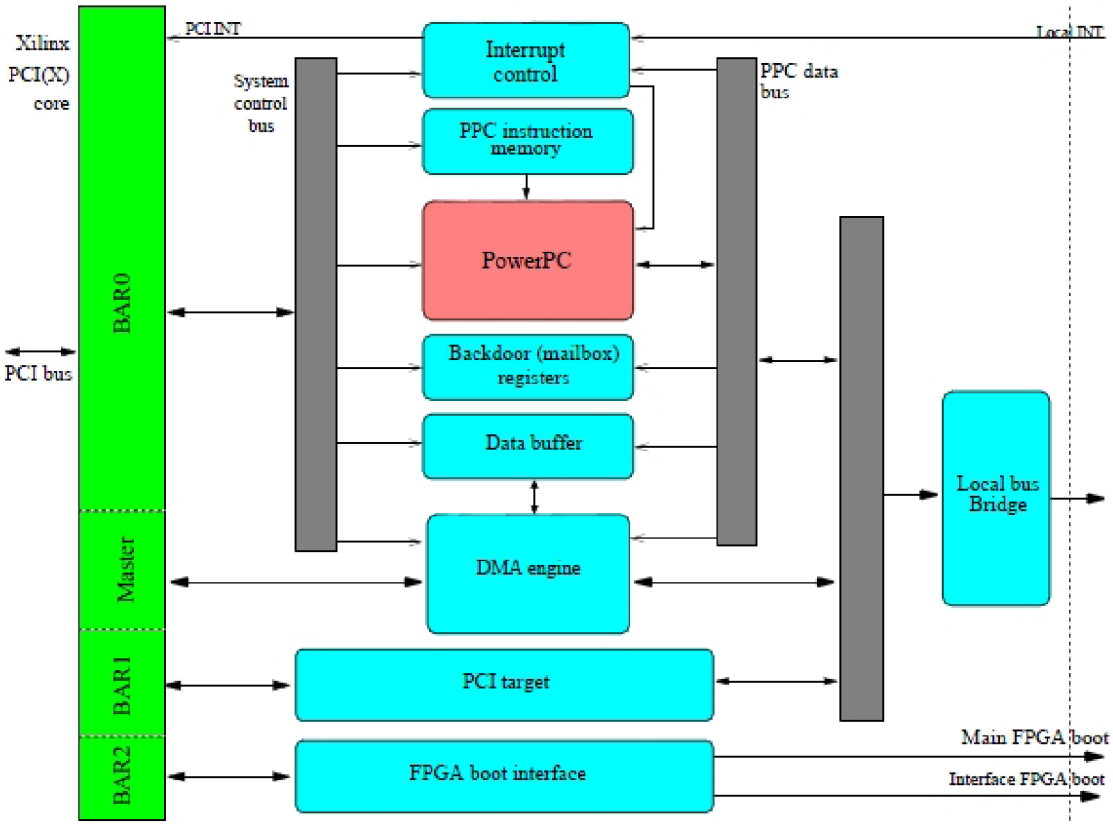
Overenie funkčnosti úspešného namapovania sa vykoná zistením verzie adresného priestoru PowerPc. Ďalej sa v procese pripojenia zaregistruje prerušovacia rutina a zariadenia sa nastaví do pozície *bus master*a.

Modulu je preposielané systémové volanie `ioctl`, ktoré voľne rozširuje o implementáciu špecifických príkazov závislých na matičnej karte. V prípade známych príkazov, ktorých implementácia sa nachádza v bázovom **combo6core** module, preposiela späť(`COMBO6_IOC_BUS_ATTACH`, `COMBO6_IOC_BUS_DETACH`, `COMBO6_IOC_INFO`, `COMBO6_IOC_INFO_OLD`). Ostatné príkazy sú implementované v tomto module. Jedná sa o nahranie dát na konfiguráciu FPGA hradla a PowerPC príkazom(`COMBO6_IOC_BOOT`) a volania na manipuláciu dát v adresných registroch (`COMBO6_READ`, `COMBO6_WRITE`).

4.3 Dátový modul szedata

Ako bolo načrtnuté v úvode kapitoly, modul **szedata** predstavuje nezávislý modul obsahujúci rozhranie na správu pamäte rezervovanej pre DMA prenosy. Na tom by nebolo nič zvláštne, lenže veľkosť pamäte, ktorá je v tomto prípade žiadaná designovým modulom, si vynucuje jej pokročilú správu. Mnohé designové moduly využívajú tento modul, aby im alokoval dostatočný počet jednotiek blokov definových štruktúrou `szedata_block`.

S týmto typom blokov vnútorne pracuje program nahraný v PowerPc, ako aj v FPGA. Nakoľko je Combo6 hardvér stavaný hlavne pre veľké dátové toky, je rýchlosť a množstvo prenesených blokov medzi zariadením a ovládačom prioritne smerovaná na čo najvyššiu



Obrázek 4.1: Architektúra registrov Combo6x karty

možnú mieru efektivity. Preto designové moduly vytvárajú kruhové buffery o veľkosti rádovo tisícok záznamov. Vnútorne si modul vedie štatistiky o využiteľnosti týchto blokov.

Hlavnou úlohou tohto modulu je pri inicializácii alokovať žiadaný počet blokov a vytvoriť uzol zariadenia na ceste `/dev/szedata/N`. Pomocou uzla aplikácie využívajú systémové volania na umožnenie spracovania väčších objemov dát v užívateľskom priestore.

Jedná sa hlavne o volanie `mmap`, ktoré namapuje na základe vloženého adresného posunu štatistiky, pamäťový priestor alokovaných blokov alebo ich deskriptory.

Systémové volanie `ioctl` je v tomto module mimoriadne bohaté. Umožňuje aplikáciam pripojiť sa k jednotlivým designom(`SZEDATA_IOCTL_SUBSCRIBE_INTERFACE`, ...), získať naposledy spracovaný blok (`SZEDATA_IOCTL_RBUFFER`) a tento blok uzamknúť dovtedy, než aplikácia dokončí jeho spracovanie (`SZEDATA_IOCTL_LOCK_NEXT`, ...). Podobne ako volaním `mmap` je možné získať štatistiky (`SZEDATA_IOCTL_GET_STATS`) alebo aktuálny stav využitého priestoru, ktorý je sprístupnený aplikáciam. Dôležité sú volania, ktoré uvedú zariadenie do aktívnho stavu, kedy začne spracovávať dáta (`SZEDATA_IOCTL_START`, `SZEDATA_IOCTL_STOP`).

4.4 FlowMon modul sc6pcr

Tento modul uzatvára množinu modulov, ktoré sú potrebné na uvedenie karty do aktívneho stavu. Pri inicializácii sa interne zaregistrujú designy, ktoré je možné spracovať v PowerPc. K tomu sa využívajú služby v module **combo6core**. Následne sa zinicializuje modul **szedata**, ktorý vytvorí uzol zariadenia a alokuje pamäť pre DMA prenosy. Celý design sa

uvádza do funkčného stavu zapisovaním do PowerPc registrov.

Hlavnou úlohou je vytvoriť kruhový RX[1] buffer pre pakety. V prípade NetFlow modulu nie je potrebné vytvárať TX buffer, pretože dáta žiadnym spôsobom neupravuje a o spracovanie sa postará vnútorná logika designu FlowMon 2.2.

K vytvoreniu tejto dátovej štruktúry sa využíva služieb modulu **szedata**. Tento, ako bolo napísané v 4.3, zabezpečuje vytvorenie dátového skladu, z ktorého sa zkonštruuje konečný paketový buffer. Pre tento design sa vytvára ešte ďalší kruhový buffer, pozostávajúci zo štruktúr `pcrdma_dma`, s ktorými vnútorne pracuje PowerPc a kde každý jeden záznam obsahuje 7 deskriptorov z paketového buffera. PowerPc dostane pri štarte designu odkaz do fyzickej pamäte na prvý záznam a už len vyvoláva prerušenie s požiadavkou o spracovanie nových dát.

Z toho plynie, že súčasťou tohto modulu sú tiež rutiny určené k spracovaniu prerušení, ktoré je preposlané modulom **combo6x**. Jedným z prerušení je už spomenutá požiadavka na spracovanie paketu `PCRDMA_INTR_RX_MASK`. Ďalšie prerušenie môže obsahovať masku výskytu neočakávanej chyby `PCRDMA_INTR_ERR_MASK`, ktorej obslužná rutina zabezpečí, že sa design uvedie do pôvodného stavu pred spustením.

[1]Receive buffer alebo tiež buffer príchodzích dát z ethernetu. K nemu analogicky existuje TX buffer, z ktorého zariadenie získava dáta, ktoré posiela naspäť do ethernetu

Kapitola 5

Postup prenosu Combo6x/NetFlow

Kapitola stručne popisuje v chronologickom poradí kroky, ktorými som postupoval pri portácii ovládača z Linuxu do FreeBSD. Najskôr som zisťoval možnosti a informácie o podobných prenosoch medzi UNIX-like platformami. To je popísané v prvej sekcii. Potom som musel načerpať dostatočné množstvo vedomostí o tom, akým spôsobom sa vytvárajú ovládače pre FreeBSD a Linux, čo je v ďalšej sekcii. Nasleduje sekcia, ktorej úlohou bolo zistiť, ako funguje stávajúci ovládač pod OS Linux. Záver je venovaný finálnej portácii ovládača a problémom s ním spojených.

5.1 Použité nástroje

Počas vývoja som využíval nástroje a techniku, ktorú je vhodné na tomto mieste spomenúť. Ako textový editor, či už na tvorbu tejto dokumentácie alebo zdrojových kódov, mi slúžil po celý čas *vim*, obľúbený takmer všetkými užívateľmi UNIX-like systémov.

Ku kompilácii zdrojových kódov som používal štandardný prekladač *GNU C(gcc)* s automatickým nástrojom *BSD Make*.

Server s označením *Server IBM Industrial PC*, ktorý som mal k dispozícii obsahuje Combo6x kartu s add-on kartou 4PFS. Na serveri je nainštalovaný OS Cent, ktorý má vytvorené 4 jadrá. Práve druhé jadro je najvhodnejšie, keďže obsahuje funkčné zavádzanie karty do systému. Vďaka tomu som mohol experimentovať s rôznymi nástrojmi v prostredí Linux, ako aj vo FreeBSD. Operačný systém, na ktorý budem portovať ovládač, je v tomto čase najnovšia STABLE distibúcia FreeBSD 7.1. FreeBSD je nainštalovaný na samostatnom pevnom disku. Nemalé komplikácie spôsobila základná doska (INTEL), ktorá z neznámych dôvodov odmietala pracovať s diskami MAXTOR na zbernici SATA.

5.2 Analýza dostupných riešení

Problém majority Linuxových ovládačov nad ovládačmi pre OS FreeBSD je už dlho známy. Z tohto dôvodu bolo potrebné zistiť, či existujú všeobecné postupy, ako takýto prenos realizovať, alebo nástroje, ktoré takýto proces dokážu automatizovať. Po analýze možných riešení ostali dve. Prvé sa opiera o výskum a úspešnú realizáciu prenosu Linuxových ovládačov pomocou tzv. medzi–vrstvy. Druhé riešenie predstavuje štandardnú cestu, a to postupný prepis jednotlivých zdrojových súborov.

5.2.1 Použitie medzi–vrstvy, linux-kmod-compat

Toto riešenie sa zakladá na výskume profesora Univerzity v Pise, Luigiho Rizzo [7], ktoré dospelo do štádia plnohodnotného portu *linux-kmod-compat*. Problém prenosu ovládačov rieši tým, že vytvoril vrstvu medzi natívnym Linuxovým ovládačom a FreeBSD jadrom. Princípom je definovanie Linuxových hlavičkových súborov, ktoré majú oproti originálnym súborom tú istú signatúru, avšak ich implementácia zodpovedá jadru FreeBSD.

Po výmene niekoľkých emailov s autorom som sa rozhodol postupovať tak, ako mi sám doporučil. Citujem:*„In this particular case I would probably try to do ad-hoc modifications to the specific linux driver source to adapt it to the freebsd kernel.“* Ako hlavný problém jeho riešenia uvádza fakt, že prepísané hlavičkové súbory sú zamerané na USB zariadenia, zatiaľ čo v našom prípade sa jedná o PCI zariadenie, ktoré je o mnoho zložitejšie, a preto by tam filozofia medzi–vrstvy mohla zlyhať.

5.2.2 Postupný prepis zdrojových súborov

Možným riešením práce je postupný prepis zdrojových súborov ovládača do cieľového OS. Táto varianta už bola úspešne realizovaná študentom Fakulty Informatiky Masarykovej Univerzity v Brne, Mgr. Jiřím Slabým pre OS NetBSD[1] [8]. Tento ovládač sa nachádza vo verzovacom systéme projektu Liberouter. Podľa slov autora, do ovládača neboli zaznamenané žiadne zmeny od doby jeho vytvorenia. Za dôvod neaktivity nad NetBSD ovládačom pokladám fakt, že jeho modulová hierarchia je diametrálne odlišná od Linuxovej verzie. Z toho vyplýva, že vznikli dva ovládače na jeden typ zariadenia.

Na základe tejto skutočnosti som vyvinul maximálne úsilie na to, aby ovládač vo FreeBSD bol čo najviac podobný Linuxovej verzii, aj napriek mierne odlišnej konvencii, ktorá sa používa vo FreeBSD. Vďaka tomuto prístupu bude možné zaimplementovať zmeny s takou istou efektivitou, ako je tomu u východzieho ovládača v OS Linux.

Tento prístup nie je OS FreeBSD vôbec cudzí, ako príklad môže slúžiť ovládač *Direct Rendering Manager* nachádzajúci sa v systéme v adresári `/sys/dev/drm`. Podobný prístup kompatibility som zvolil aj ja. Z tohto dôvodu sa medzi hlavičkovými súbormi nachádza rozhranie `/kernel/include/linux_compat.h`, ktoré definujú rôzne makrá syntakticky zhodné s volaniami v systéme Linux, avšak vnútorne využívajúcimi volania FreeBSD.

5.3 Získavanie zdrojov informácií o tvorbe ovládačov

Základným predpokladom pri tvorbe ovládača je znalosť cieľovej platformy. Pokiaľ ide o prenos ovládača z jednej platformy na druhú, táto požiadavka sa zdvojnásobuje. V tomto prípade som sa musel zoznámiť so základnými konceptami používanými v operačnom systéme Linux, ako aj vo FreeBSD. Tieto systémy majú spoločné rysy, no mnohé riešenia prístupu k niektorým technológiam sú odlišné.

Najskôr som potreboval získať vedomosti hlavne o jadre FreeBSD. K tomu mi výborne poslúžila kniha [6], ktorá však neobsahuje dôležité informácie potrebné k vývoju ovládačov. Po čase som zistil, že všeobecne chýbajú dokumenty, ktoré by dostatočne pokryli problematiku ovládačov v tomto systéme. Tu som narazil na kontrast oproti systému Linux, ku

[1] NetBSD je otvorený operačný systém typu BSD, ktorý vznikol na začiatku 90. rokov. Oproti FreeBSD bol prioritnejšie zameraný na podporu rôznych platforiem. V súčasnosti je možné NetBSD nasadiť na viac ako dvadsiatich platformách.

ktorému existuje už tretie vydanie knihy [3], ktorá sa zaoberá vývojom ovládačov. Takmer všetky znalosti, ktoré som získal, sú hlavne z tejto publikácie. Povedomie o tom, ako sa vytvárajú ovládače pre FreeBSD, som načerpal hlavne analyzovaním už implementovaných riešení v zdrojovom kóde FreeBSD. Takýto postup je náročný na čas, a preto jedným z cieľov práce je pokryť chýbajúcu problematiku v prijateľnom rozsahu bakalárskej práce.

5.4 Analýza stávajúcich ovládačov Combo6x pre OS Linux

Po naštudovaní základných znalostí o tvorbe ovládačov som sa pustil do analýzy dostupných riešení pre Combo6 karty a jej designy. Túto aktivitu sprevádzali veľmi podobné problémy, ktoré už boli popísané v predchádzajúcej sekcii. Po zistení nepríjemného faktu absencie akejkoľvek vývojovej dokumentácie ku všetkým ovládačom projektu Liberouter som začal postupne analyzovať zdrojový kód.

Najťažšou úlohou bolo zistiť modulovú hierarchiu ovládača a z nej potom určiť tie moduly, ktoré sú potrebné k projektu FlowMon. Po čase som začal komunikovať s Mgr. Jiřím Slabým o rôznych prístupoch a riešeniach, ktoré sú v jednotlivých moduloch použité. Postupne som sa stretal s rozdielmi v prístupe operačných systémov k riešeniu tej istej technológie. Najvýznamnejšie odlišnosti sú podrobnejšie popísané v následujúcej časti.

5.5 Implementácia

Počas portácie modulov ovládača sa vyskytlo niekoľko odlišností riešení v OS Linux a FreeBSD. Moduly boli prenášané v takom slede, v akom sa postupne nahrávajú do jadra systému.

V tomto čase je funkčnosť ovládača vo FreeBSD nestabilná. Nakoľko má byť tento ovládač súčasťou produktu Liberouter a má sa do budúcnosti ďalej rozvíjať, práce na ladení tejto komponenty budú pokračovať aj po odovzdaní tohto dokumentu. Najaktuálnejšiu verziu ovládača bude možné nájsť vo verejnom verzovacom systéme projektu Liberouter.

Nestabilita je spôsobená tým, že rozhranie užívateľských aplikácií bolo navrhované voči ovládaču na OS Linux. To v tomto prípade zapríčinilo, že toto rozhranie musí spĺňať aj ovládač pre FreeBSD. Významné dôvody nestability sú popísané v následujúcich odstavcoch.

Ako bolo už povedané, modul **szedata** alokuje a spravuje veľkú časť DMA pamäte. Túto časť rozdelí na menšie celky, ktoré vystupujú v systéme ako samostatné bloky. Tento modul poskytuje službu, kedy pri systémovom volaní *mmap()* namapuje celú túto pamäť užívateľskému procesu. V prípade FreeBSD sa nedá synchronizovať len blok takto alokovanej pamäte, ale až celá mapa, čo môže spôsobiť komplikácie pri vysokých zaťaženiach siete.

Problematická časť je ladenie ovládača. V tomto prípade nie je možné využívať štandardného nástroja, akým je debugger. Modul sa nahraje do systému, kde ho debugger nemôže uchopiť. Preto je vývoj ovládačov značne komplikovanejší. O to viac komplíkacií spôsobí, ak sa v kóde nachádza nejaký prvok nestability. V tom prípade je celé jadro systému znovu zavedené.

Ďalšie komplikácie sú spôsobené tým, že v prípade chybného alebo opakovaného nahrania firmvéru, dochádza k nedefinovanému chovaniu karty. PowerPc síce na karte odpovedá na určitú časť volaní, ale na niektoré poskytuje chybné dáta. Týmto spôsobom som strávil niekoľko hodín hľadaním chýb v kóde. Pri takomto probléme sa musí vypnúť a zapnúť celý server. Je to z toho dôvodu, že karta si aj po reštarte systému uchováva informácie vo svojom PowerPc a FPGA hradle.

5.5.1 Procfs vs. Sysctl

Súčasťou modulu **combo6core** a **szedata** je aj implementácia rozhrania umožňujúceho dynamickú správu parametrov týchto modulov z užívateľského prostredia. Vďaka tomu môže užívateľ meniť chovanie modulu počas jeho behu bez kompilácie a nahrania modulu do jadra systému.

V Linuxovej verzii ovládača je k tomuto účelu použitý virtuálny súborový systém *procfs*. Tento prístup využíva vlastnosti špeciálnych súborov. Každý jeden proces, ktorý je v systéme spustený, teda aj modul ovládača, má svoj uzol v adresári `/proc`. Aplikačný softvér potom systémovým volaním *read()* získa informácie uložené v jadre systému, ktoré tam nastaví modul. Naopak volaním *write()* môže aplikácia nastaviť vlastnosti.

FreeBSD k tomu účelu používa rozhranie *sysctl*. Tento nástroj funguje na báze nastavovania a čítania informácií zo stromovej štruktúry *sysctl*.

Obe tieto technológie sa nachádzajú v obidvoch systémoch. Lenže každý systém uprednostňuje iný prístup. V prípade Linuxu je volanie *sysctl* implementované nad technológiou *procfs*, a preto opakované využívanie tohto prístupu je príliš drahé na systémové prostriedky. Naopak OS FreeBSD má rozhranie *sysctl* implementované priamo v jadre už od verzie 4.4BSD. Preto sa doporučuje používať tento prístup na rozdiel od *procfs*.

Po komunikácií s jedným z autorov pôvodných ovládačov sme sa zhodli na tom, že dynamickú správu parametrov nie je nutné implementovať, keďže nemá momentálne žiadne využitie. Pre každý prípad je vo FreeBSD verzii ovládača nachystané rozhranie *sysctl*, ktoré sa využíva na poskytnutie detailnejších informácií o karte v **combo6core** module.

5.5.2 Pamäť DMA a jej synchronizácia

Ako alokovať pamäť pre DMA prenosy v OS FreeBSD bolo popísané v 3.5. Operačný systém Linux pristupuje k alokácii DMA pamäte odlišným spôsobom. Alokácia je vykonaná jediným volaním systémovej funkcie, do ktorej sa vloží odkaz, ktorý bude po úspešom vykonaní obsahovať fyzickú adresu pamäte alokovaného priestoru. S touto adresou potom pracuje zariadenie. V návratovej hodnote sa nachádza odkaz do virtuálnej pamäte.

Oproti spôsobu, ktorý je použitý vo FreeBSD, je tento prístup podstatne jednoduchší, no vývojár dochádza o mnohé možnosti, ktorými môže spôsob alokácie ovplyvňovať a spravovať. Avšak koncept nastavovania príznakov pri alokácii je v dnešnej dobe, kedy počítače disponujú rádovo gigabajtmi operačnej pamäťe, v určitých prípadoch zbytočne zložitý a neprehľadný.

Táto odlišnosť v svojej podstate nespôsobuje pri alokácii až také komplikácie. Tie sa vyskytnú až pri synchronizácii pamäte so zariadením. FreeBSD synchronizuje celú mapu, naopak pri synchronizácii v Linuxe stačí vložiť len fyzickú adresu a veľkosť pamäte, ktorá má byť synchronizovaná. To umožňuje v Linuxe alokovať väčší priestor pamäti, ktorý je možné ďalej štrukturovať na menšie celky. Každý tento blok je potom samostatne synchronizovateľný. Tento prístup sa u DMA prenosov vo FreeBSD použiť nedá.

Kapitola 6

Testovanie a porovnanie funkčnosti ovládača

Kapitola popisuje nástroje, ktoré sa používajú v súvislosti s projektom FlowMon. Ďalej pomocou týchto nástrojov sa dokáže, že výsledok snahy portácie ovládača je úspešný. Všetky tieto nástroje museli byť revidované hlavne z dôvodu importovania správnych hlavičkových súborov, ktoré sú závislé od použitého operačného systému. V niektorých prípadoch boli vložené podmienené preklady, ktoré namiesto technológie *procfs* využívajú štandardné volanie systému *ioctl()*.

Keďže nie je možné v rozumnom rozsahu dokumentu venovať sa každému nástroju hlbšie, každý z nich bude len stručne charakterizovaný s príkladom použitia. Pred začiatkom používania týchto nástrojov je potrebné mať v systéme nahrané všetky požadované moduly.

6.1 csid

Nástroj vykoná systémové *ioctl()* volanie s požiadavkou základných informácií o hardvére. Toto volanie prebieha interne pre všetky ďalšie nástroje, čím si overujú dostupnosť požadovaných zdrojov.

```
[root@flowmon csid]# ./csid
combo6x sfpro xc2vp20
```

6.2 csxtool

Aplikácia sa využíva na úvodné nahranie dostupných designov, ktoré podporujú matičnú kartu Combo6x. Tento nástroj primárne pracuje s XML súborom, v ktorom sú nakonfigurované jednotlivé popisy designov a ich identifikátory. Aplikácia definuje množinu príkazov, ktoré sú určené hlavne k spracovaniu daného XML súboru. Dôležitý príkaz *boot* zabezpečí nahranie designu do FPGA a PowerPC.

```
[root@flowmon csxtool]#./csxtool -c boot -f /netflow/02_05/design.xml
```

6.3 flomonctl

Použitie aplikácie spočíva v počiatočnom nakonfigurovaní matičnej karty do designu FlowMon sondy. Všetky predchádzajúce aplikácie boli použiteľné v celom spektre designov. Táto

je už priamo závislá na designe, ktorým si nakonfiguruje logiku karty do takého stavu, aby zodpovedala požadovanej funkčnosti designu. Pomocou tejto aplikácie môžeme aj počas behu karty nastavovať rôzne vlastnosti designu FlowMon.

```
[root@flowmon flowmonctl]#./flowmonctl -c sample_hold -v 2 -s 10 -t 0
```

Ďalšia ukážka nahraje skompilovaný program VHDL do PowerPc a tým zavedie design FlowMon sondy do srdca karty.

```
[root@flowmon flowmonctl]#./flowmonctl -c init -e /netflow/02_05/
                                                          flowmon.bin
```

6.4 flowmoncol

Týmto nástrojom sa spustí kolektor, ktorý zbiera dáta od exportérov na báze protokolu NetFlow. Spusteniu NetfFlow kolektoru predchádza celý rad konfiguračných nastavení, ktoré je možné nájsť na manuálových stránkach projektu Liberouter.

Následujúca ukázka spustí kolektor na na porte 5000.

```
flowmoncol -v 5 -i 10 -l 5000 -m 4
```

6.5 flowmon_nf5

Aplikácia predstavuje analyzátor dát protokolu NetFlow v5. Týmto spôsobom sa spustí exportér a pripojí sa na kolektor korý beží na adrese *localhost:5000*.

```
flowmon_nf5 -v 5 localhost:5000
```

Kapitola 7

Záver

V práci som sa snažil vysvetliť a predviesť čo najviac technológií, ktoré sa používajú pri vývoji ovládačov pre OS FreeBSD. Zameral som sa hlavne na DMA prenosy, členenie a rozmanitosť modulov a spracovanie prerušení. Všetky tieto prístupy som mal aplikovať na prenos ovládača zariadenia Combo6x karty z Linuxu do FreeBSD.

Tento systém poskytuje rozhranie, ktoré umožňuje efektívne vytvárať ovládače zariadení. Rozhranie spadá do UNIX–ových štandardov a obsahuje nástroje k využívaniu najmodernejších technológií. To môžeme využiť aj v prípade portácie ovládačov z iných operačných systémov. Modifikovanie takýchto komponent je flexibilné a rýchle. V takomto prípade by sa nemalo zabúdať na dodržanie aplikačného rozhrania už so stávajúcimi aplikáciami.

Počas práce som narazil na značné problémy, ktoré podstatne zdržali vývoj oproti zamýšlanému plánu. Z tohto dôvodu nie je výsledok portácie v takej podobe, aby som bol s ním spokojný po stránke funkčnosti i dokumentácie.

Za veľký problém považujem neprehľadnú a roztrieštenú dokumentáciu, ktorá by pokryla problematiku vývoja ovládača pre FreeBSD. Mnohé konkurenčné operačné systémy disponujú podrobnou dokumentáciou, ktorá poskytuje vývojárom potrebné informácie a oporu. Aj napriek tomu, že sú si tieto systémy v mnohom podobné v kľúčových technológiach, ako DMA, je ich rozhranie a logika odlišné. Ako príklad môžu slúžiť identické signatúry funkcií v OS FreeBSD a NetBSD, no forma použitia je odlišná.

Za ešte väčší problém považujem chýbajúcu vývojovú dokumentáciu či analýzu ovládačov a rozhrania užívateľských aplikácií k projektu Liberouter a jeho designom. Pri takto zameranom projekte je tvorba dokumentov, týkajúcich sa tak dôležitých komponent, rozhodujúca pre ich rozšírenie do povedomia odbornej verejnosti.

Do riešenia práce som sa pustil bez predchádzajúcich znalostí jadra FreeBSD, vývoja ovládačov zariadení, ako aj karte Combo6x. Po spracovaní tejto témy som získal množstvo skúseností v tejto problematike, no v mnohých oblastiach oboch projektov mám stále medzery. Ich dopĺňanie analyzovaním zdrojového kódu je mentálne i časovo náročné.

Písanie tejto práce bolo pre mňa v mnohom prínosné, najmä v navrhovaní ovládačov a ich štrukturovaní do logických modulov. Dynamická správa modulov v sebe ponúka možnosť flexibilnej práce s celým jadrom systému. Nezávislosť najnižších vrstiev zabezpečuje stabilitu systému aj pri neočakávaných chybách. To mi umožnilo nahliadať na vývoj zdrojových komponent aplikácií z oveľa nezávislejšieho pohľadu. Taktiež forma zdrojového kódu ovládačov mi rozšírila obzor o nové postupy používané v tomto odvetví, ktoré by som chcel v budúcnosti aplikovať pri vývoji optimalizovanejších a štruktúrovanejších aplikácií.

Literatura

[1] Cesnet: Oficiální stránky projektu Liberouter. [online].
URL http://www.liberouter.org/

[2] Cesnet: Stránky združení CESNET. [online].
URL http://www.cesnet.cz/

[3] Corbet, J.; Rubini, A.; Kroah-Hartman, G.: *Linux Device Drivers, Third Edition.* O'Reilly Media, Inc., 2005, ISBN 0-596-00590-3, 616 s.

[4] Čeleda, P.; Kováčik, M.; Krejčí, R.; aj.: Software for NetFlow Monitoring Adapter. [online], 2005.
URL http://www.cesnet.cz/doc/techzpravy/2005/netflow/

[5] Foundation, T. F.: Oficiální stránky projektu FreeBSD. [online].
URL http://www.freebsd.org/

[6] McKusick, M. K.; Neville-Neil, G. V.: *The Design and Implementation of the FreeBSD Operating System.* Pearson Education, Inc., 2005, ISBN 0-201-70245-2, 720 s.

[7] RIZZO, L.: Building Linux Device Drivers on FreeBSD. [online], poslední aktualizave: 18 Nov. 2008.
URL http://info.iet.unipi.it/~luigi/FreeBSD/linux_bsd_kld.html

[8] Slabý, J.: PowerPC v FPGA na kartách COMBO6. [online], 2006.
URL http://is.muni.cz/th/98734/fi_b/

Příloha A

Kostra ovládača zariadenia pre OS FreeBSD

```
/*
 * Simple KLD to play with the PCI functions.
 *
 * Murray Stokely
 */

#include <sys/param.h>        /* defines used in kernel.h */
#include <sys/module.h>
#include <sys/systm.h>
#include <sys/errno.h>
#include <sys/kernel.h>       /* types used in module initialization */
#include <sys/conf.h>     /* cdevsw struct */
#include <sys/uio.h>      /* uio struct */
#include <sys/malloc.h>
#include <sys/bus.h>      /* structs, prototypes for pci bus stuff */

#include <machine/bus.h>
#include <sys/rman.h>
#include <machine/resource.h>

#include <dev/pci/pcivar.h>   /* For pci_get macros! */
#include <dev/pci/pcireg.h>

/* The softc holds our per-instance data. */
struct mypci_softc {
    device_t    my_dev;
    struct cdev *my_cdev;
};

/* Function prototypes */
static d_open_t     mypci_open;
static d_close_t    mypci_close;
```

```
static d_read_t     mypci_read;
static d_write_t    mypci_write;

/* Character device entry points */

static struct cdevsw mypci_cdevsw = {
    .d_version =    D_VERSION,
    .d_open =   mypci_open,
    .d_close =  mypci_close,
    .d_read =   mypci_read,
    .d_write =  mypci_write,
    .d_name =   "mypci",
};

/*
 * In the cdevsw routines, we find our softc by using the si_drv1 member
 * of struct cdev.  We set this variable to point to our softc in our
 * attach routine when we create the /dev entry.
 */

int
mypci_open(struct cdev *dev, int oflags, int devtype, d_thread_t *td)
{
    struct mypci_softc *sc;

    /* Look up our softc. */
    sc = dev->si_drv1;
    device_printf(sc->my_dev, "Opened successfully.\n");
    return (0);
}

int
mypci_close(struct cdev *dev, int fflag, int devtype, d_thread_t *td)
{
    struct mypci_softc *sc;

    /* Look up our softc. */
    sc = dev->si_drv1;
    device_printf(sc->my_dev, "Closed.\n");
    return (0);
}

int
mypci_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct mypci_softc *sc;

    /* Look up our softc. */
```

```
    sc = dev->si_drv1;
    device_printf(sc->my_dev, "Asked to read %d bytes.\n", uio->uio_resid);
    return (0);
}

int
mypci_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct mypci_softc *sc;

    /* Look up our softc. */
    sc = dev->si_drv1;
    device_printf(sc->my_dev, "Asked to write %d bytes.\n", uio->uio_resid);
    return (0);
}

/* PCI Support Functions */

/*
 * Compare the device ID of this device against the IDs that this driver
 * supports.  If there is a match, set the description and return success.
 */
static int
mypci_probe(device_t dev)
{

    device_printf(dev, "MyPCI Probe\nVendor ID : 0x%x\nDevice ID : 0x%x\n",
        pci_get_vendor(dev), pci_get_device(dev));

    if (pci_get_vendor(dev) == 0x11c1) {
        printf("We've got the Winmodem, probe successful!\n");
        device_set_desc(dev, "WinModem");
        return (BUS_PROBE_DEFAULT);
    }
    return (ENXIO);
}

/* Attach function is only called if the probe is successful. */

static int
mypci_attach(device_t dev)
{
    struct mypci_softc *sc;

    printf("MyPCI Attach for : deviceID : 0x%x\n", pci_get_devid(dev));

    /* Look up our softc and initialize its fields. */
    sc = device_get_softc(dev);
```

```
    sc->my_dev = dev;

    /*
     * Create a /dev entry for this device.  The kernel will assign us
     * a major number automatically.  We use the unit number of this
     * device as the minor number and name the character device
     * "mypci<unit>".
     */
    sc->my_cdev = make_dev(&mypci_cdevsw, device_get_unit(dev),
        UID_ROOT, GID_WHEEL, 0600, "mypci%u", device_get_unit(dev));
    sc->my_cdev->si_drv1 = sc;
    printf("Mypci device loaded.\n");
    return (0);
}

/* Detach device. */

static int
mypci_detach(device_t dev)
{
    struct mypci_softc *sc;

    /* Teardown the state in our softc created in our attach routine. */
    sc = device_get_softc(dev);
    destroy_dev(sc->my_cdev);
    printf("Mypci detach!\n");
    return (0);
}

/* Called during system shutdown after sync. */

static int
mypci_shutdown(device_t dev)
{

    printf("Mypci shutdown!\n");
    return (0);
}

/*
 * Device suspend routine.
 */
static int
mypci_suspend(device_t dev)
{

    printf("Mypci suspend!\n");
    return (0);
```

```
}

/*
 * Device resume routine.
 */
static int
mypci_resume(device_t dev)
{

    printf("Mypci resume!\n");
    return (0);
}

static device_method_t mypci_methods[] = {
    /* Device interface */
    DEVMETHOD(device_probe,     mypci_probe),
    DEVMETHOD(device_attach,    mypci_attach),
    DEVMETHOD(device_detach,    mypci_detach),
    DEVMETHOD(device_shutdown,  mypci_shutdown),
    DEVMETHOD(device_suspend,   mypci_suspend),
    DEVMETHOD(device_resume,    mypci_resume),

    { 0, 0 }
};

static devclass_t mypci_devclass;

DEFINE_CLASS_0(mypci, mypci_driver, mypci_methods, sizeof(struct mypci_softc));
DRIVER_MODULE(mypci, pci, mypci_driver, mypci_devclass, 0, 0);
```