ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA ELEKTROTECHNICKÁ

BAKALÁŘSKÁ PRÁCE

Operační systémy PikeOS a LINUX

Praha, 2008 **Autor: Michal Hrouda**

Prohlášení

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

V Praze dne 12. 6. 2009

Zeman Michal

podpis

Poděkování

Tímto bych chtěl poděkovat především vedoucímu mé bakalářské práce za pomoc při psaní této práce. Velké poděkování patří též firmám Energocentrum Plus s.r.o. a Mikroklima s.r.o. za jejich pomoc a za poskytnutí nezbytných podkladů a vývojových prostředků, bez kterých by tato práce nemohla vzniknout. Děkuji také firmě Sysgo za jejich technickou podporu při řešení vzniklých potíží.

Abstrakt

V této bakalářské práci je popsán operační systém Linux a s ním související produkt PikeOS firmy Sysgo, provozovaný na modulu s procesorem Freescale MPC5200. V první části jsem se zaměřil na samotný procesorový modul, zatímco v následujících na samotné systémy Linux a PikeOS, jejich popis a způsob kompilace a konfigurace pro uvedený modul.

Abstract

This bachelor thesis describes Linux operating system and system created by Sysgo company named PikeOS. Both of system is operated on embedded module equiped with Freescale MPC5200 processor. First chapter describes embedded module while the others describes Linux and PikeOS systems, its features
and methods for compiling and configuring for given module.

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídicí techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Michal Hrouda**

Studijní program: Elektrotechnika a informatika (bakalářský), strukturovaný
Obor: Kybernetika a měření

Název tématu: **Operační systémy PikeOS a Linux**

Pokyny pro vypracování:

1. Přeneste operační systém Linux na desku s Power PC MPC5200. Realizujte ovladače pro sériová rozhraní SPI a I2C.
2. Na stejném HW zprovozněte operační systém Pike OS.
3. Na jednom z oddílů PikeOS zprovozněte OS Linux. Realizujte komunikaci mezi jednotlivými oddíly, tj. mezi nativním oddílem PikeOS a oddílem s OS Linux.

Seznam odborné literatury:

Dodá vedoucí práce

Vedoucí: Ing. Pavel Burget

Platnost zadání: do konce zimního semestru 2008/2009

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

L.S.

doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 25. 2. 2008

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Tato práce se zabývá zprovozněním operačních systémů Linux a PikeOS na embedded modulu osazeném procesorem MPC5200 s příznačným názvem Shark. Výběr těchto systémů a použitého hardware je podřízen předpokládanému nasazení cílového systému do průmyslového prostředí pro účely řízení.

Procesor MPC5200 je představitelem architektury PowerPC, jde o 32bitový procesor, který v sobě integruje jednotku pro operace s plovoucí řádovou čárkou v dvojnásobné přesnosti, jednotku správy paměti, ethernet a další periferie vhodné pro řídící účely.

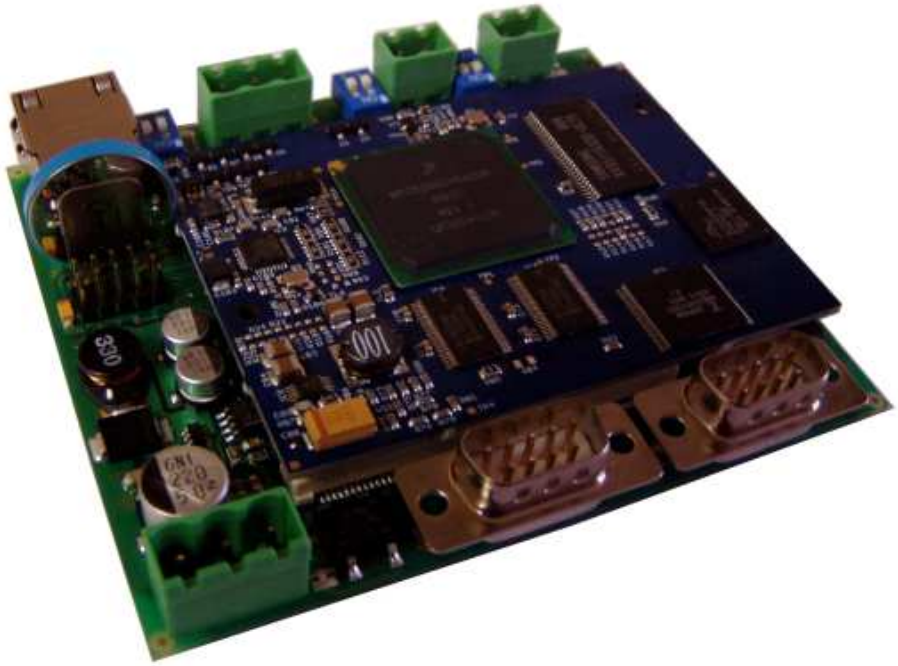
PikeOS (4) je systém určený pro embedded zařízení, kde je vyžadována vysoká spolehlivost a robustnost. Nejedná se však o plnohodnotný *operační* systém, jeho hlavní podstatou je virtualizační jádro, které vytváří prostředí pro provoz již plnohodnotných operačních systémů a nebo tzv. nativních úloh. Ve spolupráci s plně konfigurovatelným časovým plánováním tohoto jádra je možné vytvořit úplný realtime systém s vysokou spolehlivostí, neboť časově kritické činnosti jsou vykonávány nativní úlohou a interakce s uživatelem je zajištěna aplikací běžící v prostředí Linuxu.

Systém Linux je plnohodnotný operační systém s širokou škálou podporovaných architektur a platforem. Výhoda Linuxu v embedded systémech tkví v zajištění hardwarově závislých částí a jejich zpřístupnění formou snadněji použitelných rozhraní a dále v jednodušším vývoji cílové aplikace. V tomto případě lze většinu aplikace vyvinout a odladit na stolním PC bez nutnosti použití drahých hardware debuggerů pro cílovou platformu.

V první části (2) této práce naleznete základní popis zvoleného procesoru , embedded modulu a základnové desky pro tento modul. V druhé části (3) je pak postup kompilace a konfigurace jádra Linuxu pro spuštění na této desce, ve třetí (4) naleznete informace týkající se kompilace a konfigurace PikeOS a v poslední části jsou popsány možnosti komunikace mezi nativní úlohou a Linuxem běžící v rámci PikeOS.

Kapitola 2

Popis hardware



2.1 Procesor MPC5200B

2.1.1 Popis procesoru MPC5200B

Procesor MPC5200B v sobě integruje vysoce výkonné jádro e300 s rozsáhlým množstvím periferií zaměřených na komunikace a systémové integrace. Jádro e300 je založeno na architektuře jader PowerPC. MPC5200B představuje inovativní I/O sub-systém, který odděluje správu periferií od vlastního jádra e300. MPC5200B podporuje

architekturu dvojité externí sběrnice. Ta se skládá z rychlé sběrnice pro SDRAM, která je připojena přímo na jádro e300 a sběrnice s názvem LocalPlus bus, která je používána jako obecný interface pro připojení dalších periferních zařízení a ladícího prostředí.

Tento procesor byl vybrán z následujících důvodů

- Velký výpočetní výkon
- Integrována FPU a MMU
- Vhodné periferie pro řídící účely (6x UART, CAN, USB), ethernet řadič
- Oddělená sběrnice pro SDRAM a ostatní periferie
- K dispozici v automotive provedení $\Rightarrow$ spolehlivost, garance dlouhého výrobního cyklu
- Podporován většinou Realtime operačních systému a Linuxem

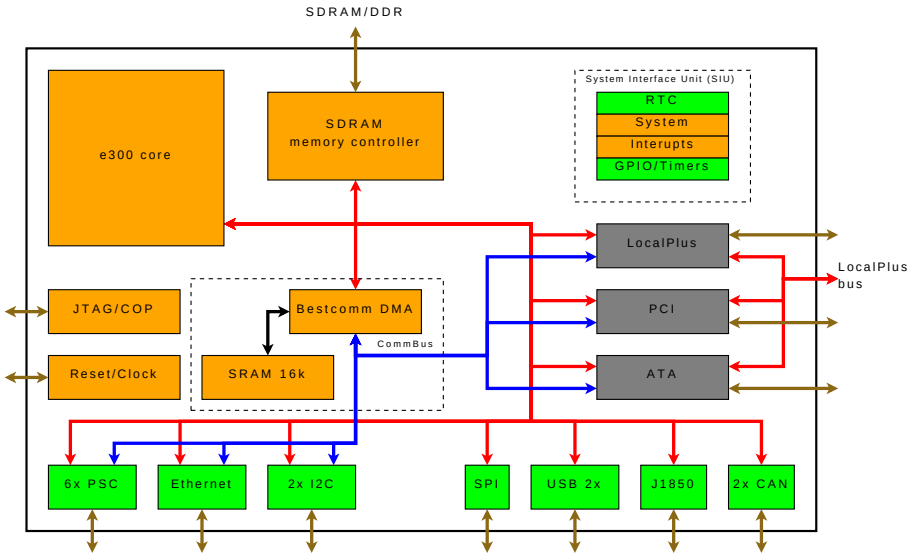
Základní vlastnosti procesoru MPC5200 lze shrnout do následujících bodů

- Jádro e300
  - Superskalární architektura
  - 760 MIPS při 400 MHz (v průmyslovém rozsahu teplot -40°C - 85°C)
  - Jednotka pro operace s plovoucí řádovou čárkou v dvojnásobné přesnosti
  - Jednotka správy paměti
- SDRAM/DDR interface
  - Podpora pracovní frekvence až 132 MHz
  - Podpora režímů SDR a DDR
  - 256 MB adresní prostor na jeden signál Chip select
  - 32bitová šířka datové sběrnice
  - Přímá podpora inicializace a refresh pamětí
- Flexibilní externí sběrnice LocalPlus bus
  - Podpora ROM/Flash/SRAM a dalších paměťově mapovaných zařízení
  - Osm programovatelných signálů Chip select
  - Nemultiplexovaný režim s šířkou dat 8/16/32bit a až 26bit adresy
  - Multiplexovaný režim s šířkou dat 8/16/32bit a až 25bit adresy
- Řadič PCI kompatiblní s verzí 2.2

- Řadič ATA
- Šest programovatelných sériových kontrolérů (PSC)
  - Lze používat v režimu UART/Soft Modem/I2S/AC97/Plně duplexní SPI/IRDA
- Fast ethernet řadič
- Řadič Host USB verze 1.1 (OHCI), k dispozici 2 porty
- Dva řadiče sběrnice I2C
- Řadič SPI
- Dva řadiče sběrnice CAN verze 2.0 A/B
- Ladící rozhraní dle standardu IEEE 1149.1

Na obr. 2.1 je zobrazeno zjednodušené blokové schéma tohoto procesoru. Z blokového schématu je jasně patrná ona dvojitá externí sběrnice, pro SDRAM a LocalPlus bus.

Řadič SDRAM/DDR je připojen přímo na jádro e300, kdy použitím na čipu integrované 16kB instrukční a 16kB datové vyrovnávací paměti je dosaženo vysokého výkonu procesoru pro výpočetně náročné aplikace.
Procesor je dále vybaven integrovaným inteligentním řadičem přímého přístupu do paměti (DMA) BestComm, který umožňuje na jádře nezávislou obsluhu přerušení od periferií, jejich nízkoúrovňovou správu a přesuny bloků paměti.



Obrázek 2.1: Zjednodušené blokové schéma procesoru MPC5200

Tímto bych ukončil stručný přehled vlastností tohoto procesoru, pro zájemce o více informací odkazuji na firemní dokumentaci firmy Freescale ([1]).

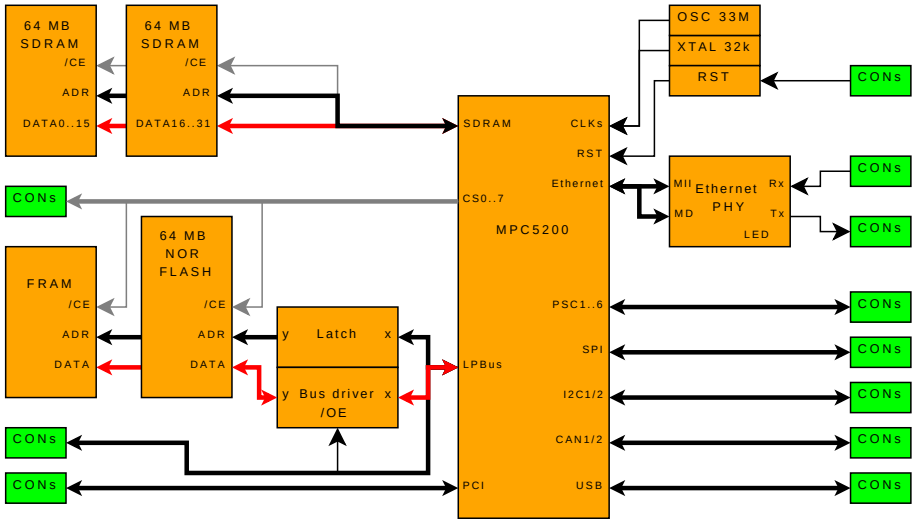
2.2 Embedded modul Shark

Embedded modul Shark ([2]) je založen na popsaném procesoru MPC5200B. Modul je postaven na základě podobného modulu německé firmy TQ Components ([3]) s názvem TQM5200 a je tak s ním pinově kompatibilní, další vlastnosti už jsou však upraveny pro plánované použití.

Vlastnosti modulu Shark

- Procesor: *MPC5200B*
- RAM: *až 128 MB SDRAM, 132 MHz*
- Flash: *až 64 MB NOR Flash*, k dispozici i 128 kB FRAM (náhrada EEPROM)
- Osazený ethernetový budič (PHY) - stačí přidat jen oddělovací transformátor a konektor
- Všechny I/O piny vyvedeny na dvojici 120 pinových board-to-board konektorů
- Potřeba pouze jednoho napájecího napětí *3.3 V*
- Rozměry: *80 x 60 x 8 mm*

Výrobce modul dodává s již nahraným bootloaderem uBoot ([4]), tím je připraven k okamžitému použití bez nutnosti speciálního hardwarového vybavení. Ke komunikaci s modulem slouží sériová konsole, kterou ve výchozím nastavení realizuje jednotka PSC1 (pozn. uBoot ale čísluje PSC od nuly, čili v uBootu bude zobrazena jako PSC0) v konfiguraci 115200 Baudů, 8 datových bitů, 1 stop bit a žádná parita.



Obrázek 2.2: Blokové schéma modulu Shark MPC5200

2.3 Ovládání bootloaderu uBoot

Pro potřeby následujících kapitol je třeba znát několik příkazů bootloaderu, jedná se hlavně o nastavení ethernetu a metody stahování potřebných binárních souborů do paměti modulu.
Chování bootloaderu lze ovlivnit pomocí několika systémových proměnných, my budeme potřebovat následující

Tabulka 2.1: uBoot - Systémové proměnné

Proměnná	Popis
ethaddr	MAC adresa ethernetového řadiče (formát 00:11:22:33:44:55), pokud je nastavena a toto nastavení uloženo, nelze ji již měnit
ipaddr	IP adresa modulu
serverip	IP adresa serveru, ze kterého se budou stahovat soubory
preboot	Příkazy, které se mají provést při startu bootloaderu
bootcmd	Příkazy, které se mají provést pro boot systému
bootdelay	Doba ve vteřinách určující čas, než dojde k automatickému vykonání příkazů v **bootcmd** po startu bootladeru
bootargs	Parametry příkazové řádky, které se předají startovanému operačnímu systému
autoload	Určuje, zda se bude bude při požadavku DHCP/BOOTP automaticky stahovat i definovaný soubor (možné hodnoty YES/NO)
loadaddr	V případě automatického stahování udává adresu, kam se má stažený soubor nahrát (hodnota je uvedena v hexadecimální soustavě)
filename	Název automaticky stahovaného souboru

help

syntaxe: help *[příkaz]*
Vypíše nápovědu k zadanému příkazu, pokud není příkaz uveden, vypíše se seznam všech podporovaných příkazů

dhcp

syntaxe: dhcp
Slouží k automatickému získání IP adresy pomocí DHCP serveru. Pokud je

proměnná *autoload* nastavena na *yes* dojde též ke stažení binárního souboru definovaného v proměnné *filename* na adresu danou proměnnou *loadaddr* pomocí protokolu TFTP ze serveru *serverip*.

bootp

syntaxe: bootp [adresa] [soubor]
Příkaz obdobný příkazu dhcp, ale používá se protokol BOOTP. Parametry *adresa* a *soubor* jsou obdobou k systémovým proměnným *loadaddr* a *filename*.

tftp

syntaxe: tftp [adresa] [soubor]
Příkaz provede stažení souboru definovaného proměnnou *soubor* na adresu *adresa*, soubor se stahuje pomocí TFTP protokolu ze serveru definovaného systémovou proměnnou *serverip*.

boot

syntaxe: boot
Vykoná příkazy definované v systémové proměnné *bootcmd*.

bootm

syntaxe: bootm [adresa [args]]
Spustí boot aplikace z paměti na adrese *adresa* s případnými argumenty *args*. Pro boot operačního systému se také uplatňuje systémová proměnná *bootargs*, která se předává jako parametry příkazové řádky.

setenv

syntaxe: setenv jmeno hodnota
Slouží k definování proměnných. Provede definici proměnné *jmeno* s hodnotou *hodnota*. Pro případ vymazání proměnné se zadává prázdná hodnota.

printenv

syntaxe: printenv [jmeno]
Slouží k vypsání obsahu proměnné *jmeno*, pokud není jméno uvedeno, dojde k vypsáni obsahu všech definovaných proměnných.

saveenv

syntaxe: saveenv
Uloží nastavení systémových proměnných do trvalého úložiště, v případě modulu Shark se jedná o pamět Flash.

ping

syntaxe: ping ip_adresa

Jednoduchý příkaz pro testování funkčnosti sítě.

md

syntaxe: md[.b/.w/.l] adresa [pocet]

Slouží k vypsání obsahu paměti od adresy *adresa*. Pokud je definován *pocet* dojde k vypsání daného počtu objektů. Modifikátory b, w a l slouží k definování velikosti objektu - b .. byte (8b), w .. word (16b) a l .. long (32b).

mw

syntaxe: mw[.b/.w/.l] adresa hodnota [pocet]

Slouží k zápisu hodnoty *hodnota* do paměti na adresu *adresa* a případně na následujících *count* adres. Modifikátory b, w a l slouží k definování velikosti objektu - b .. byte (8b), w .. word (16b) a l .. long (32b).

cp

syntaxe: cp[.b/.w/.l] zdroj cil pocet

Slouží ke zkopírování počtu *pocet* objektů z adresy *zdroj* na adresu *cil*. Modifikátory b, w a l slouží k definování velikosti objektu - b .. byte (8b), w .. word (16b) a l .. long (32b).

erase

syntaxe: erase start konec

Slouží k vymazání sektorů v FLASH paměti, sektory jsou určeny počáteční adresou *start* a koncovou *konec*. (pozn. Na modulu Shark je u FLASH paměti velikost sektoru 128 kB, počáteční adresa musí tedy mít spodních 17 bitů nulových, koncová naopak jedničkových)

Příklad stažení binárního souboru do modulu

Tento příklad demonstruje nastavení statické IP adresy a stažení binárního souboru *uImage* do modulu na adresu *0x400000*

```
setenv ipaddr 192.168.0.121
setenv serverip 192.168.0.6
setenv autoload no
tftp 400000 uImage
```

pokud se jedná o image operačního systému lze jej spustit zadáním příkazu

```
setenv bootargs ...
bootm 400000
```

Kapitola 3

Linux

3.1 Pojem Linux

Linux je moderní modulární operační systém podobný Unixu. Používá monolitické jádro, které má pod sebou správu procesů, řízení sítě, periferií a souborových systémů. Toto jádro je možné zkompilovat pro většinu dostupných architektur, z těch nejznámějších jmenujme alespoň x86 / x86_64, ARM, PowerPC či MIPS. Pro každou z podporovaných architektur je k dispozici výběr z několika předdefinovaných konfigurací pro použitou základnovou desku. Ovladače zařízení jsou přímou součástí stromu zdrojových kódů jádra. Existují dva způsoby použití ovladače:

- Zakompilování ovladače přímo do výsledného souboru jádra
  - Výhody - Vyšší bezpečnost, menší velikost oproti řešení s moduly
  - Nevýhody - Hůře se ladí, pro současnou podporu velkého množství hardware by vycházela neúměrná velikost jádra
  - $\Rightarrow$ Vhodné pro ovladače nezbytně nutné k základní inicializaci hardware
- Kompilace ovladače jako modul
  - Výhody - Snažší vývoj a ladění, výhodnější pro tvorbu univerzálních kompilací, menší paměťová náročnost - v paměti je jen to, co je třeba
  - Nevýhody - Jistá režie potřebná k fungování modulů - načítání / uvolňování modulů, kompatibilita modulů napříč různými verzemi jádra

K dispozici jsou ovladače pro nejrůznější hardware, ať už pro standartní hardware vyskytující se ve stolních počítačích tak i pro speciální integrované periferie v procesorech pro embedded aplikace a periferní obvody připojitelné přes sběrnice typu I^2C

nebo SPI a další.
Hlavním rozdílem Linuxu od ostatních operačních systémů je to, že jeho jádro a příslušné aplikace jsou z většiny šířeny pod licencí GNU GPL a jsou k dispozici úplně zdarma včetně všech zdrojových kódů. Toto tedy umožňuje, že si každý může zkompilovat systém přesně na míru k jeho požadavkům.

3.2 Kompilace Linuxu a aplikací

Cílem této sekce je popsání celého procesu kompilace jádra, přidružených aplikací až ke spuštění celého systému na embedded modulu Shark.
K vlastní kompilaci budeme potřebovat následující:

- Embedded linux development kit pro PowerPC (zkráceně ELDK) (všechny soubory prezentované v této práci se vztahují k verzi 4.1, nicméně v době dokončování je již k dispozici verze 4.2) [4]
- Linuxové jádro (verze 2.6, v dalším textu bude odkazováno na verzi 2.6.23.13) [6]
- Sada aplikací a utilit pro správu a používání systému, v našem případě využijeme BusyBox verze 1.9 [8]
- Přístup k počítači s nainstalovaným systémem Linux (v mém případě se jedná o Suse Linux 10.1)

3.2.1 Nastavení hostitelského systému a instalace ELDK

Pro potřeby práce s embedded modulem je nutné nebo vhodné mít na hostitelském počítači nasintalovány následující aplikace:

- TFTP server (nutný, slouží pro stahování binárních souborů do modulu, dále v textu je předpokládána cesta k rootu serveru /tftpboot/)
- DHCP server (není bezpodmínečně nutný, ale zvyšuje univerzálnost konfigurace uložené v modulu)
- Minicom či jakýkoliv jiný sériový terminál (nutný, slouží ke komunikaci s modulem pomocí sériové konzole)

- Přístup k NFS serveru (nutný, slouží k zpřístupnění kořenového adresáře, před jeho nakopírováním do FLASH paměti)

Všechny výše uvedené aplikace jsou k dispozici na instalačních médiích Suse Linux.

3.2.1.1 TFTP Server

Konfigurace TFTP odpovídá požadavkům, stačí tedy v souboru */etc/xinet.d/tftp* změnit u řádku

```
disable = yes
```

hodnotu *yes* na *no* a pomocí následujícího příkazu restartovat démona xinetd (musíte být pochopitelně uživatel root)

```
/etc/init.d/xinetd restart
```

3.2.1.2 DHCP Server

Pokud bude využíván DHCP server, tak jeho konfiguarce spočívá v editaci souboru */etc/dhcpd*, pro nás je postačující následující obsah tohoto souboru

```
# konfigurace dhcp serveru

ddns-update-style none; ddns-updates off;

subnet 192.168.0.0 netmask 255.255.255.0
{
    default-lease-time 14400;
    max-lease-time 14400;
    option domain-name-servers 192.168.0.1;
    option routers 192.168.0.1;
    option subnet-mask 255.255.255.0;
    range 192.168.0.31 192.168.0.100;

    host mpc5200-uboot-or-linux
    {
        hardware ethernet 00:04:9f:00:27:5f;
        fixed-address 192.168.0.121;
        next-server 192.168.0.6;
    }
}
```

Toto nastavení zajistí, že modul dostane vždy IP adresu 192.168.0.121 (bude funkční i BOOTP) a výchozí server pro TFTP bude v U-bootu počítač s IP adresou 192.168.0.6.

Server spustíme následujícím příkazem, zařízení na kterém bude server spuštěn je síťové zařízení *eth0*

```
/usr/sbin/dhcpd eth0
```

3.2.1.3 Minicom

Konfigurace minicomu spočívá v nastavení jména souboru zařízení sériového portu (čili ono /dev/ttyS* pro pevné porty nebo /dev/ttyUSB* pro USB porty) a parametrů sériového portu zmíněných dříve v textu (zde), konfigurační režim se spouští zadáním příkazu

```
minicom -s
```

3.2.1.4 NFS Server

Pro služby NFS serveru je nutné do souboru /etc/exports vložit následující obsah (předpokládá se, že adresář s kořenovým adresářem bude na cestě /tftpboot/...

```
/tftpboot                       *(no_root_squash,rw)
```

Poté je nutné NFS server restartovat

```
/etc/init.d/nfsserver restart
```

Dalším krokem je Embedded Linux development kit, ten je distribuován buď jako obraz ISO nebo jako samostatné RPM balíčky. Pro značnou velikost onoho ISO obrazu (cca 800 MB) a potřebu jen malé jeho části, je vhodnější stáhnout z FTP serveru (ftp://ftp.denx.de/pub/eldk/4.1/ppc-linux-x86/distribution) pouze následující RPM balíčky:

- crosstool-powerpc-devel-0.35-9.i386.rpm (adresář RMPS, velikost 12 MB)
- crosstool-targetcomponents-ppc_6xx-0.35-9.ppc.rpm (adresář ppc_60x/RPMS, velikost 20 MB)
- u-boot-ppc_6xx-1.2.0-1.ppc.rpm (adresář ppc_60x/RPMS, velikost 9 MB)

První z nich je překladač a linker, jde o cross-compiler (tzn. že na jedné platformě kompilujeme pro jinou, v našem případě tedy na *x86* kompilujeme pro *powerpc*) gcc

a kolekci binutils též pro powerpc. Druhý balíček jsou knihovny a nezbytné soubory pro cílovou platformu (stanou se tedy částí kořenového adresáře). Kompilátor a linker jsem si nainstaloval namísto do */usr* do */opt/eldk*. Pro použití kompilátoru je nutné ještě zadat následující příkazy

```
export PATH=\$PATH:/opt/eldk/bin
export CROSS\_COMPILE=powerpc-linux-
```

Do adresáře /opt/eldk/powerpc-linux/ je nutné zkopírovat adresáře *include* a *lib* z RPM balíčku targetcomponents.
Nyní je nutné do zvoleného adresáře dekomprimovat soubor se staženým jádrem Linuxu, v mém případě jsem se rozhodl pro domovský adresář (též je možné použít osvědčený adresář */usr/src*). Po dekomprimaci vytvoříme následující symbolické odkazy:

- /opt/eldk/powerpc-linux/include/**asm** na /home/*/linux-2.6.23.13/include/asm-powerpc
- /opt/eldk/powerpc-linux/include/**asm-generic** na /home/*/linux-2.6.23.13/include/asm-generic
- /opt/eldk/powerpc-linux/include/**linux** na /home/*/linux-2.6.23.13/include/linux

Posledním krokem je zkopírování utility *mkimage* z ubootu do */usr/bin*. Tímto je příprava hostitelského počítače hotova. Poznámka na konec, celý kořenový adresář budeme vytvářet v adresáři /tftpboot/rootfs

3.2.2 Kompilace jádra Linuxu

Kompilace jádra je klíčovým krokem v přípravě na spuštění Linuxu na modulu Shark. Prvním krokem je aplikace příslušných oprav (patchů) na stažené a rozbalené Linuxové jádro. Konkrétně jde o doplnění ovladače pro následující periferie

- Bestcomm DMA controller
- FEC - **F**ast **E**thernet **C**ontroller

Tyto úpravy jsou nutné neboť v době psaní této práce nebyly potřebné ovladače pro platformu *PPC* součástí standartního release jádra, potřebný soubory se jmenuje *linux_bestcomm_fec.patch* a je součástí přiloženého CD. Aplikace patche se provede zkopírováním tohoto souboru do adresáře s rozbaleným jádrem a vykonáním následujícího příkazu

```
patch -p1 < linux_bestcomm_fec.patch
```

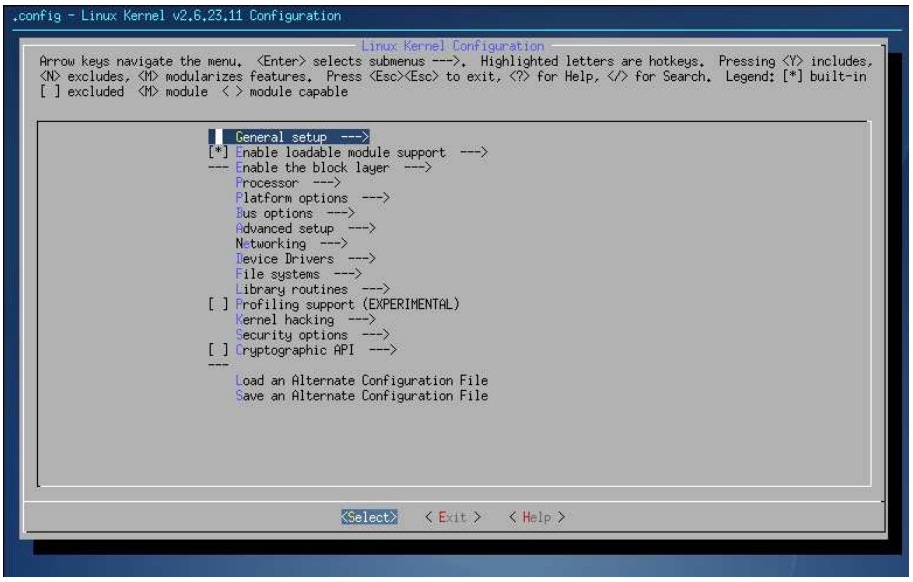
Nyní již můžeme přistoupit k vlastní konfiguraci jádra, výchozím bodem pro nás bude konfigurace pro desku Freescale Lite5200B (jde o vývojový kit pro MPC5200), které se modul Shark podobá. Tuto konfiguraci aplikujeme spuštěním příkazu

```
make ARCH=ppc lite5200_defconfig
```

Po skončení tohoto příkazu následuje podrobné dokonfigurování všech ovladačů a vlastností jádra. Pro tyto účely je k dispozici několik metod výběru vhodných komponent a konfigurace, pro informaci uvedu následující

- *make config* - konfigurace v textovém režimu, spočívající v odpovědích na otázky typu zahrnout komponentu Ano/Ne/Modul
- *make menuconfig* - konfigurace také v textovém režimu, ale pomocí konfiguračních dialogů a formou stromového seznamu komponent (GUI vytvořeno pomocí knihovny NCourses)
- *make xconfig* - konfigurace pomocí grafického rozhraní v prostředí X11, principielně podobné volbě menuconfig

Osobně volím konfiguraci přes *menuconfig*, konfigurace je po uživatelské stránce vcelku přátelská a je možné též touto metodou kompilovat jádro i přes vzdálenou konsoli bez nutnosti řešit tunelování X serveru, který je náročnější na objem přenesených dat.



Obrázek 3.1: Linux - Rozhraní pro konfiguraci jádra

Po spuštění příkazu

```
make ARCH=ppc menuconfig
```

se nám zobrazí rozhraní dle předchozího obrázku (3.1). Konfigurace je rozdělena do několika kategorií, z nichž nejdůležitější sekce jsou *Processor*, *Networking*, *Device Drivers* a *File systems*. Výpis kompletní konfigurace je na přiloženém CD.

Kompilaci celého jádra spustíme pomocí sady následujících příkazů

```
make ARCH=ppc uImage
make ARCH=ppc modules
make ARCH=ppc modules\_install INSTALL\_MOD\_PATH=/tftpboot/rootfs
```

Pro vysvětlení uvedu, že první příkaz slouží ke kompilaci vlastního binárního souboru jádra, druhý pro kompilaci ovladačů zvolených jako moduly a poslední ke zkopírování zkompilovaných modulů do adresáře, kde je umístěn kořenový adresář pro modul Shark.
Pro otestování základní funkcionality jádra je možné jej nahrát do modulu, boot by měl samozřejmě skončit s chybou ohledně nemožnosti připojení kořenového adresáře. Pro tyto potřeby si již dopředu nadefinujeme v u-bootu proměnnou bootargs následovně

```
console=ttyPSC,115200 root=/dev/nfs rw nfsroot=/tftpboot/rootfs ip=dhcp
```

3.2.3 Kompilace potřebných aplikací

V tomto kroku je potřeba zkompilovat sadu aplikací pro fungování a práci se systémem. Konkrétně jde například o aplikace zajišťující start a inicializaci systému, správu modulů nebo příkazový interpret. Místo kompilace několika samostatných aplikací, jsem zvolil ucelenou sadu s názvem Busybox (www.busybox.net). Konfigurace a kompilace probíhá podobným způsobem jako v případě jádra, pro konfiguraci je opět dostupné grafické rozhraní formou stromové struktury napsané pomocí NCourses.
Výpis kompletní konfigurace je stejně jako v případě jádra umístěn na přiloženém CD.
Celý proces lze shrnout do následujících kroků

```
make menuconfig
make
make install
```

Pro výběr umístění, kam zkompilované soubory zkopírovat, slouží volba v menu 'Busybox Settings' → 'Installation Options' → 'Busybox installation prefix'.

3.2.4 Příprava kořenového adresáře

V této fázi máme v příprave kořenového adresáře již nahrány potřebné aplikace a moduly pro jádro, zbývá nám tedy dokopírovat knihovny a vytvořit konfiguraci systému. V kořenovém adresáři vytvoříme následující adresáře: dev, etc, proc, sys a var. Knihovny zkopírujeme z balíčku crosstool-targetcomponents, do kořenového adresáře z něj zkopírujeme pouze */etc*, */lib* a */usr/share*.
Dalším krokem je vytvoření souborů zařízení v adresáři /dev. Těmi nepostradatelnými jsou: console, kmem, kmsg, null. Jejich vytvoření zajistíme následujícími příkazy

```
mknod /tftpboot/rootfs/dev/console c 5 1
mknod /tftpboot/rootfs/dev/kmem c 1 2
mknod /tftpboot/rootfs/dev/kmsg c 1 11
mknod /tftpboot/rootfs/dev/null c 1 3
```

Obsah souborů v adresáři /etc zde z důvodů přehlednosti vypisovat nebudu, jsou umístěny na přiloženém CD v souboru linux_etc.tar.gz. V adresáři /tftpboot/rootfs vytvoříme symbolický odkaz *linuxrc* směřující /bin/busybox kořenového adresáře. Tímto je příprava hotova.

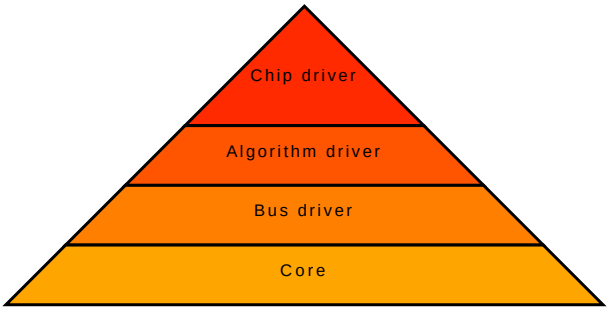
3.2.5 Start systému

Jádro máme připraveno, kořenový adresář také, můžeme přistoupit k poslednímu kroku a to ke startu systému. Start provedeme pomocí násleudjících příkazů v u-bootu

```
dhcp
tftp 400000 uImage
set bootargs console=ttyPSC0,115200 root=/dev/nfs nfsroot=/tftpboot/rootfs rw ip=dhcp
bootm
```

3.3 I2C Subsystém

V této sekci bych se rád zmínil o struktuře I2C subsystému obsaženém v Linuxu. Tento subsystém lze rozdělit do čtyř vrstev dle následujícího obrázku



Obrázek 3.2: Linux - Vrstvy I2C subsystému

Ve stručnosti lze jednotlivé vrstvy popsat takto

- *Core* - samotný základ celého subsystému
- *Bus driver* - má na starost řízení konkrétního řadiče I2C
- *Algorithm driver* - zajišťuje přenos konkrétních bloků dat po sběrnici, využívá při tom metod *bus driveru*
- *Chip driver* - zajišťuje obsluhu konkrétního čipu na sběrnici pomocí posílání zpráv přes *algorithm driver*

Já jsem se zaměřil pouze na vrstvu *Chip driver*, jako příklad využiji ovladač pro RTC M41T00, tento ovladač je vlastně přepsaný původní ovladač dodávaná s jádrem verze 2.6.23, využívající nový systém registrace I2C ovladaču. Bohužel se mi jej však nepodařilo na desce Shark s tímto RTC zprovoznit a proto bylo rozhodnuto ovladač z části přepsat na použití původního systému I2C a tím bylo docílené požadované funkčnosti.
Pro fungování ovladače jsou nejdůležitější funkce s následujícími předpisy

```
static __init int ds1307_init(void);
static __exit void ds1307_exit(void);
```

Tyto funkce jsou volány při inicializaci a deinicializaci ovladače a jejich jedinou akcí je zaregistrování příslušného ovladače pro I2C zařízení. Ten je popsán strukturou *i2c_driver* a definován takto

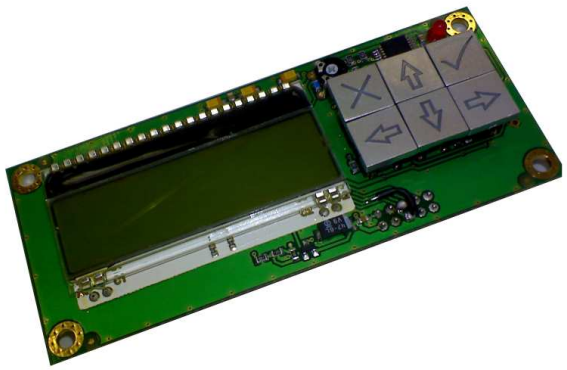
```
static struct i2c_driver ds1307_driver = {
        .driver = {
                        .name        = "rtc-ds1307",
        },

        .id                = I2C_DRIVERID_STM41T00,
        .attach_adapter        = &ds1307_attach,
        .detach_client        = &ds1307_detach,
};
```

Funkce *ds1307_attach* a *ds1307_dettach* jsou volány při přiřazení adapteru k ovladači. Cílem *attach* funkce je zjištění přítomnosti zařízení na sběrnici, pokud je nalezeno vrací se nulová hodnota, jinak záporný chybový kód. Existenci zařízení má na starosti funkce *i2c_probe*, ta provede prvotní ověření podporovaných funkcí I2C adapteru, inicializaci a nastavení potřebných struktur a pokusí se přečíst čas ze zařízení. Po tomto následuje kontrola přečteného data (v případě tohoto ovladače je většina těchto kontrol zakomentována z důvodů prvotního nastavení neboť v tomto okamžiku je hodnota registrů čistě náhodná a toto většinou vede k selhání detekce i v případě, že je vše v pořádku). Pokud žádná z těchto kontrol neskončí chybou, je zaregistrováno nové RTC zařízení, to musí definovat dvě funkce - čtení a nastavení času, za toto jsou zodpovědné funkce *ds1307_get_time* a *ds1307_set_time*.
Kompletní zrojový kód je součástí přílohy A.1.

3.4 SPI Subsystém

Zatímco v případě I2C je v jádře subsystém hiearchicky navržen, v případě SPI subsystému tomu doposud není. Cílem bude vytvořit jednoduchý modul ovladače znakového zařízení, které pomocí přímé obsluhy SPI řadiče obsaženého v procesoru MPC5200 komunikuje se znakovým LCD typu EA DOGM([5]).



Obrázek 3.3: SPI Display

Pro fungování ovladače jsou nejdůležitější funkce s následujícími předpisy, analog-

icky jako u I2C subsystému

```
static __init int spilcd_init(void);
static __exit void spilcd_exit(void);
```

Tyto funkce jsou volány při inicializaci a deinicializaci ovladače a jejich jedinou akcí je zaregistrování příslušného znakového zařízení. Definicí tohoto zařízení je struktura file_operations, která definuje funkce provedené při volání tradičních funkcí pro práci se soubory. Pro LCD jsou definovány následující operace, otevření souboru, zápis do souboru a I/O control. Struktura bude tedy mít tuto definici

```
struct file_operations spilcd_fops = {
        .owner      = THIS_MODULE,
        .poll       = spilcd_poll,
        .ioctl      = spilcd_ioctl,
        .write      = spilcd_write,
        .open       = spilcd_open,
        .release    = spilcd_release,
};
```

Význam funkcí je zřejmý již z jejich názvů, zde tedy uvádím stručný popis včetně jejich potřebných prototypů

- *spilcd_ioctl* - je zavolána při použití ioctl z uživatelské aplikace
  *static int spilcd_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)*
  v uživatslkém módu jí odpovídá funkce *ioctl*
- *spilcd_write* - volána při operaci zápisu do souboru
  *static ssize_t spilcd_write(struct file *file, const char *buf, size_t count, loff_t *off)*
  v uživatslkém módu jí odpovídá funkce *write*
- *spilcd_open* - volána při otevírání souboru
  *static int spilcd_open(struct inode *inode, struct file *filp)*
  v uživatslkém módu jí odpovídá funkce *open*
- *spilcd_relase* - volána při zavření souboru
  *static int spilcd_release(struct inode *inode, struct file *filp)*
  v uživatslkém módu jí odpovídá funkce *close*

Výpisy zmíněných funkcí jsou uvedeny v příloze B.1. Driver nemá implementovánu funcki čtení, vzhledem k tomu, že LCD toto neumožňuje.

Za iniciliazaci displaye je zodpovědná fuknce spilcd_open, což je provedeno ve dvou krocích. Tím prvním je nastavení a připrava SPI hardware (pro podrobný popis odkazuji na katalogový list procesoru MPC5200), druhým již zmíněná inicializace LCD, která je provedena dle doporuření výrobce uvedeného v katalogovém listu LCD. Tím

dojde k výchozímu nastavení parametrů displaye, jeho vymazání a umístění kurzoru do levého horního rohu (na pozici [0,0]).

Pro zápis dat na display je určena funkce spilcd_write, její kód je přímočarý, provede pouhé odeslání dat pomocí SPI sběrnice do displaye. Při zápisu jsou však respektovány výskyty znaků nové řádky (LF, 0xa) a návrat vozíku (CR, 0xd), kdy provede příslušnou změnu pozice kurzoru, také hlídá aby text z jedné řádky "'nepřetekl"' do řádky druhé.

Výmaz displaye, změna pozice, kontrastu a nuceně vyvolaná inicializace jsou řešeny pomocí I/O volání, přes metodu ioctl. Driver implementuje celkem 4 tyto funkce s přiřazenými čísly od nuly do tří. Ty jsou přiřazeny následovně

Tabulka 3.1: LCD ovladač - I/O volání

Číslo volání	**Popis**	**Argument**
0	Inicializace LCD	Pokud je nenulový udává výchozí kontrast, jinak nastavena hodnota přibližně v třetině rozsahu, povolený rozsah 0-63
1	Výmaz displaye	*nevyužito*
2	Nastavení pozice	Vyšších 8b pozice X (rozsah 0-15), nižších 8b pozice Y (rozsah 0-2)
3	Nastavení kontrastu	Požadovaný kontrast, rozsah 0-63

Pro použití těchto kódů se dá vhodně použít makro *_IO(IOC_TYPE, cislo)*, kde IOC_TYPE je pro tento ovladač rovno 91. Pro použití v systému je ještě nutné vytvořit daný soubor zařízení, to provedeme pomocí příkazu

```
mknod /dev/lcd c 120 20
```

Driver se registruje do systému jako znakové zařízení (pro to to **c**), s hlavním číslem 120 a vedlejším 20.

Na závěr uvedu pro příklad krátký úsek kódu v jazyce C pro nastavení pozice na 4. znak na řádce 1 (číslovány od nuly) a výpis textu "'Shark LCD"'

```
#include <sys/ioctl.h>

...
int fd = open("/dev/lcd", O_WR);
if(fd > 0)
 {
  ioctl(fd, _IO(91, 2), 0x0301);
  ...
  write(fd, "Shark LCD", 9);
  ...
  close(fd);
 }
...
```

a též výstup z konsole na display.

```
cat /etc/passwd > /dev/lcd
```

Kapitola 4

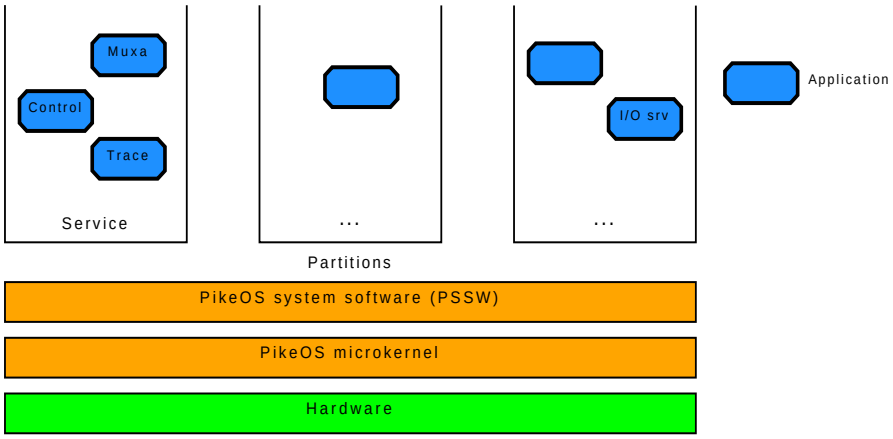
PikeOS

4.1 Popis

PikeOS ([7]) je platforma pro vývoj embedded zařízení, kde může běžet více operačních systémů a aplikací simultánně v odděleném a robustním prostředí. Architektura PikeOS je založena na mikrokernelu, které poskytuje minimální skupinu služeb. PikeOS rozlišuje dva typy úloh - vlákna(threads) a úlohy(tasks).

- *thread* - je část spustitelného kódu, která má přístup k datům a zásobníku, v PikeOS se jedná o nejmenší plánovací jednotku. Každé vlákno má přiřazen časový slot, task a přidělené zdroje
- *task* - popisuje oddělený paměťový prostor, který je sdílen všemi vlákny dané úlohy

Jádro PikeOS používá preemptivní multitasking, implementuje plánování na základě priorit a podporuje správu časového plánování a zdrojů. Nad tímto mikrojádrem je implementována vrstva PikeOS System Software (PSSW), která poskytuje služby, které bývají implementovány v tradičních monolitických jádrech, jako je spouštění nových aplikací či přístup k souborovému systému. Tato vrstva též implementuje chráněné prostředí pro běh aplikací tzv. Partitions. Tyto aplikace mohou být úlohy od jednodušších nativních úloh PikeOS až po plnohodnotný operační systém.
Na následujícím obrázku (4.1) je zobrazena architektura PikeOS.



Obrázek 4.1: Architektura PikeOS

4.2 Vrstva PSSW

Jak bylo uvedeno dříve, jedná se společně s mikrojádrem o nejdůležitější vrstvu celého PikeOS. Tato vrstva je složena ze dvou částí, tou první je vlastní PSSW modul, který je postaven přímo nad mikrojádrem a je společný všem partitions. Tento modul poskytuje rozhraní pro druhou část, kterou tvoří PSSW knihovna standardně přilinkovaná k jednotlivým běžícím partitions.
PSSW modul řídí následující činnosti

- Konfiguraci systému - založena na tabulce VMIT (Virtual machine initialization table), ta je pevnou součástí binarního ROM image a je načítána během startu systému
- Přístup k souborovému systému
- Řízení partitions - vytváření, restart, ukončení a získávání stavu partition
- Řízení aplikací - vytváření a spouštění
- Monitoring stavu celého systému
- Řízení časového plánování včetně možnosti změny celého plánovacího schématu
- Vzájemná komunikace procesů - sdílená pamět (SHM), FIFO, signály

4.3 Partitions

Na partition se můžeme dívat jako na kontejner pro uživatelské aplikace, se staticky přiřazenou skupinou zdrojů a privilegií. Tyto partitions jsou vytvářeny během startu systému z VMIT a *nemohou* být za běhu *měněny*. Každá tato partition má několik atributů, mezi které patří

- Partition Identification - jedinečné nenulové číslo
- Privilege level - úroveň oprávnění. Partition může být buď *normální* nebo *servisní*. Systémová partition může navíc oproti normální provádět následující činnosti: restartovat a spouštět jiné partition, měnit plánovací schéma, zastavit a restartovat celý systém
- Maximální priorita
- Přiřazená časová partition
- Přístupová práva k souborům
- a další ... (podrobněji uvádí manuál k PikeOS, [7])

Každé partition se též nastavuje přístup k systémové paměti a celého adresního prostoru. Kvůli použité jednotce správy paměti (MMU - memory managment unit) a virtualizaci daná partition nemá přístup do adresního prostoru, je tedy minimálně nutné definovat určitý region v operační paměti pro data a zásobník. Stejným způsobem je také možné přiřadit přístup dané partition k periferním zařízením. Velikost takto přiřazovaným segmentů je však vázána velikostí stránky, která je 4kB, toto v případě přiřazování integrovaných periferií většinou přináší vedlejší efekty. Tyto efekty vedou k tomu, že dané partition nelze ve většině případů přiřadit pouze konkrétní periferii, ale i další, které se nacházejí v přidělované stránce.

Všechna tato přiřazení včetně dalších parametrů (přístupová práva, definice SHM a FIFO, ...) se definují staticky při zakládání integračního projektu a jsou součásti VMIT.

PikeOS rozlišuje několik základních typů partitions

- Nativní úloha PikeOS - nejnižší možná aplikační vrstva
- Posixová úloha PikeOS - jde vlastně o nativní úlohu s vloženou vrstvou překládající funkce definované standartem Posix na odpovídající volání služeb PSSW
- Obecná partition - určená pro spouštění plnohodnotných operačních systémů (např. ElinOS)

4.3.1 Native

Nativní úloha PikeOS je úloha běžící přímo nad vrstvou PSSW. Vývojář má tím pádem největší kontrolu nad chodem aplikace. Znamená to však, že pro veškeré operace je nutné volat přímo funkce PSSW, které jsou někdy příliš obecné a tudíž zesložiťují výsledný zdrojový kód. Vzhledem k absenci POSIX nelze používat TCP/IP stack, je však možné vytvořit v POSIX partition jakýsi I/O server s TCP/IP stackem, který komunikuje s nativní partition pomocí portů či sdílené paměti a nakomunikovaná data předává dále na síť.

4.3.2 Posix

Posixová úloha je zjednodušeně řečeno nativní úloha, nad kterou je postavena vrstva překládající Posixové funkce na funkce vrstvy PPSW, robustnost a bezpečnost této úlohy tedy není dotčena, jinak řečeno, "'Posixová úloha a nativní si jsou z hlediska mikrokernelu téměř rovnocené"'. Oproti nativní úloze běží v každé posixové úloze několik vláken, které zajišťují fungování celé této mezivrstvy, jde například o zajištění konzole či fungování volání meziprocesní komunikace.

4.3.3 ELinOS

Pro nás nejzajímavějším typem úlohy je partition s bežícím systémem ElinOS. Jedná se o speciálně upravné linuxové jádro, do standartního jádra je doplněna architektura s názvem *P4*. Tato architektura řeší komunikaci nejspodnější vrstvy linuxového jádra s mikrojádrem PikeOS a překládá linuxová volání jednotky správy paměti na volání funkcí a služeb vrstvy PSSW.

4.4 Prostředky komunikace mezi partitions

PikeOS poskytuje následující protředky komunikace mezi partitions:

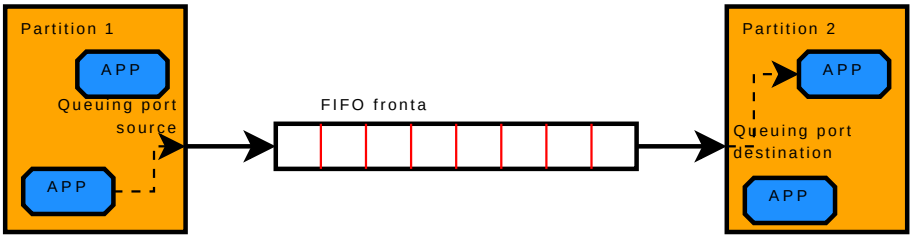
- Porty - lze si je představit jako fronty FIFO, rozlišujeme dva typy těchto portů
  - Queuing port

  - Sampling port
- Sdílená paměť
- Signály

Přístup k těmto prostředkům je pomocí volání funkcí dané vrstvy, v případě nativní úlohy se jedná o funkce vrstvy PSSW a v případě POSIX či ElinOS jde o volání patřičných Posix funkcí. V případě portů je nutné upozornit na to, že se jedná vždy o jednosměrný prostředek, pro obousměrnou komunikaci partitions je tedy potřeba dvojice těchto portů.

4.4.1 Queuing porty

Queuing porty se z hlediska uživatele chovají přesně jako fronta FIFO, viz následující obrázek. Specifikem tohoto portu je vlastnost, že zpráva zůstává v portu tak dlouho, dokud není z fronty odebrána. Operace nad queuing portem může být jak blokující, s definovatelným timeoutem, tak i neblokující.



Obrázek 4.2: PikeOS - Queuing port

Konfigurace portu spočívá v nadefinování portu v integračním projektu, propojení dvojice portů a nastavení následujících vlastností portu

- Jméno portu - dle tohoto jména se poté na port přistupuje
- Směr portu - source (data do portu zapisujeme) nebo destination (data z portu čteme)
- Maximální velikost zprávy
- Maximální počet uchovávaných zpráv

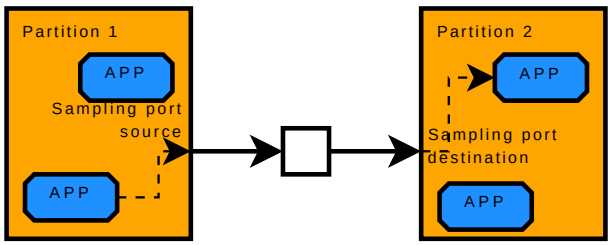
Pro obsluhu portu jsou v PSSW knihovně definovány následující funkce

- vm_qport_open
- vm_qport_write
- vm_qport_read
- vm_qport_pstat, vm_qport_stat
- vm_qport_abort

V Posixu (a tedy i v ElinOSu) se porty ovládají přímo pomocí metod definovaných pro práci se soubory, kde otevíraný soubor je příslušný soubor zařízení. V posixu jsou umístěny na cestě */qport/[nazev portu]*, v ElinOSu jsou standartně umístěny na cestě */etc/vmport*.

4.4.2 Sampling porty

Sampling porty se z hlediska uživatele chovají jako pouhá proměnná, zprávy se neukládají do fronty a tedy pokud si aplikace na druhém konci nestihne hodnotu z portu vyzvednout před příchodem nové, je tato hodnota ztracena, naproti tomu pokud přečteme port vícekrát aniž by do něho byla zapsána nová hodnota, přečteme pokaždé poslední zapsanou hodnotu. Operace nad portem jsou vždy neblokující.

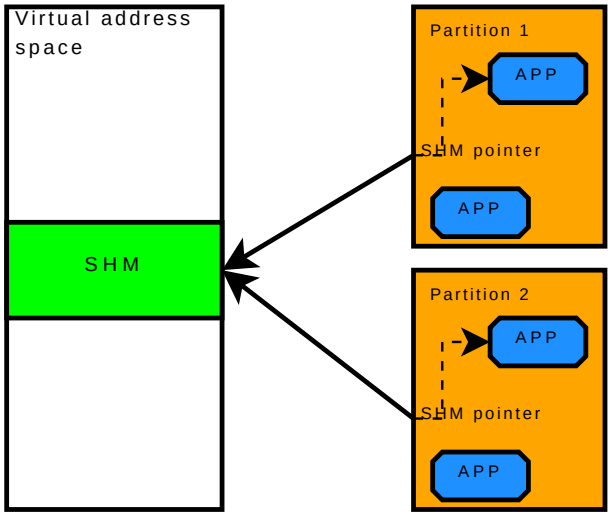


Obrázek 4.3: PikeOS - Sampling port

Pro obsluhu portu jsou v PSSW knihovně definovány stejné funkce jako pro queuing porty s tím rozdílem, že místo qport obsahují sport. V Posixu (a tedy i v ElinOSu) se k sampling portům přistupuje pomocí I/O volání nad příslušným souborem zařízení. V posixu jsou umístěny na cestě */sport/[nazev portu]*, v ElinOSu jsou standartně umístěny na cestě */etc/vmsport*.

4.4.3 Sdílená paměť

Pro výměnu většího množství dat nebo pro komunikace typu bod ⇒ multi-bod je vhodnější využít principu sdílené paměti. Ten umožňuje aby více partitions mělo přístup k shodnému bloku paměti, toto řešení má však jeden principielní problém. I v případě kdy více partitions čte data, která jedna z nich vystavuje, musíme řešit problém jejich synchronizace. PikeOS bohužel neposkytuje API, které by umožňovalo synchronizovat jednotlivé partitions mezi sebou, toto lze pochopit jen díky tomu, že se jedná o realtime systém, kde by metody synchronizace mohly vést vlivem nedomylšených podmínek k potenciálnímu zablokování partitions a tím k pádu systému, což je nepřípustné.



Obrázek 4.4: PikeOS - Sdílená paměť

Se sdílenou pamětí se pracuje podobně jako se souborem, pří nadefinování sdílené paměti je k dispozici v PikeOS soubor na "'jednotce"' SHM (pro příklad, celá cesta může vypadat takto: *SHM:/mojeshm*), v posixu existují přímo funkce pro sdílenou paměť, kterým se předává jen název oblasti a v ElinOSu jsou soubory umístěny v adresáři */dev/vmfileshm/0-n*. Po otevření tohoto souboru je možné pomocí paměťového mapu přistupovat ke sdílené oblasti.

4.4.4 Signály

Signály jsou asynchronní události, kdy běžící vlákno signalizuje událost jinému vláknu. Signál může nabývat pouze dvou stavů, buď je signalizovaný nebo není. Nad signálem je možné provádět následující operace, jeho poslání a čekání na nastavení či volání obslužné rutiny při příjmu signálu.

4.5 Komunikace partitions prakticky

Tato sekce si klade za cíl realizaci jednoduchého systému, kdy budeme z prostředí ElinOS posílat krátkou zprávu do nativní úlohy PikeOS, v tomto případě půjde o Posix partition. Jako komunikační prostředky byly zvoleny *sampling porty*, které v tomto případě realizují praktický případ předávání aktuální hodnoty sdílené veličiny mezi partitions.

Celý proces lze rozdělit do několika následujících kroků

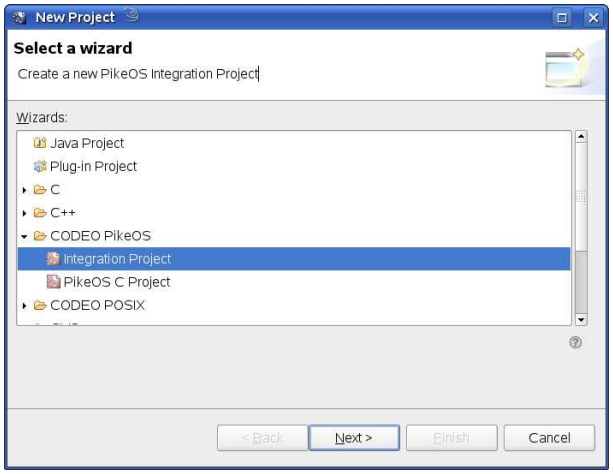
- Tvorba integračního projektu
- Tvorba Posix aplikace
- Příprava a kompilace ElinOS
- Vytvoření portů a jejich propojení
- Tvorba Linux aplikace pro odeslání zprávy

Nyní se podíváme na jednotlivé kroky podrobněji.

4.5.1 Integrační projekt

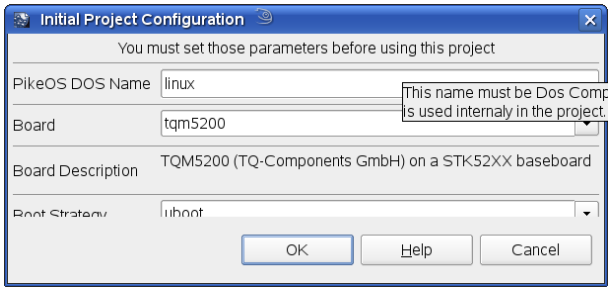
Integrační projekt je nejdůležitější část celého procesu tvorby systému na bázi PikeOS. Tento projekt definuje jaké partitions jsou v systému obsaženy, jaké zdroje mají přiřazeny, konkrétně například množství operační paměti, přiřazené paměťové regiony, přístupová práva k souborům (zahrnuje i práva k sdílené paměti), seznam procesů a pro nás momentálně nejdůležitější seznam definovaných sampling a queuing portů. Integrační porjekt dále obsahuje definice sdílené paměti, propojení portů mezi partitions a plánovací schéma nebo schémata. Celá konfigurace je popsána sadou *XML souborů*, my však budeme využít dodáváné integrované vývojové prostředí (IDE)

s názvem Codeo, které je založené na IDE Eclipse a obsahuje průvodce a nástroje pro snadnou úpravu těchto konfiguračních souborů.



Obrázek 4.5: PikeOS - Vytvoření integračního projektu

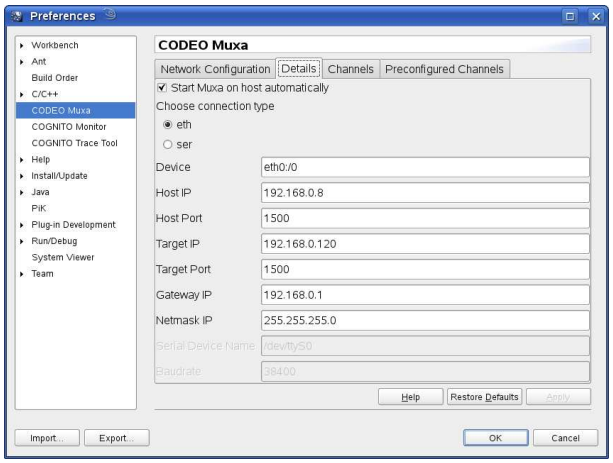
Vytvoříme tedy pomocí průvodce integrační projekt založený na šabloně *devel-linux*, při dotazu na typ desky volíme TQM5200 a bootovací strategii uboot.



Obrázek 4.6: PikeOS - Výběr cílové architektury

Po dokončení průvodce je integrační projekt vytvořen a připraven ke konfiguraci. Ke komunikaci s jednotlivými partitions se využívájí konzole realizované přes ethernetové rozhraní a pomocí prostředku zvaného *muxa*. Konfigurace tohoto se provádí statickým přiřazením IP adresy modulu (součástí binárního obrazu) a to samé je nutné provést i v IDE. Nastavení je přístupné pod menu *Window* → *Preferences* → *CODEO muxa*, kde na záložce *details* vyplníme požadované parametry. Poté na záložce *Network Configuration* použijeme možnost *Export Muxa Configuration* a nastavení si uložíme do

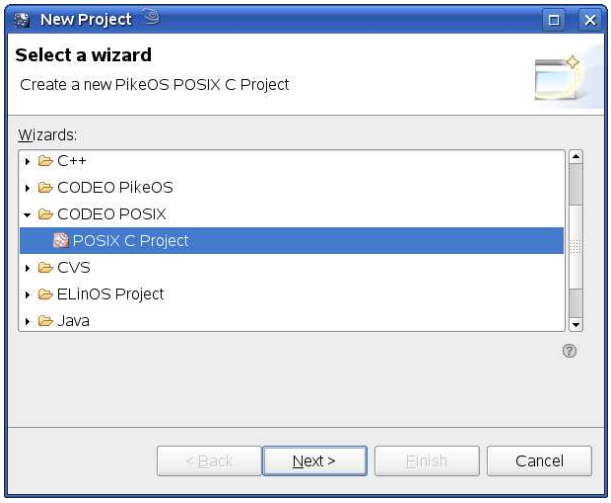
souboru. Zmáčkneme tlačítko *Apply* a *Ok*. Poté si otevřeme soubor project.xml.conf v integračním projektu, rozbalíme položku *service* a použijeme pravé tlačítko myši nad položkou *muxa*, zde vybereme volbu *Import values form file* a načteme dříve vyexportovaný soubor.



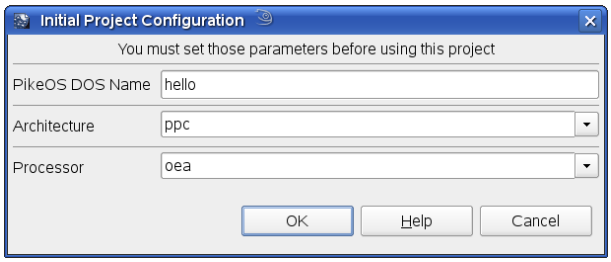
Obrázek 4.7: PikeOS - Konfigurace muxa

4.5.2 Tvorba Posix aplikace

Posix aplikaci vytvoříme opět pomocí průvodce a na základě šablony *hello*, při dotazu na počáteční iniciliazaci projektu volíme jako architekturu *ppc* a procesor *oea*. Průvodce vytvoří základní kostru aplikace, od které odvodíme náši aplikaci.



Obrázek 4.8: PikeOS - Vytvoření posix aplikace



Obrázek 4.9: PikeOS - Incializace Posix aplikace

Aplikace je velice jednoduchá a snadno pochopitelná, na začátku je provedeno otevření patřičného souboru portu a vypsány jeho parametry, poté se v nekonečné smyčce čtou data z portu a vypisují se na konzoli. Příslušný zdrojový kód je uveden v příloze C.1. Posledním krokem je přiřazení aplikace do integračního projektu, to se provede otevřením souboru
project.xml.conf v Posix aplikaci a poté nastavením volby *Integration project* na cestu k němu.

4.5.3 Příprava a kompilace ElinOS

V případě ElinOSu je kompilace jednodušší než v případě samotného Linuxu, je to díky grafickému rozhraní s názvem ELK (*což je zkratka pro Embedded Linux Kit*).

Zde si přehledně zvolíme architekturu cílové platformy, použitou desku, ovladače i co má obsahovat výsledný kořenový adresář (ten se vytvoří automaticky jako soubor *.tgz). Po kompilaci nás zajímá tato trojice souborů, *linux.kernel, linux.params a onen soubor *.tgz*, jméno archivu s kořenovým adresářem je závislé na názvu projektu. Soubor linux.kernel, jak již název napovídá, je zkompilované jádro linuxu pro PikeOS a linux.params obsahuje příkazovou řádku předávanou jádru (zde tedy nepomůže proměnná *bootargs* v ubootu), archiv kořenového adresáře si někam rozbalíme a zpřístupníme přes NFS server. Potřebný linux.params by měl mít pro boot přes NFS minimálně následující obsah

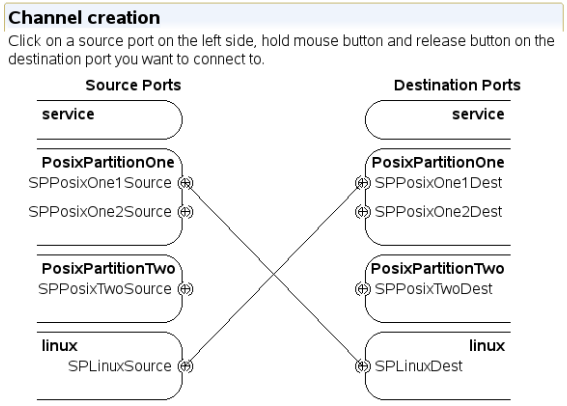
```
fpeth=0,eth0:/1 console=ttyFP0,muxa:/linux ip=bootp devfs=mount root=/dev/nfs
nfsroot=/tftpboot/rootfs-final rw
```

Pro přidání do integračního projektu je třeba udělat následující, otevřít soubor project.xml.conf, kliknutím pravým tlačítkem na název projektu zvolíme *Add* → *Application*, v poli Features List zvolíme linux a v poli Modes List dáme muxa. Tímto vytovříme novou partition, projekt uložíme a otevřeme si v seznamu souborů soubor na cestě *templates/[název partition].rbx.inc* v každém řádku *file* upravíme cestu v atributu *resource*, aby směřovala na vygenerované soubory jádra a jeho soubor s parametry.

Tímto je příprava ElinOSu dokončena.

4.5.4 Vytvoření portů a jejich propojení

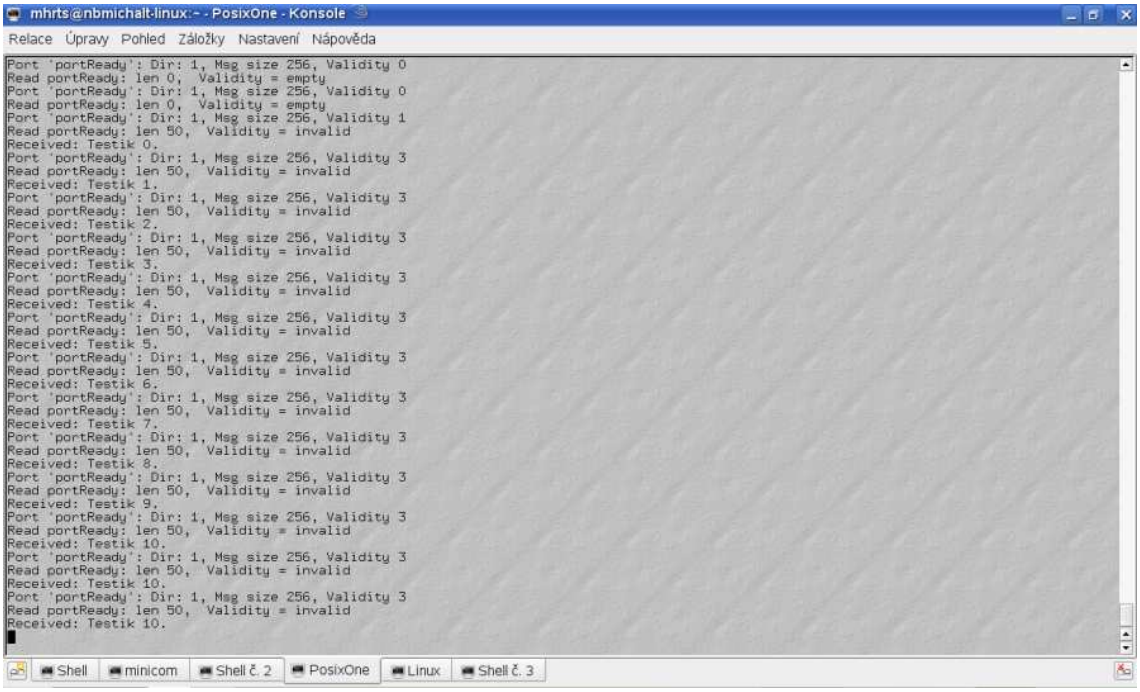
Porty se vytvářejí pomocí editoru souboru *vmit.xml*. Pro vytvoření sampling portu jsou nutné následující kroky. Ve stromu položek si rozbalíme *PartitionTable*, vybereme partition ve které chceme port vytvořit a rozbalíme ji a poté klikneme pravým tlačítkem na položku *SamplingPortList*, kde vybereme Add, vyplníme název portu, jeho směr (Source - data zapisujeme, Destination - data čteme) a maximální velikost zprávy. Tímto postupem vytvoříme jeden zdrojový port v partition odpovídající ElinOSu s názvem SPLinuxSource a jeden cílový v partition Posixu s názvem SPPosixOne1Dest, oba s maximální velikostí 256 B. Posledním krokem je propojení těchto portů, čímž vznikne komunikační kanál, to se provede na kartě *Channel View*. Pomocí metody Drag&Drop natáhneme spoj mezi těmito dvěma porty.



Obrázek 4.10: PikeOS - Definice propojení portů

4.5.5 Tvorba Linux aplikace pro odeslání zprávy

V předchozích krocích došlo k vytvoření celého systému, který je již po kompilaci Posix aplikace a integračního projektu možné načíst do modulu Shark a spustit, jednotlivé konzole jsou přístupné pomocí telnetu na adrese 127.0.0.1 a portech 1502 (Posix) a 1509 (Linux). Zbývá ještě vytvořit aplikaci, která na straně ElinOSu bude posílat na daný port nějaké zprávy. Tato aplikace se víceméně od Posix aplikace nijak výrazně neliší (ElinOS jakožto Linux, definuje též funkce standartu Posix), čili jediná změna je, že data nečte nýbrž zapisuje. Výpis zdrojového kódu je opět uveden v příloze C.2.



Obrázek 4.11: PikeOS - Výstup z posixové aplikace

Kapitola 5

Závěr

Cílem této práce byla kompilace a spuštění systému Linux na embedded modulu Shark a také demonstrace možností realtime systému PikeOS. Tyto úkoly byly úspěšně realizovány, tak jak je popsáno v předchozím textu.

Systém Linux byl zkompilován s podporou většiny integrovaných periferií obsažených v mikroprocesoru MPC5200b s kořenovým adresářem připojovaným přes NFS. V případě PikeOS, byl systém zkompilován naopak jako minimální, s podporou pouze nejnutnějších komponent, kořenový adresář je, stejně jako v případě Linuxu, připojován pomocí NFS. V případě systému Linux byl proveden dlouhodobý test stability, kdy byl modul nasazen do provozu pro řízení v délce 14 dní. Během této doby nebyly zjištěny jakékoliv problémy zapříčiněné programovým či hardwarovým selháním.

Jistý problém celého řešení tkví v následující věci, jádro obou systémů i kořenový adresář jsou stahovány či připojovány z jiného počítače. Z tohoto vyplývá, pro nasazení modulu je potřeba mít na síti k dispozici takto vybavený počítač, což samozřejmě vnáší do takovéto koncepce slabé místo. Dalo by se tedy říci, že toto řešení nemá žádnou výhodu, lze ji však najít ve snažší správě programového vybavení modulu. Představme si nasazení modulu v aplikaci, kde jeden tento modul zabezpečuje řízení malého technologického celku. Několik těchto celků tvoří rozsáhlý výrobní areál a je tedy zbytečné aby na každém tomto celku bylo v paměti flash nahráno stejné programové vybavení, které zabere přibližně polovinu její kapacity.

Díky tomuto konceptu lze modul osadit menšími paměťmi, což vede ke snížení ceny a nebo lze přebytečné místo využít pro další komponenty.

Literatura

[1] *Freescale,* **Doukentace k MPC5200**
`http://www.freescale.com`

[2] *Mikroklima,* **Embedded modul SHARK**
`http://www.mikroklima.cz`

[3] *TQ Components GMBH,* **Modul TQM5200**
`http://tq-components.de`

[4] *Univerzální bootloader U-Boot*
`http://www.denx.de/wiki/UBoot`

[5] *Electronic Assemblies* **LCD Display EA-DOGM**
`http://www.lcd-module.de`

[6] *Linux kernel*
`http://www.kernel.org`

[7] *Sysgo AG - Realtime Solutions* **PikeOS**
`http://www.sysgo.com`

[8] *Busybox*
`http://busybox.net/`

Příloha A

I2C ovladač pro RTC M41T00

A.1 rtc-ds1307

```
1  /*
2   * rtc-ds1307.c - RTC driver for some mostly-compatible I2C chips.
3   *
4   *  Copyright (C) 2005 James Chapman (ds1337 core)
5   *  Copyright (C) 2006 David Brownell
6   *
7   * This program is free software; you can redistribute it and/or modify
8   * it under the terms of the GNU General Public License version 2 as
9   * published by the Free Software Foundation.
10  */
11
12 #include <linux/module.h>
13 #include <linux/init.h>
14 #include <linux/slab.h>
15 #include <linux/i2c.h>
16 #include <linux/string.h>
17 #include <linux/rtc.h>
18 #include <linux/bcd.h>
19
20
21
22 /* We can't determine type by probing, but if we expect pre-Linux code
23  * to have set the chip up as a clock (turning on the oscillator and
24  * setting the date and time), Linux can ignore the non-clock features.
25  * That's a natural job for a factory or repair bench.
26  *
27  * This is currently a simple no-alarms driver.  If your board has the
28  * alarm irq wired up on a ds1337 or ds1339, and you want to use that,
29  * then look at the rtc-rs5c372 driver for code to steal...
30  */
31 enum ds_type {
32   ds_1307,
```

```
33    ds_1337,
34    ds_1338,
35    ds_1339,
36    ds_1340,
37    m41t00,
38    // rs5c372 too?  different address...
39  };
40
41  static unsigned short normal_i2c[] = { I2C_CLIENT_END };
42  I2C_CLIENT_INSMOD;
43
44  /* RTC registers don't differ much, except for the century flag */
45  #define DS1307_REG_SECS    0x00  /* 00-59 */
46  # define DS1307_BIT_CH     0x80
47  # define DS1340_BIT_nEOSC   0x80
48  #define DS1307_REG_MIN     0x01  /* 00-59 */
49  #define DS1307_REG_HOUR    0x02  /* 00-23, or 1-12{am,pm} */
50  # define DS1307_BIT_12HR    0x40  /* in REG_HOUR */
51  # define DS1307_BIT_PM     0x20  /* in REG_HOUR */
52  # define DS1340_BIT_CENTURY_EN  0x80  /* in REG_HOUR */
53  # define DS1340_BIT_CENTURY 0x40  /* in REG_HOUR */
54  #define DS1307_REG_WDAY    0x03  /* 01-07 */
55  #define DS1307_REG_MDAY    0x04  /* 01-31 */
56  #define DS1307_REG_MONTH   0x05  /* 01-12 */
57  # define DS1337_BIT_CENTURY 0x80  /* in REG_MONTH */
58  #define DS1307_REG_YEAR    0x06  /* 00-99 */
59
60  /* Other registers (control, status, alarms, trickle charge, NVRAM, etc)
61   * start at 7, and they differ a LOT. Only control and status matter for
62   * basic RTC date and time functionality; be careful using them.
63   */
64  #define DS1307_REG_CONTROL  0x07    /* or ds1338 */
65  # define DS1307_BIT_OUT    0x80
66  # define DS1338_BIT_OSF    0x20
67  # define DS1307_BIT_SQWE    0x10
68  # define DS1307_BIT_RS1    0x02
69  # define DS1307_BIT_RS0    0x01
70  #define DS1337_REG_CONTROL  0x0e
71  # define DS1337_BIT_nEOSC   0x80
72  # define DS1337_BIT_RS2    0x10
73  # define DS1337_BIT_RS1    0x08
74  # define DS1337_BIT_INTCN   0x04
75  # define DS1337_BIT_A2IE    0x02
76  # define DS1337_BIT_A1IE    0x01
77  #define DS1340_REG_CONTROL  0x07
78  # define DS1340_BIT_OUT    0x80
79  # define DS1340_BIT_FT     0x40
80  # define DS1340_BIT_CALIB_SIGN  0x20
81  # define DS1340_M_CALIBRATION 0x1f
82  #define DS1340_REG_FLAG    0x09
83  # define DS1340_BIT_OSF    0x80
84  #define DS1337_REG_STATUS  0x0f
85  # define DS1337_BIT_OSF    0x80
```

```
86  # define DS1337_BIT_A2I   0x02
87  # define DS1337_BIT_A1I   0x01
88  #define DS1339_REG_TRICKLE  0x10
89
90
91
92  struct ds1307 {
93    u8      reg_addr;
94    u8      regs[8];
95    enum ds_type    type;
96    struct i2c_msg    msg[2];
97    struct i2c_client client;
98    struct i2c_client dev;
99    struct rtc_device *rtc;
100 };
101
102 struct chip_desc {
103   char      name[9];
104   unsigned    nvram56:1;
105   unsigned    alarm:1;
106   enum ds_type    type;
107 };
108
109 static const struct chip_desc chips[] = { {
110   .name   = "ds1307",
111   .type   = ds_1307,
112   .nvram56  = 1,
113 }, {
114   .name   = "ds1337",
115   .type   = ds_1337,
116   .alarm    = 1,
117 }, {
118   .name   = "ds1338",
119   .type   = ds_1338,
120   .nvram56  = 1,
121 }, {
122   .name   = "ds1339",
123   .type   = ds_1339,
124   .alarm    = 1,
125 }, {
126   .name   = "ds1340",
127   .type   = ds_1340,
128 }, {
129   .name   = "m41t00",
130   .type   = m41t00,
131 }, };
132
133 static inline const struct chip_desc *find_chip(const char *s)
134 {
135   unsigned i;
136
137   for (i = 0; i < ARRAY_SIZE(chips); i++)
138   {
```

```
139       if (strnicmp(s, chips[i].name, sizeof chips[i].name) == 0)
140         return &chips[i];
141     }
142     return NULL;
143   }
144
145   static int ds1307_get_time(struct device *dev, struct rtc_time *t)
146   {
147     int    tmp;
148     struct i2c_client *client = to_i2c_client(dev);
149     struct ds1307 *ds1307 = container_of(client, struct ds1307, client);
150
151     /* read the RTC date and time registers all at once */
152     ds1307->msg[1].flags = I2C_M_RD;
153     ds1307->msg[1].len = 7;
154
155
156     tmp = i2c_transfer(client->adapter, ds1307->msg, 2);
157     if (tmp != 2) {
158       dev_err(dev, "%s error %d\n", "read", tmp);
159       return -EIO;
160     }
161
162     dev_dbg(dev, "%s: %02x %02x %02x %02x %02x %02x %02x\n",
163         "read",
164         ds1307->regs[0], ds1307->regs[1],
165         ds1307->regs[2], ds1307->regs[3],
166         ds1307->regs[4], ds1307->regs[5],
167         ds1307->regs[6]);
168
169     t->tm_sec = BCD2BIN(ds1307->regs[DS1307_REG_SECS] & 0x7f);
170     t->tm_min = BCD2BIN(ds1307->regs[DS1307_REG_MIN] & 0x7f);
171     tmp = ds1307->regs[DS1307_REG_HOUR] & 0x3f;
172     t->tm_hour = BCD2BIN(tmp);
173     t->tm_wday = BCD2BIN(ds1307->regs[DS1307_REG_WDAY] & 0x07) - 1;
174     t->tm_mday = BCD2BIN(ds1307->regs[DS1307_REG_MDAY] & 0x3f);
175     tmp = ds1307->regs[DS1307_REG_MONTH] & 0x1f;
176     t->tm_mon = BCD2BIN(tmp) - 1;
177
178     /* assume 20YY not 19YY, and ignore DS1337_BIT_CENTURY */
179     t->tm_year = BCD2BIN(ds1307->regs[DS1307_REG_YEAR]) + 100;
180
181     dev_dbg(dev, "%s secs=%d, mins=%d, "
182       "hours=%d, mday=%d, mon=%d, year=%d, wday=%d\n",
183       "read", t->tm_sec, t->tm_min,
184       t->tm_hour, t->tm_mday,
185       t->tm_mon, t->tm_year, t->tm_wday);
186
187     /* initial clock setting can be undefined */
188     return rtc_valid_tm(t);
189   }
190
191   static int ds1307_set_time(struct device *dev, struct rtc_time *t)
```

```
192 {
193   struct i2c_client *client = to_i2c_client(dev);
194   struct ds1307 *ds1307 = container_of(client, struct ds1307, client);
195   int    result;
196   int    tmp;
197   u8     *buf = ds1307->regs;
198
199   dev_dbg(dev, "%s secs=%d, mins=%d, "
200     "hours=%d, mday=%d, mon=%d, year=%d, wday=%d\n",
201     "write", t->tm_sec, t->tm_min,
202     t->tm_hour, t->tm_mday,
203     t->tm_mon, t->tm_year, t->tm_wday);
204
205   *buf++ = 0;   /* first register addr */
206   buf[DS1307_REG_SECS] = BIN2BCD(t->tm_sec);
207   buf[DS1307_REG_MIN] = BIN2BCD(t->tm_min);
208   buf[DS1307_REG_HOUR] = BIN2BCD(t->tm_hour);
209   buf[DS1307_REG_WDAY] = BIN2BCD(t->tm_wday + 1);
210   buf[DS1307_REG_MDAY] = BIN2BCD(t->tm_mday);
211   buf[DS1307_REG_MONTH] = BIN2BCD(t->tm_mon + 1);
212
213   /* assume 20YY not 19YY */
214   tmp = t->tm_year - 100;
215   buf[DS1307_REG_YEAR] = BIN2BCD(tmp);
216
217   switch (ds1307->type) {
218   case ds_1337:
219   case ds_1339:
220     buf[DS1307_REG_MONTH] |= DS1337_BIT_CENTURY;
221     break;
222   case ds_1340:
223     buf[DS1307_REG_HOUR] |= DS1340_BIT_CENTURY_EN
224         | DS1340_BIT_CENTURY;
225     break;
226   default:
227     break;
228   }
229
230   ds1307->msg[1].flags = 0;
231   ds1307->msg[1].len = 8;
232
233   dev_dbg(dev, "%s: %02x %02x %02x %02x %02x %02x %02x\n",
234     "write", buf[0], buf[1], buf[2], buf[3],
235     buf[4], buf[5], buf[6]);
236
237   result = i2c_transfer(client->adapter, &ds1307->msg[1], 1);
238   if (result != 1) {
239     dev_err(dev, "%s error %d\n", "write", tmp);
240     return -EIO;
241   }
242   return 0;
243 }
244
```

```
245 static const struct rtc_class_ops ds13xx_rtc_ops = {
246   .read_time  = ds1307_get_time,
247   .set_time = ds1307_set_time,
248 };
249
250 static struct i2c_driver ds1307_driver;
251
252 static int ds1307_probe(struct i2c_adapter *adapter, int address, int kind)
253 {
254   struct ds1307   *ds1307;
255   int      err = -ENODEV;
256   int      tmp;
257   const struct chip_desc  *chip;
258   struct i2c_client   *client;
259   struct rtc_device   *rtc;
260
261   chip = find_chip("m41t00");
262   if (!chip) {
263     dev_err(&adapter->dev, "unknown chip type\n");
264     return -ENODEV;
265   }
266
267   if (!i2c_check_functionality(adapter,
268       I2C_FUNC_I2C | I2C_FUNC_SMBUS_WRITE_BYTE_DATA))
269     return -EIO;
270
271   if (!(ds1307 = kzalloc(sizeof(struct ds1307), GFP_KERNEL)))
272     return -ENOMEM;
273
274   client = &ds1307->client;
275   client->addr = address;
276   client->adapter = adapter;
277   client->driver = &ds1307_driver;
278
279   strlcpy(client->name, ds1307_driver.driver.name, I2C_NAME_SIZE);
280
281   ds1307->msg[0].addr = client->addr;
282   ds1307->msg[0].flags = 0;
283   ds1307->msg[0].len = 1;
284   ds1307->msg[0].buf = &ds1307->reg_addr;
285
286   ds1307->msg[1].addr = client->addr;
287   ds1307->msg[1].flags = I2C_M_RD;
288   ds1307->msg[1].len = sizeof(ds1307->regs);
289   ds1307->msg[1].buf = ds1307->regs;
290
291   ds1307->type = chip->type;
292
293   switch (ds1307->type) {
294   case ds_1337:
295   case ds_1339:
296     ds1307->reg_addr = DS1337_REG_CONTROL;
297     ds1307->msg[1].len = 2;
```

```
298
299       /* get registers that the "rtc" read below won't read... */
300       tmp = i2c_transfer(adapter, ds1307->msg, 2);
301       if (tmp != 2) {
302         pr_debug("read error %d\n", tmp);
303         err = -EIO;
304         goto exit_free;
305       }
306
307       ds1307->reg_addr = 0;
308       ds1307->msg[1].len = sizeof(ds1307->regs);
309
310       /* oscillator off?  turn it on, so clock can tick. */
311       if (ds1307->regs[0] & DS1337_BIT_nEOSC)
312         i2c_smbus_write_byte_data(client, DS1337_REG_CONTROL,
313           ds1307->regs[0] & ~DS1337_BIT_nEOSC);
314
315       /* oscillator fault?  clear flag, and warn */
316       if (ds1307->regs[1] & DS1337_BIT_OSF) {
317         i2c_smbus_write_byte_data(client, DS1337_REG_STATUS,
318           ds1307->regs[1] & ~DS1337_BIT_OSF);
319         dev_warn(&client->dev, "SET TIME!\n");
320       }
321       break;
322     default:
323       break;
324     }
325
326   read_rtc:
327     /* read RTC registers */
328
329     tmp = i2c_transfer(adapter, ds1307->msg, 2);
330     if (tmp != 2) {
331       pr_debug("read error %d\n", tmp);
332       err = -EIO;
333       goto exit_free;
334     }
335
336     /* minimal sanity checking; some chips (like DS1340) don't
337      * specify the extra bits as must-be-zero, but there are
338      * still a few values that are clearly out-of-range.
339      */
340     tmp = ds1307->regs[DS1307_REG_SECS];
341     switch (ds1307->type) {
342     case ds_1340:
343       /* FIXME read register with DS1340_BIT_OSF, use that to
344        * trigger the "set time" warning (*after* restarting the
345        * oscillator!) instead of this weaker ds1307/m41t00 test.
346        */
347     case ds_1307:
348     case m41t00:
349       /* clock halted?  turn it on, so clock can tick. */
350       if (tmp & DS1307_BIT_CH) {
```

```
351         i2c_smbus_write_byte_data(client, DS1307_REG_SECS, 0);
352         dev_warn(&client->dev, "SET TIME!\n");
353         goto read_rtc;
354       }
355       break;
356     case ds_1338:
357       /* clock halted?  turn it on, so clock can tick. */
358       if (tmp & DS1307_BIT_CH)
359         i2c_smbus_write_byte_data(client, DS1307_REG_SECS, 0);
360
361       /* oscillator fault?  clear flag, and warn */
362       if (ds1307->regs[DS1307_REG_CONTROL] & DS1338_BIT_OSF) {
363         i2c_smbus_write_byte_data(client, DS1307_REG_CONTROL,
364             ds1307->regs[DS1307_REG_CONTROL]
365             & ~DS1338_BIT_OSF);
366         dev_warn(&client->dev, "SET TIME!\n");
367         goto read_rtc;
368       }
369       break;
370     case ds_1337:
371     case ds_1339:
372       break;
373     }
374
375     tmp = ds1307->regs[DS1307_REG_SECS];
376     tmp = BCD2BIN(tmp & 0x7f);
377 //   if (tmp > 60)
378 //     goto exit_bad;
379     tmp = BCD2BIN(ds1307->regs[DS1307_REG_MIN] & 0x7f);
380 //   if (tmp > 60)
381 //     goto exit_bad;
382
383     tmp = BCD2BIN(ds1307->regs[DS1307_REG_MDAY] & 0x3f);
384 //   if (tmp == 0 || tmp > 31)
385 //     goto exit_bad;
386
387     tmp = BCD2BIN(ds1307->regs[DS1307_REG_MONTH] & 0x1f);
388 //   if (tmp == 0 || tmp > 12)
389 //     goto exit_bad;
390
391     tmp = ds1307->regs[DS1307_REG_HOUR];
392     switch (ds1307->type) {
393     case ds_1340:
394     case m41t00:
395       /* NOTE: ignores century bits; fix before deploying
396        * systems that will run through year 2100.
397        */
398       break;
399     default:
400       if (!(tmp & DS1307_BIT_12HR))
401         break;
402
403       /* Be sure we're in 24 hour mode.  Multi-master systems
```

```
404         * take note...
405         */
406        tmp = BCD2BIN(tmp & 0x1f);
407        if (tmp == 12)
408          tmp = 0;
409        if (ds1307->regs[DS1307_REG_HOUR] & DS1307_BIT_PM)
410          tmp += 12;
411        i2c_smbus_write_byte_data(client,
412            DS1307_REG_HOUR,
413            BIN2BCD(tmp));
414      }
415
416      if ((err = i2c_attach_client(client)))
417          goto exit_free;
418
419      dev_info(&client->dev, "chip found\n");
420
421      rtc = rtc_device_register(client->name, &client->dev, &ds13xx_rtc_ops,
           THIS_MODULE);
422      if (IS_ERR(rtc)) {
423        err = PTR_ERR(rtc);
424        dev_err(&client->dev,
425          "unable to register the class device\n");
426        goto exit_free;
427      }
428
429      i2c_set_clientdata(client, rtc);
430      return 0;
431
432    exit_bad:
433      dev_dbg(&client->dev, "%s: %02x %02x %02x %02x %02x %02x %02x\n",
434          "bogus register",
435          ds1307->regs[0], ds1307->regs[1],
436          ds1307->regs[2], ds1307->regs[3],
437          ds1307->regs[4], ds1307->regs[5],
438          ds1307->regs[6]);
439
440    exit_free:
441      kfree(ds1307);
442      return err;
443    }
444
445    static int __devexit ds1307_remove(struct i2c_client *client)
446    {
447      struct ds1307 *ds1307 = i2c_get_clientdata(client);
448
449      rtc_device_unregister(ds1307->rtc);
450      kfree(ds1307);
451      return 0;
452    }
453
454    static int ds1307_attach(struct i2c_adapter *adapter)
455    {
```

```
456   return(i2c_probe(adapter, &addr_data, ds1307_probe));
457 }
458
459 static int ds1307_detach(struct i2c_client *client)
460 {
461   struct ds1307 *ds1307 = container_of(client, struct ds1307, client);
462   int err;
463   struct rtc_device *rtc = i2c_get_clientdata(client);
464
465   if (rtc)
466     rtc_device_unregister(rtc);
467   if ((err = i2c_detach_client(client)))
468     return err;
469
470   kfree(ds1307);
471
472   return(0);
473 }
474
475 static struct i2c_driver ds1307_driver = {
476   .driver = {
477     .name = "rtc-ds1307",
478 //     .owner  = THIS_MODULE,
479   },
480
481   .id    = I2C_DRIVERID_STM41T00,
482   .attach_adapter = &ds1307_attach,
483   .detach_client  = &ds1307_detach,
484 };
485
486 static int __init ds1307_init(void)
487 {
488   return i2c_add_driver(&ds1307_driver);
489 }
490 module_init(ds1307_init);
491
492 static void __exit ds1307_exit(void)
493 {
494   i2c_del_driver(&ds1307_driver);
495 }
496 module_exit(ds1307_exit);
497
498 MODULE_DESCRIPTION("RTC driver for DS1307 and similar chips");
499 MODULE_LICENSE("GPL");
```

Příloha B

SPI ovladač LCD

B.1 lcddriver.c

```
1  /* $Id: lcddriver.c,v 1.0 2008/05/11 22:00:00 $
2   *
3   * DOGM display over SPI driver
4   *
5   * Copyright (c) 2008 Michal Hrouda
6   *
7   * Authors:     Michal Hrouda      (initial version), based on GPIO driver from
        ETRAX
8   */
9
10 #include <linux/module.h>
11 #include <linux/sched.h>
12 #include <linux/slab.h>
13 #include <linux/ioport.h>
14 #include <linux/errno.h>
15 #include <linux/kernel.h>
16 #include <linux/fs.h>
17 #include <linux/string.h>
18 #include <linux/poll.h>
19 #include <linux/init.h>
20 #include <linux/delay.h>
21 #include <linux/syscalls.h>
22 #include <asm-ppc/mpc52xx.h>
23
24 #include "lcddriver.h"
25
26 //#define DEBUG 1
27
28 static char spilcd_name[] = "EA-DOGM_SPI_driver";
29
30 static int spilcd_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
        unsigned long arg);
```

```
31 static ssize_t spilcd_read(struct file *file, char *buf, size_t count, loff_t *off
       );
32 static ssize_t spilcd_write(struct file * file, const char * buf, size_t count,
       loff_t *off);
33 static int spilcd_open(struct inode *inode, struct file *filp);
34 static int spilcd_release(struct inode *inode, struct file *filp);
35
36 static unsigned int spilcd_poll(struct file *filp, struct poll_table_struct *wait)
       ;
37
38 volatile struct mpc52xx_spi __iomem *spi_module = NULL;
39 /* private data per open() of this driver */
40
41 struct spilcd_private {
42   struct spilcd_private *next;
43   /* These fields are generic */
44   int x;
45   int y;
46 };
47
48 static ssize_t spilcd_sendbuffer(struct file *file, volatile struct mpc52xx_spi *
       spi, const char *buf, size_t count, u8 rs)
49 {
50   int i = 0, iCounter = 30;
51
52   if(!spi)
53     return(0);
54
55   spi->port_data_register = 0x00 | ((rs == 1) ? 0x01 : 0x00); // /CS = 0, RS
56   for(i = 0; i < count; i++)
57   {
58     spi->data_register = buf[i];
59     iCounter = 100;
60     // Wait for transfer end
61     while( (spi->status_register & MPC52xx_SPI_SPIF) == 0);
62     // Additional small delay
63     do
64     {
65       volatile int j = 5;
66     } while(iCounter--> 0);
67
68     spi->status_register = 0;
69   }
70   spi->port_data_register = 0x08; // /CS = 1
71
72   msleep(1);
73
74   return(i);
75 }
76
77 static void spilcd_initdisplay(struct file *file, volatile struct mpc52xx_spi *spi
       , int arg)
78 {
```

```
79    struct spilcd_private *priv_data = (struct spilcd_private*)file->private_data;
80    char buffer[2];
81    char c = arg & 0xff;
82    if(c == 0)
83      c = 0x12;
84
85    buffer[0] = 0x39; spilcd_sendbuffer(file, spi_module, buffer, 1, 0);
86    buffer[0] = 0x15; spilcd_sendbuffer(file, spi_module, buffer, 1, 0);
87    buffer[0] = 0x54 | (c >> 4);  spilcd_sendbuffer(file, spi_module, buffer, 1, 0);
88    buffer[0] = 0x6e; spilcd_sendbuffer(file, spi_module, buffer, 1, 0);
89    buffer[0] = 0x70 | (c & 0x0f);  spilcd_sendbuffer(file, spi_module, buffer, 1,
          0);
90    buffer[0] = 0x38; spilcd_sendbuffer(file, spi_module, buffer, 1, 0);
91    buffer[0] = 0x0c; spilcd_sendbuffer(file, spi_module, buffer, 1, 0);
92    buffer[0] = 0x01; spilcd_sendbuffer(file, spi_module, buffer, 1, 0); msleep(10);
93    buffer[0] = 0x06; spilcd_sendbuffer(file, spi_module, buffer, 1, 0);
94
95    priv_data->x = 0;
96    priv_data->y = 0;
97  }
98
99  static void spilcd_clear(struct file *file, volatile struct mpc52xx_spi *spi)
100 {
101   char command = 0x01;
102   spilcd_sendbuffer(file, spi, &command, 1, 0);
103 }
104
105 static void spilcd_setposition(struct file *file, volatile struct mpc52xx_spi *spi
        , int x, int y)
106 {
107   struct spilcd_private *priv_data = (struct spilcd_private*)file->private_data;
108   char command = 0x80;
109
110   x = x % 16; // range correction
111   y = y % 3;  // range correction
112
113   command |= (y << 4) | x;
114   spilcd_sendbuffer(file, spi, &command, 1, 0);
115
116   priv_data->x = x;
117   priv_data->y = y;
118 }
119
120 static void spilcd_setcontrast(struct file *file, volatile struct mpc52xx_spi *spi
        , int contrast)
121 {
122   char buffer[8];
123
124   contrast = contrast & 0x3f;
125
126   buffer[0] = 0x39;
127   buffer[1] = 0x15;
128   buffer[2] = 0x54 | (contrast >> 4);
```

```
129   buffer[3] = 0x6e;
130   buffer[4] = 0x70 | (contrast & 0x0f);
131
132   spilcd_sendbuffer(file, spi, buffer, 5, 0);
133 }
134
135 static unsigned int spilcd_poll(struct file *file, poll_table *wait)
136 {
137   return 0;
138 }
139
140 static ssize_t spilcd_read(struct file *file, char *buf, size_t count, loff_t *off
      )
141 {
142   return 0;
143 }
144
145 static ssize_t spilcd_write(struct file *file, const char *buf, size_t count,
      loff_t *off)
146 {
147   int written = 0;
148   struct spilcd_private *priv_data = (struct spilcd_private*)file->private_data;
149
150   if(spi_module)
151   {
152     int ctr = 0;
153     for(ctr = 0; ctr < count; ctr++)
154     {
155       char c = buf[ctr];
156       switch(c)
157       {
158         case 10:
159         {
160           if(priv_data->y < 2)
161             priv_data->y++;
162
163           spilcd_setposition(file, spi_module, priv_data->x, priv_data->y);
164         } break;
165         case 13:
166         {
167           priv_data->x = 0;
168
169           spilcd_setposition(file, spi_module, priv_data->x, priv_data->y);
170         } break;
171         default:
172         {
173           if( (priv_data->x < 16) && (priv_data->y < 3) )
174           {
175             spilcd_sendbuffer(file, spi_module, &c, 1, 1);
176             priv_data->x++;
177           }
178         } break;
179       }
```

```
180         written++;
181
182       }
183       return(written);
184     }
185     else
186       return(-EIO);
187   }
188
189   static int spilcd_open(struct inode *inode, struct file *filp)
190   {
191     struct spilcd_private *priv;
192     void __iomem *sys_mbar = (void __iomem *) MPC52xx_MBAR_VIRT;
193
194     int p = iminor(inode);
195
196     if (p != SPILCD_MINOR)
197       return -EINVAL;
198
199     priv = kmalloc(sizeof(struct spilcd_private), GFP_KERNEL);
200
201     if (!priv)
202       return -ENOMEM;
203
204     // Initialize file structure
205     spi_module = (struct mpc52xx_spi *)(sys_mbar + MPC52xx_SPI_OFFSET);
206
207     filp->private_data = (void *)priv;
208
209     // SPI module initialization
210     if(spi_module)
211     {
212       spi_module->baud_rate = 0x74; // Clk = 512.6 kHz (IPB = 132 MHz)
213       spi_module->control_register_1 = MPC52xx_SPI_MSTR | MPC52xx_SPI_CPOL |
              MPC52xx_SPI_CPHA;
214       spi_module->control_register_2 = MPC52xx_SPI_SPC0;
215       spi_module->data_dir_register = 0x0f;
216
217       // Enable SPI
218       spi_module->control_register_1 |= MPC52xx_SPI_SPE;
219
220       spi_module->port_data_register = 0x08; // /CS = 1
221
222       spilcd_initdisplay(filp, spi_module, 0);
223     }
224
225     return 0;
226   }
227
228   static int spilcd_release(struct inode *inode, struct file *filp)
229   {
230     struct spilcd_private *todel = (struct spilcd_private *)filp->private_data;
231
```

```
232   if(spi_module)
233   {
234     spi_module->port_data_register = 0x08; // /CS = 1
235     spi_module->control_register_1 = 0;
236   }
237
238   if(todel)
239   {
240       kfree(todel);
241   }
242   return 0;
243 }
244
245 /* Main device API. ioctl's to read/set/clear bits, as well as to
246  * set alarms to wait for using a subsequent select().
247  */
248
249 static int spilcd_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
      unsigned long arg)
250 {
251   struct spilcd_private *priv_data = (struct spilcd_private*)file->private_data;
252
253   // Check SPI base addr
254   if(!spi_module)
255           return -EIO;
256
257   if(_IOC_TYPE(cmd) != SPILCD_IOCTYPE)
258       return -EINVAL;
259
260   switch(_IOC_NR(cmd))
261   {
262     case SPILCD_IOC_INIT:
263     {
264       // Display initialization
265       spilcd_initdisplay(file, spi_module, arg);
266     } break;
267     case SPILCD_IOC_CLEAR:
268     {
269       // Display clear
270       spilcd_clear(file, spi_module);
271
272     } break;
273     case SPILCD_IOC_SETPOSITION:
274     {
275       // Set cursor position
276       spilcd_setposition(file, spi_module, (arg >> 8) & 0xff, arg & 0xff);
277     } break;
278     case SPILCD_IOC_CONTRAST:
279     {
280       // Set LCD contrast
281       spilcd_setcontrast(file, spi_module, arg);
282     } break;
283   }
```

```
284   return 0;
285 }
286
287 struct file_operations spilcd_fops = {
288   .owner        = THIS_MODULE,
289   .poll         = spilcd_poll,
290   .ioctl        = spilcd_ioctl,
291   .write        = spilcd_write,
292   .open         = spilcd_open,
293   .release      = spilcd_release,
294 };
295
296
297 /* main driver initialization routine, called from mem.c */
298
299 static __init int spilcd_init(void)
300 {
301   int res;
302
303   /* do the formalities */
304
305   res = register_chrdev(SPILCD_MAJOR, spilcd_name, &spilcd_fops);
306   if (res < 0)
307   {
308     printk(KERN_ERR "EA-DOGM: couldn't register device.\n");
309     return res;
310   }
311
312   printk("EA-DOGM SPI driver v0.1, (c) 2008 Michal Hrouda\n");
313
314   return res;
315 }
316
317 static __exit void spilcd_exit(void)
318 {
319   unregister_chrdev(SPILCD_MAJOR, spilcd_name);
320 }
321 /* this makes sure that spilcd_init is called during kernel boot */
322
323 module_init(spilcd_init);
324 module_exit(spilcd_exit);
325
326 MODULE_LICENSE("Proprietary. Send bugs to ...");
```

B.2 lcddriver.h

```
1  #ifndef _spilcdH
2  #define _spilcdH
3
4  #define SPILCD_MAJOR 120     /* experimental MAJOR number */
5  #define SPILCD_MINOR 20
6
7  #define SPILCD_IOCTYPE 91
8
9  #define SPILCD_IOC_INIT    0
10 #define SPILCD_IOC_CLEAR   1
11 #define SPILCD_IOC_SETPOSITION   2
12 #define SPILCD_IOC_CONTRAST 3
13
14 struct mpc52xx_spi {
15   u8 control_register_1;
16   u8 control_register_2;
17   u8 reserved1[2];
18   u8 baud_rate;
19   u8 status_register;
20   u8 reserved2[3];
21   u8 data_register;
22   u8 reserved3[3];
23   u8 port_data_register;
24   u8 reserved4[2];
25   u8 data_dir_register;
26 };
27
28 #define MPC52xx_SPI_OFFSET   0x0F00
29
30 // SPI Control register 1
31 #define MPC52xx_SPI_SPIE   0x80
32 #define MPC52xx_SPI_SPE    0x40
33 #define MPC52xx_SPI_MSTR   0x10
34 #define MPC52xx_SPI_CPOL   0x08
35 #define MPC52xx_SPI_CPHA   0x04
36 #define MPC52xx_SPI_SSOE   0x02
37 #define MPC52xx_SPI_LSBFE  0x01
38
39 // SPI Control register 2
40 #define MPC52xx_SPI_SPISWAI 0x02
41 #define MPC52xx_SPI_SPC0   0x01
42
43 // SPI Status register
44 #define MPC52xx_SPI_SPIF   0x80
45 #define MPC52xx_SPI_WCOL   0x40
46 #define MPC52xx_SPI_MODF   0x10
47
48 #endif
```

Příloha C

PikeOS komunikace partitions

C.1 Zdrojový kód Posix partition

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <sys/mman.h>
6  #include <unistd.h>
7  #include <fcntl.h>
8  #include <errno.h>
9  #ifdef POSIXDEBUG
10 #include <sys/debug.h>
11 #endif
12 #include <sys/sport.h>
13 #include <string.h>
14
15 void do_fstat(int fd, const char *s)
16 {
17   _sport_ioc_t ioc;
18   int iResult = ioctl(fd, SPORT_PSTAT, &ioc);
19
20   printf("Port %s: Dir: %d, Msg size %ld, Validity %d\n", s, ioc.sport_op.pstat.
         direction, ioc.sport_op.pstat.msg_size, (int)ioc.sport_op.pstat.validity);
21 }
22
23 int main(void)
24 {
25   int stop = 0;
26
27   puts("POSIX application starting up.");
28 #ifdef POSIXDEBUG
29   gdb_breakpoint();
30 #endif
31
```

```
32  int  portReady, portAck;
33
34  portReady = open("/sport/SPPosixOne1Dest", O_RDONLY);
35  if(portReady == -1)
36  {
37    printf("Failed to open SPPosixOne1Dest, return value = %d\n", errno);
38  }
39  do_fstat(portReady, "SPPosixOne1Dest");
40
41  int iCtr = 0;
42  while (!stop) {
43    int iReaden;
44    char cBuffer[260];
45
46    _sport_ioc_t ioc;
47
48    int iResult = ioctl(portReady, SPORT_PSTAT, &ioc);
49    printf("Port 'portReady': Dir: %d, Msg size %ld, Validity %d\n", ioc.sport_op.
          pstat.direction, ioc.sport_op.pstat.msg_size, (int)ioc.sport_op.pstat.
          validity);
50
51    ioc.sport_op.read.buf = cBuffer;
52    ioc.sport_op.read.buf_size = 256;
53    ioc.sport_op.read.read_size = 0;
54
55    iResult = ioctl(portReady, SPORT_READ, &ioc);
56    if(iResult != -1)
57    {
58      printf("Read portReady: len %d,  Validity = ", ioc.sport_op.read.read_size);
59      switch(ioc.sport_op.read.validity)
60      {
61        case SPORT_EMPTY: printf("empty"); break;
62        case SPORT_AVAIL: printf("avail"); break;
63        case SPORT_VALID: printf("valid"); break;
64        case SPORT_INVALID: printf("invalid"); break;
65        case SPORT_NOT_APPLICABLE: printf("n/a"); break;
66        case SPORT_UNKNOWN: printf("unknown"); break;
67      }
68      printf("\n");
69
70      if( (ioc.sport_op.read.read_size > 0) /*&& (cBuffer[0] == 1)*/ )
71      {
72        printf("Received: %s.\n", cBuffer);
73      }
74    }
75    else
76    {
77      printf("Failed to read portReady errno = %d\n", errno);
78    }
79    sleep(1);
80  }
81
82  if(portReady != -1)
```

```
83      close(portReady);
84
85    return 0;
86  }
```

C.2 Zdrojový kód aplikace pro ElinOS

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/types.h>
4   #include <sys/stat.h>
5   #include <sys/mman.h>
6   #include <unistd.h>
7   #include <fcntl.h>
8   #include <errno.h>
9   #include </opt/elinos-4.1/linux/linux-p4-2.6.15/drivers/p4/vmsport_ioctl.h>
10
11  void do_fstat(int fd, const char *s)
12  {
13          struct stat sb;
14          fstat(fd, &sb);
15
16          printf("Port %s: Msg size %ld, # of msg %ld, available space/msgs %ld\n",
                s, sb.st_blksize, sb.st_blocks, sb.st_size);
17  }
18
19
20  int main(void)
21  {
22          int stop = 0;
23
24          puts("POSIX application starting up.");
25
26          int  portAck;
27
28          portAck = open("/dev/vmsport/SPLinuxSource", O_WRONLY);
29          if(portAck == -1)
30          {
31                  printf("Failed to open SPPosixOne1Source, return value = %d\n",
                        errno);
32          }
33          do_fstat(portAck, "SPLinuxSource");
34
35          int iCtr = 0;
36          while (!stop) {
37          stop=iCtr == 10;
38                  int iReaden;
39                  char cBuffer[260];
```

```
40
41                  sprintf(cBuffer, "Testik %d", iCtr++);
42
43                  _sport_ioc_t ioc;
44                  ioc.sport_op.write.buf = cBuffer;
45                  ioc.sport_op.write.buf_size = 50;
46
47                  int iResult = ioctl(portAck, SPORT_WRITE, &ioc);
48                  if(iResult == -1)
49                  {
50                          printf("Failed to write portAck errno = %d\n", errno);
51                  }
52
53                  sleep(1);
54          }
55
56
57
58          return 0;
59  }
```

Příloha D

Obsah přiloženého CD

K této práci je přiloženo CD, na kterém jsou uloženy zdrojové kódy.

- *bp_2008_michal_hrouda.pdf* - Tato práce ve formátu PDF
- apps
  Aplikace nutné zkompilovat pro embedded modul
  - *.config* - Konfigurační soubor pro Busyboxu
  - *busybox-1.9.0.tar.gz* - Zdrojové kódy Busyboxu
- devel
  Aplikace nutné pro vývoj
  - *crosstool-powerpc-devel-0.35-9.i386.rpm* - Kompilátor a linker pro PowerPC architekturu
  - *u-boot-ppc_6xx-1.2.0-1.ppc.rpm* - Zdrojové kódy bootloaderu uBoot
- kernel
  Jádro systému linux
  - *driver_i2c* - Ovladač pro RTC DS1307
  - *driver_spi* - Ovladač pro LCD EA-DOGM
  - *linux_bestcomm_fec.patch* - Patch pro přidání podpory ethernetového řadiče
  - *.config* - Konfigurační soubor pro jádro
  - *linux-2.6.23.13.tar.gz* - Zdrojové kódy jádra
- target
  Soubory pro zkopírovaní na embedded modul
  - *crosstool-targetcomponents-ppc_6xx-0.35-9.ppc.rpm* - Potřebné knihovny

- *linux_etc.tar.gz* - Obsah adresáře /etc