OS Linux na platformě ARM

A Linux implementation on an ARM platform

Bc. Roman Došek

Diplomová práce
2013

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2012/2013

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Roman Došek**
Osobní číslo:	**A11397**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**prezenční**
Téma práce:	**OS Linux na platformě ARM**

Zásady pro vypracování:

1. **Prostudujte platformy PXA320 (ARMv5) a Tegra (ARM Cortex A9) z hlediska možností implementace OS Linux.**
2. **Analyzujte, zda pro zavedení systému lépe vyhovuje projekt U-Boot nebo Barebox.**
3. **Implementujte OS Linux na platformě s PXA320. Kromě zavedení systému se soustřeďte také na ovladače perfierií ethernet, CAN, seriové linky atd.**
4. **Integrujte do systému aplikační knihovny, nainstalujte a otestujte vzorovou aplikaci.**
5. **Zvažte možnost využití mechanismu snapshotů pro zrychlení zavádění operačního systému a aplikace.**

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **VENKATESWARAN, Sreekrishnan. Essential Linux device drivers. Upper Saddle River: Prentice Hall, 2008, xxx, 714 s. ISBN 978-0-132-39655-4.**
2. **CORBET, Jonathan. Linux device drivers. 3rd ed. Sebastopol: O'Reilly, 2005, xviii, 615 s. ISBN 05-960-0590-3.**
3. **LOVE, Robert. Linux kernel development. 3rd ed. Upper Saddle River: Addison-Wesley, 2010, xx, 332 s. ISBN 06-723-2946-8.**
4. **GRÖTKER, Thorsten. The developer's guide to debugging. 2nd ed. North Charleston, S.C.: [CreateSpace Independent Publishing Platform], 2012, xx, 242 s. ISBN 978-1470185527.**
5. **STALLMAN, Richard M a Roland MCGRATH. GNU make: a program for directing recompilation : GNU make version 3.79.1 : June, 2002. Boston: Free Software Foundation, 2002, vi, 184 s. ISBN 18-821-1482-5.**
6. **Building embedded Linux systems. 2nd ed. Sebastopol: O'Reilly, 2008, xx, 439 s. ISBN 978-0-596-52968-0.**

Vedoucí diplomové práce: **Ing. Tomáš Dulík, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **22. února 2013**

Termín odevzdání diplomové práce: **22. května 2013**

Ve Zlíně dne 22. února 2013

L.S.

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

ABSTRAKT

Tato práce se zabývá implementací operačního systému Linux na platformě ARM. V práci je rozebírána historie, současný stav ARM platformy i její budoucnost. V praktické části jsou popsány některé linuxové nástroje, které jsou využitelné při práci s Linuxem na ARMu. Práce rovněž analyzuje i současný stav zavaděčů Linuxu na ARMech a některé problémy, které se s těmito zavaděči vyskytují. Důraz je kladen na použití Linuxu na embedded zařízeních.

Klíčová slova: Linux, ARM, embdedded, Qt

ABSTRACT

This thesis is about implementation of a Linux operating system on an ARM platform. It contains analysis of Linux ARM port history, current state as well as close future. Practical part of the thesis contains description of tools usable during development of a complete Linux system on an ARM platform. Since Linux has to be run by a bootloader, thesis contains description of available ARM bootloaders, their state and some of the problems found during work with them. Strong focus on Linux for embedded devices is expressed.

Keywords: Linux, ARM, embedded, Qt

Mé hlavní poděkování patří mojí rodině za podporu a motivaci během studia, bez ní by tato práce nikdy nevznikla.

Ing. Tomáši Dulíkovi, PhD. bych chtěl poděkovat za vytvoření podmínek k práci, motivaci, ochotu a cenné rady.

A díky patří i firmě UNIS a jejím zaměstnancům za poskytnutí možnosti pracovat na tak zajímavé práci a ochotu pomoci při řešení problémů.

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně ……

podpis diplomanta

OBSAH

ÚVOD

Cílem této práce je zmapovat použití na Linuxu na zařízeních na platformě ARM. Důvodem pro výběr právě této platformy je její současný raketový rozvoj, který je způsoben větší orientací na spotřební elektroniku (chytré telefony, tablety) a útlum na trhu se standardními PC. Mimo to se začínají objevovat první servery na platformě ARM, kde má Linux tradičně vysoké zastoupení.

Tam, kde se dříve používaly jednoduché mikropočítače, programované kompletně v jazyku C a bez systému, nebo s jednoduchým RTOS systémem, dnes tlak na vývoj nových funkcí vynucuje použití System-on-chip (SoC) řešení s kompletním operačním systémem. V práci je proto diskutováno to, jaké jsou důvody k použití Linuxu na embedded platformách a jaké k němu existují alternativy. Jako každé řešení, i Linux má přes svou flexibilitu některé slabší stránky.

Dále práce obsahuje krátký ohled do historie Linuxu na ARM procesorech a důvody, které vedly k současnému stavu. Zároveň obsahuje i krátký pohled na to, co se pro Linux na ARM procesorech v budoucnosti změní.

V praktické části je možné nalézt požadavky, které jsou kladeny na podporu pro zařízení, pro běh Linuxu a co dělat, pokud podpora pro dané zařízení není k dispozici. Následuje praktický návod, jak vytvořit kompletní linuxový systém pro procesorovou desku, včetně zavaděče.

I. TEORETICKÁ ČÁST

1 PROGRAMOVÉ PROSTŘEDÍ NA EMBEDDED SYSTÉMECH

Procesory s architekturou ARM se v dnešní době využívají v obrovském množství systémů a na trhu je k dispozici velký počet různě výkonných procesorů od různých výrobců. Zpravidla se pro vyvíjený systém používá co nejlevnější procesor, což také znamená, že jeho výkon nebude příliš přesahovat nad požadované specifikace.

Pro různé druhy systémů je vhodný různý přístup – například tam, kde bude procesor pouze číst data z A/D převodníku a po sběrnici je bude posílat dál, nemá význam zavádět jakýkoliv operační systém.

1.1 Bez systému

Mikroprocesor neobsahuje žádný druh operačního systému. Program je složený pouze z ovladačů hardwaru a vlastního programu. Tento přístup má nejmenší nároky na výkon procesoru a je možné ho při pečlivém zvážení použít i v časově-kritických aplikacích.

1.2 RTOS

V případě komplexnějšího programu, ve kterém běží více úloh zároveň, a je nutné úlohy v pravidelných intervalech střídat převažuje použití nějakého realtime operačního systému. Přestože je v názvu operační systém, jednoduché RTOS příliš nesplňují definici běžného operačního systému, neboť neposkytují žádnou abstrakci od hardwaru. Uživatelský program stále přímo manipuluje s hardwarem prostřednictvím nízkoúrovňových ovladačů nebo i přímou manipulací s registry procesoru.

Existuje celá řada těchto operačních systémů a liší se především tím, jakým způsobem dochází k přepínání úloh, licencí a podporovanými platformami.

Příkladem těchto RTOS jsou FreeRTOS, CooCox CoOS, MicroC/OS-II, ThreadX nebo třeba Nucleus RTOS.

1.3 Operační systém

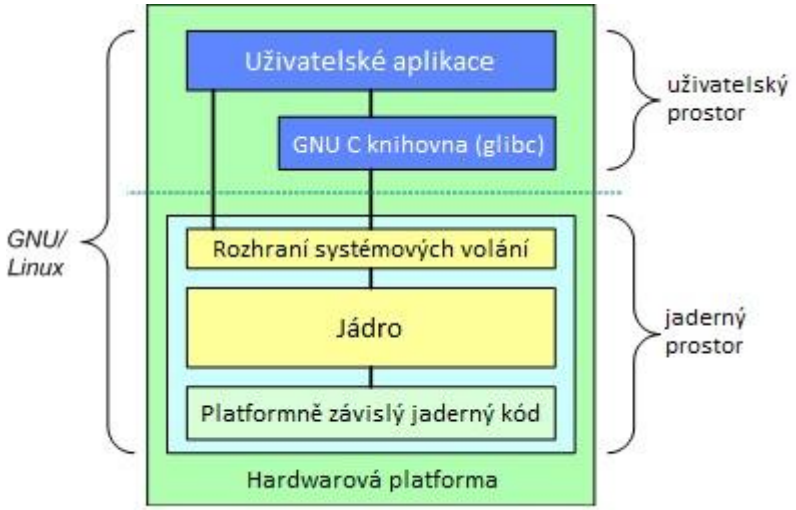
Úlohou operačního systému na vestavěných platformách je provádět abstrakci od hardwaru a umožnit tak napsat více portovatelný software a výrazně zjednodušit vývoj komplexních programů tím, že se velká část práce přesune právě na operační systém. Nejrozšířenější

operační systémy pro vestavěné systémy navíc umožňují i realtimový běh. Daní za použití celého operačního systému je ztráta části výkonu.

1.3.1 Linux

Linux je opensource operační systém s monolitickým jádrem. Novější jádra tohoto systému je možné nastavit tak, aby se chovala realtimově.

Více o Linuxu je v kapitole 2 Linux.

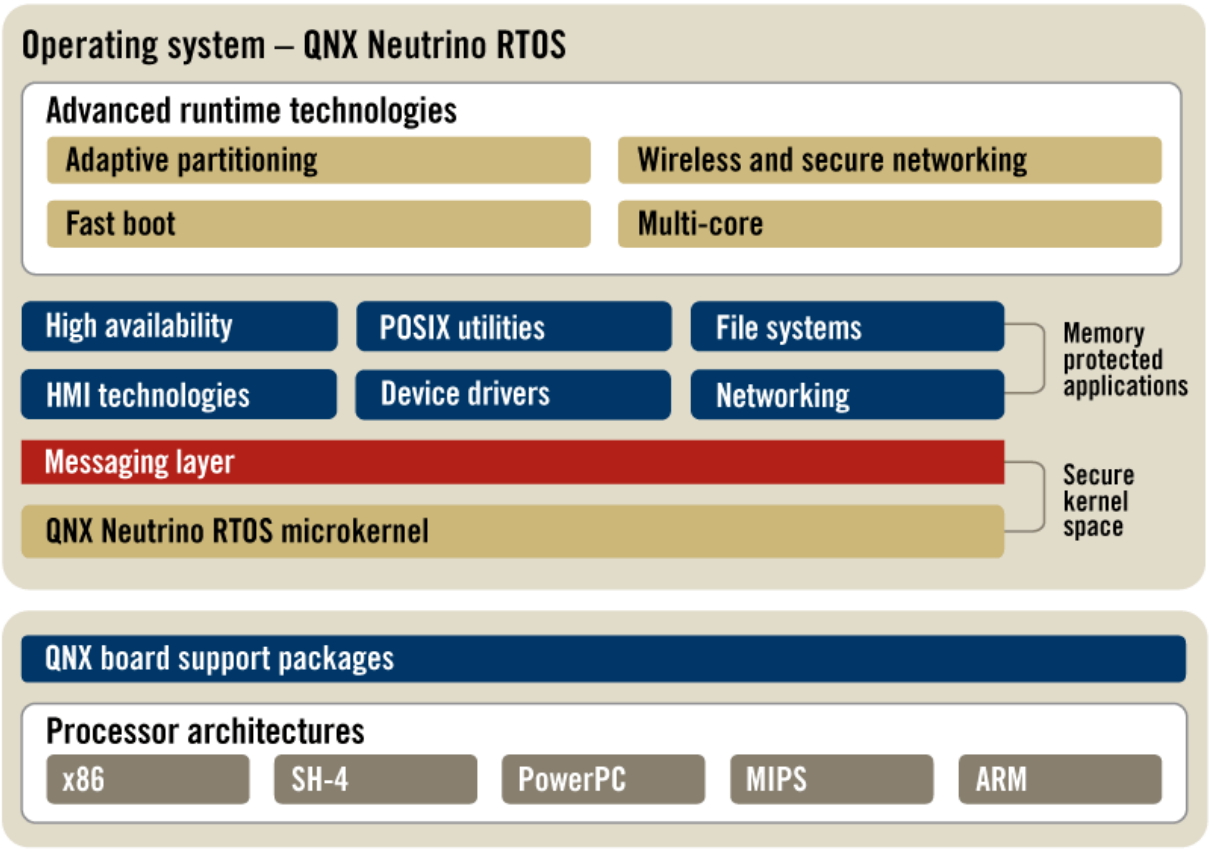


Obrázek 1 – struktura linuxového systému

1.3.2 QNX Neutrino

QNX Neutrino je komerční realtime operační systém, který je určený k nasazení na vestavěných systémech. Výrobce poskytuje několik variant tohoto systému, které splňují nejrůznější certifikace. Díky tomu je tento systém používán v průmyslu, medicíně, obraně a jiných kritických aplikacích.

Na rozdíl od Linuxu, který má monolitické jádro je QNX Neutrino založený na microkernel architekuře. Díky tomu je možné jednotlivé komponenty zapínat a vypínat dle potřeby a v případě selhání kterékoliv komponenty samotný systém chybu přežije a komponenta je automaticky restartována. Na druhou stranu zase tato architektura v sobě nese vyšší nároky, protože komunikace mezi mikrojádrem a ovladači je komplikovanější. [7]



Obrázek 2 – Architektura operačního systému QNX Neutrino [7]

QNX Neutrino je kompatibilní se standardem POSIX, což umožňuje snadné portování aplikací napsaných pro jiný Unix-like systém.

1.3.3 VxWorks

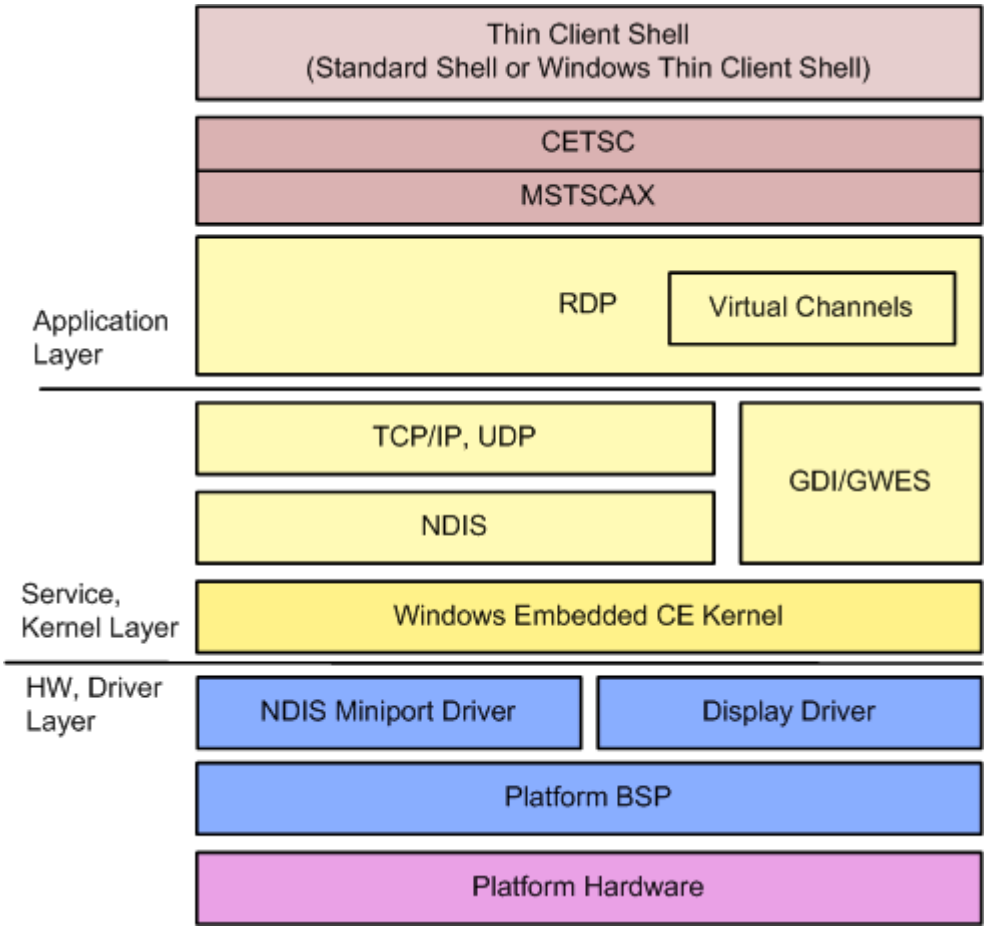
VxWorks je systém s velmi podobnými vlastnosti, jako má QNX Neutrino. Stejně jako on je to komerční microkernel operační systém, který byl vyvíjen od roku 1987 firmou Wind River Systems. Firmu Wind River Systems koupila v roce 2009 firma Intel.

VxWorks je velmi používaný v kosmické technice (satelity, sondy apod.), jako příklad může být jeho nasazení ve známém vozítku NASA Mars Science Laboratory Curiosity.

1.3.4 Windows Embedded Compact

Windows Embedded Compact, dříve známý jako Windows Embedded CE (zkráceně WinCE), je operační systém vyvíjený firmou Microsoft a určený pro nasazení v embedded systémech.

Jeho jádro využívá takzvanou hybridní architekturu, což je kompromis mezi microkernelem a monolitickým jádrem. Díky tomu jsou některé části jádra vzájemně izolovány.



Obrázek 3 – architektura operačního systému Windows Embedded CE

1.3.5 Srovnání

Následující tabulka zachycuje porovnání systémů v některých klíčových oblastech.

Operační systém	Hard realtime	Soft realtime	Architektura	Licence
Linux	Ne	Ano	Monolitická	GNU GPLv2
QNX Neutrino	Ano	Ano	Microkernel	Proprietární
VxWorks	Ano	Ano	Microkernel	Proprietární
Windows Embedded Compact	Ne	Ano	Hybridní	Proprietární

Tabulka 1 – Srovnání operačních systémů

2 LINUX

Linux je jeden z nejvíce rozšířených operačních systémů. Původně byl vyvinut pouze pro platformu 386, ale postupně byl spolu s rostoucí popularitou portován na řadu dalších platforem. V nejaktuálnější stabilní verzi (3.8) podporuje 26 platforem.

S postupným vývojem hardwaru a odpovídající dostupností dostatečně výkonných procesorů se neustále zvyšuje jeho podíl ve vestavěných systémech. Hlavními důvody pro jeho úspěch jsou otevřený model vývoje a nulová cena.

Jako každé reálné technické řešení má i Linux některé problémy, které mohou odrazovat o jeho použití. Některé z nich jsou dány historickým vývojem a jsou v budoucnosti řešeny, jiné problémy přetrvávají. Jedním z největších problémů Linuxu a otevřeného software obecně je nedůvěra k opensource modelu a jeho filozofii, podle které by měly mít užitek všichni přispívající. To vedlo v minulosti řadu firem k nezveřejnování vlastního kódu a nezačleňování ho do hlavní řady jádra. V poslední době se naštěstí smýšlení mnoha firem změnilo a jejich přispívání do jádra pro ně znamená i ušetření nákladů na vývoj.

Linux navíc není původně realtimový operační systém a přestože lze jeho chování vyladit tak, aby se choval realtimově, může to být v některých aplikacích překážkou. Protože má navíc linuxové jádro monolitickou architekturu, sdíly jaderné moduly jeden stejný paměťový prostor a pád modulu tak často znamená pád celého systému, což není v kritických aplikacích tolerovatelné.

Široká podpora různého hardwaru a platforem zase způsobuje agresivní narůstání množství kódu v jádře, což značně zhoršuje jeho budoucí udržovatelnost a vzniká tím i větší prostor pro chyby. Přes vysoké snahy maintainerů jádra není vždy kód psán stejným stylem, což zhoršuje orientaci.

Politikou Linuxu je také nerozbíjet userspace software, díky čemuž musí být v některých případech problémy, nebo špatná návrhová rozhodnutí řešena ne úplně ideálními cestami.

2.1 Linux na platformě ARM

První pokusy zprovoznit linuxové jádro na platformě ARM prováděl Russell King, dnes maintainer ARM větve Linuxu, již v roce 1995. V té době ještě nebyl Linux vůbec portabilní – podporována byla pouze platforma x86. [8] V dnešní době je platforma ARM nejrychleji rostoucí ze všech podporovaných architektur. Pro srovnání – v jednom období bylo mezi dvěma po sobě jdoucími verzemi jádra ve větvi ARM 70 tisíc řádků změn a proti tomu 5 tisíc řádků změn ve větvi x86. [9]

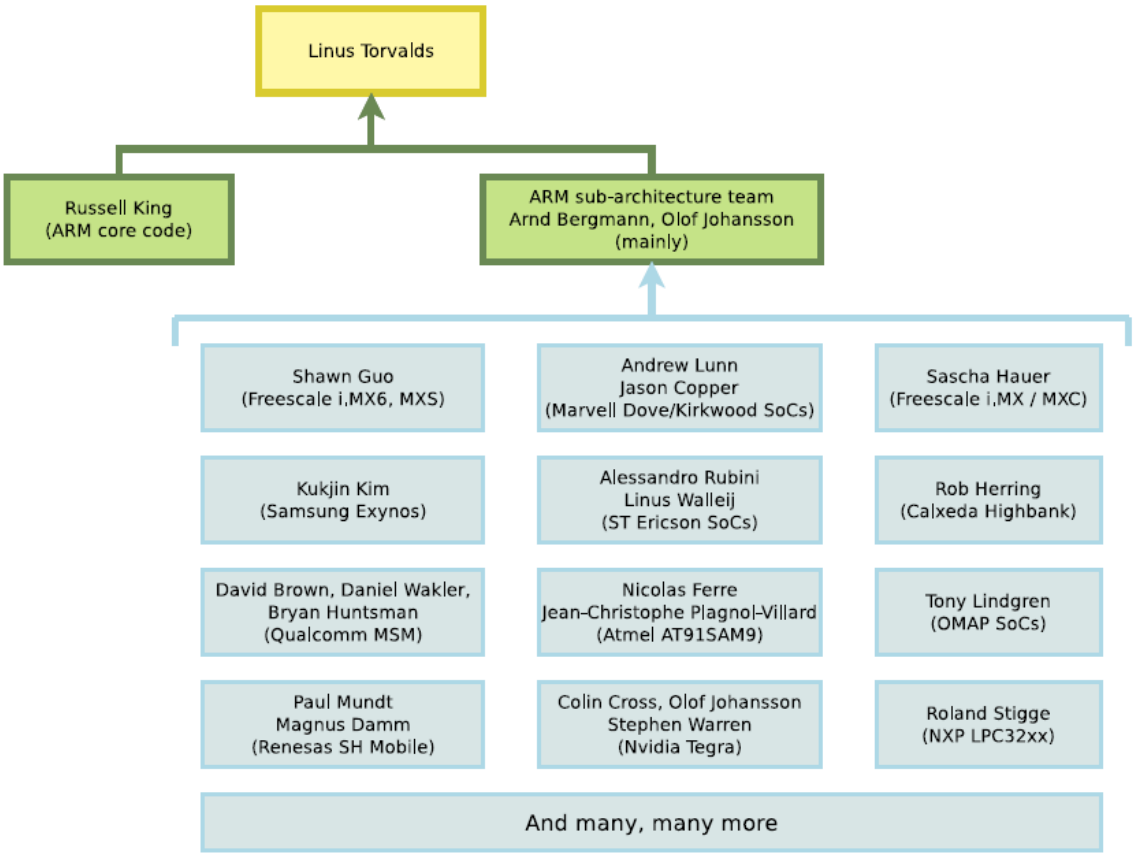
2.1.1 Historické problémy

V dřívější době byl ARM používaný pouze na embedded zařízeních a až v poslední době se začaly objevovat PC/Laptopy založené na ARM procesorech a dokonce i servery. Až do verze Linuxu 3.7 neexistovala generická podpora platformy ARM, která by umožňovala nabootovat jedno jádro na různých zařízeních, a pro každé zařízení muselo být zkompilováno vlastní jádro.

Vzhledem k velkému počtu výrobců na ARM trhu často docházelo k tomu, že výrobci často do linuxového jádra přispívali kódem, který duplikoval činnost již existujícího kódu od jiného výrobce nebo byl dokonce konfliktní. Tato situace časem kulminovala do toho stádia, že hlavní maintainer ARM větve Russell King přestal stíhat kontrolovat posílané patche a někteří výrobci kvůli tomu začali posílat patche přímo Linusovi Torvaldsovi, a přitom obcházeli stromovou strukturu, která je v rámci vývoje linuxového jádra stanovená. To nakonec také vedlo k odmítání nových patchů pro podporu nových desek a bouřlivé diskuzi, která měla za cíl problémy v ARM větvi vyřešit.

2.1.2 Vylepšená podpora

Velcí výrobci ARM procesorů (Freescale, IBM, Samsung, ST-Ericsson, Texas Instruments) a společnost ARM mezitím založili novou společnost – Linaro a začali spolupracovat na vylepšování podpory ARM platformy v Linuxu. Kromě starého ARM stromu udržovaného Russellem Kingem také vznikl další strom arm-soc, který v současné době spravují Arnd Bergmann a Olof Johansson. Cílem tohoto stromu je začleňovat podporu pro nová ARM zařízení bez zbytečných duplicit v kódu.



Obrázek 4 – stávající udržovací struktura pro ARM [10]

Na platformě x86 existuje automatická detekce hardwaru – ta je možná, protože většina x86 hardwaru je napojená na PCI/PCI-e sběrnici. Na procesorových deskách na platformě ARM ale není žádná univerzální sběrnice, na které by byly připojené zařízení, proto se nakonec přešlo na podporu desek pomocí stromové struktury Flattened Device Tree (FDT), která zachycuje, jak je který hardware připojený na jaké sběrnici, jaké používá adresní rozsahy a podobně. Díky tomu již nedochází ke zbytečné duplikaci kódu a pro různé desky stačí přidat pouze jeden Device Tree Source (dts) soubor.

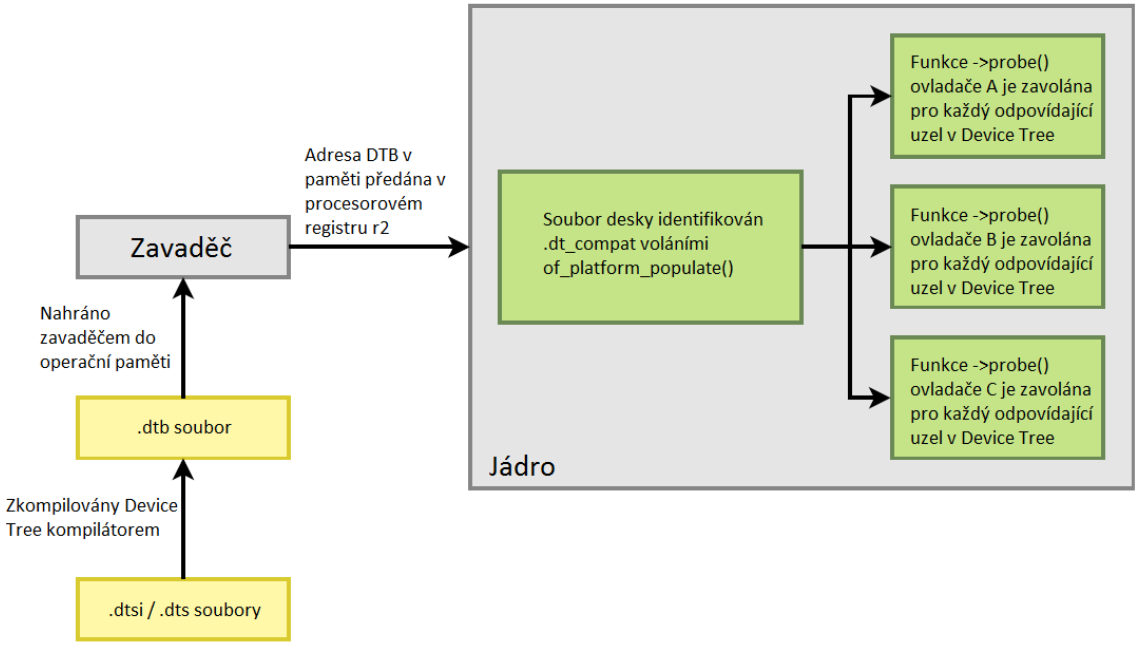
Starší, ale stále velmi používané subarchitektury jsou postupně pročišťovány a konvertovány na podporu FDT a nové subarchitektury už musí FDT podporovat.

Generická podpora pro ARM platformu, začleněná v jádře 3.7, je pokračováním předchozích snah o pročištění. Díky ní je možné v jádře povolit podporu několika různých desek a vytvořit tak jádro, které je schopné nabootovat na různém hardwaru. Dříve bylo naproti tomu nutné mít pro každý odlišný hardware vlastní jádro.

2.1.2.1 Device Tree

V minulosti bylo nutné pro každou novou desku napsat specifický kód, který byl pro všechny desky velmi podobný a málo přehledný. Device Tree je způsob, jak přesunout specifické detaily pro jednotlivé desky do člověkem čitelných souborů a ponechat ve zdrojových kódech pouze obecnou inicializaci, která při bootu využije předaný device tree soubor obsahující detaily o platformě.

Device Tree obsahuje stromovou strukturu komponent, díky které je možné určit, jak je která komponenta zapojená na desce, jaké využívá adresní rozsahy, s jakým hardwarem je kompatibilní a podobně. Zdrojový device tree soubor (Device Tree Source) se před použitím musí zkompilovat pomocí device tree kompilátoru (Device Tree Compiler) a výsledkem je binární soubor device tree (Device Tree Blob). Tento soubor pak při bootu musí zavaděč předat jádru. Existují také Device Tree Source Include, které slouží k uchovávání společných částí mezi různými deskami založenými například na stejném procesoru. [10]



Obrázek 5 – inicializace s pomocí Device Tree [10]

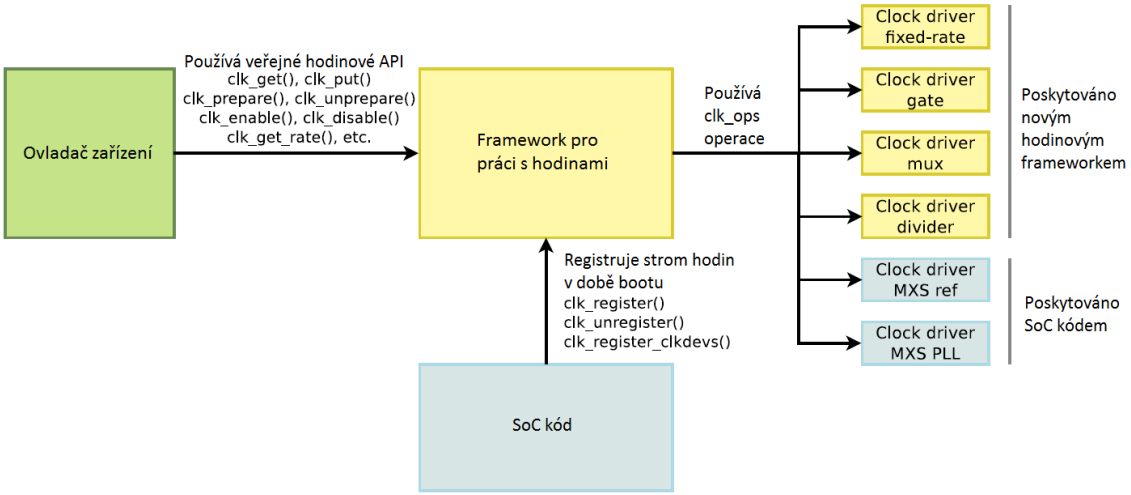
2.1.2.2 Nový framework pro práci s hodinami

Různé hardwarové části integrovaného systému jsou řízené různými hodinami, které pracují na různých frekvencích. Většina těchto hodin je součástí komplikovaného stromu,

ve kterém jsou rodičovské hodiny vstupem do potomků. Velká část těchto hodin je softwarové konfigurovatelná (zapnutí/vypnutí, změna frekvence) a musí být možné s nimi za běhu manipulovat pro podporu různých šetřících režimů.

Ve starších jádrech byl seznam hodin definován a kontrolován kódem pro podporu dané platformy. Existovala sice struktura pro ovládání hodin a společné API, ale obojí bylo pro každou subarchitekturu definováno a implementováno jinak, i když mezi různými subarchitekturami byly pouze malé rozdíly.

Do linuxového jádra byl s verzí 3.4 poprvé přidán nový framework pro manipulaci s hodinami, který implementuje společné API a definuje datové struktury, pomocí kterých si může podpůrný kód definovat svoje hodiny. [10]



Obrázek 6 – použití nového frameworku pro práci s hodinami [10]

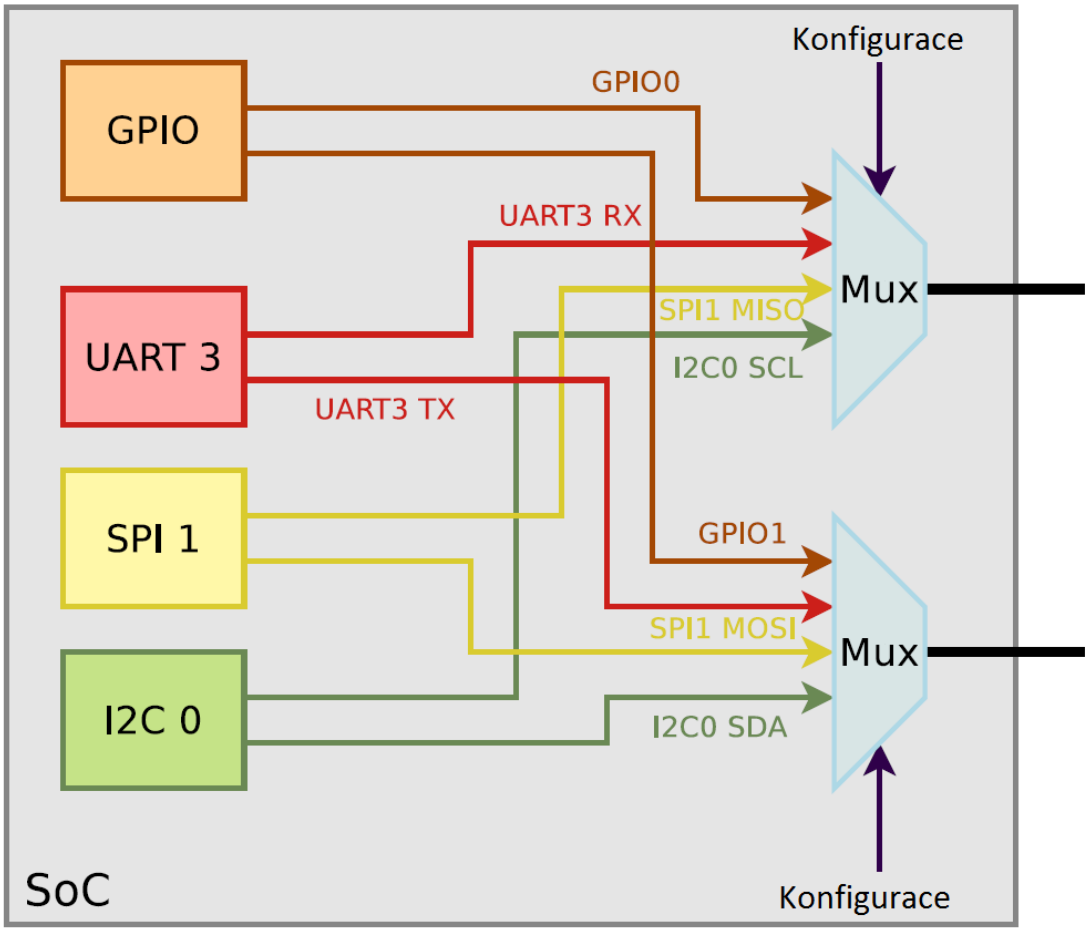
2.1.2.3 Pin muxing

Dnešní integrované čipy obsahují velké množství periferií, jejichž množství může přesahovat počet dostupných pinů. Z tohoto důvodu jsou tyto piny multiplexovány – mohou být použity jako funkce A, B, C, nebo GPIO.

Příkladem těchto funkcí jsou:

- SDA/SCL vodiče pro I2C sběrnice
- MISO/MOSI/CLK vodiče pro SPI
- RX/TX/CTS/DTS vodiče pro UARTy

Na obrázku Obrázek 7 je znázorněn princip multiplexování pinů procesoru.



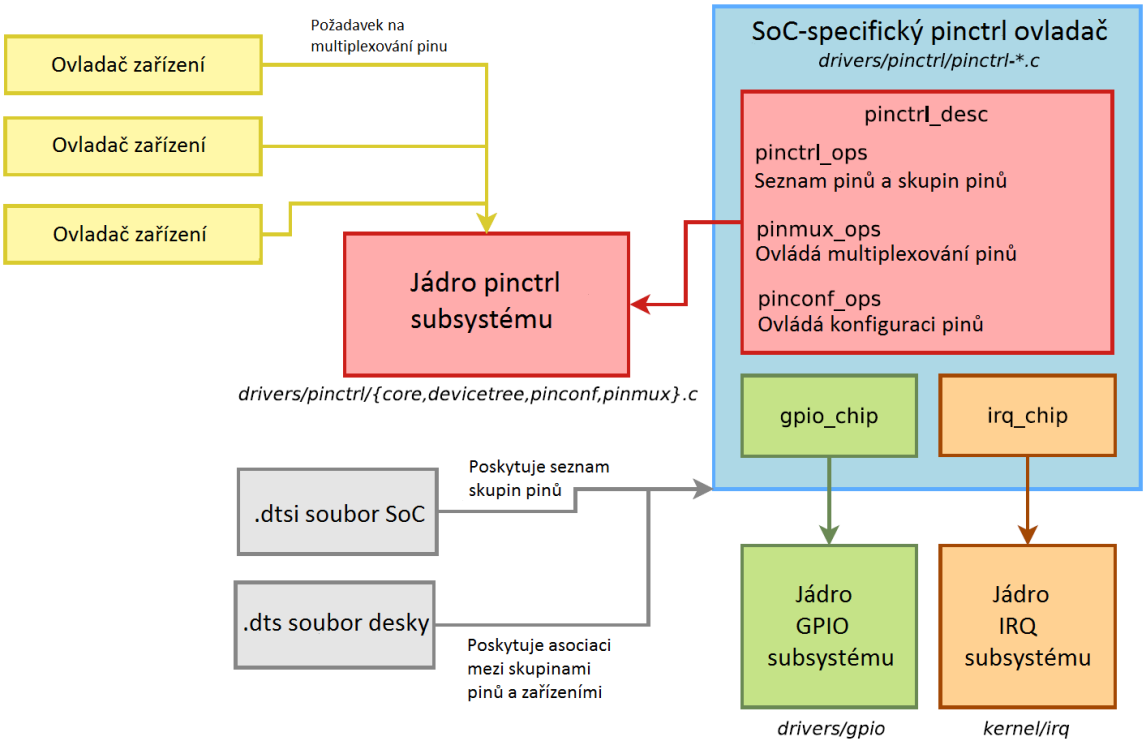
Obrázek 7 – princip multiplexování pinů procesoru [10]

V jakém módu bude každý pin použit je konfigurováno softwarově a závisí to na tom, jak bude tento čip zapojen na nosné desce. V linuxovém jádře měla každá ARM subarchitektura vlastní kód starající se o multiplexování pinů. API tohoto kódu proto bylo pro každou subarchitekturu specifické a multiplexování pinů muselo být nastavováno v kódu pro podporu procesoru a nemohlo být měněno z ovladačů.

Problémy s tímto přístupem se snaží řešit nový pinctrl subsystém, jehož hlavním autorem je Linus Walleij, z firmy Linaro/ST-Ericsson. Implementaci tohoto subsystému je možné nalézt v *drivers/pinctrl* a poskytuje následující:

- API pro registraci pinctrl ovladače pro entity které mají k dispozici seznam pinů, jejich funkcí a vědí jak je konfigurovat. Používáno ovladači specifickými pro SoC ke zveřejnění možností multiplexování pinů.
- API pro ovladače zařízení dovolující požádat o multiplexování určité sady pinů
- Interakci s GPIO frameworkem

Následující obrázek znázorňuje funkci pinctrl subsystému.



Obrázek 8 – funkce pinctrl subsystému [10]

3 PODMÍNKY PRO BĚH LINUXU NA ZAŘÍZENÍ

3.1 Křížový překladač

Křížový překladač (v angličtině Cross-Compiler) je takový překladač, který generuje spustitelný kód pro jinou platformu, než na které je překladač spuštěn. Mezi křížové překladače se řadí všechny překladače, které jsou distribuovány ve vývojových prostředích od jednotlivých výrobců mikropočítačů – například Code Composer Studio, IAR Workbench, Tasking a jiné.

Pro zkompilování zavaděče, linuxového jádra a zbytku systému však nejsou tyto překladače vhodné, neboť zkompilování těchto částí zpravidla s těmito překladači nikdo netestoval a navíc část zdrojových kódů obsahuje použité neoficiální rozšíření jazyka, která jsou součástí kompilátoru obsaženým v GNU Compiler Collection (GCC). Z tohoto důvodu je v současné době nejvhodnější použít křížový překladač založený na GCC.

Buď je možné použít některý z předpřipravených křížových překladačů, například CodeSourcery ARM GNU/Linux tool chain nebo si vytvořit vlastní křížový překladač, například pomocí nástroje Crosstool-ng. Obecné předpřipravené překladače mají tu výhodu, že jsou otestované a rychle a snadno dostupné. Na druhou stranu ale kvůli podpoře co nejvíce modelů procesorů nevyužívají všechny možnosti procesoru, mohou obsahovat podporu pro nepotřebné věci nebo jim naopak může chybět podpora pro něco potřebného.

Proto jsou předpřipravené překladače vhodné spíš jako rychlá testovací možnost, ale cílový systém by měl být zkompilován překladačem na míru.

3.2 Zavaděč

Zavaděč (v angličtině bootloader) je malý program, jehož úkolem je inicializovat minimum zařízení připojených k procesoru a následně zavést další systém (Linux, WinCE, atd.).

U stolních/přenosných počítačů založených na platformě x86 je na základní desce v EEPROM či jiné paměti uložený BIOS případně EFI/UEFI. Ten detekuje bootovací zařízení a z něj spouští zavaděč dalšího systému, případně může rovnou zavést daný systém. Na vestavěných zařízeních naproti tomu úlohu BIOSu přebírá zavaděč. Konkrétní bootovací sekvence navíc závisí na typu použité paměti.

V případě, že je zavaděč uložen v paměti NOR, může procesor spustit zavaděč přímo odtamtud. V případě NAND pamětí je situace komplikovanější, protože kód uložený v NAND paměti nelze přímo spustit a musí být nejdříve nahrán do RAM paměti. Procesor ale nemá dopředu informaci o tom, jak velký zavaděč je, proto NAND paměť obsahuje v několika prvních blocích ještě další předzavaděč, který zavádí buď plný zavaděč, nebo rovnou následující systém. Tento předzavaděč se označuje různými termíny, například Initial Program Loader (IPL) nebo Secondary Program Loader (SPL). Předzavaděč už má v sobě uloženou informaci, jak velký je plný zavaděč nebo jádro systému a proto ho může zkopírovat do RAM a opět na něj přeskočit.

Celý postup lze tedy v případě NAND pamětí shrnout takto:

- Procesor obsahuje v permanentní paměti krátký kód, který z NAND paměti překopíruje prvních několik bloků NAND paměti, ve kterých je uložený předzavaděč, do RAM paměti a provede JMP na nahraný kód.
- Předzavaděč zkopíruje další zavaděč nebo jádro systému do RAM a provede JMP.
- V případě, že byl v předchozím kroku zkopírován zavaděč, se v něm provede inicializace nutných periferií a poté se zavede systém. Další systém může být přitom zaváděn z SD karty, USB zařízení nebo třeba sítě.

Ne pro všechny platformy existuje takový předzavaděč, který je schopný přímo zavést jádro systému, proto je tento mezikrok často nutný, i když je systém uložený v NAND paměti.

Pro použití na vestavěných systémech existují pouze 3 univerzální zavaděče – Das U-boot, Barebox a RedBoot. Ani pro jeden testovaný modul nebyla v RedBootu podpora, proto jsou dále popisovány pouze zavaděče Das U-boot a Barebox.

3.2.1 Das U-boot

Das U-boot (Universal Bootloader) je opensource zavaděč pro použití ve vestavěných systémech. Je dostupný pro velké množství různých architektur. Při spuštění a přerušení bootování U-boot přejde do CLI režimu, ve kterém je možné provádět různé jednoduché činnost jako je aktualizace zavaděče/jádra/systému, úprava bootovací sekvence a podobně. Toto prostředí obsahuje proměnné prostředí, které se navíc mohou chovat jako jednoduché skripty – například je možné vytvořit skript složený ze 4 příkazů, který automaticky stáhne

novou verzi systému z TFTP serveru a přepíše původní systém v NAND paměti. Kromě vlastních proměnných, které je možné jednoduše nadefinovat buď za běhu nebo již při přípravě zavaděče, obsahuje U-Boot také několik proměnných, které mají speciální význam a které je nutné mít nastaveny pro správnou funkci, případně nabootovaní dalšího systému. V příloze Příloha I: Příkazy U-bootu lze nalézt seznam příkazů podporovaných v CLI U-bootu včetně popisu.

3.2.2 Barebox

Barebox je zavaděč původně odvozený od zavaděče U-Boot, který se ale snaží o stejný styl kódu a kvalitu, jako má linuxové jádro. Přestože je Barebox odvozený z U-Bootu, nemá podporu pro tak velké množství procesorů/desek/zařízení. Příloha P II:Příkazy Bareboxu obsahuje seznam příkazů použitelných v CLI Bareboxu.

Následující seznam obsahuje některé klíčové vlastnosti, které odlišují Barebox od ostatních dostupných zavaděčů.

- **POSIXové souborové API**

Barebox využívá široce uznávané funkce open/close/read/write/lseek spolu s modelem reprezentování zařízeí pomocí souborů. Díky tomu je API důvěrně známe komukoliv se zkušenostmi s programováním pod Unixovými systémy.

- **Konzole**

Konzole v Bareboxu obsahuje standardní příkazy jako je ls/cd/mkdir/echo/cat,…

- **Souborový systém prostředí**

Na rozdíl od U-Bootu Barebox nezneužívá proměnné prostředí k vytváření skriptů. Po spuštění zavaděče se uživateli nasktne pohled na konzoli a něco, co vypadá jako souborový systém. Ve skutečnosti je to ale jednoduchý ar archív, který je příkazem loadenv načtený z flash paměti do ramdisku a příkazem saveenv uložený zpět.

- **Podpora souborových systémů**

Při spuštění je souborový systém prostředí (env) připojen do / a souborový systém zařízení připojený do /dev, aby bylo možné přistupovat k jednotlivým zařízením. Další souborové systémy mohou být připojeny podle potřeby.

- **Model ovladačů (zapůjčený z Linuxu)**

Barebox následuje linuxový ovladačový model: zařízení mohou být specifikována v hardwarově specifickém souboru a ovladače jsou zodpovědné za zařízení pokud mají stejné jméno.

- **Zdroj hodin**

Používá se stejné API pro práci se zdroji hodin jako v Linuxu.

- **Kconfig/Kbuild**

K sestavení Bareboxu se používá systém Kconfig/Kbuild známý z linuxového jádra, který uložňuje paralelizované sestavování a odstraňuje nutnost mít v kódu hodně ifdefů.

- **Sandbox**

Při vyvíjení Bareboxu je možné sestavit Barebox jako 'sandbox', což zkompiluje Barebox jako standardní POSIXovou aplikaci pro Linux. Tato aplikace může být spuštěna jako normální příkaz a dokonce má i přístup k síti. Soubory z lokálního souborového systému mohou být použity k simulaci zařízení.

- **Parametry zařízení**

Barebox obsahuje model parametrů pro zařízení – každé zařízení může specifikovat vlastní parametry a tyto parametry existují pro každou instanci tohoto zařízení. Parametry mohou být měněny na příkazové řádce ve stylu \<devid\>.\<param\>=”…”. Například změnu IPv4 adresy síťové karty zastupované zařízením eth0 je možné provést příkazem *'eth0.ip=192.168.0.7*' a *'echo $eth0.ip'*.

- **Getopt**

Barebox má odlehčenou implementaci funkce getopt(). Díky tomu není nutné identifikovat parametry příkazů jen podle pozice, což může mít negativní vliv na čitelnost.

- **Integrovaný editor**

Skripty mohou být upravovány malým integrovaným fullcreen editorem. Tento editor má pouze nutné minimum funkcí: posouvání kurzoru, vkládání znaků, uložení a ukončení.

3.3 Linuxové jádro

Linuxové jádro je možné sestavit ručně, nebo ho nechat sestavit v rámci nástrojů pro tvorbu embedded systému, které jsou popsány v kapitole Nástroje pro tvorbu systému.

Ve všech případech však musí být daný procesor + periferie podporovány buď přímo v jádře, nebo musí existovat patche, které tuto podporu přidávají.

Nastavení linuxového jádra závisí z velké části na hardwaru, na kterém bude provozováno a na softwaru, který bude na platformě instalován. Pouze výjimečně se ve vestavěných systémech používá systém modulů, neboť jádro zpravidla nepotřebuje podporovat více různých konfigurací a je na míru vytvořeno danému hardwaru.

3.3.1 Adresářová struktura jádra pro podporu platformy ARM

Hlavní adresář obsahující zdrojové kódy pro platformu ARM je *arch/arm*. V tomto adresáři je několik souborů a podadresářů, které obsahují hlavičkové soubory, zdrojové soubory a další.

Význam a umístění důležitých souborů a adresářů:

Kconfig – konfigurační soubor pro nástroj Kconfig, který popisuje jednotlivé volby pro platformu ARM

Makefile – soubor obsahující instrukce pro pro sestavovací systém make

boot – obsahuje podpůrné nástroje jádra týkající se bootování

boot/dts – obsahuje Device Tree Source (dts) a Device Tree Source Include(dtsi) soubory, které popisují hardwarové zapojení desek, její periferie a podobně

configs – obsahuje předpřipravené konfigurace jádra pro specifické platformy, v budoucnosti pravděpodobně dojde k odstranění/neaktualizování tohoto adresáře

mach-<název_subarchitektury> – obsahuje soubory specifické pro danou subarchitekturu – podpora procesoru, správu napájení, správu paměti, časovače, obecná inicializace desky,… Konkrétní uspořádání tohoto adresáře se značně liší.

plat-<název> – některé podobné subarchitektury využívají plat-<název> pro společný kód

3.3.2 Podpora starších subarchitektur

Pro starší a již málo používané subarchitektury je zachován systém, při kterém je veškerý popis hardwaru na desce realizovaný v C souboru pro danou desku. Problém s tímto přístupem je ten, že při jakékoliv změně zapojení, nebo třeba výměně jednoho z čipů, je nutné provádět úpravy ve zdrojových kódech. Navíc v případě 2 totožných desek s například různým síťovým řadičem je nutné buď brát každou desku jako samostatnou platformu, nebo do souboru přidat preprocesorové makra pro kontrolu kompilace, které mají za následek výrazně horší čitelnost kódu a náchylnost k chybám. Navíc způsobují rychle rostoucí komplexitu konfigurace jádra. Problém je rovněž s procesorovými moduly, které nejsou v návrhu zohledněny vůbec a jejichž použití má podobné následky.

3.3.3 Podpora novějších subarchitektur

Pro novější subarchitektury se používá struktura Flattened Device Tree (FDT). Kromě zredukování duplikovaného kódu a postupné standardizaci není díky FDT nutné dělat pro podporu nových desek zásahy ve zdrojovém kódu jádra. Podpora FDT není hotová pro všechny ARM aubarchitektury – pouze pro ty aktuálně používané a nově přidávané.

3.3.4 Porovnání obou přístupů

Pro porovnání je použita ARM subarchitektura nVidia Tegra, která ve starších jádrech je tvořena starším způsobem se zvláštními soubory se zdrojovým kódem pro každou podporovanou desku a v novějších jádrech již využívá Device Tree. Implementace této subarchitektury je v jádře v adresáři *arch/arm/mach-tegra*.

Pro porovnání bude jako starší jádro sloužit verze 3.0.71 a jako novější bude nejnovější mainline – 3.9-rc5. Následující tabulka srovná obě jádra z pohledu podpory různých desek.

Kritérium	3.0.71	3.9-rc5
Počet souborů v *arch/arm/mach–tegra*	44	50
Z toho soubory pro podporu konkrétních desek	14	3
Počet podporovaných desek	6	18

Tabulka 2 – porovnání starší a novější implementace subarchitektury

Při porovnání počtu souborů je pak patrné, že mezi těmito verzemi jádra ubylo 11 souborů pro podporu konkrétních desek a přibylo 17 souborů vylepšující podporu této subarchitektury (podpora uspávání, řízení frekvence procesorů a další.)

4 MECHANISMUS SNAPSHOTŮ

Tento mechanismus, někdy nazývaný suspend to disk nebo hibernace, umožňuje snížit dobu načítání systému tím, že se při vypínání systému uloží funkční stav a při startu systému se tento snapshot načte zpět do paměti. Množství ušetřeného času silně závisí na velikosti nasazeného systému, rychlosti permanentní paměti a rychlosti procesoru. Linux obecně tento přístup v hlavní větvi jádra na platformě ARM nepodporuje a důvodů pro to je několik. Prvním důvodem je nutnost spolupráce ovladačů hardwaru se systémem na vytváření snapshotů. Ovladače musí podporovat uložení aktuálního stavu a zpětně obnovení bez nutnosti provádět kompletní inicializaci. Pokud se obnovení ze snapshotu provádí až v linuxovém jádře, je ušetřený čas poměrně nízký, protože se před obnovením ze snapshotu musí inicializovat ovladače. Řešením je provádět obnovení ze snapshotu už ze zavaděče, ale to opět vyžaduje určitou minimální inicializaci ovladačů.

I přes problémy, které obnovení ze snapshotu provázejí je zdokumentováno několik případů, kdy se pomocí systému snapshotů na různých zařízeních dosáhnout významné úspory času. Bez vyjímek ale na zprovoznění pracovali lidé s vysokými zkušenostmi s linuxovým jádrem, protože jsou nutné poměrně velké změny v jádře, jaderných ovladačích a zavaděči. [23]

Pro starší linuxová jádra jsou k dispozici sady patchů, které tuto funkcionalitu do jisté míry přidávají, ale jejich funkčnost je závislá na konkrétním hardwarovém vybavení a použitých ovladačích. Tyto patche lze najít pod názvem *swsusp* nebo *suspend2* for ARM. [24]

Navíc je možné použít řešení QuickBoot od firmy Ubiquitous, které je ale komerční. [25]

II. PRAKTICKÁ ČÁST

5 LINUXOVÉ NÁSTROJE

Tato kapitola je zařazena z toho důvodu, že pro použití jakéhokoliv nástroje pro vytvoření linuxového systému s nimi uživatel pravděpodobně přijde do styku. Tato práce však ke čtení předpokládá určitou zkušenost s Linuxem a programováním, proto jsou popsány pouze ne úplně běžné nástroje.

5.1 Konfigurační nástroj Kconfig

Kconfig je konfigurační mechanismus, který původně vznikl ke konfiguraci linuxového jádra a který postupně adaptovali další open source projekty jako je BusyBox, Buildroot, crosstools–ng nebo uClibc. Tento mechanismus vytváří stromovou strukturu konfiguračních voleb, mezi kterými mohou být různé vzájemné závislosti a kterými je možné nakonfigurovat například linuxové jádro přesně na míru.

Existuje více variant konfiguračních „frontendů“ jak pro konzoli (menuconfig) tak pro grafické prostředí (xconfig). Výstupem konfigurace je jeden soubor, který má běžně název .config. V něm jsou veškeré volby, které byly pomocí konfiguračního nástroje nastaveny.

Příklad volby nastavené na ano:

CONFIG_ARM=y

Příklad volby nastavené na ne:

 # CONFIG_MODULE_FORCE_LOAD is not set

Příklad volby nastavené na hodnotu:

CONFIG_INIT_ENV_ARG_LIMIT=32

Z tohoto konfiguračního souboru je pak ještě možné vyrobit takzvaný defconfig, což je soubor, který má stejnou syntaxi, pouze jsou vynechány volby, které jsou nastavené stejně, jako je výchozí stav (jsou redundantní).

Jedna z výhod Kconfigu je jeho jednoduchá syntaxe. Každá volba (s výjimkou nejvýše postavených voleb) má nějaké rodiče a případně potomky. Každá volba je viditelná pouze, pokud je povolen její rodič. Voleb je několik typů: bool, tristate, string, hex, int. Řádky v kconfig souboru začínají klíčovým slovem, za kterým může následovat několik

argumentů. Každá konfigurační volba začíná řádkem, na kterém je *config NAZEV_VOLBY* a na dalších řádcích následují její atributy.

Na následující ukázce bude popsán základ syntaxe Kconfigu (ukázka pochází ze souboru jádra v *arch/arm/mach–tegra/Kconfig*.)

```
config MACH_HARMONY
       bool "Harmony board"
       depends on ARCH_TEGRA_2x_SOC
       select MACH_HAS_SND_SOC_TEGRA_WM8903 if SND_SOC
       help
         Support for nVidia Harmony development platform
```

Zdrojový kód 1 – ukázka syntaxe Kconfig souborů

config MACH_VENTANA

Udává, že se jedná o konfigurační volbu s názvem *MACH_HARMONY*, ve vygenerovaném konfiguračním souboru bude tato volba pojmenována *CONFIG_MACH_HARMONY* a pod stejným názvem je tento symbol identifikován v Makefilech a zdrojových souborech. Díky tomu lze například určitou část kódu zkompilovat pouze, pokud je symbol nastaven na ano, viz Zdrojový kód 2 – zdrojový kód podmíněný nastavením Kconfig symbolu.

```
#if defined(CONFIG_MACH_HARMONY)
/* Harmony specific code here */
#endif //CONFIG_MACH_HARMONY
```

Zdrojový kód 2 – zdrojový kód podmíněný nastavením Kconfig symbolu

bool "Harmony board"

Tento řádek udává 2 informace. První z nich že se jedná o typ volby bool, tedy o volbu s možnostmi pouze ano nebo ne. Druhá část udává, pod jakým názvem bude volba zobrazována při konfiguraci pomocí menuconfigu apod.

depends on ARCH_TEGRA_2x_SOC

Tímto je zajištěno, že tato volba nelze vybrat, pokud není zatržena nadřízená volba, což je v tomto případě *ARCH_TEGRA_2x_SOC*.

select MACH_HAS_SND_SOC_TEGRA_WM8903 if SND_SOC

Tímto je zajištěno, že v případě, že uživatel povolí tuto volbu a zároveň i volbu *SND_SOC*, automaticky se povolí i odpovídající symbol *MACH_HAS_SND_SOC_TEGRA_WM8903*.

help

Udává, že na následujícím řádku je popsána nápověda k dané volbě.

5.2 Ladění a kontrola

5.2.1 ldd

Program ldd vypisuje sdílené knihovny, na kterých předložená sdílená knihovna nebo spustitelný soubor závisí. Díky tomu lze snadno zjistit chybějící závislosti, případně že program využívá nesprávnou sdílenou knihovnu.

```
$ ldd /bin/bash
        linux-vdso.so.1 =>  (0x00007fff074e7000)
        libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007fac0e556000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fac0e352000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fac0df92000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fac0e796000)
```

Zdrojový kód 3 – ukázka použití ldd

5.2.2 gdb

GDB(GNU Debugger) je v linuxovém světě nejpoužívanější debugger. Jeho nevýhodou je poměrně obtížné ovládání, proto se často používá prostřednictvím nějakého frontendu(DDD, plugin v Eclipse apod.) Pokud však prostřednictvím něj spustíme program, který je na cílové platformě nestabilní, je možné pomocí něj zjistit, v čem je chyba. Spuštění programu pod GDB se provádí následujícím způsobem.

```
$ gdb --args echo "Hello GDB"
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
```

```
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /bin/echo...(no debugging symbols found)...done.
(gdb) run
Starting program: /bin/echo Hello\ GDB
Hello GDB
[Inferior 1 (process 16474) exited normally]
(gdb
```

Zdrojový kód 4 – ukázka použití GDB

5.2.3 strace

Strace je linuxový nástroj pro diagnostiku a ladění. Jeho úkolem je zachytávat a vypisovat systémová volání, která provádí volaný program. Pro každé systémové volání vypisuje jeho název, argumenty se kterými je voláno i návratovou hodnotou. Analýzou těchto volání lze detekovat a případně vyřešit řadu chyb i v programech bez dostupného zdrojového kódu. Mezi nejsnadněji detekovatelné chyby patří selhání při otevírání souboru nebo nenalezené sdílené knihovny. Na ukázce Zdrojový kód 5 – ukázka běhu strace je nástroj strace aplikován na příkaz *echo "strace demonstration"*.

```
$ strace echo "strace demonstration"
execve("/bin/echo", ["echo", "strace demonstration"], [/* 10 vars */]) = 0
brk(0)                                  = 0x1200000
uname({sys="Linux", node="tqm", ...})   = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x76fcb000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = -1 ENOENT (No such file or directory)
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0\210z\1\0004\0\0\0"..., 512) =
512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1176952, ...}) = 0
mmap2(NULL, 1213696, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x76e7e000
mprotect(0x76f9a000, 28672, PROT_NONE)  = 0
mmap2(0x76fa1000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
0x11b) = 0x76fa1000
mmap2(0x76fa4000, 9472, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
0) = 0x76fa4000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x76fca000
set_tls(0x76fca4c0, 0x76fcab98, 0x76fca4c0, 0x76fcd048, 0x76fcd048) = 0
```

```
mprotect(0x76fa1000, 8192, PROT_READ)   = 0
mprotect(0x76fcc000, 4096, PROT_READ)   = 0
getuid32()                              = 0
brk(0)                                  = 0x1200000
brk(0x1221000)                          = 0x1221000
write(1, "strace demonstration\n", 21strace demonstration
)  = 21
exit_group(0)                           = ?
+++ exited with 0 +++
```

Zdrojový kód 5 – ukázka běhu strace

Jak je na ukázce vidět, i jednoduché příkazy způsobí velké množství systémových volání. Pro detekování chyb v nestabilním programu jsou však zpravidla klíčové poslední volání před pádem aplikace, proto není nutné číst celé výpisy.

6 NÁSTROJE PRO TVORBU SYSTÉMU

Vzhledem k množství a rychlosti tvorby nových embedded systémů vznikly nástroje, které celou činnost značně usnadňují a integrují v sobě většinu požadovaných nástrojů. K dispozici je poměrně veliké množství těchto nástrojů, proto jsou dále popsány pouze ty, které byly v rámci práce vyzkoušeny.

6.1 Buildroot

Buildroot je sada Makefilů a patchů, pomocí kterých je možné vygenerovat křížový překladač, zavaděč, linuxové jádro a root filesystém. [11]

Pro konfiguraci buildrootu se používá jazyk Kconfig, který je popsán v samostatné kapitole. Konfigurace Buildrootu je možná pomocí rozhraní pro CLI nebo GUI. Samotný Buildroot v sobě obsahuje velké množství balíčků a je poměrně jednoduché přidat další.

6.1.1 Stažení nástroje Buildroot

Buildroot je možné stáhnout v několika verzích. Přibližně každé 3 měsíce vychází stabilní verze. Pro ty, co vyžadují co možná nejnovější verze jsou k dispozici denní snapshoty nebo přímý přístup do Git repozitáře.

6.2 PTXdist

Stejně jako projekt Buildroot je i projekt PTXdist zaměřený na vygenerování kompletních systémů pro vestavěné zařízení. A podobně se i jeho struktura skládá z velké části z Makefilů a Kconfig souboru. Oproti Buildrootu ale PTXdist působí výrazně přehledněji a předvídatelněji. Jednotlivé sestavené části lze libovolně vyřazovat a překompilovávat, instalaci jednotlivých balíků lze provést po krocích a snadněji tak analyzovat potencionální chyby. Navíc za vývojem nástroje PTXdist stojí firma Pengutronix, která se zabývá zakázkovým portováním Linuxu na různé platformy a hostuje i projekt Barebox. [12]

Součástí je i nástroj pro generování kostry pro různé druhy nových balíčků, který výrazně usnadňuje přidávání nového softwaru.

6.2.1 Části nástroje PTXdist

1) ptxdist program

Program ptxdist je nainstalován na vývojovém počítači, spouští se pro vyvolání všech akcí, jako je zkompilování nějakého balíčku, vygenerování obrazů, jádra apod. Obyčejně se program ptxdist spouští ve workspace adresáři, který obsahuje všechny potřebné soubory ke správnému běhu.

2) *Konfigurační systém*

Konfigurační systém se používá pro upravení konfigurace, která obsahuje informace o tom, která balíčky se mají sestavit a jaké jsou nastavení projektu.

3) *Patche*

PTXdist obsahuje mechanismus pro automatické aplikování patchů na různé balíčky. To je nutné z toho důvodu, že řada balíčků obsahuje chyby – především v ohledu na křížové překládání a je tak nutné na ně aplikovat v rámci PTXdist opravy.

4) Popis balíčků

Pro každou softwarovou komponentu PTXdist obsahuje „recept“ – soubor příkazů a akcí, který je nutný pro získání, sestavení a nainstalování dané komponenty do systému. Každý balíček má navíc vlastní soubor pro konfigurační systém.

5) Toolchainy

PTXdist neobsahuje žádný předkompilovaný toolchain, ale je schopný si sestavit toolchainy, které jsou k dispozici v rámci projektu OSELAS.Toolchain().

6) Balíček pro podporu desky

Board Support Package(BSP) – Předkonfigurovaný balíček pro podporu konkrétní desky. Zpravidla se dodává k nějakému hardwaru, ale existují i obecné BSP připravené pro přizpůsobení na míru konkrétní platformě.

[13]

6.2.2 Stážení a zkompilování nástroje PTXdist

Následující seznam linuxových příkazů je možné použít ke zkompilování nástroje PTXdist. Program si v případě nesplněných závislostí může vyžádat instalaci dalších balíčků. Alternativou k uvedenému postupu je použít verzi dostupnou v balíčkovacím systému použité distribuce.

```
$ wget http://www.ptxdist.org/software/ptxdist/download/ptxdist-2013.01.90.tar.bz2
$ tar xzf ptxdist-2013.01.90.tar.bz2
$ cd ptxdist-2013.01.90
$ ./configure
$ make
$ sudo make install
```

Zdrojový kód 6 – stáhnutí a kompilace PTXdist

6.3 OpenEmbedded

Kromě již zmíněných nástrojů existuje i framework OpenEmbedded, za kterého dále vychází Yocto Project a Embeddded Linux Development Kit (ELDK). Tento nástroj již nevyužívá Makefily, ale namísto toho je složený z takzvaných BitBake receptů, které jsou odvozené od systému, jaký používá linuxová distribuce Gentoo.

Cílem OpenEmbedded a odvozených je spíše vytvoření distribuce s balíčkovacím systémem než vygenerování jednoho systému. Pro velkou část projektů je tento postup zbytečně komplikovaný a nepřináší užitek.

7 VYTVOŘENÍ PODPORY PRO NOVOU DESKU

7.1 Podpora v zavaděči

Nejlepší postup pro vytvoření podpory zavaděče pro novou desku je vyjít z již existující podpory pro podobnou desku. Zavaděče však trpí několika problémy, z nichž asi nejvýraznějším je nedostatečná nebo nekorektní dokumentace.

7.1.1 U-boot

V případě U-bootu není přes velmi rozsáhlou dokumentaci týkající se běhu U-bootu žádná oficiální dokumentace týkající se portování U-bootu na nové desky, i když lze nalézt informace z různých dalších zdrojů. Tyto postupy však jsou jen na určitou verzi U-bootu a mohou být zastaralé. Mezi použitelné zdroje lze zařadit následující: [14], [15], [16], [17].

Následující seznam kroků je možné použít jako výchozí cestu pro vytvoření podpory U-bootu pro novou desku. Některé desky však mohou vyžadovat přidání dalších ovladačů, nebo další přizpůsobení.

1. Stažení vhodné verze U-bootu – oficiální stabilní verze, větev pro danou platformu, nebo verze upravená od výrobce dané desky
2. Nalezení co nejpodobnější již podporované desky
3. Vytvoření konfiguračního souboru v *include/configs/jméno_desky.h* na základě již existujícího konfiguračního souboru pro jinou desku
4. Vytvoření podpůrného kódu v adresáři *board/jméno_desky/*
5. Přidání desky do souboru MAKEALL – v nových verzích U-bootu již není potřeba
6. Přidání desky do souboru Makefile, opět podobně jako u vzorové desky
7. Zkontrolovat, jestli je definovaný odpovídající MACH_TYPE v souboru include/asm-arm/mach-types.h, pokud ne, je nutné ho přidat.

[14]

7.1.2 Barebox

Barebox má v dokumentaci sekci, která se zabývá portováním na novou desku, ale není úplně kompletní a některé postupy – například pojmenování souborů – nejsou v existujících zdrojových kódech dodržovány, což zhoršuje orientaci. I pro Barebox je možné nalézt další zdroje týkající se jeho portování, například [18], [19], [20].

Pro portování Bareboxu je vhodné využít obdobný postup, jako pro portování U-bootu, tedy vyjít z podpory pro nějakou podobnou desku a pouze ji přizpůsobit. Navíc je možné využít jako základ kód z U-bootu, který obsahuje podporu pro větší množství desek.

7.2 Podpora v linuxovém jádře

Náročnost vytváření podpory pro novou desku silně závisí na verzi použitého jádra a subarchitektury. Ve starších jádrech bez podpory Device Tree je nutné pro podporu nové desky vytvořit jeden nebo více souborů se zdrojovým kódem, případně formou podmíněné kompilace poupravit již existující soubory.

Protože je nyní doporučování používat Device Tree, je dále popisován pouze tento způsob.

7.2.1 Struktura Device Tree Source souborů

Každý DTS soubor je uspořádán do stromové struktury, který je tvořena Device Tree uzly (Nodes). Až na výjimky každý uzel popisuje nějaké zařízení nebo sběrnici a obsahuje vlastnosti, které jsou zapisovány stylem název = hodnota. Jako příklad je uvedená zkrácená ukázka pro podporu tlačítkové klávesnice připojené na GPIO piny procesoru.

```
/* Uzel popisující GPIO klávesnici s názvem gpio-keys */
gpio-keys {
                /* Klíč compatible udává, jaký typ zařízení uzel *
                 * popisuje a jaký ovladač se pro něj použije    */
                compatible = "gpio-keys";

                /* Uzel popisující jednu klávesu s názvem function-1  */
                function-1 {
                        /* Popisek klávesy */
                        label = "Function 1";
                         /* Definice GPIO pinu, na kterém *
```

```
                        * je tlačítko připojeno            */
                       gpios = <&gpio3 21 1>;
                        /* Kód klávesy, který se při stisknutí *
                         * tlačítka v systému objeví          *
                         * (zde je to F1)                     */
                       linux,code = <59>; /* KEY_FN_1 */
               };

               /* Uzel popisující jednu klávesu s názvem function-2  */
               function-2 {
                       label = "Function 2";
                       gpios = <&gpio2 27 4>;
                       linux,code = <60>; /* KEY_FN_2 */
               };
};
```

Zdrojový kód 7 – ukázka Device Tree uzlu

8 VYTVOŘENÍ SYSTÉMU

Nástroje pro tvorbu systému již nemusejí obsahovat žádné platformě specifické věci, které by bylo nutné přizpůsobovat a stačí je k vygenerování funkčního systému správně nastavit.

8.1 Buildroot

Nejprve je nutné mít stažený vybraný archív s Buildrootem - informace kde stáhnout tento archív je v kapitole 6.1.1 Stažení nástroje Buildroot. Po rozbalení tohoto archívu do adresáře, ve kterém chceme vytvářet systém již postačuje z adresáře s buildrootem zavolat příkaz *make menuconfig*, pomocí kterého buildroot nakonfigurujeme přesně podle požadavků. Nakonfigurovaný buildroot poté spustíme příkazem *make*. Jednoduchost Buildrootu má však svoje úskalí – pokud proces vyžaduje použití vlastních konfiguračních souborů, vlastního upraveného jádra a podobně, stává se aktuální konfigurace nepřenositelná na jiný počítač. Řešením je používat namísto pevných cest k souborům cesty, které jsou relativní k adresáři Buildrootu, nebo vytvořit skript s definicí těchto cest do proměnných prostředí a volat Buildroot z něj. Ukázku takového skriptu je možné nalézt v příloze P III: Skript pro práci s Buildrootem.

8.2 PTXdist

Dále uvedený postup předpokládá, že PTXdist už je v systému nainstalovaný buď pomocí postupu popsaného v popsaný v kapitole 6.2.2 nebo pomocí balíčkovacího systému.

Před započetím práce je nutné mít nainstalovaný alespoň jeden toolchain, který bude použit ke zkompilování a sestavení všech nutných součástí. Pro PTXdist se doporučuje použít toolchain sestavený pomocí OSELAS.Toolchain(), což je systém pro vygenerování toolchainu využívající PTXdist. Toolchainy je možné získat několika způsoby. Tím nejjednodušším je nainstalovat hotový toolchain pomocí balíčkovacího systému – například pro Ubuntu/Debian existuje repozitář, který stačí přidat mezi zdroje balíčkovacího systému a poté z něj nainstalovat vybraný toolchain.[21] Alternativně lze stáhnout některé již sestavené toolchainy, které distribuují někteří výrobci desek.

V případě, že pro použitou platformu není dostupný žádný hotový toolchain, nepoužíváme distribuci založenou na Debianu nebo požadujeme některé specifické věci, které generický toolchain neposkytuje, je nutné si sestavit vlastní toolchain.

8.2.1 Sestavení OSELAS.Toolchain()

Prvním krokem je stažení archívu obsahující PTXdist konfiguraci pro sestavení toolchainu ze stránek PTXdist zde: *http://www.ptxdist.de/oselas/toolchain/download/*. Následně je nutné archív rozbalit, přejít do rozbaleného adresáře, vybrat požadovaný toolchain a spustit kompilaci.

Pro nejaktuálnější verzi OSELAL Toolchainu (2012.12.1) a procesor s jádrem Cortex A8 by postup vypadal následovně:

```
$ tar xf OSELAS.Toolchain-2012.12.0.tar.bz2

$ cd OSELAS.Toolchain-2012.12.0

$ ptxdist select ptxconfigs/arm-cortexa8-linux-gnueabi_gcc-4.7.3_glibc-2.16.0_binutils-
2.22_kernel-3.6-sanitized.ptxconfig

$ ptxdist go
```

Zdrojový kód 8 – sestavení OSELAS toolchainu

Zkompilovaný toolchain se nainstaluje do adresáře */opt/OSELAS.Toolchain-2012.12.0/arm-cortexa8-linux-gnueabi/gcc-4.6.2-glibc-2.14.1-binutils-2.21.1a-kernel-2.6.39-sanitized/*. Před prvním použitím je ještě doporučeno k vygenerovanému toolchainu nastavit práva pouze pro čtení, aby se zabránilo jeho poškození.

8.2.2 Vytvoření a přizpůsobení BSP

V terminologii emebedded systémů se kolekce nástrojů a programů pro specifickou desku nazývá BSP (Board Support Package) neboli balík pro podporu desky. Tento termín nemá žádný zavedený překlad, proto se dále využívá zkratka BSP.

V rámci PTXdist se BSP skládá z konfiguračních souborů, patchů, specifických balíků a podobně. Na webu PTXdist lze stáhnout několik hotových BSP, další jsou k dispozici například od některých výrobců evaluation kitů. Jako příklad bude použit Generic BSP, který je možné stáhnout na adrese *http://oselas.com/oselas/bsp/pengutronix/download*. Po extrahování a přejití do extrahovaného adresáře již je možné spouštět příkazy PTXdistu, pomocí kterých se projekt nakonfiguruje. Následující příklad ilustruje přípravu Generic BSP.

```
$ wget http://oselas.com/oselas/bsp/pengutronix/download/OSELAS.BSP-Pengutronix-Generic-2012.12.0.tgz

$ tar xf OSELAS.BSP-Pengutronix-Generic-2012.12.0.tgz

$ cd OSELAS.BSP-Pengutronix-Generic-2012.12.0.tgz
```

Zdrojový kód 9 – získání BSP

8.2.2.1 Adresářová struktura BSP v PTXDistu

Adresář	**Popis**
configs	Obsahuje konfigurační soubory ptxdistu a jednotlivých balíčků (například jádra, zavaděče, apod.) V podadresáři *configs/platform-název_platformy/patches* se obvykle umisťují patche jádra a zavaděče.
local_src	Obsahuje archívy nebo adresáře pro balíčky specifické k projektu – například vlastní aplikace v projektu.
patches	Obsahuje patche jednotlivých balíčků konfigurovaných přes *menuconfig*.
platform-název_platformy	V tomto adresáři probíhá kompilace všech součástí a sestavování cílového obrazu. Sestavené obrazy je možné nalézt v podadresáři *images*.
projectroot	Obsahuje konfigurační soubory kopírované do cílového systému. Například konfigurace sítě, skripty pro start aplikace po startu apod.
rules	Obsahuje konfigurační soubory s příponou .in, které ovládají zobrazování v menuconfigu a závislosti balíčku a soubory příponou .make, které popisují proces sestavování balíčku, udávají jaké výsledné soubory se kopírují do cílového systému a další.
src	Do této složky se stahují veškeré používané balíčky.
tests	Tato složky obsahuje automatizované testy, kterými je možné ověřit správnost sestaveného systému

Tabulka 3 – adresářová struktura BSP v PTXdistu

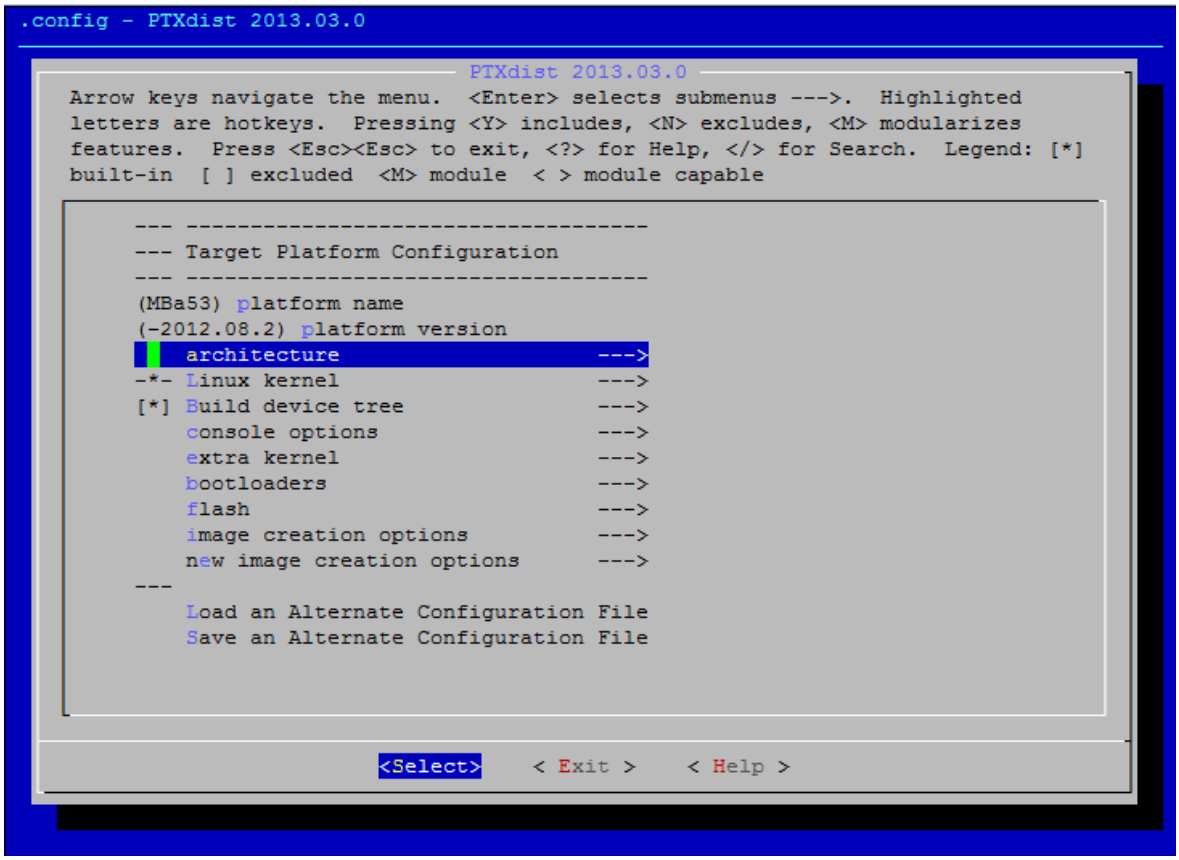
8.2.2.2 Konfigurace platformy

První věc, kterou je nutné udělat, je nastavit toolchain, který bude projekt využívat. To je možné učinit dvěma způsoby. Jedním z nich je použití příkazu *ptxdist toolchain /cesta/k/toolchainu*, nebo ho lze nastavit v dialogu vyvolaném příkazem *ptxdist platformconfig*.

```
$ ptxdist toolchain /opt/OSELAS.Toolchain-2011.11.1/arm-cortexa8-linux-gnueabi/gcc-
4.6.2-glibc-2.14.1-binutils-2.21.1a-kernel-2.6.39-sanitized/bin'
```

Zdrojový kód 10 – nastavení toolchainu pomocí příkazu ptxdist toolchain

Příkaz *ptxdist platformconfig* oproti tomu vyvolá dialog, který je možné vidět na Obrázek 9 – ptxdist platformconfig.



Obrázek 9 – ptxdist platformconfig

V tomto dialogu je možné nastavit toolchain v sekci architecture → toolchain. V sekci architecture lze dále nastavit řadu důležitých voleb, jako je cílová architektura a různá

nastavení zvyšující bezpečnost. V dalších sekcích lze pak ovlivnit, jestli se bude v rámci systému sestavovat linuxové jádro, jaký bude použit zavaděč, generované obrazy a další.

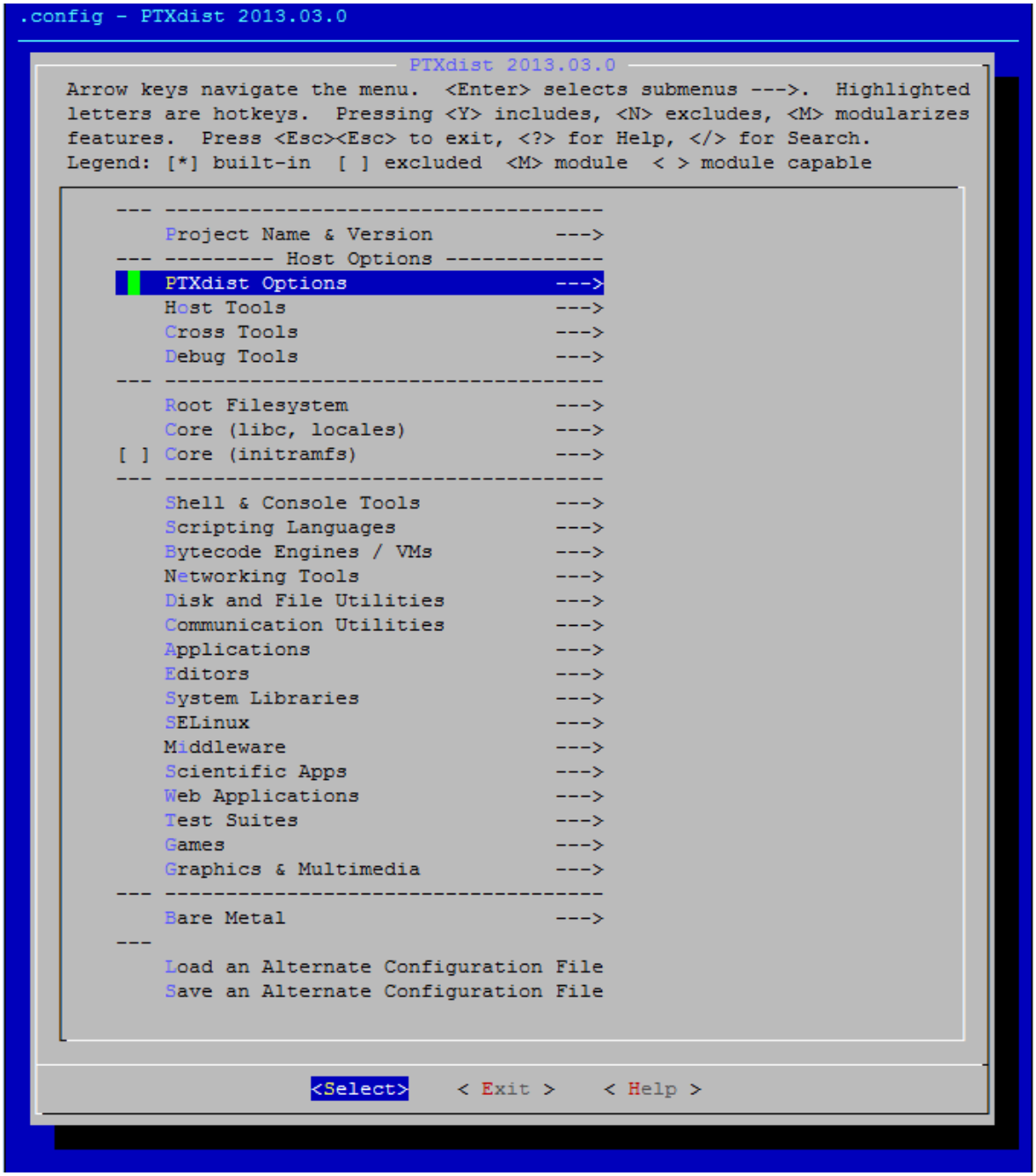
8.2.2.3 Konfigurace jádra a softwaru

V rámci PTXdist je možná nakonfigurovat jádro příkazem *ptxdist kernelconfig*, který zobrazí to samé jako *make menuconfig* v adresáři jádra. Před spuštěním příkazu musí být korektně nastavená platforma, jinak by se v kernelconfigu mohly objevit nastavení pro jinou procesorovou architekturu, nebo by se vůbec nespustila – například v případě špatně nastaveného toolchainu.

V konfiguraci jádra je nutné vybrat správný procesor, desku, ovladače a jiné nastavení, která jsou specifická pro použitou desku. Vybrané nastavení by také měly odrážet požadovanou aplikaci a neměly by být vybrány nepotřebné součásti. Také je výhodnější všechny součásti kompilovat jako součásti jádra a ne jako moduly, protože zavádění modulů dynamicky sebou nese zbytečný overhead.

Konfigurace softwaru se provádí příkazem *ptxdist menuconfig*. V zásadě zde platí obdobné zásady jako u konfigurace jádra, tedy že by měly být vybrány jen nutné součásti. Množství instalovaného softwaru sice nemá vliv na dobu bootu, ale zbytečně prodlužuje dobu kompilace systému a může zanést zbytečné bezpečnostní rizika.

PTXdist obsahuje velké množství aplikací, z nichž některé jsou přizpůsobeny pro křížové překládání na jiné platformy. Kromě všech možných linuxových nástrojů a aplikací je zde možné nalézt i multimediální aplikace, knihovny Qt & GTK, nebo třeba Java Runtime Environment a množství balíků stoupá s každou verzí. V případě, že nějaký software chybí, je také poměrně jednoduché vytvořit vlastní balík a integrovat ho.



Obrázek 10 – volby softwaru v PTXdist

8.2.3 Vytvoření vlastních balíčků

I přes veliké množství software v PTXdistu je často nutné přidat vlastní software. Je sice možné vlastní programy samostatně přeložit a poté nakopírovat do výsledných obrazů, ale zpravidla je výrazně jednodušší vytvořit pro vlastní software balíčky, které zajistí kompilaci a kopírování do cílového systému samy. V PTXdistu jsou pro vytváření vlastních balíčků šablony a průvodce, který celý proces usnadňuje.

8.2.3.1 Použití průvodce

Pro vytvoření nového balíčku pomocí průvodce slouží příkaz *ptxdist newpackage typ_balíčku*. Pokud je tento příkaz spuštěn bez zadaného typu balíčku, vypíše možnosti, jaké typy balíčků jsou k dispozici, viz Zdrojový kód 11 – použití ptxdist newpackage bez zadání typu balíčku.

```
$ ptxdist newpackage


usage: 'ptxdist newpackage <type>', where type is:


  host                      create package for development host
  target                    create package for embedded target
  cross                     create cross development package
  klibc                     create package for initramfs built against klibc


  src-autoconf-lib          create autotoolized library
  src-autoconf-prog         create autotoolized binary
  src-autoconf-proglib      create autotoolized binary+library
  src-cmake-prog            create cmake binary
  src-qmake-prog            create qmake binary
  src-linux-driver          create a linux kernel driver
  src-make-prog             create a plain makefile binary
  src-stellaris             create stellaris firmware
  font                      create a font package
  file                      create package to install existing files
  kernel                    create package for an extra kernel
  barebox                   create package for an extra barebox
  image-tgz                 create package for a tgz image
  image-genimage            create package for a genimage image
```

Zdrojový kód 11 – použití ptxdist newpackage bez zadání typu balíčku

Kompletní popis k čemu který je balíček lze nalézt v PTXdist manuálu [22].

Příklad ukázaný v Zdrojový kód 12 demonstruje vytvoření balíčku pro vlastní aplikaci postavenou na frameworku Qt s použitím qmake.

```
$ ptxdist newpackage src-qmake-prog


ptxdist: creating a new 'src-qmake-prog' package:


ptxdist: enter package name...........: MyQtDemoApp
ptxdist: enter version number.........: 1.00
ptxdist: enter package author.........: Roman Dosek
ptxdist: enter package section........: project_specific


generating rules/MyQtDemoApp.make
generating rules/MyQtDemoApp.in


local_src/myqtdemoapp-1.00 does not exist, create? [Y/n] Y
./
./wizard.sh
./install.pri
./@name@.cpp
./@name@.pro
```

Zdrojový kód 12 – ukázka vytvoření balíčku s pomocí průvodce

8.2.3.2 Úprava .in a .make souborů

Pro některý software jsou balíčky vytvořené pomocí průvodce již kompletně funkční a není nutné je dále upravovat, jindy je ale nutné balíčky vytvořené pomocí průvodce upravovat, například kvůli kopírování dalších souborů do cílového systému.

8.2.3.3 Testování balíčku

Sestavení každého balíčku je rozděleno do několika částí, a tyto části lze provádět samostatně a ověřit tak každý krok při vytváření nového balíčku.

1) Stažení

Příkaz: *ptxdist get název_balíčku*

Popis: Zdroj, odkud se balíček stahuje je definován v příslušném make souboru. Zdrojem může být http, ftp, git, nebo třeba lokální soubor (file).

2) Extrahování

Příkaz: *ptxdist extract název_balíčku*

Popis: Extrahování balíčku v závislosti na příponě, jako umá soubor definovanou v make souboru.

3) Příprava

Příkaz: *ptxdist prepare název_balíčku*

Popis: V této fázi se spouští zpravidla configure skript pro přípravu balíčku ke kompilaci, ale v závislosti na typu balíčku je možné zde spouštět libovolné příkazy.

4) Kompilace

Příkaz: *ptxdist compile název_balíčku*

Popis: Ke zkompilování balíčku se zpravidla používá příkaz make. PTXdist obsahuje i několik šablon pro nejčastěji používané kompilační systémy (Autotools, qmake, CMake,…)

5) Instalace

Příkaz: *ptxdist install název_balíčku*

Popis: Nainstaluje potřebné soubory do hostitelského systému – například knihovny, na kterých závisí další programy. Adresář, do kterého se soubory kopírují je *sysroot*.

6) Cílová instalace

Příkaz: *ptxdist targetinstall název_balíčku*

Popis: Nainstaluje soubory určené pro cílový systém do adresáře *root*.

8.2.4 Kompilace a vytvoření obrazů

Pokud je v PTXdistu vše správně nastaveno, je možné přejít k samotné kompilaci a vytvoření obrazů. Pro kompilaci slouží příkaz *ptxdist go*. Ten zajistí kompilaci všech balíčků, které ještě nejsou zkompilovány. V případě, že se během běhu *ptxdist go* nevyskytly žádné chyby, lze vytvořit výsledné obrazy příkazem *ptxdist images*. Jaké typy obrazů budou vytvořeny a v jakých formátech závisí na výběru, který je možné učinit v konfiguraci platformy.

9 TESTOVANÉ PLATFORMY

V rámci práce byly otestovány 2 procesorové moduly, které se lišily prakticky ve všech ohledech, s výjimkou společné architektury. Pro každý modul je dále uvedena tabulka, která shrnuje základní informace o modulu. Kromě těchto 2 modulů byl ještě testován Linux na modulu Colibri T20 (Nvidia Tegra 2), desce Leopardboard (TMS320DM365) a desce Beagleboard (AM37x).

Cílem práce bylo zprovoznit na nějakém modulu všechny požadované periferie, mezi které patřily: síťová karta, SD karta, CAN sběrnice, LVDS LCD displej, hardwarová GPIO klávesnice, rezistivní dotykový displej a hardwarové dekódování videa.

Na modulu Colibri PXA320 bylo dosaženo všech požadavků s výjimkou funkční síťové karty v Linuxu a hardwarové akcelerace videa, proto se dále přešlo na modul TQMa53, u kterého výrobce deklaroval většinu požadovaných vlastností jako funkčních. Tento modul navíc má vestavěný CAN oproti modulu Colibri T20, u kterého je použit stejný externí čip, jako měl modul Colibri PXA320, a který při vyšších přenosových rychlostech trpěl chybami.

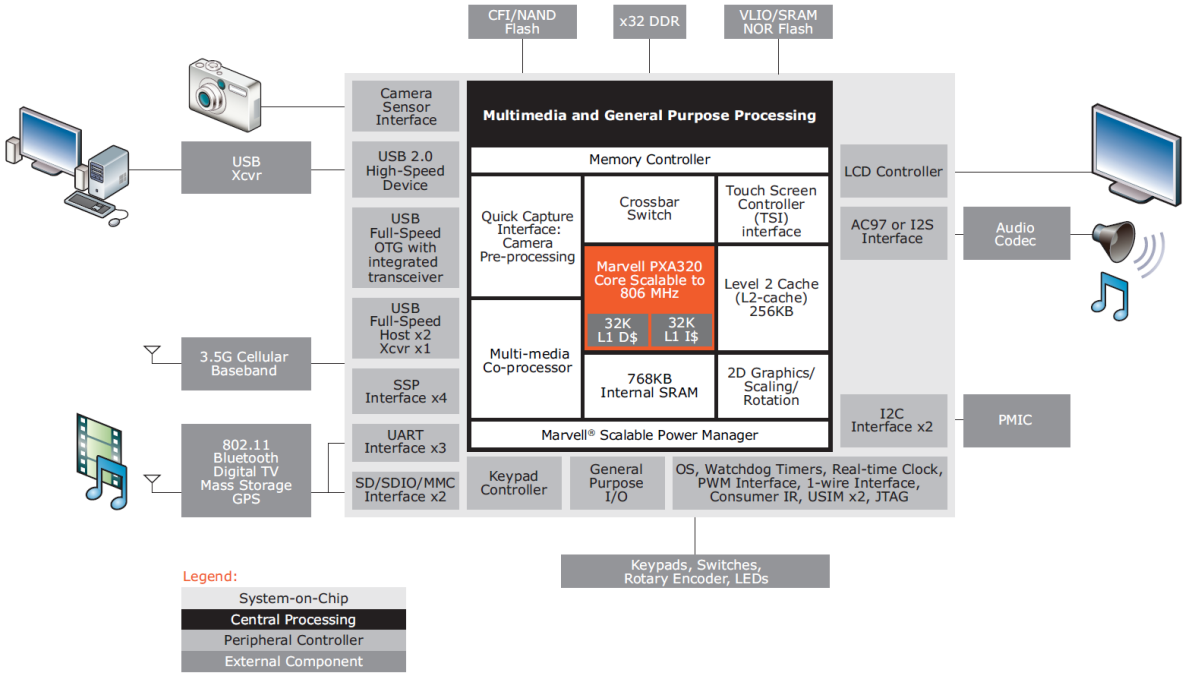
9.1 Modul Colibri PXA320

Název modulu:	Colibri PXA320
Výrobce modulu:	Toradex AG
Procesor:	Marvell PXA320
Nosná deska:	Colibri Evaluation Board a UNIS CMFD2.0
Použitý zavaděč:	U–boot
Použitý nástroj pro vytvoření systému:	Buildroot

Tabulka 4 – Základní informace o modulu Colibri PXA320

Procesor PXA320 od firmy Marvell je pokračovatelem starších procesorů řady StrongARM od Intelu. Jeho jádrem je XScale – implementace ARMv5TE architektury. Jeho sériová výroba začala již v roce 2006, ale do dnešních dnů je stále k dostání a je garantována dostupnost modulů s tímto procesorem až do roku 2017.

Maximální taktovací frekvence tohoto procesoru je 806MHz, má 256KB L2 cache a 32-bitové DDR rozhraní. Na následujícím obrázku Obrázek 11 – Diagram procesoru Marvell PXA320 jsou znázorněny rozhraní a části tohoto procesoru.

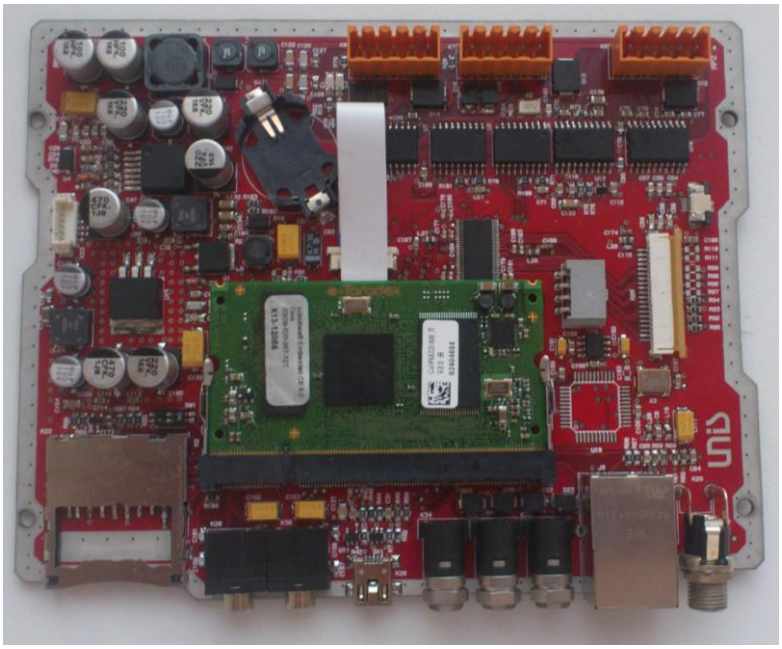


Obrázek 11 – Diagram procesoru Marvell PXA320

Pro tento modul byl použit zavaděč U-boot, protože byl jediný, který podporu tohoto modulu a nosné desky obsahoval. Jako nástroj pro tvorbu systému byl vybrán Buildroot, protože byl doporučen při zadávání práce.

Výrobce modulu dává na svém webu k dispozici upravené linuxové jádro, které je však bohužel poměrně zastaralé a podobně je na tom i poskytovaný zavaděč. Do novějších řad jádra se sice dostala podpora tohoto modulu, ale některé části (například podpora síťového čipu pro novější revize modulu) stále chybí a vzhledem ke stáří modulu a čipu je vysoce nepravděpodobné, že by byla ještě někdy přidána.

Tento modul byl testován se dvěma nosnými deskami – jednak vývojová deska od výrobce modulu, jednak speciální nosná deska vyrobená firmou UNIS. Se stejným zavaděčem modul fungoval na obou deskách, linuxové jádro však bylo nutné mít pro každou desku jiné.



Obrázek 12 – pohled na modul Colibri PXA320 na nosné desce od firmy UNIS

Závěrem bohužel nelze tento modul doporučit pro vývoj nových zařízení s Linuxem. Pro novější revize modulu není dostupná kompletní podpora v novějších linuxových jádrech ani zavaděči.

9.2 Modul TQMa53

Název modulu:	TQMa53
Výrobce modulu:	TQ Group GmbH
Procesor:	Freescale i.MX53
Nosná deska:	MBa53
Použitý zavaděč:	Barebox
Použitý nástroj pro vytvoření systému:	PTXdist

Tabulka 5 – základní informace o modulu TQMa53

Pro modul TQma53 dodává výrobce již předpřipravený balíček s podporu desky (BSP) pro PTXdist, proto nebylo nutné dělat mnoho úprav – pouze upravit výběr instalovaného softwaru a upravit DeviceTree soubory tak, aby odpovídali všemu připojenému hardwaru.

Jako nosná deska pro tento modul byla použita deska od výrobce modulu MBa53.

Nejproblémovějším požadavkem na tomto modulu se stalo hardwarové dekódování videa, pro které v době zakoupení modulu nebyla v systému podpora. Pro podporu videa na procesoru použitém v tomto modulu existují hned 3 možnosti. První z nich je oficiální ovladač od firmy Freescale. Tento ovladač však trpí několika problémy. Největším problémem je vyžadování uzavřeného firmwaru, díky čemuž tento ovladač od počátku kvůli linuxové politice neměl šanci dostat se do hlavní větve jádra, a proto funguje pouze na poměrně starém opatchovaném jádře, které výrobce poskytuje. Navíc je závislý na několika dalších softwarových částech, které sice Freescale v jisté podobě poskytuje, ale které jsou závislé na určité verzi opatchovaného linuxového jádra. Díky tomu vznikly další 2 alternativy – ovladač vyvíjený ve firmě Pengutronix, který je sice čistěji napsaný a lze ho snáze použít v aktuálním jádře, ale také je závislý na uzavřeném firmwaru a ovladač CODA, který je nakonec dostupný v hlavní řadě jádra.

Výrobce tohoto modulu – firma TQ Group GmbH – nakonec v nedávné době zařadila do nejnovějšího balíčku pro podporu desky nejnovější jádro a spolu s ním i ovladač CODA, který nyní umožňuje hardwarové dekódování videa.

Obrázek 13 – pohled na modul TQMa53 na desce MBa53

10 SOFTWAROVÉ ŘEŠENÍ NA CÍLOVÉ PLATFORMĚ

Tato kapitola shrnuje software instalovaný na cílové platformě a jeho použití.

10.1 Init systém

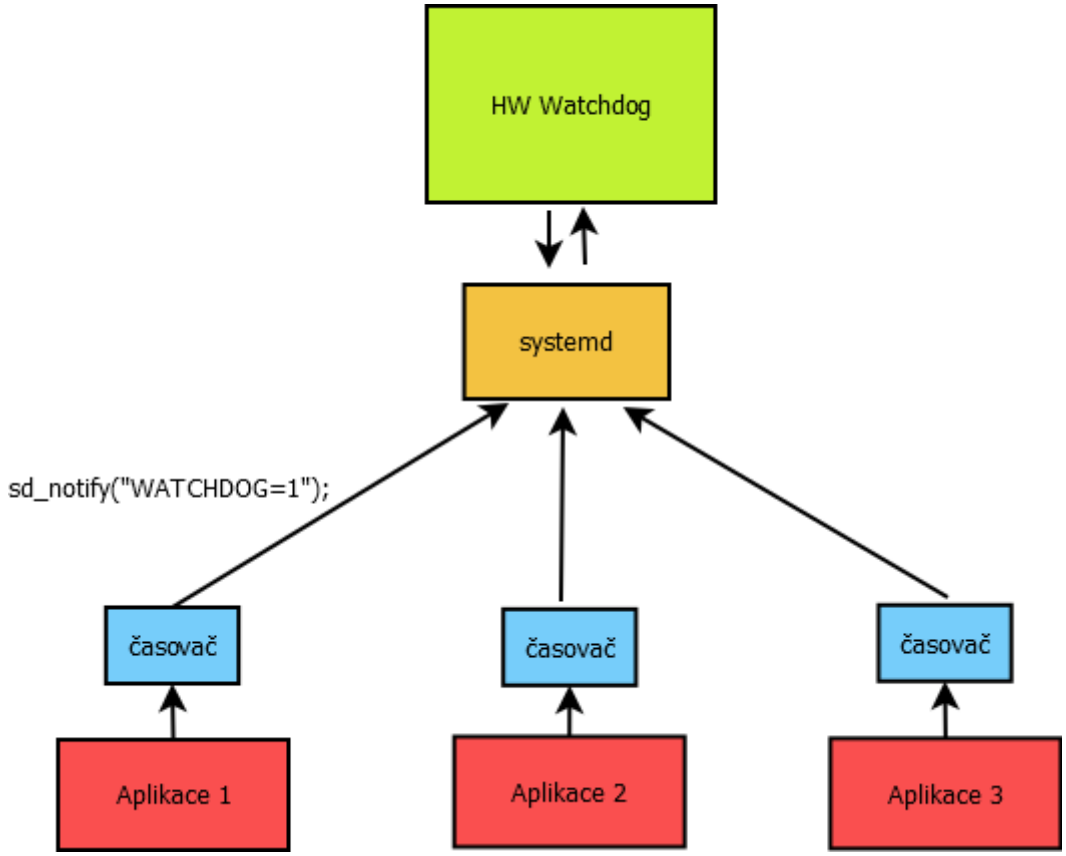
Jako init systém byl použit *systemd*, který je poměrně velkou novinkou. Důvodů k jeho použití bylo hned několik. V první řadě není založen na shellových skriptech jako starší systémy *System V init* nebo *BSD init*, ale používá deklarativně zapsané soubory, ve kterých jsou jasně zapsané zavislosti jednotlivých služeb. Mimo zrychlení způsobené tím, že není nutné interpretovat shell skripty, je tak nyní s použitím tohoto init systému celý bootovací proces možné paralelizovat bez obav, že by docházelo k uváznutí procesů.

Systemd provádí sledování procesů a umožňuje nadefinovat akce, které se mají provést v případě ukončení procesů. Služby je navíc možné spouštět prostřednictvím Socketů nebo D-Busu.

10.2 Watchdog

Použitý modul TQMa53 obsahuje hardwarový watchdog, díky kterému je možné modul zrestartovat v případě potíží. Nejjednodušší použití watchdogu spočívá v tom, že po jeho zapnutí je nutné v určitých časových intervalech zapisovat do specifického souboru. Protože je ale v rámci aplikace nutné sledovat více částí, byl zvolen postup, který zahrnuje systemd jako prostředníka pro použití watchdogu.

1. Hardwarový watchdog je v pravidelných intervalech notifikován pomocí systemd. V případě zaseknutí systemd provede systém hard reset.
2. Aplikace v pravidelných intervalech notifikuje systemd. Pokud aplikace neodešle notifikaci, systemd se ji pokusí zrestartovat a pokud se to v určitém počtu pokusů nebo čase nepodaří, restartuje systém.



Obrázek 14 – struktura použití watchdogu

10.3 Sběrnice CAN

Pro práci se sběrnicí CAN v rámci aplikace je použita knihovna *SocketCAN*. Počáteční nastavení sběrnice se provádí příkazem *canconfig* z balíčku *canutils*. Na následující ukázce je nastavení CAN portu na komunikační rychlost 100kbps.

```
ifconfig can1 down
canconfig can0 bitrate 100000 sample-point 0.8
canconfig can0 ctrlmode berror-rate on
ifconfig can0 up
echo 256 > /sys/class/net/can0/tx_queue_len
```

Zdrojový kód 13 – nastavení CAN sběrnice v Linuxu

Testování CAN sběrnice je rovněž možné provádět s pomocí dalších nástrojů z balíku canutils. Pro odeslání zprávy po sběrnici CAN je to příkaz cansend a pro přijetí candump.

10.4 Akcelerované přehrávání videa

Pro přehrání videa s hardwarovou akcelerací je nutné použít přehrávač, který má v sobě podporu pro použitý čip. Pro tuto platformu je to multimediální framework GStreamer, který je na Linuxu široce rozšířený a je využíván velkou částí linuxových audio a video přehrávačů jako backend. GStreamer je pipeline framework – jeho filozofií je dávat k dispozici elementy, ze kterých si následně uživatel sestaví celou pipeline k přehrávání. Příkladem těchto elementů je zdroj, filtr, dekodér a výstupní zařízení. Tím je zajištěna vysoká flexibilita a je možné provádět různá předzpracování, přidávat filtry a efekty, nebo zobrazovat na různá výstupní zařízení. Bohužel to ale také přidává na komplexitě a uživatel musí mít vysokou představu o tom, jak jednotlivé elementy poskládat pro funkční výstup.

GStreamer umožňuje i automatické sestavení pipeline, avšak to je závislé na prioritě jednotlivých elementů a v současné době na této platformě nefunguje. Na následujících ukázkách je několik příkladů, jak vypadá celá pipeline na platformě TQMa53.

```
gst-launch-0.11 \
                filesrc location=/root/test.mkv \
                ! matroskademux \
                ! h264parse \
                ! video/x-h264,stream-format=byte-stream,alignment=au \
                ! v4l2filter device=/dev/video/by-name/coda \
                ! v4l2sink device=/dev/video/by-name/vout
```

Zdrojový kód 14 – přehrání videa ve formátu H264 v matroska kontejneru

```
gst-launch-0.11 \
                souphttpsrc location=http://10.5.8.217/video.mjpg \
                ! multipartdemux \
                ! jpegdec \
                ! v4l2sink device=/dev/video/by-name/vout
```

Zdrojový kód 15 – přehrání videa ve formátu MJPEG z HTTP serveru

ZÁVĚR

Tato práce se zabývala implementací operačního systému Linux na platformě ARM. Hlavní částí teoretické části je popis historie a současné situace, v jaké Linux na platformě ARM je. Jako zdroj největších historických problémů byla identifikována nedůvěra k otevřenému vývojovému modelu, který Linux má, neochota do něj přispívat a nízká spolupráce mezi výrobci hardwaru. Naštěstí se tato situace postupně zlepšuje a do linuxového jádra je začleňován dále jen kód, který splňuje zpřísněné požadavky. Výrobci hardwaru dokonce udělali tak bezprecedentní krok, jako je ustavení společné společnosti Linaro, která se zabývá podporou Linuxu na ARM procesorech a odstraňováním duplicit mezi kódem různých výrobců.

Jedním ze zdrojů problémů je i fakt, že platforma ARM nemá obdobu sběrnice PCI/PCI–e jako je tomu na platformách x86/x86_64. Tento nedostatek je nyní řešen používáním speciálních DeviceTree souborů, které ve stromové struktuře popisují, jakým způsobem je hardware k procesoru připojen a s jakým nastavením.

Jako zavaděč se více osvědčil Barebox, který sice obsahuje podporu menšího množství hardwaru, ale způsob práce s ním více připomíná práci s plnohodnotným linuxovým systémem. Za jednu z jeho největších výhod oproti U-bootu lze také považovat fakt, že skripty lze ukládat do souborů a díky tomu se snadněji editují, udržují i používají a lze na první pohled rozlišit mezi standardním příkazem zavaděče a uživatelským skriptem.

Z nástrojů na vytvoření systému se subjektivně lépe pracovalo s nástrojem PTXdist. Nástroj Buildroot je naproti tomu velice vhodný pro vytvoření jednoduchého systému pro cílovou desku, ale je nutné počítat s tím, že vytvořená konfigurace je obtížněji přenositelná na jiný počítač. PTXdist umožňuje větší nastavení a je lépe připraven na práci v týmu a úpravy balíčků, daní za to mírně větší komplexita nástroje.

V práci byly testovány 2 procesorové moduly, každý s jiným nástrojem pro tvorbu systému a zavaděčem. Velmi se tam projevil rozdíl mezi starším modulem Colibri PXA320, který byl primárně určen pro provoz se systémem Windows Embedded a jehož linuxová podpora byla do jádra přidávána dobrovolníky a modulem TQMa53, který má naproti tomu linuxovou podporu garantovanou od výrobce a Linux je pro něj primární platformou.

Výsledkem práce je reálně nasaditelný systém pro modul TQMa53 a Colibri PXA320, který je dále využíván firmou UNIS.

ZÁVĚR V ANGLIČTINĚ

This thesis is about implmentation of Linux operating system on an ARM platform. One of the main parts is description of history and current state of Linux ARM port. As the biggest source of historical problems with Linux ARM port, scepticism from the side of hardware making companies was identified. That led to reluctance to contribute into Linux kernel as well as low level of cooperation. Luckily this situation improves over time and companies making hardware even made the unprecedented step of financing company focused on Linux ARM code.

One of the sources of problems is also a fact, that ARM platform in comparison with x86/x86_64 doesn't have analogy to PCI/PCI-e bus. This shortcoming is currently solved by using special DeviceTree files, which describes hardware interconnection and setup in tree structure.

The comparison of bootloaders on ARM was won by Barebox bootloader because of his cleaner interface which mimics feeling of full-blown Linux-like system. One of his greatest advantages in comparison with U-boot was its scripting system, which doesn't abuse environment variables for scripts and saves them in in file-like manner. This leads to easier orientation for user.

To make a complete system for an ARM-based board, the PTXdist tool is subjectively better. The Buildroot tool is on the other hand very easy to use - the only shortcoming is that generated and configured system is hard to move into another computer. PTXdist allows for more configurations, easier collaboration and tailoring of packages, which has slight impact on its complexity.

Two processor modules were tested, each with a different tool for generating a system and a bootloader. Comparison revealed major differences between slightly old module Colibri PXA320, which was primarily designated to work with Windows Embedded system and module TQMa53, which has Linux support guaranteed by the manufacturer and Linux is a primary platform for it.

Result of this work is apllicable system for TQMa53 and Colibri PXA320 modules, which is consequently used by the UNIS company.

SEZNAM POUŽITÉ LITERATURY

[1] VENKATESWARAN, Sreekrishnan. Essential Linux device drivers. Upper Saddle River: Prentice Hall, c2008, xxx, 714 s. ISBN 978-0-132-39655-4.

[2] CORBET, Jonathan. Linux device drivers. 3rd ed. Sebastopol: O´Reilly, 2005, xviii, 615 s. ISBN 05-960-0590-3.

[3] LOVE, Robert. Linux kernel development. 3rd ed. Upper Saddle River: Addison-Wesley, c2010, xx, 332 s. ISBN 06-723-2946-8.

[4] GRÖTKER, Thorsten. The developer's guide to debugging. 2nd ed. North Charleston, S.C.?: [CreateSpace Independent Publishing Platform], c2012, xx, 242 s. ISBN 978-1470185527.

[5] STALLMAN, Richard M a Roland MCGRATH. GNU make: a program for directing recompilation : GNU make version 3.79.1 : June, 2002. Boston: Free Software Foundation, 2002, vi, 184 s. ISBN 18-821-1482-5.

[6] Building embedded Linux systems. 2nd ed. Sebastopol: O´Reilly, 2008, xx, 439 s. ISBN 978-0-596-52968-0.

[7] QNX Neutrino RTOS. QNX [online]. ©2004–2013 [cit. 2013-04-02]. Dostupné z: http://www.qnx.com/products/neutrino–rtos/neutrino–rtos.html#overview

[8] The History of ARM Linux. The ARM Linux Project [online]. © 2013 [cit. 2013-04-02]. Dostupné z: http://www.arm.linux.org.uk/docs/history.php

[9] RUSLING, David. Kernel Upstreaming. In: Linaro Blog [online]. 2011 [cit. 2013-04-02]. Dostupné z: http://www.linaro.org/linaro-blog/2011/06/09/kernel-upstreaming/

[10] PETAZZONI, Thomas. FREE ELECTRONS. Linux kernel: consolidation in the ARM architecture support. 2012. Dostupné z: http://free-electrons.com/pub/conferences/2012/lsm/arm-kernel-consolidation/arm-kernel-consolidation.pdf

[11] Buildroot [online]. © 1999–2005 Erik Andersen, 2006-2012 The Buildroot developers [cit. 2013–04–02]. Dostupné z: http://buildroot.uclibc.org/

[12] PTXdist – Reproducible Embedded Linux Systems [online]. 2013 [cit. 2013-04-02]. Dostupné z: http://www.ptxdist.org/software/ptxdist/index_en.html

[13] Application Note – Installing PTXdist-2012.12.0. Hildesheim, 2012. Dostupné z: http://www.ptxdist.org/software/ptxdist/appnotes/AppNote_InstallingPtxdist.pdf

[14] OPDENACKER, Michael. FREE ELECTRONS. Porting U-boot. © 2004-2009. Dostupné z: http://free-electrons.com/doc/porting-u-boot.pdf

[15] HALLINAN, Christopher. Bootloaders in Embedded Linux Systems. In: Pearson Education, Informit [online]. 2010 [cit. 2013-04-02]. Dostupné z: http://www.informit.com/articles/article.aspx?p=1647051

[16] Porting U-Boot to a new board. STMICROELECTRONICS. STLinux [online]. © 2008-2013 [cit. 2013-04-02]. Dostupné z: http://www.stlinux.com/u-boot/porting

[17] RAGHUNANDAN, ES. Porting U-Boot to a New Board [online]. 2012 [cit. 2013-04-02]. Dostupné z: http://portinguboottoanewboard.blogspot.com

[18] Adapting a new Board. In: Barebox Developer's Manual [online]. 2011 [cit. 2013-04-02]. Dostupné z: http://barebox.org/documentation/barebox-2011.05.0/dev_board.html

[19] Barebox – Getting started. Elec4fun [online]. 2011 [cit. 2013-04-02]. Dostupné z: http://www.elec4fun.fr/2011-03-30-10-16-30/2011-03-31-13-08-45/startwithbarebox

[20] Barebox. In: Crash Course wiki [online]. 2012 [cit. 2013-04-02]. Dostupné z: http://www.crashcourse.ca/wiki/index.php/Barebox

[21] OSELAS®.Toolchain(). In: PTXdist [online]. 2013 [cit. 2013-04-02]. Dostupné z: http://www.ptxdist.de/oselas/toolchain/index_en.html

[22] PENGUTRONIX. How to become a PTXdist Guru. Hildesheim, 2012. Dostupné z: http://www.pengutronix.de/software/ptxdist/appnotes/OSELAS.BSP-Pengutronix-Generic-arm-Quickstart.pdf

[23] DONGWOOK, Kang. ETRI. Snapshot Booting on Embedded Linux. Daejeon, 2011. Dostupné z: http://elinux.org/images/c/c3/Elc2011_kang.pdf

[24] Suspend to Disk for ARM. In: Embedded Linux Wiki [online]. 2011 [cit. 2013-05-08]. Dostupné z: http://elinux.org/Suspend_To_Disk_For_ARM

[25] QuickBoot. Qbiquitous [online]. © 2013 [cit. 2013-05-08]. Dostupné z: http://www.ubiquitous.co.jp/En/products/middleware/quickboot/

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

API Application Programming Interface

ARM Advanced RISC Machine

BSP Board Support Package

CAN Controller Area Network

CLI Command Line Interface

DTB Device Tree Blob

DTC Device Tree Compiler

DTS Device Tree Source

ELDK Embedded Linux Developmnet Kit

FDT Flattened Device Tree

GCC GNU Compiler Collection

GDB GNU Debugger

GPIO General Purpose Input/Output

LCD Liquid Crystal Display

LVDS Low-voltage differential signaling

OE OpenEmbedded

RTOS Real-Time Operating Systém

SoC System-on-Chip

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM ZDROJOVÝCH KÓDŮ

SEZNAM PŘÍLOH

PŘÍLOHA P I: PŘÍKAZY U-BOOTU

Příkazy U–bootu jsou rozděleny do kategorií podle funkce a pro každou desku lze v konfiguračním souboru definovat, které z příkazů U–Bootu budou dostupné. Z těchto příkazů je navíc možné skládat další příkazy, které se poté spouští příkazem run.

Informační příkazy

bdinfo – vypíše obsah struktury Board Info

coninfo – vypíše dostupná konzolá zařízení a informace o

flinfo – vypíše informace o flash paměti

iminfo – vypíše hlavičku aplikačního obrazu

help – zobrazí nápovědu

Příkazy pro manipulaci s pamětí

base – vypsání nebo nastavení offsetu adresy

crc32 – výpočet kontrolního součtu

cmp – porovnání obsahu paměti

cp – kopírování obsahu paměti

md – zobrazení obsahu paměti

mm – změna obsahu paměti (automatická inkrementace)

mtest – jednoduchý test RAM paměti

mw – zápis do paměti (naplnění)

nm – změna obsahu paměti (konstantní adresa)

loop – nekonečná smyčka na rozsahu adres

Příkazy pro práci s Flash pamětí

erase – vymazání Flash paměti

protect – povolení nebo zakázání zámku pro zápis do Flash paměti

mtdparts – definování MTD oddílů kompatibilních s Linuxem

Příkazy pro spouštění programů

source – spuštění skriptu z paměti

bootm – spuštění aplikačního obrazu z paměti

go – spuštění aplikace z adresy

Příkazy pro práci se sítí a sériovou linkou

bootp – spuštění obrazu ze sítě s použitím protokolu BOOTP/TFTP

dhcp – spuštění DHCP klienta pro získání IP adresy/bootovacích parametrů

loadb – načtění binárního souboru přes sériovou linku (kermit)

loads – načtení S–Record souboru přes sériovou linku

rarpboot – spuštění obrazu ze sítě s použitím protokolu RARP/TFTP

tftpboot – spuštění obrazu ze sítě s použitím protokolu TFTP

Příkazy pro manipulaci s proměnnými prostředí

printenv – vypíše proměnné prostředí

saveenv – uloží proměnné prostředí do permanentní paměti

setenv – nastaví proměnno u prostředí

run – spustí příkaz z proměnné prostředí

bootd – spuštění výchozího obrazu

Příkazy pro práci s Flattened Device Tree

fdt addr – výběr FDT se kterým se bude pracovat

fdt list – výpis jedné úrovně

fdt print – rekurzivní výpis

fdt mknode – vytvoření nového uzlu

fdt set – nastavení vlastností uzlu

fdt rm – odstranění uzlu nebo jeho vlastností

fdt move – přesunutí FDT blobu na novou adresu

fdt chosen – opravení dynamických informací

Speciální příkazy

i2c – I2C subsystém

Ostatní příkazy

echo – výpis argumentů do konzole

reset – provedení resetu CPU

sleep – pozastavení běhu na určitou dobu

version – vypíše verzi monitoru

? – alias k příkazu 'help'

PŘÍLOHA P II: PŘÍKAZY BAREBOXU

Jelikož je Barebox forkem U-bootu, některé příkazy jsou shodné, nicméně Barebox se snaží chovat co nejvíc podobně Linuxu, proto je jako v Linuxu podobný systém adresářů a příkazy.

Standardní příkazy

clear – vyčištění obrazovky
cat – zobrazí obsah souboru
cd – změna aktuálního adresáře
mkdir – vytvoření adresáře
cp – zkopírování souboru
edit – otevře editor souboru
pwd – vypíše aktuální pracovní adresář
mount – připojení oddílu
ls – vypsání obsahu adreáře
rm – smazání souboru
rmdir – smazání adresáře
time – vypsání aktuálního času
umount – odpojení oddílu
global – vytvoření globální proměnné
export – exportuje proměnnou
false
exec – spuštění skriptu
insmod – načtení modulu
passwd – nastavení hesla k zavaděči
sh – spustí skript
true
test – příkaz pro porovnání hodnot
lsmod – výpis načtených modulů

Příkazy pro práci s GPIO a LED

gpio_direction_input – nastaví GPIO pin jako vstupní

gpio_direction_output – nastaví GPIO pin jako výstupní

gpio_get_value – získání hodnoty GPIO pinu

gpio_set_value – nastavení hodnoty GPIO pinu

led – nastavení hodnoty LED

trigger – přiřazení triggeru k LED

Informační příkazy

cpuinfo – zobrazí informace o CPU (specifické pro ARM)

devinfo – zobrazí informace o známých zařízeních a ovladačích

reginfo – zobrazí informace o některých registrech (specifické pro mpc5200)

iomem – zobrazí informace o I/O prostředcích a jejich využití

help – zobrazí nápovědu

Příkazy pro práci s UBI oddíly

ubiattach – připojení zařízení do UBI

ubimkvol – vytvoření UBI oddílu

ubirmvol – smazání UBI oddílu

Příkazy pro spouštění programů

source – spuštění skriptu z paměti

bootm – spuštění aplikačního obrazu z paměti

go – spuštění aplikace z adresy

linux16 – spuštění obrazu linuxu (pouze pro x86 platformu)

Příkazy pro manipulaci s pamětí

crc – výpočet kontrolního součtu

nand – manipulace s vadnými bloky v NAND paměti

protect – zakáže zápis do Flash paměti

unprotect – povolí zápis do flash paměti

md – zobrazí obsah paměti

erase – vymazání flash paměti

addpart – vytvoření oddílů na zařízení nebo souboru

delpart – odstranění oddílů na zařízení nebo souboru

memcmp – porovná obsah paměti

memcpy – zkopíruje obsah paměti

meminfo – zobrazí informace o paměti

mw – zapsání hodnoty do paměti nebo souboru

Příkazy pro manipulaci s proměnnými prostředí

printenv – vypíše proměnné prostředí

saveenv – uloží proměnné prostředí do permanentní paměti

setenv – nastaví proměnno u prostředí

loadenv – načtění proměnných prostředí z trvalého úložiště

magicvar – vypíše proměnné prostředí se speciálním významem a jejich význam

Příkazy pro práci se sítí a sériovou linkou

tftp – přenesení souboru přes tftp protokol

ping – ping síťového zařízení

nfs – načtení souboru z NFS serveru

dfu – spustí update firmwaru s pomocí DFU (Device Firmware Update) protokolu

Ostatní příkazy

echo – výpis argumentů do konzole

reset – provedení resetu CPU

sleep – pozastavení běhu na určitou dobu

version – vypíše verzi monitoru

usb – (znovu)detekuje USB zařízení

bmp – zobrazení bmp obrázku do framebufferu

automount – automaticke připojení adresáře při prvním přístupu

readline – načtení řádku z konzole

timeout – počkání jestli nastane timeout (lze přerušit vstupem uživatele)

unlzo – Dekomprimuje komprimovaný LZO soubor

Příkazy pro práci s Flattened Device Tree

oftree – načtení, dupování a uvolnění DeviceTree blobů

of_node – vytváření a mazání uzlů v DeviceTree

of_property – vytváření, modifikace a mazání vlastností DeviceTree uzlu

PŘÍLOHA P III: SKRIPT PRO PRÁCI S BUILDROOTEM

```
#!/bin/bash

# This script downloads, configures and builds everything needed to
# support linux on PXA320 based board
# ==> Toolchain, Linux kernel, Buildroot and U-boot
# Specifically made for project UNIS CMFD2
# Author: Roman "Format" Dosek <formatsh@gmail.com>

# Prepare script options
ENABLE_AUTOMATIC=0
ENABLE_PREPARE=0
ENABLE_CONFIGURE=0
ENABLE_BUILD=0
ENABLE_DOWNLOAD=0
while getopts adp:c:b: option
do
        case "${option}"
        in
            a) ENABLE_AUTOMATIC=1;;
            d) ENABLE_DOWNLOAD=1;;
            f) ENABLE_PREPARE=1 && PREPARE_OPTIONS=${OPTARG};;
                c) ENABLE_CONFIGURE=1 && CONFIGURE_OPTIONS=${OPTARG};;
            b) ENABLE_BUILD=1 && BUILD_OPTIONS=${OPTARG};;
        esac
done

# ************************** Functions ****************************

# Colorized output
printr() {
      echo $(tput bold) $(tput setaf 2) $* $(tput sgr0)
}

printenv() {
      echo $(tput bold) $(tput setaf 2) $1 $(tput setaf 1) $2 $(tput
sgr0)
}


# Global variables setup
# These variables are rerquired for relocateable configs
# which also means that everything that uses them HAS TO be built by
# this script or these variables has to be set by hand
setupGlobalVariables() {
      printr "Setting up global variables....."

      # These variable has to be global, because they ARE referenced from
      # other scripts
      export FIRE_PROJNAME=unis_cmfd20
      export FIRE_ARCH=arm
      export FIRE_LIBABI=uclibcgnueabi
      export FIRE_TOOLCHAIN=$FIRE_ARCH-$FIRE_PROJNAME-linux-$FIRE_LIBABI
      export FIRE_LINUX_TAG=v3.3

      export FIRE_DIR=/home/unis_fire
      export FIRE_CONFIGS=$FIRE_DIR/configs
      export FIRE_BUILD_TOOLCHAIN=$FIRE_DIR/toolchain-build
      export FIRE_TOOLCHAINDIR=$FIRE_DIR/toolchains/$FIRE_TOOLCHAIN
      export FIRE_KERNEL=$FIRE_DIR/kernel/linux-$FIRE_LINUX_TAG
      export FIRE_PATCHES_GENERAL=$FIRE_DIR/patches/general
      export FIRE_PATCHES_KERNEL=$FIRE_DIR/patches/kernel_old
```

```
	export FIRE_BUILDROOT=$FIRE_DIR/buildroot
	export FIRE_SKELETON=$FIRE_DIR/skeleton
	export FIRE_UBOOT_BRANCH=openpxa_new
	export FIRE_UBOOT_SPL=n

	#Download variables
	export FIRE_DOWNLOAD_DIR=$FIRE_DIR/tarballs

	#Generic compilation variables
	export ARCH=arm
	export PATH=$FIRE_TOOLCHAINDIR/bin:$PATH
	export CROSS_COMPILE=$FIRE_TOOLCHAIN"-"

	printr "Variables set as: "
	printr "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
	printenv "Project name" $FIRE_PROJNAME
	printenv "Root directory:" $FIRE_DIR
	printenv "Toolchain:" $FIRE_TOOLCHAIN
	printenv "Dir containing all configs:" $FIRE_CONFIGS
	printenv "Dir containing kernel sources:" $FIRE_KERNEL
	printenv "Patches for kernel:" $FIRE_PATCHES_KERNEL
	printenv "Other patches:" $FIRE_PATCHES_GENERAL
	printenv "Buildroot dir:" $FIRE_BUILDROOT

	printenv "Architecture " $ARCH
	printenv "Current PATH settings: " $PATH
	printenv "Cross compilation: " $CROSS_COMPILE
	printr "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
}

# Download All
download() {
	mkdir $FIRE_DOWNLOAD_DIR
	cd $FIRE_DOWNLOAD_DIR
	if [ "$(ls -A)" ]; then
		printr "Download dir is NOT empty, continuing may replace
some existing file."
		printr "Do you wish to proceed?"
		select yn in "Yes" "No"; do
			case $yn in
				Yes ) printr "OK..proceeding" ;break;;
				No ) printr "Terminated..." ;exit;;
			esac
		done
	fi

	wget http://crosstool-ng.org/download/crosstool-ng/crosstool-ng-
1.16.0.tar.bz2
	wget ftp://ftp.denx.de/pub/u-boot/u-boot-2010.09.tar.bz2
	wget http://buildroot.net/downloads/buildroot-2012.08.tar.gz
	wget http://www.kernel.org/pub/linux/kernel/v3.0/linux-3.3.tar.gz
}


# Prepare crosstool-ng
prepareToolchain() {
	mkdir $FIRE_DIR/crosstool-build
	cd $FIRE_DIR/crosstool-build
	tar xjf $FIRE_DOWNLOAD_DIR/crosstool-ng-1.16.0.tar.bz2 .
	cd crosstool-ng-1.16.0
	./configure
	make
	make install
```

```
	cd $FIRE_DIR
	rm -rf $FIRE_DIR/crosstool-build
}

# Prepare U-boot
prepareUboot() {
	mkdir $FIRE_DIR/uboot
	cd $FIRE_DIR/uboot
	tar$FIRE_DOWNLOAD_DIR/ xjf u-boot-2010.09.tar.bz2
	mv u-boot-2010.09/* .
}

# Prepare Buildroot
prepareBuildroot() {
	mkdir $FIRE_DIR/buildroot
	cd $FIRE_DIR/buildroot
	tar xzf $FIRE_DOWNLOAD_DIR/buildroot-2012.08.tar.gz .
}

# Prepare kernel
prepareKernel() {
	mkdir $FIRE_DIR/kernel
	cd $FIRE_DIR/kernel

	tar xzf $FIRE_DOWNLOAD_DIR/linux-3.3.tar.gz .
	cd linux-3.3

	#Auto kernel patch
	for filename in $FIRE_PATCHES_KERNEL/*
	do
		echo $filename
		patch -p1 < $filename
	done;
}

# Build Toolchain
buildToolchain() {
	printr "Building tooolchain......"
	printr "Creating toolchain build dir....."
	mkdir $FIRE_BUILD_TOOLCHAIN
	cd $FIRE_BUILD_TOOLCHAIN

	printr "Build started....."
	#ct-ng clean
	ct-ng distclean
	rm -rf .build/src/linux-custom/linux-3.6.4

	printr "Copying toolchain config....."
	cp $FIRE_CONFIGS/crosstool-ng.config $FIRE_BUILD_TOOLCHAIN/.config

	#Try to build
	ct-ng build
	#Now apply patch which fixes uclibc
	patch -p1 < $FIRE_PATCHES_GENERAL/uclibc_patch.diff
	#And build again
	rm -rf .build/src/linux-custom/linux-3.6.4
	ct-ng build
	printr "Toolchain built into dir:" $FIRE_TOOLCHAIN
}

# Build U-boot
buildUboot() {
	printr "Building U-boot"
	cd $FIRE_DIR/uboot
```

```
	printr "Applying Toradex patches"
	patch -p1 < $FIRE_PATCHES_GENERAL/u-boot-2010.09-toradex.patch
	printr "Applying custom patches"
	#nothing yet
	printr "Preparing main image"
	printr distclean
	printr clean mrproper
	CROSS_COMPILE=$FIRE_TOOLCHAIN"-" make colibri_pxa320_config
	CROSS_COMPILE=$FIRE_TOOLCHAIN"-" make
	printr "Preparing NAND SPL"
	CROSS_COMPILE=$FIRE_TOOLCHAIN"-" make colibri_pxa320_nand_config
	CROSS_COMPILE=$FIRE_TOOLCHAIN"-" make
}

# Build Buildroot
buildBuildroot() {
	printr "Building buildroot"

	cd $FIRE_BUILDROOT

	#Clean All
	make clean

	#Create default config
	make colibri_cmfd20_defconfig

	printr "Build started......"
	make all

	#Patch hexsynatax
	patch -p1 < $FIRE_PATCHES_GENERAL/hexsyntax_patch.diff
	make all
}

# Build Linux Kernel
buildKernel() {
	printr "Building kernel..."
	cd $FIRE_KERNEL
	make clean mrproper
	make colibri_cmfd20_defconfig
	make uImage
	printr "Kernel built"
}

# Configure Linux Kernel
configureKernel() {
	printr "Configure kernel..."
	cd $FIRE_KERNEL
	#make clean mrproper
	make colibri_cmfd20_defconfig
	make menuconfig && make savedefconfig && cp defconfig
arch/arm/configs/colibri_cmfd20_defconfig && cp defconfig
$FIRE_CONFIGS/colibri_cmfd20_defconfig
	printr "Kernel configuration finished, config written to
$FIRE_CONFIGS/colibri_cmfd20_defconfig"
}

# Configure Toolchain
configureToolchain() {
	printr "Configure toolchain..."
	cd $FIRE_BUILD_TOOLCHAIN
	#ct-ng clean
	cp $FIRE_CONFIGS/crosstool-ng.config $FIRE_BUILD_TOOLCHAIN/.config
```

```
	ct-ng menuconfig && cp $FIRE_BUILD_TOOLCHAIN/.config
$FIRE_CONFIGS/crosstool-ng.config
	printr "Toolchain configuration finished, config written to
$FIRE_CONFIGS/crosstool-ng.config"
}

# Configure Buildroot
configureBuildroot() {
	printr "Configure buildroot..."
	cd $FIRE_BUILDROOT
	#make clean
	make colibri_cmfd20_defconfig
	make menuconfig && make savedefconfig && cp defconfig
configs/colibri_cmfd20_defconfig
	printr "Buildroot configuration finished, defconfig saved back to
$FIRE_BUILDROOT/configs/colibri_cmfd20_defconfig"
}

# ************************** Script Logic ***************************

if [  $ENABLE_AUTOMATIC -eq 0 -a\
	$ENABLE_DOWNLOAD -eq 0 -a\
	$ENABLE_PREPARE -eq 0 -a\
	$ENABLE_CONFIGURE -eq 0 -a\
	$ENABLE_BUILD -eq 0 ]; then
		printr "Usage: ./firebuild [options]"
		printr "-a Fully automatic mode which downloads, prepares,
configures and builds everything"
		printr "Best suited for first use of this script"
		printr ""
		printr "-p [all kernel toolchain buildroot uboot] - prepares
sources for selected part(s)"
		printr ""
		printr "-b [all kernel toolchain buildroot uboot] - run build
for selected part(s)"
		printr ""
		printr "-c [all kernel toolchain buildroot] - run
configurations for selected part(s)"
		printr "   This will rewrite previous config/defconfig, use
with caution"
else
	setupGlobalVariables

	# Automatic build - Just do it all
	if [ $ENABLE_AUTOMATIC -eq 1 ]; then

		# First download all needed packages
		download

		# Prepare packages for configuration and build
		# That means uncompressing donloaded tarballs into respective
		# directories and applying patches
		prepareToolchain
		prepareKernel
		prepareBuildroot
		prepareUboot

		# Now invoke configuration of each packages to allow user to
		# tune options
		configureToolchain
		configureKernel
		configureBuildroot

		#And build packages
		buildToolchain
```

```
            buildKernel
            buildBuildroot
            buildUboot

      fi

      # Download
      if [ $ENABLE_DOWNLOAD -eq 1 ]; then
            download
      fi

      # Preparations
      if [ $ENABLE_PREPARE -eq 1 ]; then
            case "${PREPARE_OPTIONS[@]}" in  *"all"*) prepareKernel &&
prepareToolchain && prepareBuildroot && prepareUboot;; esac
            case "${PREPARE_OPTIONS[@]}" in  *"kernel"*) prepareKernel ;;
esac
            case "${PREAPRE_OPTIONS[@]}" in  *"toolchain"*)
prepareToolchain ;; esac
            case "${PREPARE_OPTIONS[@]}" in  *"buildroot"*)
prepareBuildroot ;; esac
            case "${PREPARE_OPTIONS[@]}" in  *"uboot"*) prepareUboot ;;
esac
        fi

      # Build
      if [ $ENABLE_BUILD -eq 1 ]; then
            case "${BUILD_OPTIONS[@]}" in  *"all"*) buildKernel &&
buildToolchain && buildBuildroot  && buildUboot ;; esac
            case "${BUILD_OPTIONS[@]}" in  *"kernel"*) buildKernel ;;
esac
            case "${BUILD_OPTIONS[@]}" in  *"toolchain"*) buildToolchain
;; esac
            case "${BUILD_OPTIONS[@]}" in  *"buildroot"*) buildBuildroot
;; esac
            case "${BUILD_OPTIONS[@]}" in  *"uboot"*) buildUboot ;; esac
      fi

      # Configuration
      if [ $ENABLE_CONFIGURE -eq 1 ]; then
            case "${CONFIGURE_OPTIONS[@]}" in  *"all"*) configureKernel
&& configureToolchain && configureBuildroot ;; esac
            case "${CONFIGURE_OPTIONS[@]}" in  *"kernel"*)
configureKernel ;; esac
            case "${CONFIGURE_OPTIONS[@]}" in  *"toolchain"*)
configureToolchain ;; esac
            case "${CONFIGURE_OPTIONS[@]}" in  *"buildroot"*)
configureBuildroot ;; esac
      fi
fi
```