Fakulta elektrotechnická

Katedra aplikované elektroniky a telekomunikací

Bakalářská práce

Linux v embedded aplikacích

Autor práce: Petr Hodina

Vedoucí práce: Ing. Petr Weissar, Ph.D.

Plzeň 2013

Abstrakt

Tato práce se zabývá sestavením funkčního embedded linuxového operačního systému pro embedded zařízení. V úvodní části jsou rozebrány základní informace o embedded systémech a o embedded operačních systémech s jádrem Linux. Následuje přehled dostupných embedded zařízení na trhu. Dále jsou popsány jednotlivé komponenty embedded linuxového systému. Další část se zabývá popisem sestavení embedded linuxového operačního systému. Tato část obsahuje dva postupy. První popisuje manuální sestavení krok za krokem. Druhý postup využívá nástroj Bitbake a metadata z Yocto/OpenEmbedded. Pro sestavený embedded linuxový operační systém je v poslední části práce ukázka programu pro uživatelský a jaderný prostor.

Klíčová slova

Embedded systémy, jádro Linux, Yocto/OpenEmbedded, Raspberry Pi

Abstract

Hodina, Petr. Linux in embedded applications [Linux v embedded aplikacích]. Pilsen, 2013. Bachelor thesis (Bakalářská práce). University of West Bohemia. Faculty of Electrical Engineering. Department of Applied Electronics and Telecommunications. Supervisor: Ing. Petr Weissar, Ph.D.

The content of this thesis is about building fully functioning embedded linux operating system for embedded devices. Firstly are presented basic information about embedded systems and embedded linux operating systems. These are followed by a list of available embedded devices on the market. Then each component of embedded linux operating system is described. Next several chapters describe the process of building each component of embedded linux operating system. There are two ways to accomplish that. First describes the entire process by building it manualy step by step. The other way uses a tool called Bitbake and metadata from Yocto/OpenEmbedded. Last chapters show how to build basic program for user and kernel space.

Keywords

Embedded systems, Linux kernel, Yocto/OpenEmbedded, Raspberry Pi

Prohlášení

Předkládám tímto k posouzení a obhajobě bakalářskou práci, zpracovanou na závěr studia na Fakultě elektrotechnické Západočeské univerzity v Plzni.

Prohlašuji, že jsem svou závěrečnou práci vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 270 trestního zákona č. 40/2009 Sb.

Také prohlašuji, že veškerý software, použitý při řešení této diplomové práce, je legální.

V Plzni dne 19. února 2013

Petr Hodina

.....

Podpis

Obsah

Seznam zkratek a pojmů

Embedded systém …..... Vestavěný systém.

Firmware …..... Software zařízení.

Operační systém …..... Složitý firmware.

Open-source software …..... Software s otevřeným zdrojovým kódem.

Jádro operačního systému …..... Program zprostředkující interakce mezi hardwarem a uživatelskými aplikacemi.

Linux …..... Open-source jádro operačního systému.

Distribuce GNU/Linux …..... Operační systém s jádrem Linux a GNU programy.

SRAM …..... Statická paměť RAM.

DRAM …..... Dynamická paměť RAM.

CPU …..... Procesor.

GPIO …..... General Purpose Input/Output. Vstupy/výstupy, které lze programově měnit.

GPU …..... Grafický procesor.

SoC …..... System on Chip. Systém na čipu.

SoM …..... System on Module. Systém na modulu.

Bootloader …..... První vykonávaný program po startu.

Busybox …..... Programové řešení zejména pro embedded systémy.

TFTP …..... Trivial File Transfer Protocol. Jednoduchý protokol pro přenos souborů.

ELF …..... Executable and Linkable Format. Formát spustitelných a objektových souborů.

API …..... Application Programming Interface. Rozhraní pro programování aplikací.

Nativní verze nástrojů …..... Nástroje pro stejnou cílovou architekturu procesoru.

Křížové verze nástrojů …..... Nástroje pro odlišnou cílovou architekturu procesoru.

NFS …..... Network File System. Síťový souborový systém.

Debuggování aplikace …..... Ladění aplikace.

1. Úvod

Vliv elektroniky je patrný v mnoha oblastech lidského života. Dnes ji lze nalézt i v oblastech, kde by ji před 20 až 30 lety téměř nikdo nečekal - v kancelářích, vozidlech, telekomunikacích, domácnostech. Konkrétnějšími příklady jsou pak zařízení typu mobilních telefonů, fotoaparátů, termostatů, hodinek, zabezpečení domácnosti, modemů, monitorovacích systémů lidského zdraví, praček nebo kotlů na vytápění.

Tento boom elektroniky je spojený především s rychlým rozvojem oblasti integrovaných obvodů, číslicových systémů a miniaturizace. Velkou roli zde hrála také cena, která se v posledních dekádách postupně snižovala díky inovacím ve výrobě. Tyto faktory tak zapříčinily, že různé elektronické výrobky dnes zaplavily trh.

Tento vývoj měl pozitivní dopad také na embedded systémy. Jedná se o jednoúčelový vestavěný systém, který ovládá větší mechanický nebo elektrický systém, velmi často s real-time požadavky. Centrálním procesorem je mikropočítač nebo digitální signálový procesor. [5]

Embedded systémy se od klasických stolních počítačů liší tím, že na rozdíl od flexibility a širokého rozsahu požadavků uživatelů je kladen důraz na opakované provádění předem definovaného úkonu mnohdy bez nutnosti zásahu uživatele. Uživatel může měnit již zabudovanou funkcionalitu zařízení, avšak nemůže funkcionalitu přidat či odebrat změnami softwaru. To je možné jen tehdy, pokud výrobce nabízí upgrade software zařízení, tzv. firmware. Ten je uložený v nevolativní paměti (dnes typicky FLASH) a je vydán jen při nějaké závažné chybě v systému. U některých zařízení tohoto aspektu lze využít a vyměnit firmware pro změnu funkcionality bez nutnosti měnit či nakupovat nový hardware. Náhradou dedikovaného hardwaru embedded systémem lze snáze ochránit zařízení před nezákonnou replikací, neboť takové zařízení je těžší replikovat než pouhou čistě hardwarovou formu především díky firmwaru, který dané zařízení ovládá. [5]

V raných dobách embedded systémů byl firmware téměř vždy výhradně proprietární záležitost, do jehož vývoje byly vloženy nemalé finanční investice spolu s časovou náročností. Nahlédnutí do zdrojových kódů firmwaru zařízení bylo tehdy téměř nemožné pro kohokoliv mimo zaměstnance dané vývojové společnosti. Při použití je též nutné dodržovat ustanovené licence a případně za ně i platit.

Velkou roli v rozšíření open-source operačních systémů, složitý firmware, [2] měl Linus Thorvalds, který zahájil vývoj jádra, následně pojmenovaného Linux, a též pozdější koordinaci projektu. Díky použité licenci bylo možné software volně šířit, ale i dále upravovat a distribuovat. Spolu s dalším

open-source softwarem dnes tvoří základ všech distribucí GNU/Linux. Distribuce jsou vytvářeny proto, aby uživatel nemusel sám jádro a další potřebný doplňující software skládat do funkčního celku. Velkou výhodou je jejich diverzita, neboť každá distribuce má jiné přednosti a také škálovatelnost, tedy možnost konfigurace. Rozsah nasazení operačních systémů s linuxovým jádrem je od mobilních telefonů, routerů, přes stolní počítače a servery až k superpočítačům. Díky podpoře velkých nadnárodních firemních subjektů, jejichž seznam lze najít na Linux Foundation, a jednotlivců typu Linus Thorvalds, najdeme tento operační systém i na velmi netradičních zařízeních, kde třeba i nahradil původní firmware zařízení. [6]

2. Cíle práce

Účelem této práce je podrobně popsat a porozumět procesu sestavení operačního systému spolu s nastavením jednotlivých komponent na jednom již existujícím embedded zařízení. Důvodem vzniku této práce je ověření dostupných metod pro vytvoření linuxového operačního systému na embedded zařízeních a sestavení uceleného postupu, který lze následně v budoucnosti použít. Jednotlivé cíle této práce jsou:

- Zmapování dostupných zařízení na trhu použitelných pro embedded aplikace s jádrem operačního systému Linux
- Následně na příkladu navrhnout a prakticky ukázat postup tvorby operačního systému pro jedno vybrané embedded zařízení, též jeho následnou instalaci a ověření funkcionality
- Na několika jednoduchých názorných příkladech ukázat tvorbu základních programů pro cílové zařízení

Veškeré praktické ukázky byly prováděny pod operačním systémem GNU/Linux. Pro jednoduchost a snadnou replikaci byl zvolen operační systém Ubuntu ve verzi 12.04 LTS s jádrem Linux.

3. Teoretický rozbor

3.1 Charakteristika embedded systémů

Každý embedded systém se skládá z procesoru, paměti a periferních obvodů. [3] Jádrem embedded systému je procesor. Jde o hardware, který vykonává softwarové instrukce a zároveň ovládá aktivity dalších obvodů k němu připojených. Hlavním kritériem pro výběr je potřebný výpočetní výkon, který musí dostačovat pro danou aplikaci.

Druhým nezbytným prvkem je paměť. Těch je v embedded systémech na rozdíl od běžného stolního počítače několik druhů. Zároveň též záleží na architektuře, zda jde o Harvardskou či von Neumannovu architekturu, která má společný programový i datový prostor. Proměnná data jsou obvykle uložena v paměti RAM, která je typu SRAM či DRAM. [3]

Paměť typu SRAM je rychlejší, avšak menší než paměť typu DRAM. Využívá se především při bootování, pro nahrání druhého stupně bootloaderu, a pak následně jako cache paměť. Ta však vyžaduje složitý řadič, který se stará o správnou funkci, neboť informace je v paměti uložena v buňce ve formě náboje na kondenzátoru, který se postupem času vybíjí. [3]

Program je buďto uložen na odebíratelném médiu typu SD karty či FLASH disku nebo v interní paměti typu FLASH. Mimo tyto typy se lze ještě setkat s pamětí typu EEPROM, která se ve většině případů používá pro uložení konfiguračních dat. [3]

Periferní obvody slouží k interakci a komunikaci s okolním světem. Digitální vstupy a výstupy (I/O) jsou externí piny procesoru, jejichž výstupním stavem může být logická 1 nebo logická 0. Při čtení mohou naopak číst logickou hodnotu, která opět může odpovídat logické 1 nebo 0. Lze je adresovat jednotlivě nebo skupinově, pak o nich hovoříme jako o paralelních portech. [3]

Sériová rozhraní se používají pro komunikaci, která je definována protokolem, který určuje platná operace na sériové lince. Pro čtení analogových signálů se používají analogově číslicové převodníky, které převádějí analogovou hodnotu na číslicovou. Pro zpětnou konverzi se používají číslicově analogové převodníky, které převedou číslicovou hodnotu na analogovou. Mezi další periferní obvody patří různé řadiče (např. klávesnice, LCD displeje), které slouží pro specifické účely a jejichž popis lze najít v katalogovém listu. [3]

3.2 Rozdělení embedded systémů

Existují dva typy rozdělení embedded systémů [4]:

- podle oblasti použití (např. v letectví, vesmíru, domácích spotřebičích)
- podle vlastností embedded systému

Na základě vlastností lze stanovit 4 kritéria dalšího dělení:

- velikost operační paměti dat a trvalého úložiště programu
- schopnost systému pracovat v reálném čase
- síťová konektivita
- uživatelské interakce

Z hlediska velikosti operační paměti dat a trvalého úložiště programu je v oblasti embedded linuxových systémů rozlišována trojice systémů: malé (do 2MB ROM, do 4 MB RAM), střední (do 32 MB ROM, do 64 MB RAM) a velké (od 32 MB ROM, od 64 MB RAM). Je nutno si uvědomit, že tato rozmezí nejsou pevně stanovena, avšak předurčují použití systému v praxi. Je jednodušší předimenzovat systémové zdroje než opačně, samozřejmě za cenu vyšších výrobních nákladů. [4]

Většina zařízení orientovaných na zákazníky spadá do kategorie středních systémů. Jde především o PDA zařízení, MP3 přehrávače, síťové systémy typu router a podobně. Velké systémy je možné najít třeba v leteckých simulátorech nebo některých zařízeních v telefonních ústřednách. Z hlediska CPU mají dostatečný výkon na vykonávání více jednodušších úloh či pro jednu složitější úlohu. [4]

Druhým kritériem je schopnost systému pracovat v reálném čase, tedy schopnost systému reagovat na určitý podnět za předdefinovaný časový rámec. Takový systém je označován jako hard real-time systém. Soft real-time systémem označuje systémy, které toto nemusí splňovat, avšak na podnět musí zareagovat stejně. [4]

Třetí kriterium, síťová konektivita, je velmi důležité zejména pro možnost výměny dat či aktualizace firmwaru. Jednou z nejrozšířenějších je Ethernet. [4]

Poslední kategorií, kterou však nemusí systém vždy splňovat, jsou uživatelské interakce. Ty umožňují komunikovat s uživatelem, přijímat zadaná data od uživatele a případně nabídnout výstup. V nejjednodušším případě se může jednat o spínací tlačítka a LED. U složitějších systémů se může vyskytnout dotykový displej s obrazovkou doprovázený audio výstupem skrze reproduktor. [4]

3.3 Výhody operačního systému s jádrem Linux

Hlavní výhodou operačního systému s jádrem Linux je kvalita a spolehlivost kódu. Jde o subjektivní vlastnost, kterou lze posuzovat na základě následujících kritérií [4]:

- Modulárnost a struktura zdrojového kódu zajišťující oddělení funkcionality do jednotlivých modulů, které jsou uvnitř rozděleny na jednotlivé funkce
- Jednoduchost opravy či změny kódu pro člověka obeznámeného s vnitřní funkcionalitou
- Konfigurovatelnost umožňuje určit, které části budou součástí finálního produktu
- Předvídatelnost spuštění programu, který by se měl chovat tak, jak se od něho očekává
- Při chybě by se měla spustit rutina, která problém oznámí systémovému administrátorovi přehlednou diagnostickou zprávou, případně se automaticky zotaví z chyb
- Životnost programu by měla být co největší bez nutnosti asistence a při chybových událostech by měla být zachována integrita

Zdrojový kód jádra a dalších open-source programů je volně přístupný na internetu, odkud si jej lze volně stáhnout. Do těchto internetových repozitářů přispívají kromě jednotlivých vývojářů i výrobci hardwaru. Uživatel tak není odkázán jen na komunitní podporu, ale může přímo použít softwarové řešení od daného výrobce hardwaru. Díky open-source licenci je možné zdrojové kódy používat pro libovolné účely, modifikovat či i přímo distribuovat. Největší výhodou open-source softwaru je, že je většinou úplně zdarma (open-source není ekvivalent free softwaru) a je dostupný, což odbourává náklady na počáteční vývoj, vývojové nástroje a náklady na běh. [4]

3.4 Nevýhody operačního systému s jádrem Linux

V zásadě existují celkem dvě velké nevýhody operačního systému s jádrem Linux [4]:

- Roztříštěnost
- Podpora od výrobce

První nevýhodou je roztříštěnost, která vzniká tím, že existuje více podobných projektů. Typickým příkladem jsou linuxové distribuce. Všechny mají společný cíl (být operačním systémem), ale každá volí jinou cestu a nástroje.

Druhá z nevýhod se projevuje nejvíce zejména ve firemní sféře. Podpora od výrobce totiž není licencí garantována a záleží na dobré vůli výrobce, zda ji nabídne a zda si za ni bude něco účtovat či ne. Tento ač zpočátku nevýznamný aspekt může nabývat velkého významu v některých oblastech aplikace

embedded systémů, např. chyba v zabezpečovací technice, letectví, dopravě … bez záruky na odstranění či náhrady škody.

3.5 Základní přehled embedded systémů

Na následujících několika stránkách je seznam embedded systémů, které jsou kompatibilní s operačním systémem typu Linux. Architektura je u všech procesorů stejná – ARM. Jde tedy o procesory 32-bitové procesory typu RISC, které používají zřetězené zpracování instukcí – pipeline. Každá z uvedených platforem má též GPIO piny, které umožňují připojení externích elektronických součástek či rovnou celých modulů.

3.5.1 Beaglebone

Beaglebone je levný embedded linuxový počítač o rozměrech kreditní karty, který nabízí velké množství GPIO pinů, které jsou ještě vnitřně multiplexovány pro navýšení funkcionality. Velkou výhodou je 7 kanálový analogově-číslicový převodník pro převod vstupních analogových napětí na jejich číslicovou reprezentaci. K počítači je možné připojit rozšiřující moduly, které například nabízejí grafický výstup na TFT LCD displej, připojení a regulaci napětí z baterií či zpracování dat z připojených senzorů. [7]

Základní parametry [7]:

- CPU: superskalární ARM Cortex-A8 (armv7a) taktovaný na 720MHz
- GPU: 3D grafický akcelerátor
- ARM Cortex-M3 pro správu napájení
- 2 programovatelné realtimové 32-bitové RISC jednotky
- Konektory: USB port (lze i pro napájení), Ethernet
- GPIO: 2x 46 GPIO pinů (2 I2C, 5 UART, I2S, SPI, CAN, 66 GPIO, 7ADC)

3.5.2 Raspberry Pi

Jedná se o velmi populární embedded linuxový počítač, který svým vznikem spustil lavinu v oblasti embedded linuxových počítačů. Původním záměrem organizace Rapberry Pi Foundation měla být učební pomůcka pro výuku programování do škol, avšak uplatnění se našlo téměř v každém odvětví díky své univerzálnosti. Důkazem popularity je fakt, že za necelý rok se ve světě prodalo více než 1 milión kusů [8].

Podobně jako Beaglebone i pro Raspberry vznikly rozšiřující moduly. Jedním z nejzajímavějších je jistě AlaMode modul, který nabízí možnost připojení rozšiřujících modulů, shieldů, určených pro vývojový kit Arduino, založený na rodině procesorů Atmel AVR, případně rozšiřující modul Gertboard, který nabízí analogovo-číslicové předvodníky, které Raspberry Pi chybí, a několik relé pro spínání výkonových částí. Počítač se vyskytuje ve 2 verzích, A a B. [9]

Tab. 1: Základní parametry modelů typu A a B pro Rasberry Pi. Zdroj: [10]

	Model A	**Model B**
System-on-a-chip (SoC):	Broadcom BCM2835 (CPU + GPU. SDRAM je na odděleném čipu na umístěném na horní straně BCM2835)	
CPU:	ARM11 ARM1176JZF-S taktovaný na 700MHz	
GPU:	Broadcom VideoCore IV,OpenGL ES 2.0,OpenVG 1080p30 H.264	
Memory (SDRAM)iB	256 MiB	256 MiB (do 15 října 2012); 512 MiB (od 15 října 2012)
USB 2.0 porty:	1 (součástí BCM2835)	2 (přes integrovaný USB hub)
Video výstupy:	Kompozitní video nebo kompozitní RCA, HDMI (nelze použít současně)	
Audio výstupy:	TRS connector | 3.5 mm jack, HDMI	
Interní uložiště:	Secure Digital|SD / MMC / SDIO	
Ethernet:	Žádný	10/100M
GPIO:	8 GPIO pinů, SPI, I2C, I2S, UART	
Příkon:	500 mA, (2,5 W)	700 mA, (3,5 W)
Napájení:	5 V DC přes Micro USB typ B nebo přes GPIO	
Rozměry:	85,0 x 56,0 mm	

3.5.3 UDOO

UDOO je jednodeskový počítač, který nabízí až čtyřnásobný výkon výše uvedeného Raspberry Pi. Mimo to nabízí kompatibilitu s rozšiřujícími moduly pro vývojový kit Arduino Due s procesorem Atmel ARM. UDOO vznikl jako kickstarter projekt a momentálně ještě není k dispozici (datum vydání-září 2013). [11]

Základní parametry [12]:

- CPU1: Freescale i.MX 6 ARM Cortex-A9 CPU dvou/čtyř jádrový procesor taktovaný na 1GHz
- GPU: 3 integrované grafické akcelerátory v každém procesoru pro 2D, OpenGL® ES2.0 3D and OpenVG™
- CPU2: Atmel SAM3X8E ARM Cortex-M3 CPU (stejné jako v Arduino Due)
- RAM: DDR3 1GB
- GPIO: 54 digitálních GPIO a analogový vstup, komptabilní s rozložením pinů na Arduino R3
- HDMI and LVDS + Touch (I2C signals)
- Ethernet: 10/100/1000 MBit
- Integrované moduly: WiFi modul
- Mini USB and Mini USB OTG
- USB type A (x2) and USB connector (requires a specific wire)
- Audio vstup: analogové audio vstup a vstup pro mikrofonu
- Konektory: SATA (pouze ve verzi 4 jádrového procesoru), konektor pro kameru, Micro SD
- Napájení: 5-12V DC nebo připojení baterie přes externí napájecí konektor

3.5.4 CubieBoard

Cubieboard je založen na AllWinner A10 SoC a na rozdíl od většiny levných linuxových embedded počítačů má dedikovaný ethernetový port, který umožňuje dosahovat vyšších rychlostí i při připojených USB zařízeních. Přítomnost IrDA a SATA konektoru jej předurčuje pro nasazení jako multimediálního systému pro přehrávání videí a ovládní pomocí klasického TV ovladače, případně jako NAS (network-attached storage) systému pro sdílení a zálohování dat po Ethenetu. [13]

Základní parametry [13]:

- CPU: 1G ARM cortex-A8 procesor, NEON, VFPv3, 256KB L2 cache
- GPU: Mali400, OpenGL ES GPU
- RAM: 512M/1GB DDR3 taktovaná na 480MHz
- Video výstup: HDMI 1080p
- Ethernet: 10/100M
- Interní uložiště: 4Gb Nand Flash
- Rozhraní: 2 USB porty, 1 microSD slot, 1 SATA, 1 IrDA

- GPIO: 96 pinů s funkcemi I2C, SPI, RGB/LVDS, CSI/TS, FM-IN, ADC, CVBS, VGA, SPDIF-OUT, R-TP

3.5.5 iMX233-OLinuXino

iMX233-OLinuXino je velmi kompaktní jednodeskový průmyslový linuxový počítač, který je výsledkem open source softwarového a open source hardwarového projektu OLINUXINO. To umožňuje použít všechny CAD a zdrojové soubory pro libovolné osobní i komerční účely bez omezení na výrobu a prodej. Díky průmyslovému nasazení může operovat v provozních teplotách od -25° do +85°C. Mimo průmyslovou oblast jej lze využít například v 3D tiskárnách typu rep-rap jako interpret G-kódu, levný PLC či pro automatizování domácností. [14]

Základní parametry [15]:

- CPU: iMX233 ARM926J procesor taktovaný na 454Mhz
- RAM: 64 MB
- SD-card connector for booting the Linux image
- Video výstup: TV PAL/NTSC
- USB: 2 USB vysokorychlostní porty
- Ethernet: 100 Mbit
- Audio vstup: Stereo Audio
- Audio výstup: Stereo Audio
- GPIO: 40 GPIO pinů
- Napájení: 6-16VDC
- Rozměry: 94,0mm x 54,6mm
- Dvě tlačítka a UEXT konektor pro připojení externích modulů

3.5.6 Aria G25

Jedná se o levný SMD modul s linuxovým embedded SoM předurčený pro snížení doby vývoje nízkopříkonových zařízení. Komplexní hardwarové součástky jako CPU, RAM, Ethernet a napájecí komponenty jsou integrovány v SMD pouzdrech, která jsou umístěna na osmivrstvém plošném spoji o rozměrech 40x40 mm s vývody po okraji. Ty umožňují připojení tohoto modulu k jednoduššímu a levnějšímu plošnému spoji, u kterého si návrhář vystačí se dvěma vrstvami. Modul již obsahuje

předprogramovaný bootloader, který načte operační systém z externí microSD či SD karty. K dispozici je též open-source firmware a vývojové prostředí spolu s ukázkovými zdrojovými kódy. [16]

Základní parametry [16]:

- CPU: ARM9 na frekvenci 400Mhz, Atmel AT91SAM9G25 SoC
- RAM: 128 or 256 MByte DDR2
- LAN: 10/100 Mbit
- USB: až 3 porty pro připojení USB zařízení:
- UART: až 6 sériových linek
- I2C: až 2 I2C sběrnice
- SPI: až 2 SPI sběrnice
- GPIO: až 60 GPIO vývodů
- A/D: až 4 kanály při 10 bitovém rozlišení
- Napájení: single at 3.3 Volt DC
- Úroveň signálů: TTL 3.3V
- Rozsah provozních teplot: 0-70 °C a -20 +70 °C (-E verze)
- Rozměry: 40 x 40 mm
- Váha: 5g
- Jedna uživatelem konfigurovatelná LED

4. Praktická část

4.1 Komponenty linuxové distribuce

Každá linuxová distribuce je sada obsahující linuxové jádro, knihovny a programy, které jsou obvykle soustředěné na jednom datovém médiu, případně jinak seskupeny dohromady. Následující text dává přehledný seznam komponent, které se v distribuci mohou vyskytovat. Některé z nich jsou samozřejmě nezbytné. [2]

Součástí každé distribuce je jádro. To je většinou založené na jádře Linux, které však je možné dle potřeb modifikovat. Lze najít i distribuce, které používají i jiné jádro než Linux, jmenovitě např. Hurd nebo jádra z *BSD. K jádru taktéž přiléhají utility k jeho správě, nejdůležitějšími jsou zejména nástroje pro práci s moduly. [2]

Každý počítač potřebuje bootloader, což je první vykonávaný program, který má za úkol inicializovat hardware a nahrát do paměti jádro. Složitější bootloader se typicky nazývá zavaděč a umožňuje výběr zavedení jádra z několika druhů médií a případně může nabízet i další nízkoúrovňovou funkcionalitu jako např. kontrolu integrity a test paměti. Ve světě embedded zařízení se typicky jedná o U-Boot, který usiluje o vytvoření jednotného univerzálního bootloaderu. [2]

Každý systém vyžaduje k běhu systémové knihovny. Pro větší systémy se standardně používá GNU knihovna glibc, pro embedded účely uclibc, která nabízí podobné funkce, avšak s mnohem nižšími paměťovými nároky. Kromě těchto základních knihoven se v systému mohou vyskytovat i různé další. [2]

Pro správu systému a základní práci se systémem jsou nutné základní utility. Většinou jde o nástroje ze sady GNU Core Utilites z balíčku `coreutils`. V embedded systémech se většinou nahrazují jediným binárním souborem, který může být dynamicky či staticky linkován. Tímto binárním souborem je většinou Busybox, což je sada modifikovaných základních programů, které je snazší zkompilovat pro cílové zařízení. Zároveň je snazší údržba, neboť program stačí modifikovat, zkompilovat a na cílovém zařízení pouze zaměnit binární soubor. [2]

Většina démonů a složitějších programů taktéž vyžaduje konfigurační soubory. Ty lze snadno vytvořit ze šablon, které lze najít u přiložených zdrojových souborů. Spolu s konfiguračními soubory jsou ve víceúrovňových systémech nutné skripty, které definují spuštěné služby pro různé úrovně běhu a též pro spouštění a zastavení služeb. [2]

Podobně jako stolní verze operačních systémů i embedded systémy mohou mít grafické prostředí. To je obvykle založené na X Window sytému. Většina vývojářů volí integrovaná grafická prostředí typu GNOME, KDE či LXDE, která se o nastavení X Window systému postarají sama. Velmi často je ale grafické ovládání založeno na webových technologiích, takže se výsledný obraz vytváří až na připojeném počítači. Toto řešení je typické pro síťové prvky. [2]

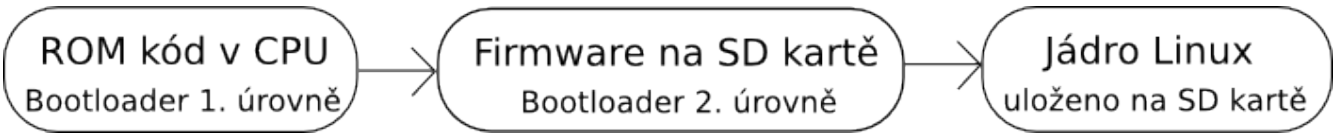
Počet aplikačních programů se u jednotlivých distribucí různí. Jejich počet je především dán účelem. U distribucí pro stolní počítače jich jsou většinou desítky i stovky. Pro embedded účely občas stačí i jeden. [2]

4.2 Bootloader

Ačkoliv je bootloader na většině zařízení spuštěn jen po krátkou dobu během startu systému, plní velmi důležitou roli, neboť je zodpovědný za inicializaci hardwaru a načtení jádra do paměti. Nastavení bootloaderu na embedded systémech má svá specifika. Je proto nezbytně nutné přečíst si katalogový list ke každému zařízení a zjistit, jak správně inicializovat hardware pro další úkony. [4]

Některá zařízení využívají malé interní ROM paměti na čipu, ve které je uložen bootloader první úrovně, který se postará o inicializaci hardware a o načtení druhého stupně bootloaderu, který si již programátor může dodat typicky na SDkartě se souborovým systémem typu FAT z důvodů jednoduché implementace a paměťového místa. Ten následně načte jádro do paměti.

Pro většinu embedded zařízení dodávají výrobci vlastní bootloadery, tudíž se vývojář nemusí o inicializaci hardware starat. Alternativou je projekt Das U-Boot, který kromě jednoduchého bootování systému nabízí i funkce pro načtení systému ze sítě pomocí technologie PXE či práce s interní programovou pamětí typu FLASH. [4]



Obr.1: Schéma bootovacího procesu pro dvojúrovňový bootloader. Zdroj: Vlastní zpracování

4.3 Kořenový souborový systém a jeho struktura

Kořenový souborový systém a jeho struktura je poměrně pečlivě svázána se systémem standardizovaných pravidel, Linux Standard Base. LSB je projektem organizace Linux Foundation, jehož cílem je standardizace celého operačního systému GNU/Linux. Jejím dodržením lze zajistit

kompatibilitu linuxových distribucí včetně přenositelnosti programů v binární formě. Není tedy nutné znovu kompilovat program pro jinou distribuci. LSB navíc poskytuje zpětnou kompatibilitu pro předchozí verze tohoto standardu. [2]

Dalším standardem je Filesystem Hierarchy Standard, který je obecným standardem pro unixové systémy. Ten obsahuje definice, které udávají, jaká je standardní adresářová struktura kořenového adresáře a význam jednotlivých adresářů. V embeddech systémech se ale mohou vyskytnout situace, kdy některé adresáře nemají význam, a proto je lze vynechat. Jedinou výjimku tvoří / , což je kořenový adresář pro všechny ostatní adresáře, který je přípojným místem, ke kterému jádro při inicializaci připojuje kořenový souborový systém. Následující seznam obsahuje nejdůležitější a nejpoužívanější adresáře: [2] Zdroj: [17]

Název	Obsah adresáře
/bin	běžné nesystémové binární spustitelné soubory
/sbin	běžné sytémové binární spustitelné soubory
/etc	konfigurační soubory
/dev	soubory zařízení
/proc	přípojné místo pro pseudosouborový systém proc obsahující především informace o procesech
/var	proměnné soubory (např. logy či dočasné soubory, které se nesmí při resertu smazat)
/tmp	dočasné soubory
/usr	uživatelské programy
/home	domovské adresáře uživatelů systému
/boot	bootovací soubory (např. obraz jádra a zavaděč systému)
/lib	systémové knihovny
/opt	aplikace třetích stran
/mnt	přípojná místa
/media	přípojná místa pro odebíratelné zařízení
/srv	servisní data

Na velké části linuxových systémů však lze narazit ještě na jeden velmi důležitý adresář. Jde o přípojný bod, `/sys`. Do tohoto stejnojmenného psedosouborového systému exportuje jádro svá data, která využívají některé uživatelské aplikace. Jedním z nich je udev, což je démon. Ten se stará o tvorbu souborů zařízení v /dev, které reprezentují jednotlivé hardwarové komponenty. Výjimku tvoří síťová rozhraní, která nemají svůj soubor zařízení, a virtuální zařízení – např. FIFO roura. [17]

4.4 Vývojové prostředí mezi vývojovým počítačem a cílovým zařízením

Předpokladem pro úspěšné sestavení embedded linuxové distribuce je definování vývojových poměrů mezi cílovým zařízením a vývojovým počítačem.

Nejběžnější sestavou je přímé spojení těchto dvou systémů pomocí fyzického spojení realizovaného metalickým či optickým kabelem. Všechna komunikace probíhá tedy přímo mezi těmito dvěma zařízeními. Při použití sériové linky probíhá komunikace mezi systémy přímo, oproti použití Ethernetu, kde je třeba realizovat zásobník pro daný síťový protokol. Výhodou je však, že lze jádro při startu stáhnout pomocí TFTP protokolu a souborový systém připojit přes NFS. [4]

Druhou možností je nepropojit systémy přímo, ale použít při vývoji paměťové médium, na které se uloží potřebná data ke sdílení mezi oběma systémy. Typickým příkladem tohoto paměťového média je microSD či SD karta. Tato varianta se používá u všech výše zmíněných embedded zařízení z důvodu flexibility a nižších nároků na systém ve vývojové fázi. [4]

Poslední možností je vývoj přímo na embedded systému [4]. Výhodou je především použití nativních verzí nástrojů a použití pouze jednoho vývojového zařízení, které je zároveň i cílovým, a tudíž odpadá i způsob komunikace a přenosu dat. Tato možnost však vyžaduje, aby cílové zařízení mělo podobné charakteristiky jako typický vývojářský počítač (vyšší nároky na RAM, úložné místo …).

4.5 Nástroje pro sestavení linuxové embedded distribuce

Většina zdrojových souborů potřebných pro sestavení embedded linuxového systému je až na několik výjimek napsána v jazyce C/C++, který lze přeložit pomocí kompilátoru, gcc. Mimo něj je však potřeba i několik pomocných nástrojů pro práci s binárními soubory, které jsou součástí balíčku binutils. Je jím zejména linker ld, preprocesor m4, program readelf pro čtení obsahu objektového a spustitelného kódu ve formátu ELF, program strip pro odstranění nepotřebných sekcí programu atd. Pro opakované a jednoduché sestavení je vhodný nástroj make, který dle pravidel v souboru Makefile sestaví námi požadovaný spustitelný soubor. Pro debuggování a trasování aplikací se hodí nástroje jako je gdb, gdbserver, strings, strace, ltrace a jiné. [18]

Kromě vývojových nástrojů je pro vytvoření potřebného run-time prostředí potřeba typicky sada standardních systémových knihoven, jde zejména o knihovnu pro jazyk C, libc. Při sestavování vlastních kompilačních nástrojů je také potřeba mít k dispozici hlavičkové soubory jádra, které obsahují popis rozhraní systému jádra Linux (API). Ty jsou také třeba, když aplikace nepoužije standardní knihovní funkci, ale volá některou ze služeb jádra přímo.

Tyto nástroje jsou společné jak pro nativní, tak i pro křížovou kompilaci. Při nativní kompilaci se aplikace vytváří pro stejnou architekturu procesoru, jako na které je vyvíjena. Křížová kompilace spočívá v kompilaci na vývojovém počítači, který má jinou architekturu než cílový. Nelze tedy použít nativní verzi nástrojů, neboť cílový procesor by přeloženým instrukcím „nerozuměl". Křížové nástroje jsou navržené tak, aby fungovaly na architektuře vývojového počítače, avšak aplikaci překládaly do instrukcí pro cílovou architekturu procesoru. [18]

Výhodou křížových nástrojů je, že umožňují práci na vývojářském počítači, které je většinou výkonnější a nabízí větší komfort při vývoji (např. použití verzovacího systému, grafické prostředí, integrované vývojové prostředí …), avšak ne každou aplikaci lze bez modifikace zkompilovat křížově. [19]

4.6 Sestavení embedded linuxového operačního systému s jádrem Linux

Teorii o sestavení funkčního OS s jádrem Linux se věnuje velké množství publikací s různou úrovní zaměření a detailů. Je důležité si uvědomit, že každá platforma má svá specifika, a proto nelze vytvořit naprosto jednotný postup, který by vždy fungoval. Problémovou oblastí je oblast kolem hardware, zejména zvolení správných křížových nástrojů, nastavení bootloaderu a jádra.

Pravděpodobně nejpokročilejší publikací a zároveň i projektem zaměřeným na podrobný popis sestavení linuxového systému je Linux From Scratch (LFS), jehož cílem je detailně popsat postup tvorby minimálního OS především pro platformy x86 a x86_64 (též označované pod názvem amd64), tedy tvorbu OS pomocí nativních kompilačních nástrojů. Tento projekt se ale postupem času rozrostl a nyní nabízí i podrobný návod pro další architektury procesorů. Jde zejména o projekt CLFS, který je zaměřen na tvorbu embedded linuxových systémů pomocí křížových kompilačních nástrojů. [20]

Další velmi šikovnou publikací je Building Embedded Linux Systems, která obsahuje zásady, postupy a názorné ukázky skriptů, které lze použít pro automatizaci tvorby embedded linuxového systému.

Postup sestavení:

- Vytvoření nového uživatelského účtu

- Tvorba adresářové struktury pro sestavení embedded linuxového operačního systému
- Stažení balíčku se zdrojovými kódy
- Tvorba FHS v adrešáři ${RPI}
- Sestavení křížových kompilačních nástrojů
- Sestavení Busybox, iana-etc a jádra Linux
- Nastavení firmware, bootovacích skriptů a dalších konfiguračních souborů
- Zapsání obrazu systému na SD kartu

4.7 Sestavení embedded linuxového OS manuálně krok za krokem pro Raspberry Pi

4.7.1 Vytvoření uživatelského účtu

Před zahájením tvorby embedded OS je důležité se přihlásit do systému jako běžný uživatel bez privilegií, které má root, aby následně nedošlo k těžko předvídatelným problémům jako např. poškození souborového systému pracovní stanice. Je též vhodné nepoužívat vlastní účet, kterým se normálně přihlašujeme do systému, a slouží nám k pracovním povinnostem. Proto ze všeho nejdříve vytvoříme nový účet v systému, to znamená skupinu a uživatele. K tomu budeme potřebovat oprávnění uživatele root. Následně se do systému přihlásíme jako nově vytvořený uživatel.

```
# Pridani nove skupiny pidev (vice informaci ... man groupadd)
groupadd pidev

# Pridani noveho uzivatele pidev (vice informaci ... man useradd)
useradd -s /bin/bash -g pidev -m -k /dev/null pidev

# Nastaveni hesla pro uzivatele pidev (vice informaci ... man passwd)
passwd pidev

# Prihlaseni do systemu jako uzivatel pidev
su - pidev
```

4.7.2 Tvorba adresářové struktury pro sestavení embedded linuxového operačního systému

Těmito příkazy jsme si tedy vytvořili nový uživatelský účet pidev a v adresáři `/home` by se nám tedy měl objevit domovský adresář uživatele pidev. Zde si vytvoříme adresářovou strukturu podobnou té z

[4] pomocí níže uvedených příkazů. Dále si vytvoříme proměnné, které vyexportujeme, aby je mohly používat i ostatní procesy. Tyto proměnné však pozbudou platnosti po odhlášení ze systému, a proto je nutné je též definovat ve skriptu `.bashrc`, který se spouští při každém přihlášení uživatele do systému. Při přihlášení do systému se však spouští shell, který si přečte příkazy ze souboru `/etc/profile` a následně `.bash_profile` v domovském adresáři, čímž hrozí kontaminace systémových proměnných, a proto je třeba modifikovat soubor `.bash_profile`**.** [20]

```
# Vytvoreni adresarove struktury v jednom prikazu (vice informaci ... man mkdir)
mkdir -pv /home/pidev/raspberrypi/{sources,firmware,tools,cross-tools,rootfs}

#Vytvoreni a vyexportovani promennych s cestou k adresarum pro jednodussi praci
export RPI=/home/pidev/raspberrypi/rootfs
export RPI_SRC=/home/pidev/raspberrypi/sources
export RPI_FW=/home/pidev/raspberrypi/firmware
export RPI_TOOLS=/home/pidev/raspberrypi/tools
export RPI_CROSS_TOOLS=/home/pidev/raspberrypi/cross-tools
```

Výpis změn ze souboru .bash_profile:

```
# Spusti novy shell, ktery ma definovany jen promenne $HOME, $TERM a $PS1
exec env -i HOME=${HOME} TERM=${TERM} PS1='\u:\w\$ ' /bin/bash
```

Výpis změn ze souboru .bashrc:

```
#vypnuti hashovaci funkce shellu, nutno prohledat cesty z promenne $PATH
set +h
#nastaveni implicitni masky prav nove vytvorenych souboru (u:rwx,g:rx,o:rx)
umask 022
export RPI=/home/pidev/raspberrypi/rootfs
export RPI_SRC=/home/pidev/raspberrypi/sources
export RPI_FW=/home/pidev/raspberrypi/firmware
export RPI_TOOLS=/home/pidev/raspberrypi/tools
export RPI_CROSS_TOOLS=/home/pidev/raspberrypi/cross-tools
#promenna pro nastaveni lokalizace programu
export LC_ALL=POSIX
#timto zajistime, ze se pouziji vzdy verze krizove a ne nativni
export PATH=${RPI_CROSS_TOOLS}/bin:/bin:/usr/bin
```

4.7.3 Stažení balíčků se zdrojovými kódy

Nyní když je vývojové prostředí připraveno, je na čase stáhnout potřebné balíčky se zdrojovými soubory do adresáře v proměnné ${RPI_SRC} a zde je rozbalit. Pro stažení většiny těchto balíčků přes HTTP či FTP lze použít nástroj *wget*. Podporuje parametr -i FILE - předání vstupního souboru s adresami požadovaných dat ke stažení, která jsou od sebe oddělená po řádcích. Balíčky umístěné na githubu lze získat prostřednictvím nástroje git, v našem případě jde především o specifické balíčky pro Raspberry Pi. [21]

```
cd ${RPI_SRC}

# Busybox
wget http://busybox.net/downloads/busybox-1.18.4.tar.bz2

# Crosstool-ng
wget http://crosstool-ng.org/download/crosstool-ng/crosstool-ng-1.17.0.tar.bz2
# CLFS bootscripts
wget  http://cross-lfs.org/files/packages/embedded-0.0.1/clfs-embedded-bootscripts-
1.0-pre5.tar.bz2

# iana-etc
wget http://cross-lfs.org/files/packages/embedded-0.0.1/iana-etc-2.30.tar.bz2

# Kernel
git clone git://github.com/raspberrypi/linux.git

# Busybox patch
wget http://patches.cross-lfs.org/embedded-dev/busybox-1.18.4-config-1.patch

# iana-etc patch
wget http://patches.cross-lfs.org/embedded-dev/iana-etc-2.30-update-1.patch

cd ${RPI_FW}

# Broadcom firmware
git clone git://github.com/raspberrypi/firmware.git
```

```
cd ${RPI_TOOLS}

# Extra tools
git clone git://github.com/raspberrypi/tools.git
```

4.7.4 Tvorba FHS v adresáři ${RPI}

Nyní vytvoříme adresářovou stukturu v kořenovém adresáři našeho nového souborového systému odpovídající specifikaci FHS, která byla detailněji rozebrána na začátku této kapitoly. Tato sekvence příkazů je typická téměř pro každý UNIXový systém. Nejde o nic jiného než o tvorbu adresářů. [4]

```
mkdir -pv ${RPI}/{bin,boot,dev,{etc/,}opt,home,lib/{firmware,modules},mnt}
mkdir -pv ${RPI}/{proc,media/{floppy,cdrom},sbin,srv,sys}
mkdir -pv ${RPI}/var/{lock,log,mail,run,spool}
mkdir -pv ${RPI}/var/{opt,cache,lib/{misc,locate},local}

#jde o specialni adresare s jinymi pravy, lze pouzit tez prikazy mkdir a chmod
install -dv -m 0750 ${RPI}/root
install -dv -m 1777 ${RPI}{/var,}/tmp

mkdir -pv ${RPI}/usr/{,local/}{bin,include,lib,sbin,src}
mkdir -pv ${RPI}/usr/{,local/}share/{doc,info,locale,man}
mkdir -pv ${RPI}/usr/{,local/}share/{misc,terminfo,zoneinfo}
mkdir -pv ${RPI}/usr/{,local/}share/man/man{1,2,3,4,5,6,7,8}

for dir in ${RPI}/usr{,/local}; do
  ln -sv share/{man,doc,info} ${dir}
done
```

4.7.5 Sestavení křížových kompilačních nástojů

Místo nastavování jednotlivých parametrů ručně a následnou kompilací jednotlivých nástrojů postupně, jsem zvolil možnost nastavit parametry křížového kompilátoru v přehledném menu a pomocí následně vygenerovaného souboru Makefile pro křížový kompilátor crosstool-ng. Důvodem bylo několikanásobné neúspěšné sestavení dle postupu na stránkách projektu CLFS. Alternativně lze kompilátor pro Raspberry Pi také stáhnout již v binární podobě, avšak vývojář se může vyskytnout v situaci, kdy tuto možnost nemá. V adresáři se zdrojovými kódy k crosstool-ng zadáme do terminálu následující příkazy [22]:

```
tar xjf crosstool-ng-1.17.0.tar.bz2
cd crosstool-ng- 1.17.0
./configure
make
make install
```

Tento příkaz rozbalí a nainstaluje crosstool-ng do systému. Nyní již lze vytvořit křížové kompilační nástroje pro cílové zařízení následujícími příkazy:

```
cd /a/directory/to/build/your/toolchain
ct-ng help
ct-ng list-samples
ct-ng show-arm-unknown-linux-gnueabi
ct-ng arm-unknown-linux-gnueabi
```

Tyto příkazy přednastaví parametry křížových kompilačních nástrojů pro ARM za použití knihovny glibc. Následně jen doladím nastavení pomocí interaktivního menu, které se zobrazí při příkazu: [22]

```
ct-ng menuconfig
```

V záložce `Paths and misc` options jsem povolil `Try features marked as EXPERIMENTAL`, neboť bez něj kompilátor pro Raspberry Pi nefunguje a dále změnil obsah `Prefix z /opt/cross/x-tools/${CT_TARGET}` na `${RPI_CROSS_TOOLS}/${CT_TARGET}`.

V podmenu `Target options` jsem nastavil položku `Target architecture` na `arm`, `Endianness na Little endian` a `Bitness` na `32-bit`. V podmenu `Operating system` jsem nastavil `Target OS na linux`. V podmenu `Binary utilities` lze nastavit položku `binutils version` na nejnovější, avšak není to nezbytně nutné. Z položek podmenu `C compiler` je vhodné povolit `Show Linaro versions (EXPERIMENTAL)`, které nám zobrazí dalších několik možných kompilátorů. Je vhodnější zvolit Linaro verzi než některý z klasických gcc kompilátorů, neboť je lépe uzpůsobena pro Raspberry Pi. Zbytek položek a submenu je možné ponechat tak, jak je. [23]

Nyní již jen stačí křížové kompilační nástroje sestavit, a to pomocí následujícího příkazu:

```
ct-ng build
```

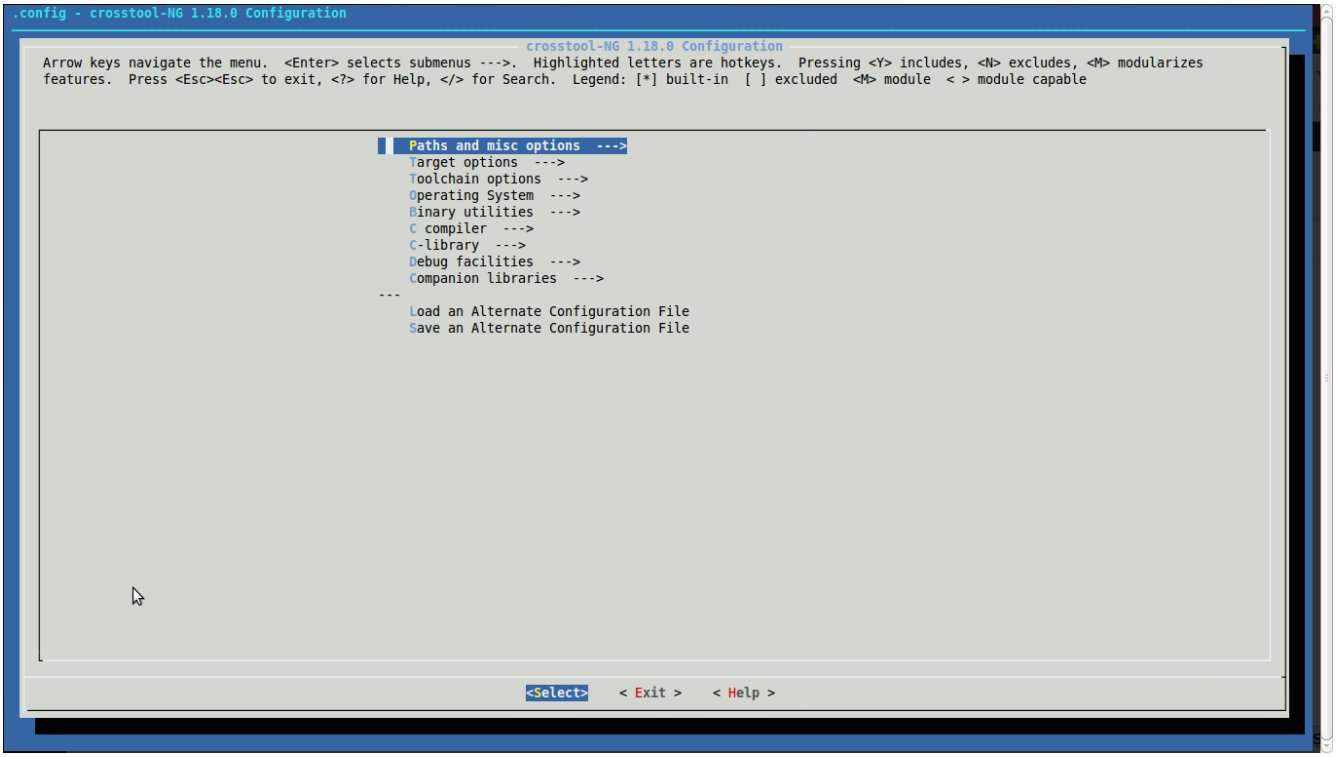
Po sestavení jsou nástroje dostupné v adresáři `${RPI_CROSS_TOOLS}/x-tools/arm-unknown-linux-gnueabi/bin`. Pro použití je ještě nutné přidat tuto cestu do systémové proměnné `PATH` a pozměnit některé další proměnné potřebné při kompilaci následovně.

Výpis změn ze souboru .bashrc:

```
export PATH="${RPI_CROSS_TOOLS}/x-tools/arm-unknown-linux-gnueabi/bin:${PATH}"
export CC="${RPI_TARGET}-gcc"
export CXX="${RPI_TARGET}-g++"
export AR="${RPI_TARGET}-ar"
export AS="${RPI_TARGET}-as"
export LD="${RPI_TARGET}-ld"
export RANLIB="${RPI_TARGET}-ranlib"
export READELF="${RPI_TARGET}-readelf"
export STRIP="${RPI_TARGET}-strip"
```

Následné provedení změn ze souboru .bashrc:

```
source ~/.bashrc
```



Obr. 2: Hlavní menu ct-ng. Zdroj: Vlastní zpracování

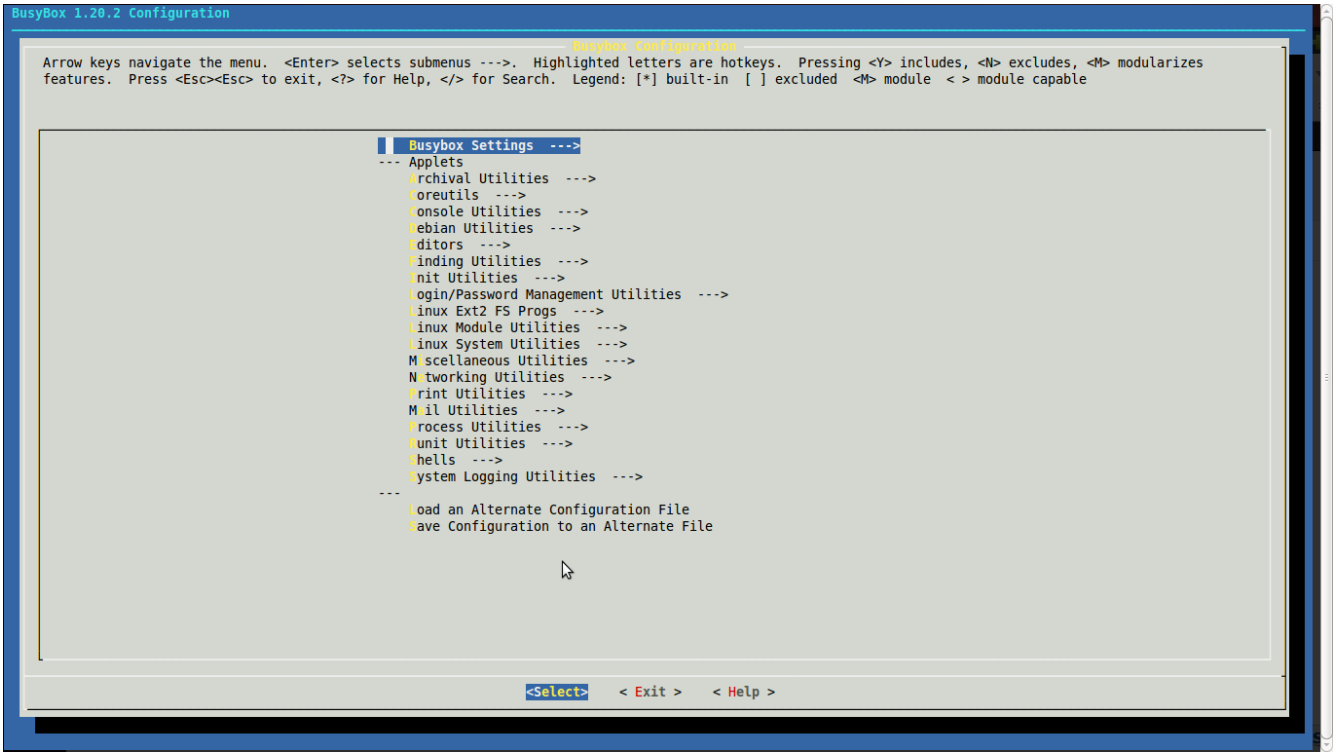
4.7.6 Sestavení Busybox

Busybox nahrazuje velké množství unixových a shellových nástrojů jedním jediným spustitelným souborem. Díky tomu má velmi nízké nároky na systémové prostředky. Cenou za tuto minimalizaci je však to, že ne každý příkaz podporuje všechny možné parametry jako jeho původní GNU verze. Busybox nahrazuje všechny spustitelné soubory tím, že při zavolání daného příkazu je onen příkaz

nahrazen symbolickém odkazem na spustitelný soubor Busyboxu, kterému je jako argument předán onen zadaný příkaz. Podle něj je pak rozeznán a spuštěn požadovaný příkaz. [24]

Následující skript extrahuje soubory Busyboxu, aplikuje na něj patch. Ten obashuje již přednastavený konfigurační soubor. Následně nastavíme tento soubor jako výchozí konfiguraci a následně jej zkompilujeme a nainstalujeme do kořenového adresáře. [24]

```
cd $RPI_SRC
tar -jxvf busybox-1.18.4.tar.bz2 && cd busybox-1.18.4
patch -Np1 -i ../busybox-1.18.4-config-1.patch
cp -v clfs/config .config
make oldconfig
make CROSS_COMPILE="${RPI_TARGET}-"
make CROSS_COMPILE="${RPI_TARGET}-" CONFIG_PREFIX="${RPI}" install
cp examples/depmod.pl ${RPI_CROSS_TOOLS}/bin
chmod 755 ${RPI_CROSS_TOOLS}/bin/depmod.pl
cd $RPI_SRC && rm -rf busybox-1.18.4
```



Obr. 3: Hlavní menu Busybox. Zdroj: Vlastní zpracování

4.7.7 Sestavení iana-etc

Balíček iana-etc obsahuje základní soubory nutné pro síťové služby a protokoly. Balíček opět extrahujeme a aplikujeme na něho patch. Příkaz make převede data ze souborů do správného formátu vyžadovaného soubory `/etc/protocols` a `/etc/services`. [25]

```
cd $RPI_SRC
tar -jxvf iana-etc-2.30.tar.bz2 && cd iana-etc-2.30
patch -Np1 -i ../iana-etc-2.30-update-1.patch
make get
make
make DESTDIR=${RPI} install
cd $RPI_SRC && rm -rf iana-etc-2.30
```

4.7.8 Sestavení jádra Linux

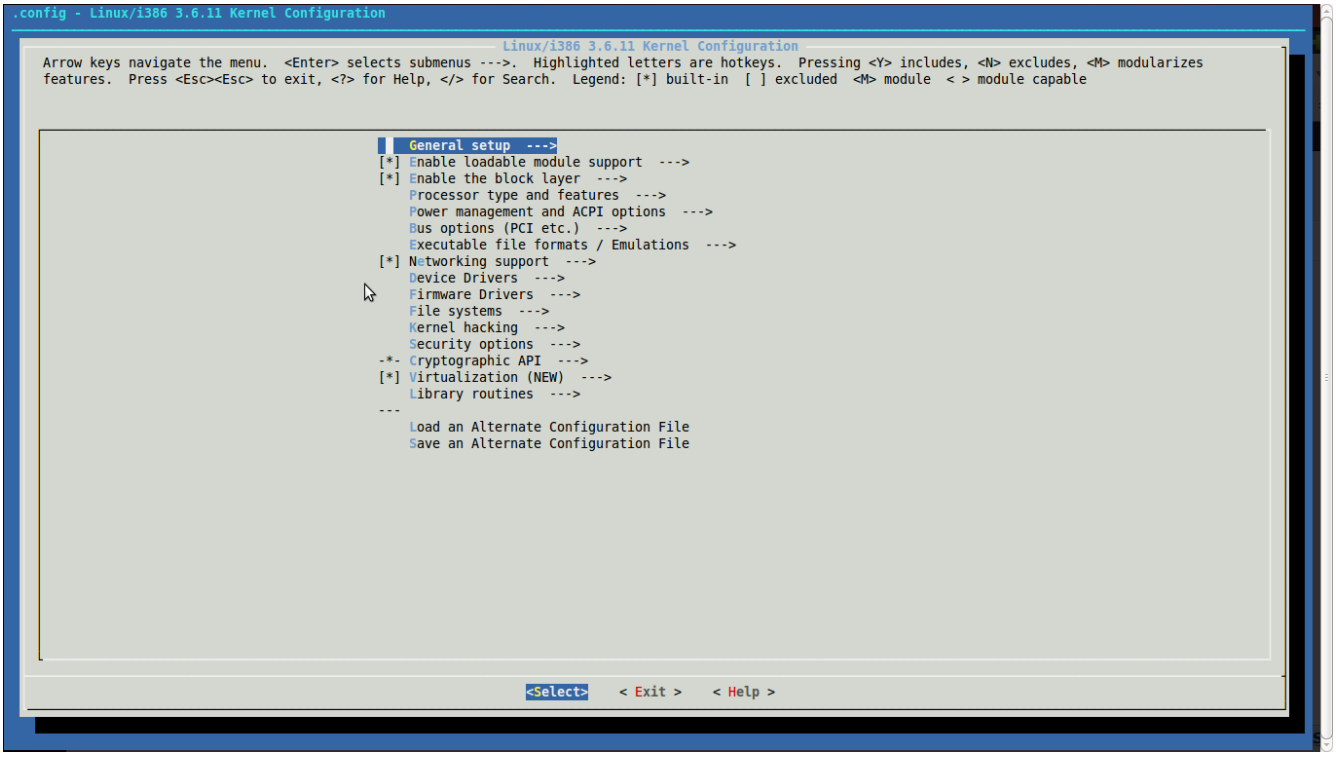
Jako zdrojové kódy jádra Linux je nutné použít ty, které již obsahují nezbytné patche specifické pro Raspberry Pi. Kromě samotných zdrojových kódů již obsahuje i volbu pro defaultní konfiguraci pro Raspberry Pi, které lze využít. Před samotnou kompilací je však ještě třeba zaměnit nativní verzi kompilačních nástrojů za křížovou pomocí proměnné `RPI_CROSS_COMPILE`. Dobrým programátorským postupem je nejdříve vyčistit adresář od souborů z dřívějších kompilací, zde to není nezbytně nutné. Následně již jen zvolím konfiguraci pro Raspberry Pi, potvrdím daný konfigurační soubor, spustím sestavovací proces. Pro modifikaci jádra lze využít příkazu *make menuconfig*. Poté nainstaluji moduly do kořenového souborového systému a pomocí nástrojů v adresáři `$RPI_TOOLS` převedu obraz jádra do požadovaného formátu pro Raspberry Pi a zkopíruji jej do adresáře `$RPI/boot/`. [23]

```
# Nastavení krizovych nastroju
RPI_CROSS_COMPILE=${RPI_CROSS_TOOLS}/bin/${RPI_TARGET}-

# Sestavení jádra
cd $RPI_SRC/linux
make mrproper
ARCH=arm make bcmrpi_cutdown_defconfig
ARCH=arm CROSS_COMPILE=${RPI_CROSS_COMPILE} make oldconfig
ARCH=arm CROSS_COMPILE=${RPI_CROSS_COMPILE} make
```

```
ARCH=arm CROSS_COMPILE=${RPI_CROSS_COMPILE} make modules_install INSTALL_MOD_PATH=$
{RPI}/boot
cd $RPI_SRC

# Prevod obrazu jadra do formatu pro Raspberry Pi
cd $RPI_TOOLS/tools/mkimage
./imagetool-uncompressed.py $CLFS_SRC/linux/arch/arm/boot/Image
cp -v kernel.img $RPI/boot/
```



Obr. 4: Hlavní menu jádra Linux. Zdroj: Vlastní zpracování

4.7.9 Firmware, bootovací skripty a systémové soubory

Spolu s jádrem upraveným přímo pro Raspberry Pi je též dodává i firmware, který zajistí bootování systému. Ten je třeba jen zkopírovat do adresáře `$RPI/boot/`. [23]

```
# cp -v $RPI_FW/firmware/boot/
{bootcode.bin,fixup.dat,fixup_cd.dat,start.elf,start_cd.elf} $RPI/boot/
```

Jako defaultní bootovací skripty pro samotný embedded linuxový systém lze použít ty z projetku CLFS. Skripty opět rozbalíme a nainstalujeme.

```
cd $RPI_SRC
tar -zxvf clfs-embedded-bootscripts-1.0-pre5.tar.bz2 && cd clfs-embedded-
bootscripts-1.0-pre5
make DESTDIR=${RPI} install-bootscripts
install -dv ${RPI}/etc/init.d
ln -sv ../rc.d/startup ${RPI}/etc/init.d/rcS
cd $RPI_SRC && rm -rf clfs-embedded-bootscripts-1.0-pre5.tar.bz2
```

Pro správnou funkci systému je ještě nutné vytvořit v kořenovém adresáři několik souborů. Jedná se o `/etc/mdev.conf` obsahující seznam zařízení, skript `/etc/profile` definující základní prostředí shellu, `/etc/passwd` obsahující systémové skupiny, `/etc/group` obsahující systémové skupiny, `/etc/inittab` obsahující defaultní akce při startu a ukončení systému, `/etc/hostname` obsahující název zařízení pro síťové účely, `/etc/hosts` obsahující jména pro jednotlivé IP adresy, `/etc/network.conf`, `/etc/udhcpc.conf` a `/etc/resolve.conf` obsahující nastavení síťových rozhraní zařízení. [23]

4.7.10 Příprava a zápis obrazu systému na SD kartu

Nyní je na čase připravit, tedy naformátovat, vyměnitelné médium, SD kartu. K tomu účelu lze využít nástroj parted, případně nástroj fdisk. Na SD kartě je nutné vytvořit 2 oddíly, v prvním se bude nacházet obsah adresáře `$RPI/boot/`, ve druhém pak samotný obsah kořenového adresáře $RPI/ bez obsahu adresáře `$RPI/boot/`. Oddíl s obsahem adresáře $RPI/boot/ musí být typu FAT32, neboť z něj načítá bootloader první úrovně bootloader druhé úrovně a jiný souborový systém nepřečte. Volba souborového systému pro oddíl obsahující obsah adresáře `$RPI/boot/` již závisí jen na souborových systémech podporovaných jádrem, které lze změnit při jeho sestavování. Tyto ovladače však musí být zabudovány v jádře, nemohou být načteny jako moduly, protože ty jsou při bootování jádra uloženy na onom souborovém systému. [23]

4.8 Sestavení embedded linuxové distribuce s použitím Yocto/OpenEmbedded pro Raspberry Pi

Kromě manuálního sestavení embedded linuxové distribuce po vzoru LFS lze též využít jeden z několika dostupných frameworků pro tvorbu operačních systémů s jádrem Linux. Zde použitým je OpenEmbedded. Jde o open-source sestavovací systém, který má za úkol zjednodušit práci při tvorbě požadovaného operačního systému pro emmbedded účely. Vývojáři poskytuje plně připravené sestavovací prostředí, které poskytuje solidní základ sestavovaného systému. OpenEmbedded framework se skládá z několika vrstev nazývaných oe-core layer, z nástroje Bitbake, který je správcem sestavovacího procesu a metadat a říká systému, jak má sestavit balíčky a samotných metadat, což jsou soubory, které obsahují informace o tom, jak sestavit balíčky. Kromě nástrojů potřebných pro sestavení poskytuje také kvalitní dokumentaci. [26]

Pomocí OpenEmbedded je možné sestavit embedded linuxovou distibuci do největší detailů. Sestavovací proces lze pozměnit, a to od malých změn týkajících se např. volby programů až po parametry samotného jádra či nastavení optimalizace kompilátoru GCC nebo si dokonce i zvolit jiný.

Yocto Project je open source projekt od Linux Foundation, který poskytuje šablony, nástroje a metody pro sestavení námi požadovaného systému. Tento projekt je zaměřen především na začínající vývojáře v oblasti embedded operačních systémů, ale jistě jej ocení i pokročilejší vývojáři. Yocto Projekt je jakousi nadstavbou, která pracuje se stabilními verzemi OpenEmbedded. K OpenEmbedded přidává malou vrstvu, která obsahuje nastavení distribuce Poky, což je distribuce s minimálními programy schopná běhu, která je vytvořená na základě důkladně otestovaných balíčku z OpenEmbedded. [26]

Pro sestavení obrazu embedded linuxové distribuce pro Raspberry Pi jsem využil meta-raspberrypi BSP (Board Support Package) vrstvu, OpenEmbedded sestavovací framework a distribuci Poky, poskytnutou v rámci Yocto Project.

Yocto Project má hosting na githubu, odkud si lze také stáhnout celý framework a patřičné soubory. Následující příkaz zkopíruje potřebné soubory do lokálního adresáře s názvem yoctoProject, který se ještě předtím vytvoří. [26]

```
git clone git://git.yoctoproject.org/poky yoctoProject
```

Nyní je ještě třeba získat zmíněnou BSP vrstvu, která obsahuje metadata s konfiguračními parametry nutnými pro Raspbery Pi. Jde zejména o nastavení jádra a architektury.

```
git clone https://github.com/djwillis/meta-raspberrypi.git
```

V kořenovém adresáři yoctoProject je teď nutné nastavit proměnné prostředí. Toho jsem docílil spuštěním skriptu, který byl součástí dříve stažených souborů. Skript nastavení všechny potřebné proměnné, vytvoří nový adresář raspberryPiBuild a zároveň ho nastaví jako aktuální adresář. [26]

```
source oe-init-build-env raspberryPiBuild/
```

Dalším krokem je nastavení sestavovacího procesu, především nastavit cílovou architekturu pro Bitbake. Pro konfiguraci lze použít libovolný textový editor. Konfigurační údaje jsou obsaženy v souboru `conf/local.conf`. Na počítači, který disponuje procesorem s více vlákny, lze nastavit proměnnou BB_NUMBER_THREADS, která toho využije a rozdělí úlohy mezi zadaný počet vláken procesoru, čímž dojde ke snížení celkového času sestavení. Hodnota proměnné je stanovena tak, že k počtu vláken procesoru přičtu 1. Při zadávání musí být všechny proměnné v uvozovkách. [26]

```
BB_NUMBER_THREADS = "9" # pro 8 vlaken procesoru
```

Jedná-li se o vícejádrový procesor, lze odkomentovat proměnnou PARALLEL_MAKE, která přijímá parametry pro program *make*. Zde zadáme opět v uvozovkách počet jader zvýšený o 1, před kterým stojí *-j*. [26]

```
PARALLEL_MAKE="-j 5" # pro 4 jadra procesoru
```

Proměnná MACHINE obsahuje cílové zařízení, pro které sestavuji daný embedded linuxový systém. V tomto případě bude mít hodnotu "raspberrypi". V konfiguračním souboru je těchto proměnných více pod sebou, avšak pouze jedna smí být odkomentovaná.

Na závěr je nutné schovat nějaké balíčky, v OpenEmbedded nazývané recepies, před nástrojem bitbake z důvodů kompatibility mezi vrstvami meta-yocto/oe-core. Pro tyto účely slouží proměnná BBMASK, která zadané balíčky vymaskuje. Balíčky se mezi sebou oddělují pomocí znaku |. [26]

```
BBMASK = "meta-raspberrypi/recipes-multimedia/libav|meta-raspberrypi/recipes-
core/systemd"
```

To jsou veškeré potřebné úpravy v tomto souboru. Pro úspěšné sestavení je však ještě třeba přidat meta-rasperrypi vrstvu do souboru `conf/bblayers.conf` . Tu přidáme na poslední řádek do proměnné BBLAYERS ve formátu `/cesta_k_adresari/meta-raspberrypi`.

```
BBLAYERS ?= " \
/home/rpi/yocto-project/meta \
/home/rpi/yocto-project/meta-yocto \
/home/rpi/yocto-project/meta-yocto-bsp \
/home/rpi/yocto-project/meta-raspberrypi \
"
```

Toto jsou veškeré nezbytné změny, které je třeba provést v souborech v adresáři yoctoProject. Pro sestavení je však potřeba řada dalších programů, které Bitbake interně používá pro sestavení. Pro sestavení na distibuci Ubuntu jsou třeba následující balíčky: [26]

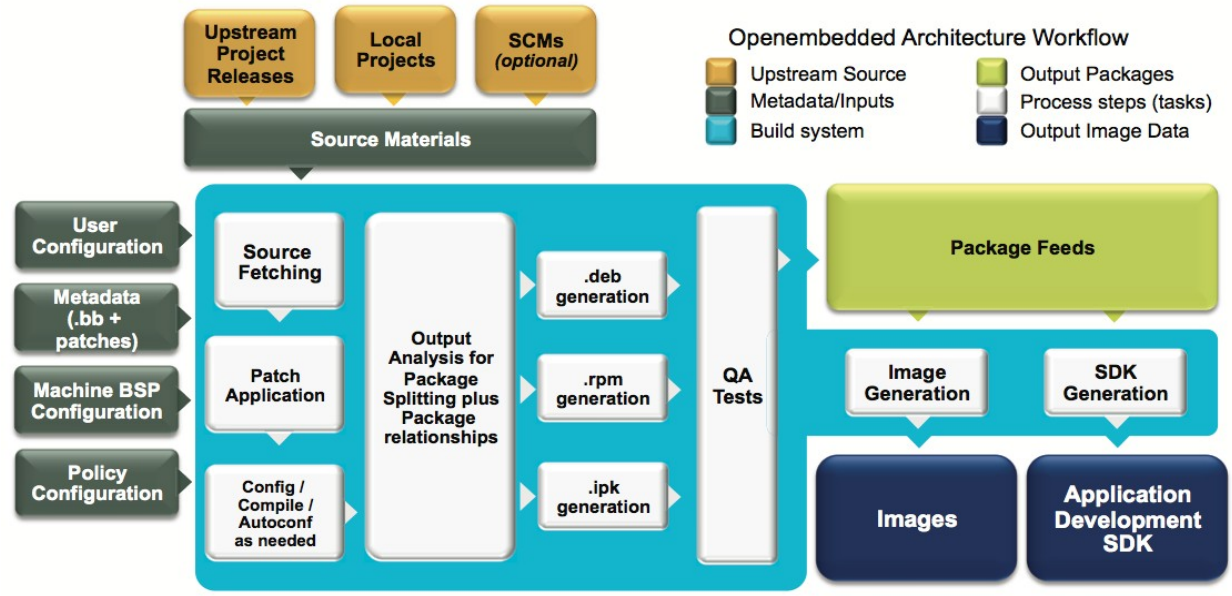
```
sed wget cvs subversion git-core coreutils \
unzip texi2html texinfo libsdl1.2-dev docbook-utils gawk \
python-pysqlite2 diffstat help2man make gcc build-essential \
g++ desktop-file-utils chrpath libgl1-mesa-dev libglu1-mesa-dev mercurial autoconf
automake groff libtool xterm
```

Sestavení embedded linuxové distibuce nyní docílíme zadáním příkazu. Doba trvání sestavení je různá. Závisí především na nastavení výše uvedených proměnných. Na 4 jádrovém procesoru o 4 GB RAM trval tento proces něco málo přes 70 minut.

```
bitbake rpi-basic-image
```

Výsledný obraz je již možno zkopírovat na SDkartu pomocí příkazu. Pro tento příkaz, jako jediný, je nutné oprávnění uživatele root: [26]

```
dd if=tmp/deploy/images/rpi-basic-image.sd-img of=/dev/soubor_zarizeni_SDkarty
```



Obr. 5: Schéma sestavení embedded linuxového operačního systému pomocí Yocto/OpenEmbedded. Zdroj: [26]

4.9 Komunikace s Raspberry Pi po sériové lince

Raspberry Pi, stejně jako většina ostatních novodobých embedded zařízení navržených s důrazem na rozměry, nemá konektor 9 pinový D-Sub konektor typu DE-9 pro připojení sériového rozhraní z důvodu úspor místa, a proto jsou vývody sériová linky umístěny mezi GPIO piny. Linka Tx je k dispozici na pinu GPIO číslo 14 a Rx na GPIO pinu číslo 15 [27]. Standartní UART komunikace obvykle probíhá dle standardu RS-232. Napěťová úroveň pro logickou úroveň 1 je typicky -12 V, pro logickou úroveň 0 je to +12 V. Napětí na GPIO pinech Raspberry Pi je však v rozmezí od 0 V do +3,3 V. Pro komunikaci dle standardu RS-232 je třeba použít převodník napětí. Pro tento účel je vhodné použít integrovaný obvod MAX3232CPE. V katalogovém listu [28] je uvedeno, že pro danou funkci je k obvodu nutné připojit 5 kondenzátorů o kapacitě 0,1 μF. Pro připojení k dnešním počítačům, které nemají sériový port, je ještě třeba použít adaptér pro převod signálů z RS-232 na USB a opačně. Na počítači následně jen stačí spustit program pro komunikaci po sériové lince jako např. minicom a nastavit komunikaci na 8bitů, 1 stop bit, bez paritního bitu a hardwarového řízení. Přenosová rychlost jednotky UART je defaultně nastavena na 115200 Bd. Lze ji změnit v souboru `/boot/cmdline.txt`, změnou položky `console=ttyAMA0,115200`. Hodnotu 115200 nahradíme požadovanou hodnotou přenosové rychlosti. Toto nastavení se však týká jen jádra. Pro

komunikaci po inicializaci jádra ještě třeba nastavit i programu getty, který následně čeká na sériové lince na příchozí spojení. Pro nové příchozí spojení následně spustí program login, který má nastarosti přihlášení do systému. Potřebné konfigurační údaje se nacházejí v souboru `/etc/inittab` na řádce 2:23:respawn:/sbin/getty -L ttyAMA0 115200 vt100, kde opět stačí zaměnit hodnotu 115200**.** [23]

Po vložení SD karty s obrazem embedded linuxového operačního systému do slotu na desce Raspberry Pi a připojení napájení přes konektor USBmicro, by se měl spustit proces načtení operačního systému. Výpis tohoto procesu lze sledovat buď na monitoru, který je k Raspberry Pi připojen pomocí HDMI nebo přes sériovou linku v okně programu minicom. Ve výpisu lze vidět informace o inicializaci samotného jádra a následně poté i bootovacích skriptů. Po jejich skončení by se mělo objevit přihlášení do systému a následný prompt či grafické prostředí pro systém sestavený pomocí Yocto Project.

4.10 Tvorba základních aplikací pro embedded zařízení

Nyní, když je embedded linuxová distribuce sestavená, její obraz nahraný do nevolativní paměti typu NAND-FLASH na SD kartě a její funkčnost na cílové platformě je ověřena, je načase začít se zabývat tvorbou programů, které budou funkční na cílové platformě v uživatelském prostoru. Podobně jako při sestavování linuxové distribuce máme na výběr ze 2 možností, nativní či křížová kompilace.

V případě jádra, systémových knihoven a dalších důležitých systémových součástí je volba víceméně jasná, neboť se skládají z velkého množství zdrojových a hlavičkových souborů s velkým množstvím závislostí a pravidel. To klade vysoké požadavky na výkon CPU a kapacitu RAM paměti, což je většinou limitující parametr embedded systémů; sestavení by trvalo neúměrně dlouhou dobu a embedded systém by na něj spotřeboval veškeré systémové prostředky. [4]

Při sestavování programů pro běh v uživatelském režimu již má smysl uvažovat o nativní kompilaci. Tu lze ocenit obzvláště v případě, že vývojář nemá k dispozici křížovou verzi kompilátoru pro danou platformu, ale na embedded systému se nachází nativní verze kompilátoru. S výhodou jej lze použít pro sestavení nenáročného programu pro uživatelský prostor či modulu pro jádro, kde ušetří čas sestavování křížových nástrojů. Na většině embedded systémů se však nativní kompilátor nenachází (např. z důvodu úspor místa), a proto je třeba použít křížovou verzi. [4]

Křížová kompilace má řadu výhod oproti nativní. Jednou z nich je, že hostitelské zařízení má většinou výkonnější CPU, více paměti RAM a úložného prostoru pro zdrojové a výsledné spustitelné soubory. Křížové nástroje nabízejí také snazší debuggování embedded systémů, neboť jej lze provádět při vývoji i později při nasazení v praxi, čehož je dosaženo pomocí nativní verze programu *gdbserver*, která se

přeloží křížovým kompilátorem a nahraje do zařízení. Vývojář se následně připojí k zařízení vzdáleně (např. po Ethernetu) a spustí v programu *gdbserver* problémovou aplikaci. Na svém lokálním počítači si spustí křížovou verzi *gdb* a připojí se k běžícímu programu *gdbserver.* Výhodou tohoto řešení oproti nativnímu vzdálenému debuggování je, že na cílovém zařízení stačí mít menší verzi aplikace bez ladicích symbolů. Aplikace s ladicími symboly poběží na křížové verzi *gdb*, který kromě toho ještě bude mít k dispozici více systémových prostředků než nativní verze *gdb* na embedded systému. [4]

Jednou z metod, jak se „vyhnout" problému s nativními či křížovými nástroji pro jazyk C/C++, je použít multiplatformní jazyk Java, případně skriptovací jazyk typu Python, Perl či Bash. Výhodou jazyka Java je, že po přeložení do bytekódu je jej možno pustit na jakékoliv platformě, pro kterou existuje JVM. S jazyky Python a Perl je to podobné, oba potřebují interpret příkazů, avšak nekompilují se. Tato výhoda však vyžaduje více systémových prostředků, a proto ji nelze využít v RT systémech. Trochu jiná situace je u Bash, který poskytuje shell v embedded systému, a proto ho lze s výjimkou RT aplikací použít. [4]

Při vývoji aplikací je též důležité jejich nasazení na cílovém zařízení, případně i zisk dat a jejich přehledná správa. V závislosti na propojení a podporovaných přenosových rychlostech existují různá řešení v závislosti na komfortu vývojáře, toku dat, latenci, atd. Tím nejjednodušším řešením je výpis adresáře pomocí příkazu *ls* a následné zkopírování požadovaného souboru pomocí příkazu *scp*. Elegantnějším řešením je použití FTP serveru a klienta. Nevýhodou však je, že klient i server musí běžet na obou zařízeních pro obousměrnou výměnu dat, což v případě embedded zařízení na sdílení dat nemusí být na škodu. Nejpohodlnějším řešením je však použití jednoho ze síťových souborových systémů, které umožňují transparentní práci se soubory v daném adresáři. U tohoto řešení se ale mohou vyskytnout problémy s latencí, které lze minimalizovat používáním cache souborů. Existují i jiná řešení, avšak tato bývají implementována ve většině linuxových distribucí a na UNIXových systémech. [4]

4.11 NFS a nastavení sdílení dat v adresáři

V zájmu jednoduchosti a přehlednosti jsem zvolil síťový souborový systém, konkrétně NFS, jehož podporu jsme zkompilovali do jádra. Network File System, zkráceně NFS, je síťový souborový systém a zároveň je též i protokolem pro výměnu dat. Z hlediska architektury se jedná o princip server-klient, což znamená, že hostitelský počítač se v tomto případě stane serverem, který si ve svém souborovém systému zvolí adresář k exportování, nastaví potřebná práva a spustí NFS démona, který poběží na pozadí systému a postará se o všechno potřebné.

Na cílovém zařízení, tedy klientovi, se stačí jen dotázat serveru na exportované adresáře a následně zvolený adresář připojit do souborového systému. Práce s adresářem probíhá na obou zařízeních naprosto transparentně a stejně jako s jinými adresáři. Výhodou tohoto řešení je, že na rozdíl od FTP řešení je potřeba jen jeden server a klient, protože při povolení čtení a zápisu se adresář stane obousměrně sdílený. Data jsou však uložena pouze na serveru, což je výhodné pro embedded systém, neboť to umožňuje použít menší kapacitu paměti pro trvalé uložení dat. (Navíc lze po připojení adresáře spustit skript, který ověří, zda není k dispozici update.) [29]

Pro nastavení NFS je třeba mít nainstalované balíčky `nfs-utils` (server) a `nfs-utils-clients` (klient). Pro nakonfigurování NFS serveru jsou třeba 3 soubory umístěné v adresáři `/etc`.

Prvním je soubor `hosts.allow`, jehož obsahem jsou povolená spojení se syntaxí `PORT:IP_ADRESA`. Do souboru je třeba zapsat `ALL:IP_ADRESA_EMBEDDED_SYSTÉMU`.

Dalším souborem je `hosts.deny`, který je prohledáván systémem, pokud se nenajde shoda v `hosts.allow`. Je-li zde nalezen, pak je spojení zakázáno. Formát je stejný jako v souboru `hosts.allow`.

Posledním souborem je `exports`, který obsahuje pravidla pro exportování adresářů přes NFS. Syntaxe souboru je `ADRESAR_KE_SDILENI IP_ADRESA(VOLBY)`. Nejzajímavější je parametr volby, který nastavíme takto `rw,sync,no_root_squash`. [29]

První volba říká, že do adresáře je povolen i zápis, druhá zajišťuje okamžitou synchronizaci a poslední nastaví vzdálenému uživateli root práva místního uživatele root. Pro zápis do těchto souborů je vyžadována práva uživatele root. Následně je třeba spustit démona NFS. Toho lze dosáhnout jednoduchým příkazem `# service nfs-kernel-server restart`. To je vše na straně serveru.

Na straně klienta je zapotřebí funkčního síťového připojení a znalost IP adresy NFS severu. Následně je třeba se dotázat na exportované adresáře pomocí příkazu `showmount -e IP_ADRESA_NFS_SERVERU` a vybraný adresář připojit do souborového systému pomocí příkazu mount. Syntaxe příkazu mount je následující: `# mount -o nolock -t nfs IP_ADRESA_NFS_SERVERU:EXPORTOVANY_ADRESAR PRIPOJNE_MISTO`

Pro trvalé připojení NFS adresáře je o něm třeba zapsat informace i do `/etc/fstab`. [29]

4.12 Testování aplikací na PC pomocí QEMU

Pro testování celého systému či jednotlivých aplikací, jejichž ukázky lze najít níže, lze použít virtualizační nástroj QEMU, který umožní emulovat cílový procesor. Lze tak spustit program přeložený pro jeden typ procesoru (např. ARM) na jiném typu procesoru (např. x86 v PC). Díky použití dynamického překladu je tak možné dosáhnout velmi dobrých výkonnostních výsledků.

Následující příkaz demonstruje použití pro systémovou emulaci námi vytvořeného embedded systému. Jádro je třeba nejprve zkopírovat z adresáře `/boot` na nějaké místo v souborovém systému počítače, na kterém se bude emulace provádět. Po zadání příkazu systém naběhne stejně jako na daném hardwaru. [30]

```
qemu-system-arm -kernel $KERNEL -cpu arm1176 -m 256 -M versatilepb -no-reboot
-serial stdio -append "root=$ROOT panic=1" -hda $IMAGE
```

Význam jednotlivých parametrů [30]:

- kernel … udává cestu k jádru emulovaného operačního systému
- cpu … udává architekturu a typ emulovaného procesoru
- m … udává velikost operační paměti v MB
- M … specifické nastavení virtuálního hardwaru
- no-reboot … při ukončení emulovaného operačního systému ukončit program QEMU
- serial … udává vstupní zařízení pro emulovaný operační systém
- append … textový řetězec předaný jako argument jádru při bootování, v systému si jej lze přečíst v /proc/cmdline
- root … udává oddíl s kořenovým souborovým systémem, které jádro během inicializace připojí

- panic … udává dobu v sekundách, po které se jádro rebootuje, dojde-li ke stavu kernel panic
- hda … udává pevný disk či oddíl s kořenovým souborovým systémem pro program QEMU

4.13 Tvorba jednoduchého programu typu "HELLO WORLD"

Nyní, když je zajištěno sdílení souborů mezi cílovým zařízením a hostitelským počítačem, je načase sestavit první program. Postup bude velmi podobný jako při sestavování OS a jeho součástí.

Nejprve jsem napsal program, následně pro něj vytvořil soubor makefile, sestavil jej, přenesl na cílový systém přes NFS a vyzkoušel se jej vzdáleně debuggovat. Výsledný spustitelný program pro architekturu ARM se bude jmenovat HelloWorld-stripped a bude se jednat o soubor bez ladicích informací a několika dalších sekcí nezmenšeného programu HelloWorld, který přijme jeden celočíselný argument pro funkci main. Tento argument určuje, kolikrát proběhne for cyklus, který na obrazovku vypíše text "Hello World, ARM!" tolikrát, kolikrát proběhne onen cyklus. Ten se následně ukončí s návratovou hodnotou daným makrem `EXIT_SUCCESS`, která má hodnotu 0. Pokud není zadán přesně jeden argument, pak program vrátí hodnotu -1 danou makrem `EXIT_FAILURE`.

Obsah zdrojového souboru main.c:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
if (argc!=2)
return EXIT_FAILURE;
int i=0;
for(;i<argv[1];i++)
printf("Hello World,ARM!");
return EXIT_SUCCESS;
}
```

Pro programy skládající se z jednoho zdrojového souboru se většinou nepíše makefile, ale pro účely této práce jsem udělal výjimku. Důvodem je to, že se jedná o křížovou kompilaci a že požaduji vytvoření zmenšené verze. Navíc lze udělat pravidlo, které smaže objektové soubory, které vznikly při procesu make.

Každý soubor makefile se skládá ze dvou částí, v první jsou obsaženy definice proměnných a v druhé jednotlivá pravidla. Výhodou souboru makefile je, že proces sestavení se stane jednoduchý, a tudíž jej

může každý replikovat, a to i bez hlubších znalostí daného programu. Proměnná NFS udává adresář, který je sdílený přes NFS.

Obsah souboru Makefile:

```
# Definice promennych
CC=$(CROSS_PREFIX)gcc
CFLAGS=-Wall
OBJ=main.o
TARGET=helloworld
NFS=/mnt/NFS            # Nutne zmenit pokud nesouhlasi
STRIP=strip
STRIP_FLAGS=-s -R .comment

# Jednotliva pravidla
all: $(TARGET)

$(TARGET): $(OBJ)
$(CC) $(CFLAGS) $(DEBUG) -o $(TARGET) $(OBJ)

#Pravidlo pro vytvoreni verze pro embedded nasazeni
strip: $(TARGET)
$(STRIP) $(STRIP_FLAGS) -o $(TARGET)-stripped $(TARGET)
# Sablona pro pravidlo
%.o: %.c
$(CC) $(CFLAGS) $(DEBUG) -c -o $@ $<
deploy:
      cp $(TARGET)-stripped $NFS
clean:
rm -f *.o
#Vylouceni kolize se soubory se stejnymi nazvy
.PHONY: clean
.PHONY: dep
.PHONY: strip
.PHONY: deploy
```

Pro sestavení programu stačí jen spustit příkaz *make*. Jelikož jsem do příkazu *make* nezadal pravidlo, vykoná pravidlo *all*, což je defaultní pravidlo, případně první pravidlo v pořadí, není-li pravidlo *all* nalezeno. Následně jsem ještě jednou zavolal *make*, avšak tentokrát s pravidlem *clean*, které smaže nepotřebné soubory. Při zobrazení výpisu adresáře, příkaz *ls -l*, bychom měli získat následující seznam souborů: [31]

```
helloworld
helloworld-stripped
main.c
makefile
```

Pro kontrolu výsledných typů souboru lze použít nástroj *file*, jenž vypíše následující informace o souborech:

```
helloworld: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 3.7.3, not stripped
helloworld-stripped: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 3.7.3, stripped
main.c: ASCII C program text
makefile: ASCII text
```

Z výpisu je vidět, že se jedná o spustitelné soubory pro architekturu ARM 32-bitů ve formátu ELF, ke kterým jsou dynamicky přilinkovány knihovny. Před spuštěním je ještě třeba nakopírovat zmenšený spustitelný soubor do adresáře, který je sdílen přes NFS. Toho lze dosáhnout opětovným zavoláním *make* s cílem *deploy*, který zavolá příkaz *cp*, jenž překopíruje požadovaný soubor. [32]

Následně je třeba se připojit na cílové zařízení pomocí SSH. Syntaxe je `ssh IP_ADRESA_CIL_ZARIZENI -l LOGIN`. Při prvním spuštění je třeba si uložit SSH fingerprint, který jednoznačně identifikuje cílový systém, a zadat heslo pro přihlášení. Následně je třeba přes příkaz *cd* změnit aktuální adresář na NFS adresář, kde se nachází spustitelný program, a pomocí *ls -l* zkontolovat, zda má soubor právo spustitelnosti – x. Pokud ne, pak je nutné jej nastavit zavoláním `chmod ugo+x helloworld-stripped`, které nastaví vlastníkovi, skupině a ostatním právo spustit soubor. Spuštění jsem provedl jednoduše pomocí `./helloworld N`, kde N je počet opakování for cyklu. Bude-li menší nebo rovno než 0, pak se nevypíše nic.

Práce vývojáře zde však nekončí, neboť tento program poskytuje dokonalou příležitost vyzkoušet si vzdálené ladění pomocí programu *gdbserver.* Jak již svým názvem naznačuje, *gdbserver* naslouchá na

daném TCP portu síťového rozhraní na příchozí spojení od *gdb*, které lze navázat i přes loopback IP adresu, a po navázání spojení vykonává jeho příkazy nad debuggovanou aplikací. [32]

Gdbserver jsem spustil s parametry `gdbserver :2001 helloworld-stripped N`, kde 2001 udává TCP port a nezadaná IP adresa před dvojtečkou znamená, že budu naslouchat na všech síťových rozhraních. Důležité je předat aplikaci argumenty již zde, neboť později již to není možné.

Na hostitelském počítači jsem spustil křížovou variantu *gdb* a jako argument jí předal nezmenšenou aplikaci. V příkazovém řádku *gdb* jsem zadal `remote target IP_ADRESA_CIL_ZARIZENI:2001` , což způsobí napojení se na *gdbserver* na cílovém zařízení. Nyní lze aplikaci ladit téměř jako lokální, jen s tím rozdílem, že latence je mnohonásobně větší, což je dáno síťovým připojením, křížovou verzí *gdb* a dále pomalejším procesorem na cílovém zařízení. Výstup aplikace se ovšem zobrazuje na konzoli se spuštěným programem *gdbserver*, nikoliv v *gdb*, a proto se text "Hello World, ARM!" zobrazí v konzoli, která je připojena k cílovému zařízení přes SSH a na níž běží *gdbserver*. [32]

4.14 Tvorba jednoduchého jaderného modulu

Jaderný modul je speciální typ spustitelného kódu, který lze dle potřeby nahrát či odebrat z jádra. Jeho účelem je rozšířit funkcionalitu jádra o nové funkce bez nutnosti překompilovat jádro a rebootovat celý systém. Ve většině případů jde o ovladače nově připojených zařízení, které by zbytečně zabíraly cenné systémové zdroje, pokud by zařízení nebylo trvale připojeno. Použití modulů je však podmíněno tím, že jádro musí být zkompilováno s podporou pro nahrávání a odebírání jaderných modulů. Pro práci se samotnými jadernými moduly jsou pak ještě potřeba nástroje, které zařídí patřičnou operaci s nimi. Mezi další činnosti patří též sestavení závislostí mezi moduly. Pro sestavení jaderného modulu se používá systém kbuild, který spolu s nástrojem make sestaví námi požadovaný modul z zdrojového kódu napsaného v jazyce C. [1]

Existuje několik druhů jaderných modulů:

- správa procesů
- správa paměti
- souborové systémy
- ovladače zařízení
- síťové služby

Pro všechny je však společné, že běží v jaderném prostoru, což je základní odlišnost od uživatelských aplikací, které běží v uživatelském prostoru. Oproti uživatelským aplikacím tedy nemají přístup ke knihovně jazyka C a jejím funkcím. To se týká i ostatních systémových knihoven.

Na rozdíl od uživatelského prostoru je zde k dispozici mnohem menší zásobník, velikost je typicky kolem 4kB. S operační pamětí je nutné zacházet opatrněji, neboť každý alokovaný byte je nutné po provedení potřebných operací před ukončením modulu opět uvolnit, neboť zde není žádný mechanizmus, který by toto provedl za programátora. Záznam o alokované paměti zůstane zapamatován jádrem, tudíž tuto paměť není možné až do dalšího restartu systému alokovat. [1]

Programy v jaderném prostoru mohou zasahovat do jakékoliv části paměti, tudíž lze snadno zapsat data na nesprávnou adresu, bez typického varování Segmentation fault, které se objeví při podobné operaci v uživatelském prostoru. Typickou programátorskou chybou je zde chyba o jedničku, zejména při práci s polem proměnných. Kritické sekce kódu musí být uzavřeny v blocích, které zaručí atomické vykonání kódu. Typickým příkladem je modifikace proměnné, která způsobí deadlock. Je tedy zřejmé, že problematika tvorby jaderných modulů je velmi široká a že klade vysoké nároky na programátora**.** [1]

Otázkou je tedy, co vlastně jaderné moduly nabízí oproti řešení pomocí démonů v uživatelském prostoru. Především jde o to, že ne všechny operace lze provádět v uživatelském režimu, dokonalým příkladem je např. správa procesů. Příkladem řešení v uživatelském prostoru je například démon pro ovládání frekvence CPU. Ten umožňuje uživateli ručně či dle zvoleného algoritmu nastavovat požadovanou frekvenci CPU. Jiným příkladem je souborový systém FUSE, který umožňuje vytvořit programové řešení souborového systému v uživatelském prostoru. Odměnou za větší latence a degradaci výkonu je to, že chyba v ovladači nezpůsobí zamrznutí sytému, ale v lepším případě jen pád programu, v horším případě i ztrátu dat. [1]

Základem každého jaderného modulu jsou makra *module_init()* a *module_exit()*. Obě přijímají ukazatele na funkce, které musí být statické. První makro je voláno při načítání modulu a musí vracet 0 datového typu *int*, jinak načítání skončí neúspěšně. Druhé makro slouží k zavolání funkce při jeho odebírání. Tato makra jsou definována v hlavičkových souborech `linux/init.h` a `linux/module.h`, které lze nalézt v `/usr/include`. Tento adresář mimo jiné obsahuje hlavičkové soubory pro knihovní funkce, které však z výše zmíněných důvodů nelze použít. Pro komunikaci s uživatelem tedy nelze použít funkci *printf()* pro výpis informací na obrazovku. Ekvivalentem funkce *printf()* je v prostoru jádra funkce *printk()*, která předaná data zapisuje do kruhového bufferu jádra. Ten je již dostupný z uživatelského prostoru, odkud si jej lze přečíst. [1]

Obsah souboru Makefile:

```
# Nazev sestavovaneho jaderneho modulu
obj-m += hello.o
# Adresar s hlavickovymi soubory jadra, pro ktere modul kompilujeme
KDIR=/usr/src/linux-headers-3.5.0-23-generic

all:
$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
rm -rf *.o *.ko *.mod *.symvers *.order
```

Obsah souboru hello.c:

```
#include <linux/init.h> // Hlavickove soubory pro inicializaci modulu
#include <linux/module.h> // Oznami jadru, ze se jedna o jaderny modul

// Vstupni funkce
static int hello_init (void){
printk(KERN_ALERT "TEST: Hello world, this is first linux module!\n");
return 0;
}
//Ukoncovaci funkce
static void hello_exit(void){
printk(KERN_ALERT "TEST: Good bye, from cylon2p0.\n");
}

// Makro definujici vstupni funkci, ta je predana jako ukazatel
module_init(hello_init);
// Makro definujici vystupni funkci, ta je predana jako ukazatel
module_exit(hello_exit);
```

Stejně jako u předchozích kompilací stačí v daném adresáři jen spustit nástroj *make*, který spolu s nástrojem *Kbuild* sestaví námi požadovaný modul. Výsledkem je soubor s koncovkou **.ko*. Pro načtení modulu do jádra je nutné zavolat příkaz *insmod* a jako argument mu předat nově vytvořený modul, i s

koncovkou a případnou cestou, pokud není v aktuálním adresáři. Zda je modul zavedený do jádra, lze zjistit z výpisu modulů aktuálně zavedených do jádra pomocí příkazu *lsmod*. Zprávu lze přečíst z kruhového buferu pomocí šikovného nástroje *dmesg*, jehož výstup lze ještě předat příkazu *grep* TEST, který vypíše pouze řádek obsahující textový řetězec TEXT. Pro odebrání modulu slouží příkaz rmmod, jehož parametrem je název modulu, tentokrát bez přípony **.ko*. [1]

Dle tohoto postupu lze vytvořit modul, který se bude starat o komunikaci s alfanumerickým LCD displejem či komunikovat po sběrnici SPI a podobně pomocí GPIO pinů. Takovýto modul již však vyžaduje hlubší znalosti a složitější zdrojový kód. Pro interakce mezi uživatelským a jaderným prostorem slouží pseudosouborový systém *sysfs*, připojovaný do adresáře `/sys`, do kterého jádro exportuje proměnné a data formou souborů.

4.15 Měření rychlosti GPIO pinů

Měření rychlosti GPIO pinů bylo provedeno pomocí digitálního osciloskopu Picoscope s výstupem do počítače přes USB. Na testovaném zařízení - Raspberry Pi - byl spuštěn program, který neustále přepínal logický stav na GPIO pinu číslo 4. Na tento pin byla připojena jedna sonda osciloskopu, druhá byla spojena se zemí. Cílem měření bylo zjistit maximální frekvenci generovaného obdélníkového signálu. Měření bylo provedeno pro několik programovacích a skriptovacích jazyků. [33]

Tab. 2: Naměřené frekvence na výstupním GPIO pinu číslo 4 pro jednotlivé jazyky. Zdroj: [33]

Jazyk	**Knihovna**	**Frekvence**
Shell	/proc/mem access	3.4 kHz
Python	RPi.GPIO	44 kHz
Python	wiringPi	20 kHz
C	Nativní knihovna	14-22 MHz
C	BCM 2835	4.7 – 5.1 MHz
C	wiringPi	6.9 – 7.1 MHz
Perl	BCM 2835	35 kHz

Ukázka programu pro měření maximální frekvence obdélníkového signálu: Zdroj:[33]

```
// Set GPIO pin 4 to output
INP_GPIO(4); // must use INP_GPIO before we can use OUT_GPIO
OUT_GPIO(4);

while(1) {
  GPIO_SET = 1<<4;
  GPIO_CLR = 1<<4;
}
```

Z tabulky je patrné, že pro skriptovací jazyky byla frekvence typicky v řádech kHz, kdežto u programovacích jazyků typu C bylo dosaženo až několik jednotek MHz. Nejvyšší frekvence bylo dosaženo při použití nativní knihovny v jazyce C s optimalizačním příznakem kompilátoru -03. Naměřené hodnoty v této tabulce odpovídají nezatíženému procesoru, při zatížení by se jednalo o zlomky uvedené frekvence. [33]

5. Závěr

Pro tuto práci byly stanoveny následující cíle:

- Zmapování dostupných zařízení na trhu použitelných pro embedded aplikace s jádrem operačního systému Linux
- Následně na příkladu navrhnout a prakticky ukázat postup tvorby operačního systému pro jedno vybrané embedded zařízení, též jeho následnou instalaci a ověření funkcionality
- Na několika jednoduchých názorných příkladech ukázat tvorbu základních programů pro cílové zařízení

Před samotným zmapováním jsem detailně popsal embedded systémy a operační systém s jádrem Linux. Následně jsem představil vybraná dostupná zařízení na trhu.

Detailně jsem popsal tvorbu embedded linuxového systému pro zařízení Raspberry Pi pomocí využití dvou možných postupů, z nichž oba byly úspěšné. Kritériem bylo spuštění nově vytvořeného operačního systému na daném zařízení, což se v obou případech podařilo.

Postup manuálního sestavení krok za krokem je vhodný především pro zkušeného vývojáře, který ví jak postupovat a jaký krok lze nahradit či pozměnit. Celý postup lze následně sumarizovat do skriptu/ů a následně jej již volat automaticky.

Vývoj pomocí Yocto/OpenEmbedded je vhodný především pro začínající vývojáře, neboť tento postup nabízí jednoduché konfigurační soubory a flexibilní nástroj Bitbake, který se o samostatné sestavení výsledného obrazu systému postará sám. Vývojář se tedy jen věnuje nastavení jednotlivých balíčků a případně i optimalizaci nástroje Bitbake. Výhodou je paralelní zpracování celého procesu a následné možné použití již sestavených souborů v jiném projektu, díky členění na jednotlivé vrstvy.

Pro splnění posledního z cílů jsem nejprve popsal nezbytně nutné nastavení vývojového prostředí a následně připravil dvě ukázky jednoduchých programů, první pro uživatelský prostor a druhou pro jaderný prostor.

Výstupem této práce je tedy použitelný embedded linuxový operační systém, který lze nasadit v praxi a kterému lze vhodnými programy navýšit funkcionalitu.

Zdroje

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

2. JELÍNEK, Lukáš. Vytváříme vlastní distribuci Linuxu: od návrhu po fungující systém. Vyd. 1. Brno: Computer Press, 2010, 304 s. ISBN 978-80-251-2433-8.

3. PINKER, Jiří. Mikroprocesory a mikropočítače: od návrhu po fungující systém. 1. vyd. Praha: BEN - technická literatura, 2004, 159 s. ISBN 80-730-0110-1.

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

5. Embedded-lab: Overview of an Embedded System. [online]. [cit. 2013-06-04]. Dostupné z http://embedded-lab.com/blog/?p=949

6. Zařízení s Linuxem. [online]. [cit. 2013-06-04]. Dostupné z http://www.root.cz/galerie/deset-zarizeni-na-ktera-se-podarilo-dostat-linux/

7. Beagle Bone. [online]. [cit. 2013-06-04]. Dostupné z http://beagleboard.org/bone

8. PlayMag: Rasberry Pi se prodalo přes jeden milion kusů, [online]. [cit. 2013-06-04]. Dostupné z http://playmag.cz/raspberry-pi-se-prodalo-pres-jeden-milion-kusu/

9. Rasberry Pi. [online]. [cit. 2013-06-04]. Dostupné z http://www.raspberrypi.org/

10. ELinux. [online]. [cit. 2013-06-04]. Dostupné z http://elinux.org/RPi_Hardware

11. Android Market. [online]. [cit. 2013-06-04]. Dostupné z http://www.androidmarket.cz/hardware-2/udoo-naslednik-raspberry-pi-a-arduina-due/

12. UDOO. [online]. [cit. 2013-06-04]. Dostupné z http://www.udoo.org/features/

13. CubieBoard. [online]. [cit. 2013-06-04]. Dostupné z http://cubieboard.org/

14. Olimex: OLinuXino. [online]. [cit. 2013-06-04]. Dostupné z https://www.olimex.com/Products/OLinuXino/

15. Olimex: OlinuXino features . [online]. [cit. 2013-06-04]. Dostupné z https://www.olimex.com/Products/OLinuXino/iMX233/iMX233-OLinuXino-MAXI/

16. Acmesystems: Aria. [online]. [cit. 2013-06-04]. Dostupné z http://www.acmesystems.it/aria

17. Thegeekstuff: Linux file systém structure. [online]. [cit. 2013-06-04]. Dostupné z http://www.thegeekstuff.com/2010/09/linux-file-system-structure/

18. uCSimply: Nativní nebo křižové nástroje. [online]. [cit. 2013-06-04]. Dostupné z http://www.ucsimply.cz/elnx/nativni-nebo-krizove-nastroje/

19. uCSimply: Křížový kompilátor. [online]. [cit. 2013-06-04]. Dostupné z http://www.ucsimply.cz/elnx/kompilator-gcc/krizovy-kompilator/

20. LinuxFromScratch. [online]. [cit. 2013-06-04]. Dostupné z http://www.linuxfromscratch.org/lfs/

21. Cross-LFS: Packages. [online]. [cit. 2013-06-04]. Dostupné z http://cross-lfs.org/view/CLFS-2.0.0/x86/materials/packages.html

22. Crosstool-ng [online]. [cit. 2013-06-04]. Dostupné z http://crosstool-ng.org/

23. Elinux: Rpi Kernel Compilation. [online]. [cit. 2013-06-04]. Dostupné z http://elinux.org/RPi_Kernel_Compilation

24. uCSimply: Busybox. [online]. [cit. 2013-06-04]. Dostupné z http://www.ucsimply.cz/elnx/sestaveni-linuxove-distribuce/zaklad-distribuce/busybox/

25. LinuxFromScratch: Iana-etc. [online]. [cit. 2013-06-04]. Dostupné z http://www.linuxfromscratch.org/lfs/view/6.7/chapter06/iana-etc.html

26. YoctoProject. [online]. [cit. 2013-06-04]. Dostupné z https://www.yoctoproject.org/docs/current/yocto-project-qs/yocto-project-qs.html

27. Elinux: Low level peripherals. [online]. [cit. 2013-06-04]. Dostupné z http://elinux.org/RPi_Low-level_peripherals

28. Datasheetcatalog: MAX3232CPE. [online]. [cit. 2013-06-04]. Dostupné z http://www.datasheetcatalog.com/datasheets_pdf/M/A/X/3/MAX3232CPE.shtml

29. Linux: NFS. [online]. [cit. 2013-06-04]. Dostupné z http://linux.die.net/man/5/nfs

30. Qemu: ARM System emulator. [online]. [cit. 2013-06-04]. Dostupné z http://qemu.weilnetz.de/qemu-doc.html#ARM-System-emulator

31. UC Simply: Make a Makefile. [online]. [cit. 2013-06-04]. Dostupné z http://www.ucsimply.cz/elnx/make-a-makefile/

32. UC Simply: Debugger GDB. [online]. [cit. 2013-06-04]. Dostupné z http://www.ucsimply.cz/elnx/debugger-gdb