České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Jiří Vaněk**

Studijní program: Otevřená informatika (magisterský)
Obor: Počítačové inženýrství

Název tématu: **CAN - USB převodník pro integraci do operačního systému GNU/Linux**

Pokyny pro vypracování:

1. Připravte krátký přehled vhodných mikroprocesorových obvodů pro realizaci převodníku USB ↔ CAN s dostatečně velkou vyrovnávací pamětí. Soustřeďte se především na levnější a menší obvody s architekturou ARM, které mají dostatečné množství paměti RAM a FLASH.
2. Seznamte se s návrhem hardware realizovaným společností PiKRON, který je založený na mikrokontroléru LPC1768 a implementujte firmware pro převodník s mikrokontrolérem LPC1768 s využitím zdrojových kódů projektu LinCAN.
3. Otestujte součinnost implementace s ovladačem pro USB zařízení LinCAN
4. Vytvořte pro převodník podporu pro současný CAN subsystém Linuxového jádra SocketCAN
5. Otestujte řešení s využitím SocketCAN subsystému a porovnejte vlastnosti s verzí využívající znakový ovladač LinCAN

Seznam odborné literatury:

[1] LPC17xx User manual, Rev. 2, NXP Semiconductors, 19 August 2010
[2] Webové stránky projektu OrtCAN http://ortcan.sourceforge.net/
[3] Webové stránky obdobného projektu pro Windows http://rs.canlab.cz/
[4] Corbet Corbet, Alessandro Rubini, Greg Kroah-Hartman, Linux Device Drivers, 3rd Edition, O'Reilly Media, Inc., 2005, ISBN-10: 0596005903, http://lwn.net/Kernel/LDD3/

Vedoucí: Ing. Pavel Píša, Ph.D.

Platnost zadání: do konce letního semestru 2012/2013

L.S.

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 8. 12. 2011

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky

Diplomová práce

CAN - USB převodník pro integraci do operačního systému GNU/Linux

Bc. Jiří Vaněk

Vedoucí práce: Ing. Pavel Píša, Ph.D.

Studijní program: Otevřená informatika, Navazující magisterský

Obor: Počítačové inženýrství

11. května 2012

Poděkování

Chtěl bych poděkovat zejména vedoucímu mé práce Ing. Pavlu Píšovi, Ph.D. za užitečné rady, připomínky a konzultace. Dále bych chtěl poděkovat své rodině a přítelkyni za podporu v průběhu mé práce.

Prohlášení

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

V Praze dne 11. 5. 2012 .....

Abstract

This diploma thesis implements the firmware for delivered hardware device that represents the CAN-USB converter. The firmware uses mechanisms from the LinCAN project. The work also includes design and implementation of the driver for the operating system GNU/Linux, which is based on SocketCAN, the current CAN Linux kernel subsystem. The implemented firmware is tested with both the character device driver LinCAN and the implemented driver using SocketCAN. There is also a comparison of these two drivers.

The work also includes a brief overview and analysis of suitable microprocessors based on ARM architecture, which are suitable for the implementation of the CAN-USB converter.

Abstrakt

Diplomová práce implementuje firmware pro dodané hardwarové zařízení představující převodník CAN-USB. Firmware využívá mechanismy projektu LinCAN. Práce dále obsahuje návrh a implementaci ovladače pro operační systém GNU/Linux, který je založen na SocketCAN, což je současný CAN subsystém Linuxového jádra. Realizovaný firmware je otestován jak se znakovým ovladačem LinCAN, tak s realizovaným ovladačem využívajícím SocketCAN a je uvedeno srovnání obou ovladačů.

Součástí práce je i stručný rozbor a přehled vhodných mikroprocesorových obvodů založených na architektuře ARM, které jsou vhodné pro realizaci CAN-USB převodníku.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

CAN (Controller Area Network) je komunikační sběrnice široce rozšířená v automobilových systémech a v průmyslových řídicích odvětvích obecně. Nejčastěji komunikuje prostřednictvím dvojice vodičů a zjednodušuje tak rozvody. Poskytuje deterministický způsob přístupu k médiu a je tak vhodná i pro použití v časově kritických aplikacích. CAN představuje spolehlivé, levné, osvědčené a dobře známé síťové řešení. Díky těmto nesporným kvalitám je nepravděpodobné, že bude nasazování CAN v dohledné době omezeno a to i přes to, že jsou k dispozici moderní řešení jako FlexRay či různé standardy pro průmyslový Ethernet.

Často je nutné ke sběrnici CAN připojit počítač pro účely konfigurace, monitorování nebo diagnostiky. To lze realizovat např. pomocí speciálních přídavných karet pro PCI rozhraní. Pro snadné a levné připojení počítače ke sběrnici CAN je však vhodný CAN-USB převodník. Rozhraním USB jsou v dnešní době vybaveny téměř veškeré osobní počítače. Naopak instalace speciální karty např. pro notebook často není ani možná.

Práce po stručném představení sběrnic USB a CAN přináší informace o architukturе ARM. Stěžejní je přehled současné nabídky mikroprocesorových obvodů založených na této architektuře a také výběr vhodného řešení pro realizaci CAN-USB převodníku. Je také k dispozici přehled nabídek od několika výrobců.

Následuje část týkající se úkolu implementace firmware pro CAN-USB převodník s mikrokontrolérem LPC1768, který využívá zdrojových kódů projektu LinCAN. Po analýze tohoto úkolu následuje popis zprovoznení itegrovaného CAN řadiče mikrokontroléru a dále popis začlenění funkcionality do struktury LinCAN.

Nejrozsáhlejší část tvoří nejdříve popis současného CAN subsystému Linuxového jádra SocketCAN a poté popis vývoje ovladače pro převodník. Tato část zahrnuje seznámení s principem ovladačů v Linuxu a jejich implementace jako jaderných modulů. Následuje několik obecných věcí týkajících se USB ovladačů v jádru Linux a pak již kompletní popis vývoje síťového ovladače CAN-USB převodníku založeného na SocketCAN.

Práci uzavírá kapitola věnovaná testování, kde je zhodnocena součinnost implementace firmware jak se znakovým ovladačem LinCAN, tak s vytvořeným síťovým ovladačem založeném na SocketCAN. Také je názorně předvedeno porovnání obou řešení.

Kapitola 2

Sběrnice CAN

2.1 Obecné principy

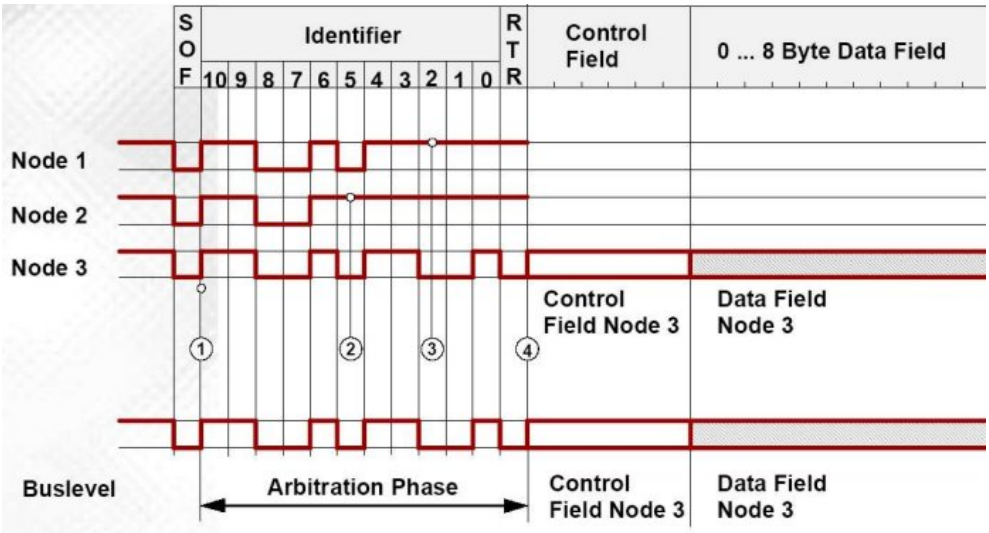
CAN je sériová sběrnice typu *multi-master* představená v roce 1986 Robert Bosh GmbH. Standard CAN definuje pouze spojovou (linkovou) vrstvu tedy řízení přístupu k médiu, kódování a dekódování rámců atd. Varianty fyzické vrstvy (parametry vedení, úrovně signálů, konektory atd.) mohou být různé. Definují je např. standardy ISO11898-2 nebo ISO11898-3.

Základní požadavek na fyzickou vrstvu je podpora přenosu dvou stavů - dominantního a recesivního. Pokud žádný uzel nevysílá dominantní stav, sběrnice zůstává v recesivním stavu. Kdykoliv nějaký uzel vysílá dominantní stav, sběrnice je v dominantním stavu.

Tato vlastnost je používána ve vrstvě přístupu k médiu (MAC) pro dosažení nedestruktivního arbitrážního mechanismu. Ten zajišťuje přístup k médiu zprávě s nejvyšší prioritou bez jakýchkoliv zpoždění. Priorita zprávy je určena jejím identifikátorem, který identifikuje obsah zprávy spíš než adresu vysílače nebo přijímače. Všechny uzly vysílají bity synchronně a při vysílání zprávy u fáze arbitráže uzel kontroluje rozdíl mezi tím co vyslal a tím co skutečně je na sběrnici. Pokud uzel zaznamená rozdíl, odpojí se. Identifikátor zprávy, který je vysílán ve fázi arbitráže je buď 11 nebo 29 bitů dlouhý v závislosti na tom, zda je používán rozšířený formát zprávy. Příklad na arbitráž sběrnice lze vidět na obrázku 2.1 (převzatém z [5][1]). Zde 3 uzly začínají vysílat v jeden okamžik. Arbitráž vyhraje poslední z nich, jelikož jím vysílaná zpráva má identifikátor s nejvyšší prioritou (nejnižší hodnotou).

Vzhledem k tomu, že arbitráž sběrnice vyžaduje synchronní bitový přenos a rychlost šíření signálu vedením je konečná, je omezena délka vedení sběrnice. Pro 125 kbit/s je maximální délka sběrnice je 500 m, pro maximální rychlost 1 Mbit/s to je 30 m [6]. Délka skutečných dat (payload) může být až 8 bytů a každá zpráva je chráněna 15 bitovým CRC. Každá zpráva je potvrzena bitem ACK. Vysílač vysílá tento bit jako recesivní. Všechny přijímače, které korektně přijmou zprávu, nastaví tento bit jako dominantní. Tímto způsobem, když žádný přijímač není je schopen přijmout zprávu korektně, vysílač může signalizovat chybu vyšším vrstvám.

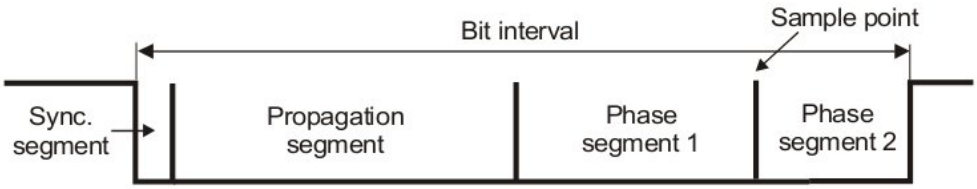
[1] Všechny obrázky v této kapitole o CAN byly převzaty z [5]



Obrázek 2.1: Ukázka arbitráže CAN sběrnice

2.2 Časování komunikace

Všechny uzly v sítí musí mít nastavenu stejnou nominální přenosovou rychlost. Aby toto bylo možné, musí mít každý uzel (resp. jeho řadič) korektně definovanou délku bitového intervalu. Programovatelná dělička (Baudrate prescaler) generuje signál s délkou označovanou jako časové kvatum (time quantum). Interval jednoho bitu je pak složen z celočíselného počtu těchto časových kvant. Ta jsou rozdělena do 4 segmentů, jak lze vidět na obrázku 2.2.

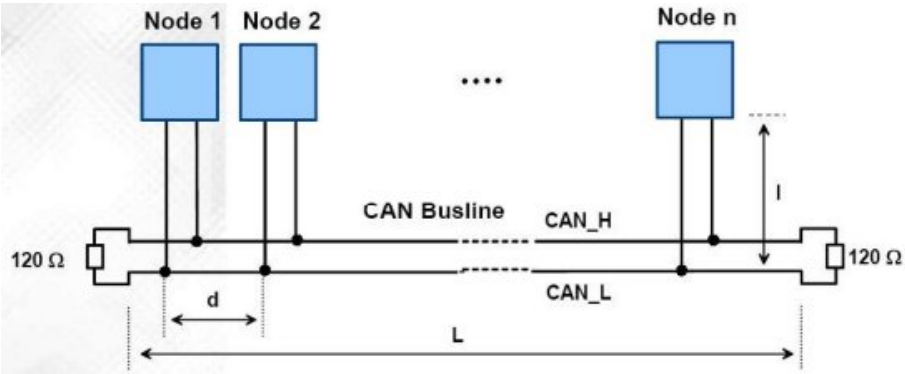


Obrázek 2.2: Časování CAN sběrnice

Synchronizační segment je dlouhý 1 časové kvantum. Propagation segment slouží ke kompenzaci zpoždění mezi uzly. Phase segment 1 a Phase segment 2 určují bod, kde řadič vzorkuje, zda je na sběrnici recesivní či dominantní úroveň. V nastavení řadiče se pak často uvádí pouze součet Propagation segment + Phase segment 1 jako Time segment 1 (tseg1) a Phase segment 2 jako Time segment 2 (tseg2). Kvůli tomu, že skutečné rychlosti se mohou mírně lišit (tolerance oscilátorů), je zaveden tzv. Synchronization Jump Width (sjw). To je počet časových kvant, o který se může prodloužit/zkrátit Phase segment 1 nebo Phase segment 2 k docílení opětovné synchronizace.

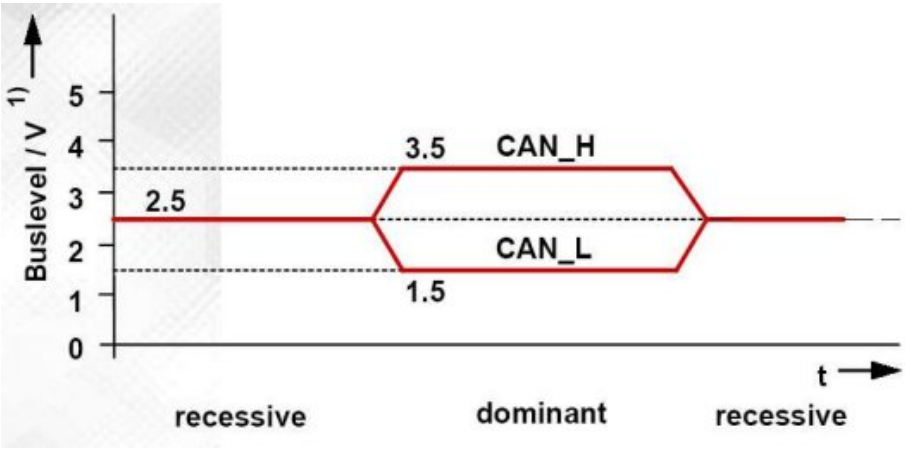
2.3 Příklad fyzické vrstvy

Jak již bylo zmíněno, varianty fyzické vrstvy mohou být rozdílné. Typickým příkladem je ovšem metalické dvouvodičové diferenciální vedení podle standardu ISO11898-2. Příklad můžeme vidět na obrázku 2.3.



Obrázek 2.3: Podoba zapojení ISO11898-2

Sběrnicová struktura se zakončovacími rezistory s hodnotou odporu 120 Ohmů. Diferenciální komunikace, kde úroveň je dána rozdílem napětí mezi vodiči CAN_H a CAN_L. Recesivní úroveň zajišťují pull-up rezistory, rozdíl CAN_H – CAN_L je blízký 0. V dominantním stavu je rozdíl CAN_H – CAN_L kladný, úrovně zajišťuje budič příslušného uzlu 2.4.



Obrázek 2.4: Napěťové úrovně ISO11898-2

Kapitola 3

Sběrnice USB

USB (Universal Serial Bus) je standard vyvíjený od roku 1994. Definuje komunikační protokol, kabely i konektory. Používá se k propojení zařízení (klávesnice, tiskárny, flash disky atd.) a hostitelského systému (PC, notebook) za účelem jejich vzájemné komunikace. Přesný popis lze najít ve specifikaci USB[1].

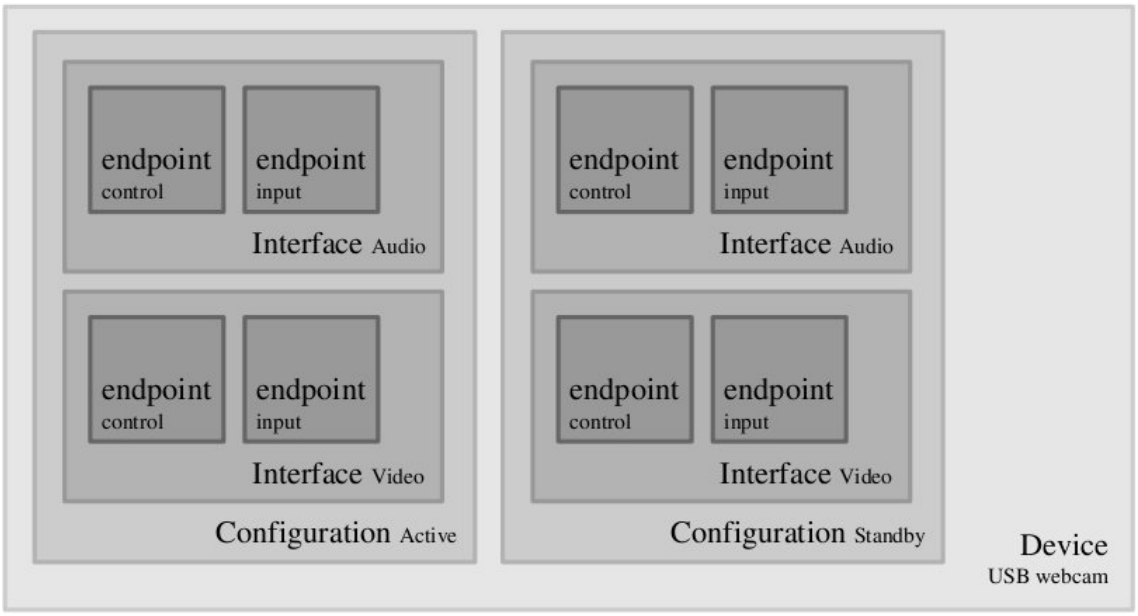
Rychlosti:

- USB 1.0 - Low Speed: až 1,5 Mb/s
- USB 1.1 - Full Speed: až 12 Mb/s
- USB 2.0 - Hi Speed: až 480 Mb/s
- USB 3.0: až 5 Gbit/s [2]

Hierarchie v zařízení (obrázek 3.1 převzatý z [9]) je popsána USB deskriptory. Ty definuje USB specifikace a jsou nezávislé na operačním systému. Existují tyto:

- **Zařízení (Device)** - reprezentuje fyzické zařízení připojené na sběrnici. Příklad: USB reproduktor s tlačítky na ovládání hlasitosti.
- **Konfigurace (Configurations)** - reprezentují stavy zařízení. Příklad: Active, Standby, Initialization.
- **Rozhraní (Interfaces)** - reprezentují jednotlivá logická zařízení a je s nimi svázán ovladač. Příklad: reproduktor, tlačítka na ovládání hlasitosti.
- **Koncové body (Endpoints)** - reprezentují jednosměrnou komunikační rouru[3]. IN - k hostovi, OUT od hosta.

[1] http://www.usb.org/developers/docs/
[2] Představena již v roce 2008, ale v současné době stále není masově rozšířena
[3] Kromě řídicího koncového bodu, který je obousměrný.



Obrázek 3.1: Přehled hierarchie USB zařízení

Kromě směru je komunikační roura definována také typem koncového bodu. Existují tyto:

- **Control** - Používána pro konfiguraci zařízení. Přes tuto rouru se získávají informace o zařízení a také se mu přes ni posílají příkazy. Každé zařízení disponuje koncovým bodem tohoto typu (tzv. EP0). Slouží k enumeraci, což je posloupnost standardizovaných požadavků a příkazů od hosta k identifikaci zařízení a zjištění potřebného ovladače. Přenos malého množství dat, kde je doručení garantováno.
- **Bulk** - Slouží k přenosům většího množství dat. Doručení je garantováno, není však zaručena šířka pásma ani možné zpoždění. Typické použití např. pro tiskárny, paměti, síťová zařízení atd.
- **Isochronous** - Slouží také k přenosům většího množství dat. Je sice garantováno maximální zpoždění, není však garantováno doručení. Typické použití je u audio/video zařízení.
- **Interrupt** - Vhodné pro časté přenosy malého množství dat. Garantováno doručení, maximální zpoždění i šířka pásma. Typické použití např. u klávesnic a myší.

Výhodou je dělení zařízení do tříd a podtříd podle jejich typu. U komunikace s takovýmito zařízeními je komunikace standardizována podobně jako tomu je u enumerace. Zařízení, které spadá do nějaké třídy, pak musí být schopno reagovat na všechny požadavky, které tato třída definuje. S tímto principem může existovat jednotný ovladač pro danou třídu zařízení. Např. typicky obsahuje operační systém jeden ovladač pro třídu flash disků. Po připojení flash disku pak není třeba dodávat speciální ovladač, ale je použit obecný ovladač pro flash disk v operačním systému. Pokud zařízení nespadá do žádné konkrétní třídy, je potřeba pro něj dodat speciální ovladač. To je třeba případ našeho CAN-USB převodníku.

Kapitola 4

Architektura ARM

4.1 Obecně

4.1.1 Popis

ARM je 32-bitová RISC (Reduced instruction set computer) architektura souboru instrukcí (ISA) vyvíjená společností ARM Holdings. Společnost nevyrábí vlastní procesory, ale věnuje se vývoji procesorových jader, které pak pod licencí poskytuje výrobcům. Výrobci pak ke standardizovanému jádru přidávají vlastní periferie a tento jednotný integrovaný obvod prezentují veřejnosti jako výsledný mikroprocesor.

Podle společnosti ARM představují jejich procesory od roku 2009 přibližně 90% veškerých vestavěných 32-bitových RISC procesorů. Jejich licenci využívá několik desítek výrobců a uplatnění nacházejí v aplikacích od mobilních telefonů, tabletů, herních konzolí až po oblasti průmyslové automatizace.

4.1.2 Historie společnosti

V roce 1983 firma *Acorn Computers* začala s projektem *Acorn RISC Machine* a v roce 1985 byl představen historicky první ARM procesor. Původní záměr byl vyrábět procesory pro PC, což vyústilo v roce 1987 k představení osobního počítače *Acorn Archimedes*, založeného na architektuře ARM. Na konci osmdesátých let spolupracovaly s Acorn firmy *Apple Computer* a *VLSI Technology* na vývoji nového jádra ARM. To dalo v roce 1990 oddělením vývojového týmu vzniknout nové společnosti *Advanced RISC Machines Ltd.* s cílem vytvořit nový mikroprocesorový standard. Význam akronymu ARM byl tedy upraven na *Advanced RISC Machine*. Již po roce pak bylo představeno první RISC jádro použitelné jako vestavěné - ARM6 a byla tak odstartována nová éra mikroprocesorů a mikropočítačů. V roce 1998 v době vstupu na burzu pak proběhlo přejmenování společnosti na současný název *ARM Holdings*.

4.2 Vhodné řešení pro realizaci převodníku

4.2.1 Možnosti realizace

V zásadě existují 3 základní možnosti, jak lze realizovat mikroprocesorový obvod s periferiemi. V našem případě s USB a CAN pro realizaci CAN-USB převodníku:

- Mikroprocesor má periferii integrovanou přímo v čipu od výrobce.
- Periferie je externí sériově vyráběný integrovaný obvod.
- Použití IP (Intellectual Property) core pro programovatelné logické obvody (FPGA, CPLD).

IP core je v návrhu hardware komplexní funkční blok, který implementuje daný element, tedy i nejrůznější periferie. Je implementován v některém z HDL (Hardware Description Language) jazyků (Verilog, VHDL) a může být tak přímo použit v programovatelných logických obvodech. Licencování závisí na vlastníkovi, existuje však také vývoj open source bloků[1], kde lze získat mnoho hardwarových návrhů pod svobodnými licencemi. Výjimkou nejsou ani implementace řadičů pro CAN a USB.

Pokud bychom volili variantu použití externího obvodu, nese to s sebou výhody i nevýhody. Příkladem čipů od firmy Philips[2] může být pro USB PDIUSBD12, pro CAN SJA1000. Výhoda právě třeba řadiče CAN SJA1000 či jiných (Intel i82527, Microchip MCP2510, Freescale MSCAN atd.) je ta, že jsou standardně používany a existuje tak pro ně vcelku široká podpora. Pro programové vybavení obsluhy těchto standardních čipů pak lze tedy např. čerpat z některých již hotových otevřených řešení.

Nevýhodou tohoto řešení je samozřejmě potřeba více místa na desce plošného spoje a dále složitost obvodu, kdy je nutné zajistit propojení řadiče s procesorem. To vede k použití mnoha propojovacích vodičů či k použití relativně pomalých sériových sběrnic (např. SPI). Další nespornou výhodou integrovaných periferií oproti externím jsou nižší energetické nároky. Výrobci procesorů (a zejména filozofie procesorů ARM na tomto staví) zavádějí vysoce optimalizovaný energetický koncept. Ten je zahrnuje různé uspávací módy s odpojováním zdrojů hodinových signálů či samotného napájení od periferií a dále použití více vnitřních sběrnic s rozdílnými pracovními frekvencemi podle potřeb jednotlivých periferií a jejich propojení přechodovými můstky. V neposlední řadě vede integrace do jednoho čipu ke zkrácení spojů a tím i zpoždění signálů.

V dnešní době je na trhu již mnoho mikroprocesorů od různých výrobců s nejrůznějšími interními periferiemi, CAN a USB nevyjímaje. Jejich použití je tak pro účely CAN-USB převodníku nejschůdnější.

4.2.2 Porovnání procesorových řad

Společnost ARM dělí svoje procesory do třech základních skupin podle jejich architektury, výkonu, schopností, efektivnosti a vhodnosti použití:

[1] např. OpenCores - http://opencores.org/
[2] Dnes má toto odvětví na starosti jejich dceřiná firma NXP

- Classic ARM Processors
- ARM Cortex Embedded Processors
- ARM Cortex Application Processors

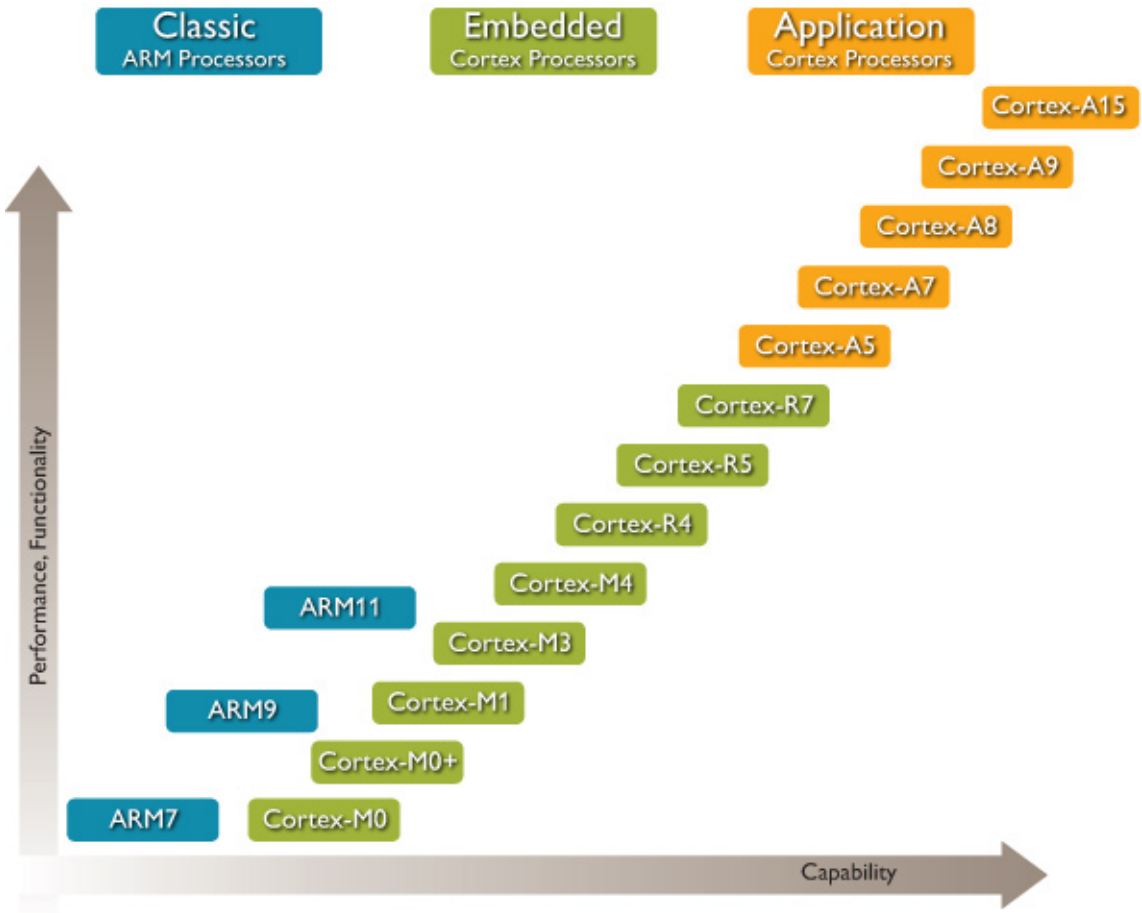
Classic ARM Processors zahrnuje procesory řady *ARM7*, *ARM9* a *ARM11*. *ARM7* je klasická rodina procesorů, která byla nesmírně úspěšná a kterou si firma ARM otevřela dveře do digitálního světa. V současné době je však překonána a její použití pro nová zařízení není doporučeno. *ARM9* a *ARM11* pak zahrnují různě výkonné procesory s rozmanitými vlastnostmi a způsoby aplikace.

ARM Cortex Embedded Processors zahrnuje dvě řady procesorů a to *Cortex-M* a *Cortex-R*. Procesory z řady *Cortex-M* jsou určeny primárně pro deterministické mikrokontrolérové[3] aplikace, procesory z řady *Cortex-R* jsou pak určeny pro výkonné real-time aplikace a jsou tak často využívány v kombinaci s nějakým operačním systémem reálného času (RTOS).

ARM Cortex Application Processors jsou procesory řady *Cortex-A*. To jsou již vysoce výkonné procesory pracující s frekvencemi v jednotkách GHz určené pro operační systémy (Android, Linux, Symbian, Windows Phone atd.) a používané v různých aplikacích od tabletů a smartphonů přes routery a digitální televize.

Rozdělení je možné vidět na obrázku 4.1 (převzatého z [7]).

[3] Mikrokontrolér (anglicky microcontroller) je označení jednočipového mikropočítače vhodného pro použití v řízení



Obrázek 4.1: Rozdělení procesorů ARM

4.2.3 Výběr vhodného procesorového jádra

Z uvedeného srovnání lze vyvodit, že pro realizaci CAN-USB převodníku nemá smysl uvažovat nad procesory z řady *Cortex-A* ani *Cortex-R*, protože pro účely jednoduchého vestavěného zařízení jsou výkonem i funkcionalitou předimenzovány, což by se zcela určitě promítlo na ceně. Lze uvažovat tedy nad řadou *Cortex-M* nebo nad některým ze skupiny klasických procesorů.

V tomto ohledu je třeba uvažovovat nad uplatněním. Sběrnice CAN je hojně využívána v průmyslové automatizaci. Nejspíše tedy má smysl pro práci s touto sběrnicí (v tomto případě ve funkci převodníku) volit stejný typ procesoru, který se používá pro její nejčastější aplikaci, tedy jako MCU (Microcontroller Unit). CAN-USB převodník by tak mohl fungovat nad stejným zařízením, které se používá pro reálnou aplikaci s mikrokontrolérem (např. ve funkci řízení motoru). To přináší mimo jiné výhodu možnosti snadného rozšíření funkcionality dané aplikace o funkcionalitu CAN-USB převodníku, např. pro diagnostické účely.

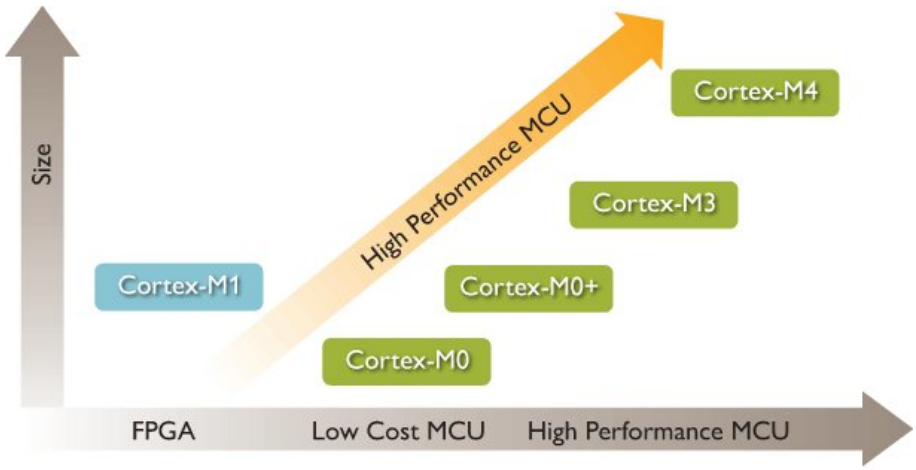
Tomuto požadavku tedy plně vyhovuje procesorová řada *Cortex-M*. Také proto, že podle společnosti ARM se tato řada stala celosvětovým průmyslovým standardem mezi mikrokont-

roléry, o cemž svědčí i fakt, že licence poskytuje více než 40ti společnostem v čele s Freescale, NXP Semiconductors, STMicroelectronics, Texas Instruments a Toshiba.

Řadu *Cortex-M* tvoří několik procesorových jader, které se liší výkonností a aplikačním využitím. Konkrétně se jedná o:

- Cortex-M0
- Cortex-M0+
- Cortex-M1
- Cortex-M3
- Cortex-M4

Cortex-M1 je navržen speciálně pro použití v programovatelných hradlových polích (FPGA). *Cortex-M0*, *Cortex-M0+* a *Cortex-M3* jsou primárně navrženy pro využití v MCU aplikacích. Vyznačují se nízkou spotřebou, malou velikostí a vysokou efektivitou zpracování. *Cortex-M3* je nejvýkonnější, obecně použitelný a podle společnosti ARM tvoří v současnosti hlavní větev v oblasti mikrokontrolérových aplikací. *Cortex-M4* vychází koncepčně z *Cortex-M3* a přidává navíc zejména funkcionalitu zpracování signálů z oblasti signálových procesorů (DSP). Přehled je možné vidět na obrázku 4.2 (převzatého z [7]).



Obrázek 4.2: Rozdělení procesorů ARM řady Cortex-M

Pro CAN-USB převodník se jeví jako nejvhodnější *Cortex-M3*. Podle provedeného rozboru poskytuje v poměru k ostatním jádrům relativně vysoký výkon s nízkými nároky na spotřebu, cenu i rozměry. Zřejmě proto je podle společnosti ARM tolik oblíben a tvoří jedničku mezi jádry mikrokontrolérů jak již bylo uvedeno výše. I kvůli tomu, že sběrnice CAN je v řídicích průmyslových oblastech hojně využívána, je použití procesoru s jádrem *Cortex-M3* vhodné. Podrobnější informace o jednotlivých procesorových řadách, samotných procesorech, aplikacích i srovnání lze nalézt na internetových stránkách společnosti ARM [7].

4.2.4 Výběr vhodného mikrokontroléru s jádrem ARM Cortex-M3

Pro výběr vhodného procesoru tedy stanovíme základní kritéria:

- jádro ARM Cortex-M3
- interní pamět FLASH alespoň 128kB
- interní pamět RAM alespoň 32kB
- integrované periferie CAN a USB (USB Device)

Tímto sítem neprošla nabídka společnosti Atmel, jelikož v době psaní této práce nenabízela žádný Cortex-M3 s integrovaným CAN. Tuto možnost nabízejí pouze u některých procesorů z rodiny ARM7 a ARM9. Společnost Freescale staví svoji nabídku mikrokontrolérů ARM pouze na jádrech Cortex-M0+ a Cortex-M4. Následuje stručný přehled nabídek procesorů od výrobců:

- NXP - tabulka 4.1
- STMicroelectronics - tabulka 4.2
- Texas Instruments - tabulka 4.3
- Fujitsu - tabulka 4.4
- Toshiba - 4.5

Jedná se spíše o orientační přehled několika produktů pro rámcovou představu, jelikož celkový výčet by byl příliš rozsáhlý a navíc by nepřinášel o moc větší vypovídající hodnotu. Položku tabulky tvoří název[4], maximální pracovní frekvence, velikost interních pamětí Flash a RAM, počet nezávisle využitelných CAN kanálů, provedení USB periferie[5] a nakonec orientační cena. Ta byla vždy stanovena pro jeden konkrétní produkt, dostupný u amerického distributora *Digi-Key Corporation*. Pokud cena není uvedena, nepodařilo se ji od distributora zjistit.

[4] jedná se o obecný název čipu, nikoliv o jméno specifické součástky, jelikož může existovat provedení s různými pouzdry a různým počtem pinů

[5] existuje-li pouze podpora pro USB Device, či navíc pro USB Host případně On-The-Go

Name	**fmax** (MHz)	**FLASH** (kB)	**RAM** (kB)	**CAN**	**USB** D, H, OTG	**Price** (USD/1ks)
LPC1754	100	128	32	1	D/H/OTG	8,25
LPC1756	100	256	32	2	D/H/OTG	8,48
LPC1758	100	512	64	2	D/H/OTG	10,63
LPC1759	120	512	64	2	D/H/OTG	10,63
LPC1764	100	128	32	2	D	8,13
LPC1768	100	512	64	2	D/H/OTG	11,25
LPC1769	120	512	64	2	D/H/OTG	11,75
LPC1774	120	128	40	2	D	8,93
LPC1778	120	512	96	2	D/H/OTG	12,13
LPC1788	120	512	96	2	D/H/OTG	13,09

Tabulka 4.1: NXP

Name	**fmax** (MHz)	**FLASH** (kB)	**RAM** (kB)	**CAN**	**USB** D, H, OTG	**Price** (USD/1ks)
STM32F105RC	72	256	64	2	D/H/OTG	9,89
STM32F107RB	72	128	48	2	D/H/OTG	8,25
STM32F215ZE	120	512	128	2	D/H/OTG	13,26
STM32F205VC	120	256	96	2	D/H/OTG	9,86
STM32F205RF	120	768	128	2	D/H/OTG	11,19
STM32F407VG	168	1024	192	2	D/H/OTG	12,78

Tabulka 4.2: STMicroelectronics

Name	**fmax** (MHz)	**FLASH** (kB)	**RAM** (kB)	**CAN**	**USB** D, H, OTG	**Price** (USD/1ks)
LM3S5632	50	128	32	1	D/H	11,38
LM3S5G51	80	384	64	2	D/H/OTG	14,19
LM3S5C56	80	512	64	1	D/H/OTG	14,52
LM3S9B95	80	256	96	2	D/H/OTG	17,49
LM3S9C97	80	512	64	2	D/H/OTG	18,70
LM3S9D90	80	512	96	2	D/H/OTG	19,40

Tabulka 4.3: Texas Instruments

Name	**fmax** (MHz)	**FLASH** (kB)	**RAM** (kB)	**CAN**	**USB** D, H, OTG	**Price** (USD/1ks)
MB9BF504N/R	80	256	32	2	D/H	8,60
MB9BF505N/R	80	384	48	2	D/H	-
MB9BF506N/R	80	512	64	2	D/H	9,00
MB9BF514N/R	144	288	32	2	D/H	-
MB9BF515N/R	144	414	48	2	D/H	-
MB9BF516N/R	144	544	64	2	D/H	-
MB9BF516T/S	144	512	64	2	D/H	-
MB9BF517T/S	144	768	96	2	D/H	-
MB9BF518T/S	144	1024	128	2	D/H	-

Tabulka 4.4: Fujitsu

Name	**fmax** (MHz)	**FLASH** (kB)	**RAM** (kB)	**CAN**	**USB** D, H, OTG	**Price** (USD/1ks)
TMPM369FD	80	512	128	1	D/H	12,00
TMPM369FY	80	256	64	1	D/H	-
TMPM368FD	80	512	128	1	D/H	-
TMPM368FY	80	256	64	1	D/H	-
TMPM368FW	80	128	48	1	D/H	-

Tabulka 4.5: Toshiba

Kapitola 5

Firmware převodníku CAN-USB

5.1 Analýza úkolu

Cílem bylo vytvořit firmware pro hardware realizovaný společností PiKRON, který je založený na mikrokontroléru LPC1768. Základem firmware byly zdrojové kódy firmware pro starší CAN-USB převodník implementovaný v roce 2008 v rámci bakalářské práce J. Kříže s názvem *Řadič sběrnice CAN připojený k PC přes sběrnici USB*[1]. Tento převodník je založen na mikrokontroléru LPC2148, obsahuje externí CAN řadič SJA1000 a podporuje ho ovladač LinCAN.

V tomto starším převodníku byl použit externí řadič SJA1000, vycházlo se tedy z podpory ovladače LinCAN pro PC pro řadič SJA1000 používaný u PCI karet. Tato funkcionalita se poté adaptovala pro použití ve vestavěném zařízení bez operačního systému. Dále bylo také nutné vytvořit emulaci propojovací sběrnice mezi řadičem a GPIO piny mikroprocesoru. V textu práce J. Kříž píše, že původně byla dokonce snaha využít funkcí pro SJA1000 v ovladači LinCAN a číst/zapisovat z/do jeho registrů přímo přes sběrnici USB. Řešení však bylo velmi pomalé a od tohoto řešení se upouští.

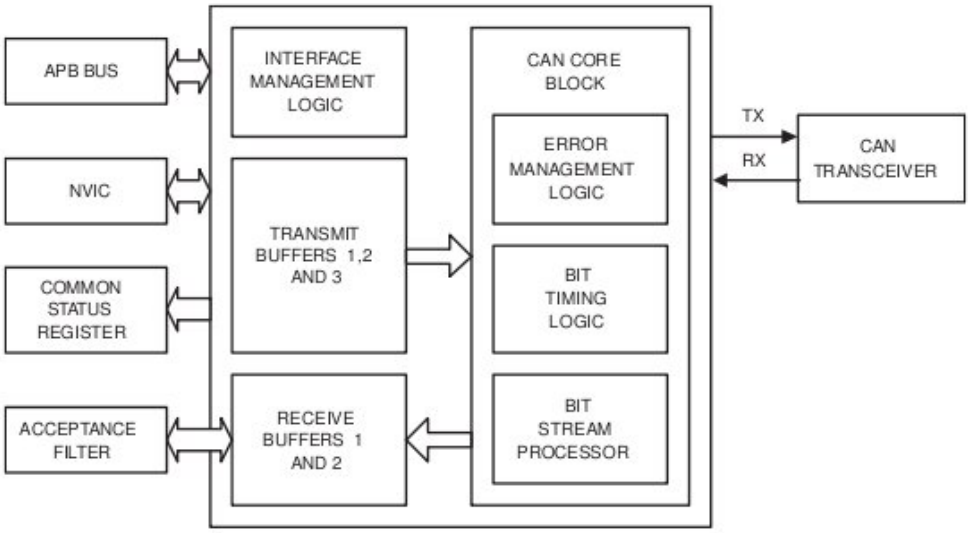
Firmware tohoto staršího převodníku využívá mechanismů LinCAN. To je znakový ovladač s důmyslným systémem vnitřních front pro specifické účely CAN komunikace. Tyto mechanismy svojí strukturou odstiňují samotnou funkcionalitu od závislosti na konkrétních typech HW a řadičů. Toho se dá dobře využít. Požadavek tedy byl navázat na předchozí převodník ve smyslu přidání podpory pro desku a CAN řadič současného převodníku a zároveň provést co možná nejmenší změny ve zbylé funkcionalitě. Takto lze pak mít jednotnou funkcionalitu pro různé převodníky a podle požadovaného typu hardware lze mezi nimi snadno volit např. podmíněnou kompilací.

5.2 Řadič CAN

Interní periferie CAN mikrokontroléru LPC1768 se skládá ze dvou částí. První je samotný řadič a druhou Acceptance Filter. Ten slouží k hardwarovému filtrování zpráv a tvoří samostatný modul. Blokové schéma řadiče lze vidět na obrázku 5.1 [4]. Stručný popis:

[1] práce je dostupná z [8]

- Rozhraní k APB bus poskytuje přístup k registrům.
- Interface Management Logic (IML) interpretuje příkazy od CPU a poskytuje mu informace o stavech a přerušeních.
- Transmit Buffer (TXB) představuje trojici odesílacích bufferů a tvoří rozhraní mezi IML a Bit Stream Processor (BSP). Každý Transmit Buffer je schopen pojmout kompletní CAN zprávu určenou pro odeslání. Zápis řídí CPU, čtení pak BSP
- Receive Buffer (RXB) představuje dvojicí přijímacích bufferů pro příjem zpráv z CAN sběrnice. Koncept dvou bufferů dovoluje CPU zpracovávat jednu zprávu zatímco další je zrovna přijímána. Zatímco u odesílání musí být specifikováno, který buffer se má zrovna použít, u příjmu se o to stará řadič a programátor toto řešit nemusí.
- Bit Stream Processor (BSP) je sekvencer, který řídí datový proud mezi odesílacími a přijímacími buffery a mezi CAN sběrnicí. Stará se také o arbitraci, bit-stuffing a detekuje chyby.
- Bit Timing Logic (BTL) se stará o časování a synchronizaci sběrnice.
- Error Management Logic (EML) dostává oznámení o chybách z BSP, zpracovává je a poskytuje BSP a IML informace o statistikách chyb.



Obrázek 5.1: Blokové schéma řadiče CAN

Nastavení registrů mikrokontroléru pro správnou funkci periferie CAN je následující:

- Zapnutí napájení. To lze přes registr Power Control for Peripherals (PCONP) kde musíme periferii povolit nastavením přislušného bitu (PCCAN1, PCCAN2).

- Nastavení hodinového signálu. To lze přes registr Peripheral Clock Selection (PCLKSEL[0—1]). Jelikož tato periferie spadá pod skupinu se sběrnicí APB0, použijeme PCLKSEL0. V tomto registru pak sybolické jméno PCLK_CAN1, PCLK_CAN2 značí dvojici bitů, jejichž hodnota určujíce dělící poměr frekvence jako zdroj hodinového signálu. Tedy např. hodnota bitů '00' = (CPU clock) / 4. Zde je nutné upozornit, že PCLK_CAN1 a PCLK_CAN2 musí být nastaveny stejně.

- Nastavení pinů procesoru. Použitý procesor je konkrétně LPC1768FDB100. V datasheetu (odkaz)a schématu zapojení lze zjistit, který pin procesoru musíme nastavit:

  – pin 46 : P0.0/RD1/TXD3/SDA1 : RD1 - CAN1 receiver input
  – pin 47 : P0.1/TD1/RXD3/SCL1 : TD1 - CAN1 transmitter output

  Potřebujeme tedy nastavit dva nejspodnější piny brány 0. Použijeme registr PINSEL0, který řídí funkce spodní poloviny brány 0. Zde jménu pinu opět odpovídá dvojice bitů, jejichž hodnota určuje jeho funkci. Je třeba tedy nastavit:

  – jméno pinu P0.0 - první alternativní funkce hodnota '01' - RD1
  – jména pinu P0.1 - první alternativní funkce hodnota '01' - TD1

  Následně je třeba nastavit režim pinů. To provedeme pomocí dvojice registrů. Prvním je PINMODE0, který určujíce nastavení pull-up/pull-down rezistorů. Dvojice bitů podle jména pinu odpovídá:

  – jméno pinu P0.0 - hodnota '00' - povolen pull-up
  – jména pinu P0.1 - hodnota '00' - povolen pull-up

  Druhým je PINMODE0_OD, který určujíce nastavení režimu open drain. Dvojice bitů podle jména pinu odpovídá:

  – jméno pinu P0.0 - hodnota '00' - normalní (ne open drain) režim
  – jména pinu P0.1 - hodnota '00' - normalní (ne open drain) režim

- Definování událostí v CAN řadiči, které mají vyvolat přerušení (příjem, odeslání atd.). K tomuto slouží CAN Interrupt Enable Register (CAN1IER, CAN2IER) [2]

- Inializace registrů řadiče. Pro fungování je nakonec nutné v registru CAN Mode (CAN1MOD, CAN2MOD) nastavit stav řadiče z reset do operačního stavu.

5.3 Implementace

5.3.1 Funkcionalita a struktura LinCAN

Inicializaci řadiče popsanou v sekci 5.2 je třeba zpracovat do programového kódu. Stejně tak ostatní funkce, které pracují s registry řadiče tzn. odesílání zpráv, přijímání zpráv, nastavení časování atd. Tuto funkcionalitu je pak třeba sloučit s koncepcí struktury použité v LinCAN.

[2] Samotné povolení přerušení od CAN periferie je třeba nastavit v NVIC.

Výsledkem je tak funkcionalita ovládající daný CAN řadič se strukturou na bázi LinCAN. Toto řešení pak může být s minimální změnou použito v kódu pro starší převodník, kde tak vznikne jednotný kód pro CAN-USB převodník viz. 5.1.

V zásadě největší změnou je volání funkce pro nastavení specifických operací pro daný hardware. V LinCAN je toto reprezentováno strukturou `struct hwspecops_t`. Alokace této struktury již proběhla a funkce dostává ukazatel na ni jako parametr. Ve funkci pak přiřadíme ukazatelům na funkce adresy námi vytvořených funkcí. Přístup pak dále probíhá přes tyto ukazatele. Tento pripcip je v LinCAN obecně používaný. Funkce tedy může vypadat:

```
1 int can_lmc1_register(struct hwspecops_t *hwspecops){
2
3         hwspecops->init_hw_data = can_lmc1_init_hw_data;
4         hwspecops->init_chip_data = can_lmc1_init_chip_data;
5
6         /* ... */
7
8         return 0;
9 }
```

Popis struktur i jejich položek lze najít v dokumentaci k LinCAN, která je dostupná z [8]. Zde pouze pro názornost uvedeny:

`init_hw_data` - volána pro inicialici struktury `candevice_t`, která reprezentuje hardwarové zařízení s CAN. Zde se nastavuje počet řadičů atd. Zjednodušený výpis:

```
1 int can_lmc1_init_hw_data(struct candevice_t *candev){
2
3         candev->nr_all_chips=1;
4         candev->flags = 0;
5
6         return 0;
7 }
```

`init_chip_data` - volána pro inicialici struktury `canchip_t`, která reprezentuje CAN řadič. Zde se nastavuje bázová adresa registrů, pracovní frekvence atd. Tato funkce je také zodpovědná za nastavení specifických operací pro daný řadič (`canchip_t->chipspecops`). Tyto operace pak reprezentují samotnou funkcionalitu daného řadiče tak, jak byla popsána na začátku této sekce (odesílaní a přijímání zpráv aj.). Zjednodušený výpis:

```
1 int can_lmc1_init_chip_data(struct candevice_t *candev, int chipnr){
2
3         // used CAN1 peripherial -> CAN1 registers base
4         candev->chip[chipnr]->chip_base_addr = CAN1_REGS_BASE;
5         // clock for CAN
6         candev->chip[chipnr]->clock = system_frequency / 4;
7
8         lpc17xx_fill_chipspecops(candev->chip[chipnr]);
9
10        return 0;
11 }
12
13 int lpc17xx_fill_chipspecops(struct canchip_t *chip){
```

```
14
15          chip->max_objects=1;
16          chip->chip_irq = CAN_IRQn;
17
18          lpc17xx_register(chip->chipspecops);
19
20          return 0;
21 }
22
23 int lpc17xx_register(struct chipspecops_t *chipspecops){
24
25          chipspecops->chip_config = lpc17xx_chip_config;
26          chipspecops->pre_write_config = lpc17xx_pre_write_config;
27          chipspecops->send_msg = lpc17xx_send_msg;
28          chipspecops->wakeup_tx = lpc17xx_wakeup_tx;
29          chipspecops->irq_handler = lpc17xx_irq_handler;
30          chipspecops->set_bittiming = lpc17xx_set_bittiming;
31
32          return 0;
33
34 }
```

5.3.2 Uživatelské požadavky

Pro datový přenos CAN zpráv po USB se používají koncové body typu bulk. Přenos řídicích požadavků pro konfiguraci (např. nastavení komunikačních parametrů sběrnice) se pak provádí přes řídicí (control) koncový bod tzv. EP0. Bližší popis je v sekci 7.4. Cílem je poskytnout tomuto převodníku podporu vytvořeným síťovým ovladačem pro GNU/Linux, který je založen na SocketCAN. Pro tento účel musela být implementována podpora dvou nových řídicích požadavků, specifikovaných makry:

- **USBCAN_VENDOR_GET_BITTIMING_CONST** - Tímto požadavkem si ovladač říká o zaslání hodnot, kterých může řadič nabývat pro účely časování.
- **USBCAN_VENDOR_SET_BITTIMING** - Tímto požadavkem ovladač nastavuje konkrétní parametry časování (Bit timing registr).

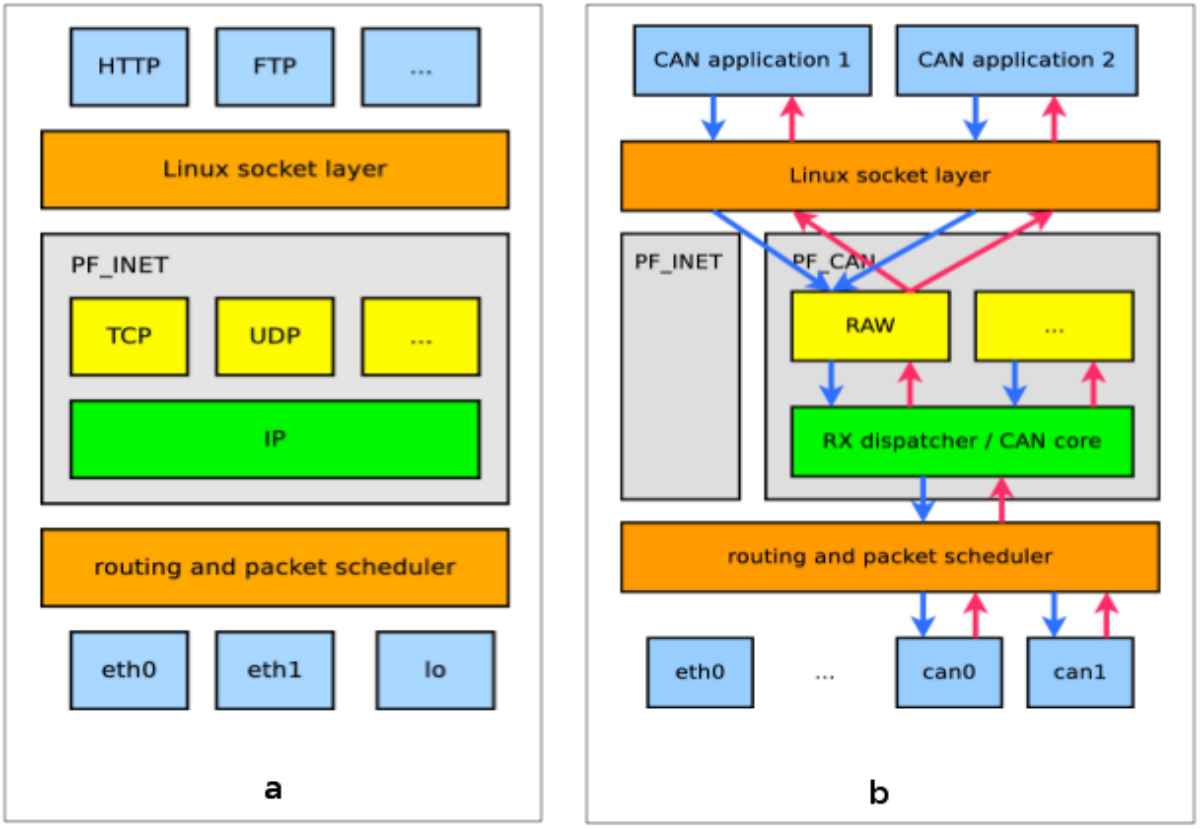
Kapitola 6

SocketCAN

SocketCAN je projekt implementující aplikační rozhraní pro přístup na sběrnici CAN a model ovladačů pro jádro Linux. Od jádra 2.6.25 je součástí mainline. SocketCAN využívá BSD Socket API, Linuxový síťový stack a zpřístupňuje zařízení CAN jako síťová rozhraní. API pro user-space aplikace je tedy koncipováno tak, aby se vývoj těchto aplikací co nejvíce přiblížil obecně známému síťovému programování pro TCP/IP protokoly pomocí socketů. Vývoj ovladačů pak vychází z tvorby síťových ovladačů pro široce rozšířený model ovladačů sítě Ethernet.

6.1 Linux a jeho síťový subsystém

Síťový substystém Linuxu je značně flexibilní. Jeho rozdělení po vrstvách s příkladem rodiny protokolů PF_INET, která implementuje TPC/IP, můžeme vidět na obrázku 6.1 v části *a* (převzatého z [3]). Z pohledu aplikační vrstvy zde existuje standardní POSIX socket API, které tvoří rozhraní jádra. User-space aplikace mohou tedy používat standardní systémová volání *socket()*, *bind()*, *read()*, *write()* apod. Dále následuje protokolová vrstva. Tu tvoří rodiny protokolů. Každá rodina protokolů může implementovat různé množství rozličných protokolů. Zde v případě PF_INET jde o protokoly TCP, UDP, apod. Dále se nachází vrstva, která má za úkol pakety předávat příslušným ovladačům síťových zařízení. Ovladače tvoří spodní úroveň a mají za úkol data fyzicky odeslat.



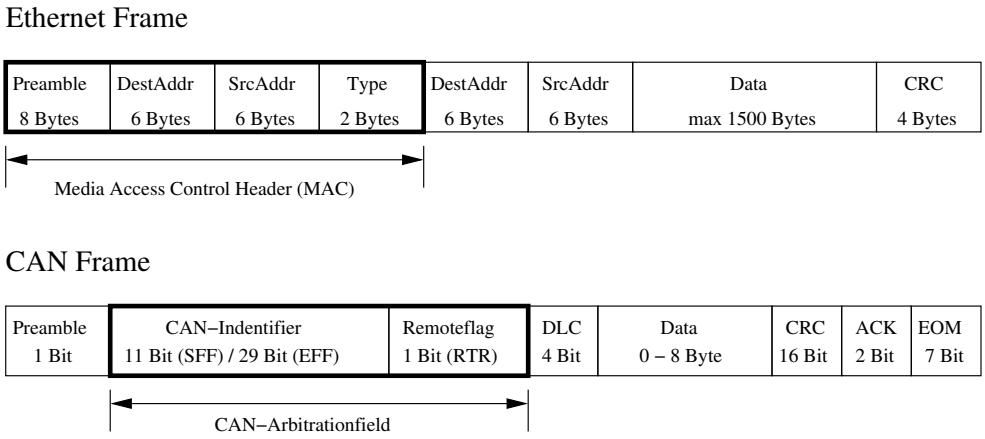
Obrázek 6.1: Síťový subsytém Linuxu a funkce PF_CAN

6.2 CAN a síťový přístup

Koncept SocketCAN staví na zavedené Linuxové síťové infrastruktuře, ovšem sběrnice CAN je charakterem odlišná od síťových standardů, které využívají TCP/IP. Zásadní je absence adresace, jakou známe např. ze sítě Ethernet. Nelze tedy přímo adresovat příjemce. Sběrnice CAN funguje na principu *broadcast*, tedy rozeslání zprávy všem ostatním připojeným uzlům. Není možné jednotlivé CAN zprávy adresovat konkrétnímu zařízení přímo na fyzické/linkové úrovni. Každá CAN zpráva obsahuje identifikátor, který je použit k arbitráži sběrnice a který můžeme chápat do jisté míry jako adresu odesílatele. Srovnání rámců z adresačního hlediska je naznačeno na obrázku 6.2.

Často je totiž každému zařízení přidělen identifikátor (nebo určitý rozsah identifikátorů), pod kterým vysílají zprávy na sběrnici. Ostatní zařízení jsou pak schopna podle tohoto identifikátoru zjistit, jestli zpráva přichází od zdroje, který patří do oblasti jejich zájmu či nikoliv. Mohou se tak rozhodnout, zda zprávu zpracovat či ignorovat.

K tomuto účelu byla vytvořena nová rodina protokolů PF_CAN. Integrace této rodiny protokolů do síťového stacku Linuxu může čerpat ze stávajících rodin síťových protokolů, jako je DECnet, Appletalk nebo AX.25, které také využívají síťovou infrastrukturu Linuxu



Obrázek 6.2: Rozdílný koncept adresace u Ethernetu a CAN

s různým hardwarem a různými protokoly [2].

6.3 Rodina protokolů PF_CAN

6.3.1 Struktura

Jádro (core) SocketCANu tvoří jaderný modul (kernel module), který implementuje rodinu protokolů PF_CAN. Samotné transportní protokoly jsou implementovány opět jako jaderné moduly, které jsou podle potřeb dynamicky nahrávány za běhu. Samotný modul jádra SocketCANu tedy nemůže být použit, aniž by nebyl nahrán alespoň jeden protokolový modul.

6.3.2 Local loopback

Rodina protokolů **PF_CAN** a její protokol **CAN_RAW** podporují tzv. *multi-application access*, tedy možnost zpřístupnit jedno CAN rozhraní více aplikacím současně. Musí tedy platit, že informace o veškerém dění na tomto rozhraní musí být nějakým způsobem sdílené, tedy přístupné všem aplikacím. Pokud se jedná o příjem, přijatý CAN rámec je přenesen ke všem aplikacím. Ovšem pokud jedna aplikace vyšle CAN zprávu, musí se to ostatní aplikace běžící na stejném systému dozvědět, jako kdyby běžely na jiných systémech propojených CAN sběrnicí s vlastními CAN rozhraními. Existuje tedy mechanismus tzv. *local loopback*, kdy jsou o vyslání CAN zprávy na sběrnici jednou aplikací informovány ostatní. Obvykle se tak stane v obsluze přerušení vyvolané při úspěšném přenosu zprávy, a to uložením právě odeslané zprávy do přijímací fronty. Je to z důvodu zachování správného pořadí CAN zpráv. Když by probíhala tato signalizace ihned po odeslání, tak např. CAN zpráva s nízkou prioritou (vysoké ID), by měla velký problém s arbitrací sběrnice. Mezitím by rozhraní mohlo klidně přijmout jiné zprávy s vyšší prioritou, než by došlo k samotnému fyzickému odeslání zprávy na sběrnici a pořadí by tak bylo nesprávné.

Příjem CAN rámce na stejném soketu, ze kterého byl odeslán je standardně vypnutý. Lze to samozřejmě změnit. Stejně tak lze změnit nastavení socketu (každého) v reakci na

loopback funcionalitu. Tato a mnohá jiná nastavení se provádí přímo z uživatelské aplikace, pomocí systémového volání *setsockopt()*.

6.3.3 Datová cesta

Pro fungování datové cesty CAN zpráv je nutná identifikace datových bufferů a síťových zařízení. Packet scheduler je totiž sdílený prostředek všech síťových zařízení, ať již se jedná o CAN, či Ethernet.

Je-li socket buffer označen jako **ETH_P_CAN**, znamená to, že obsahuje CAN zprávy. Síťovému zařízení je nastaven **ARPHRD_CAN** jako typ hardwarového rozhraní. Identifikujeme tak síťové zařízení, které bude zpracovávat datové buffery označené jako **ETH_P_CAN**. Obojí je nutné provést v ovladači daného síťového zařízení.

Touto identifikací je docíleno toho, že rodina protkolů **PF_CAN** bude zodpovědná za zpracování datových bufferů, označených jako **ETH_P_CAN**, tedy nesoucích CAN zprávy.

Kapitola 7

Ovladač převodníku CAN-USB

7.1 Jaderné moduly

Linux je modulární monolitické jádro operačního systému. Zachovává architektonický koncept monolitického jádra, tedy že jádro běží v jednom společném odděleném paměťovém prostoru (tzv. *kernel space*). Dále však umožňuje zavádět speciální moduly do systému přímo za běhu. Tento jaderný modul může reprezentovat ovladač zařízení, podporu souborového systému atd. Přímo v jádře pak bývá zakompilována většinou pouze funkcionalita důležitá pro běh systému a všechno ostatní si systém či uživatel může nahrát za běhu podle potřeby. To vede k zmenšení nároků na systémové zdroje a značné flexibilitě. Příklad nejjednoduššího jaderného modulu (převzatý z [1]):

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

1 - 2 → Vkládání hlavičkových souborů s deklaracemi potřebnými pro tvorbu modulu.

3 → Makro označující licenci, pod kterou modul spadá.

5 - 9 → Funkce, která se má zavolat po nahrání modulu do jádra.

11 - 14 → Funkce, která se má zavolat při odstanění modulu z jádra.

16 - 17 → Makra, která označují roli těchto dvou funkcí.

7, 13 → Funkce pro výpis informací do logu jádra. Použití je obdobné jako u funkce `printf()` ze standardní knihovny C pro programy z uživatelského prostoru.

Pokud bychom chtěli provést překlad tohoto triviálního příkladu do podoby jaderného modulu a ten následně zavést do jádra, postupovali bychom v následujících krocích:

Vytvoříme si pro tento účel pracovní adresář a v něm vytvoříme soubor s názvem `hello_world.c`, jehož obsahem bude příklad uvedený výše. Dále v adresáři vytvoříme soubor `Makefile`, který bude obsahovat pouze jedinou řádku:

```
obj-m = hello_world.o
```

Pak zavoláme sestavovací program `make` s následujícími parametry:

```
$ make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
```

Tímto říkáme, že program `make` načte `Makefile` z adresáře se zdrojovými kódy aktuálně běžícího jádra. Pomocí proměnné `M` řekneme, že se modul nachází v aktuálním adresáři a slovo `modules` na konci znamená, že chceme, aby se zkompilovaly pouze moduly. Výstup překladu může vypadat takto:

```
make: Entering directory '/usr/src/linux-headers-2.6.32-35-generic'
  CC [M]  /tmp/hello/hello_world.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /tmp/hello/hello_world.mod.o
  LD [M]  /tmp/hello/hello_world.ko
make: Leaving directory '/usr/src/linux-headers-2.6.32-35-generic'
```

V adresáři nám pak kromě jiného vzniknul přeložený jaderný modul `hello_world.ko`. Ten můžeme do jádra zavést následovně:

```
$ sudo insmod ./hello_world.ko
```

Zkontrolovat, že se modul zavedl do jádra, můžeme z výpisu programu `lsmod`, který zobrazuje informace o stavech modulů v Linuxu:

```
$ lsmod | head -2
Module                  Size  Used by
hello_world              680  0
```

Náš modul obsahoval základní funkcionalitu pro ověření funkčnosti, tedy výpis *Hello world* po zavedení do jádra. Z výpisu jádra můžeme zkontrolovat (vypisujeme pouze poslední řádku):

```
$ dmesg | tail -1
[ 8233.547864] Hello, world
```

Modul můžeme pak z jádra odstranit:

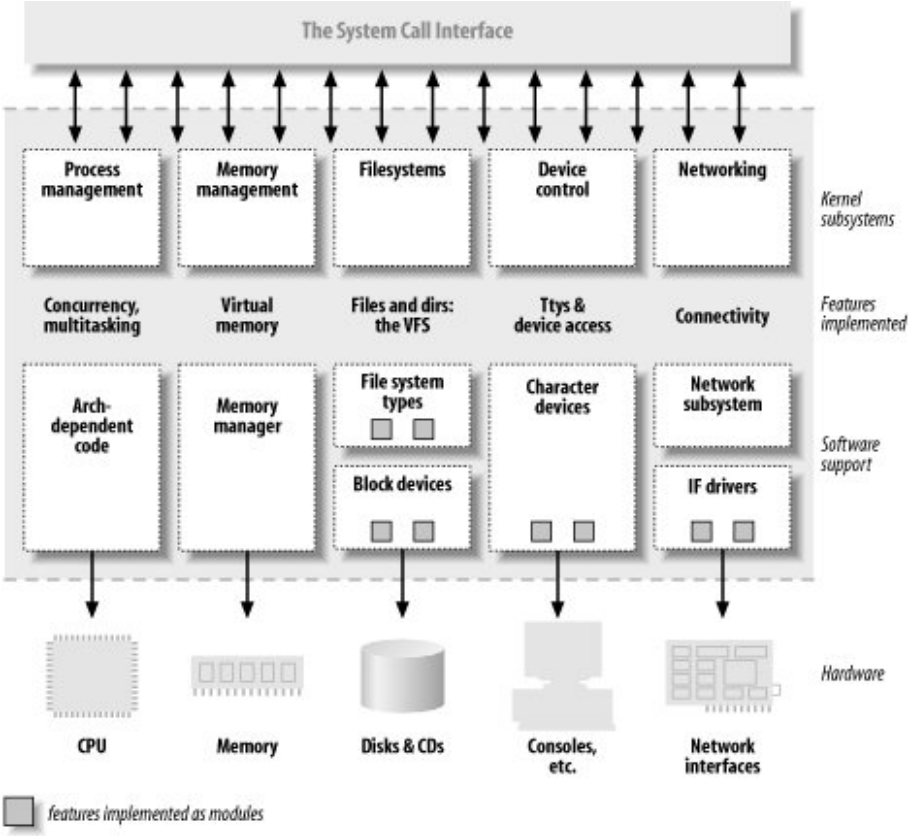
```
$ sudo rmmod hello_world
```

a opět se můžeme přesvědčit o korektním odstranění modulu z výpisu jádra:

```
$ dmesg | tail -1
[ 8819.150167] Goodbye, cruel world
```

7.2 Ovladače

Podle [1] lze jádro rozdělit do několika částí podle úlohy, jakou v systému plní. To je možné vidět na obrázku 7.1. Z uvedeného rozdělení je patrné, že existuje několik typů zařízení, tedy i několik typů ovladačů. Zároveň je vidět, že nezávisle na typu lze ovladač implementovat jako jaderný modul, popsaný v předcházející sekci 7.1.



Obrázek 7.1: Rozdělení jádra

V Linuxu tedy existují 3 základní typy zařízení:

- **Znaková zařízení** - Přístup k takovému zařízení je jako k posloupnosti bytů (znaků) obdobně jako v souborech. Obvykle poskytují pouze sekvenční přístup a nepoužívají buffery. Jsou přístupná ze systému souborů přes adresář `/dev`. Typickým příkladem může být sériový port (např. `/dev/ttyS0`), textová konzole (`/dev/console`) atd.

- **Bloková zařízení** - U blokového zařízení obecně probíhá zápis i čtení po blocích dat. Typickým zastupitelem může být pevný disk (blok 512 bytů). Linux však umožňuje aplikacím přístup k blokovým zařízením stejně jako ke znakovým. Liší se pouze způsobem, jakým jsou data interně spravována jádrem. Tedy rozhraní jádra pro blokové a znakové ovladače jsou zcela odlišná. Stejně jako u znakových jsou bloková zařízení přístupná ze systému souborů přes adresář `/dev`. Rozdílné je také to, že k blokovým zařízením může být připojem systém souborů.

- **Síťová rozhraní** - Každá síťová komunikace probíhá přes síťové rozhraní, což je zařízení zodpovědné za příjem a odesílaní datových paketů. Nejčastěji zastupuje reálné hardwarové zařízení, může však být pouze na softwarové bázi (např. *loopback*). Využívá síťového subsystému jádra. Přístup k rozhraní je umožněn přidělováním unikátních jmen jednotlivým rozhraním (např. `eth0`), není zde však možný přímý přístup přes systém souborů z `/dev` ani odjinud. Komunikace mezi jádrem a ovladačem síťových zařízení je zcela odlišná od té ve znakových a blokových ovladačích.

Pokud budeme mluvit o USB ovladači, tak základ každého je USB modul, který pracuje s USB subsystémem jádra. Samotné zařízení se pak v systému může prezentovat jako znakové zařízení (např. USB serial port), blokové zařízení (např. USB čtečka paměťových karet) nebo síťové rozhraní. Právě poslední zmíněné je náš případ v tomto ovladači CAN-USB převodníku, jelikož využíváme SocketCAN, což je implementace pro síťová rozhraní.

Na ovladače a jaderné moduly obecně jsou kladeny zvláštní nároky na správnost. Aplikace v uživatelském prostoru mají svůj oddělený paměťový prostor a při chybě jsou ukončeny operačním systémem. U modulů nic takového neplatí, fungují v rámci jednoho paměťového prostoru jádra (*kernel space*). Chybný modul proto může způsobit nestabilitu celého systému, v horším případě poškození dat. V ovladačích se nepoužívají globální proměnné. Namísto toho proměnné použitelné z více míst ovladače utvoří jednu strukturu, která pak tvoří privátní data ovladače.

7.3 USB ovladače

Linux rozlišuje dva typy ovladačů USB zařízení a to:

- **USB Device drivers** - ovladače pro hostitelský systém (obvykle osobní počítač), které ovládají jednotlivá periferní zařízení k němu připojená

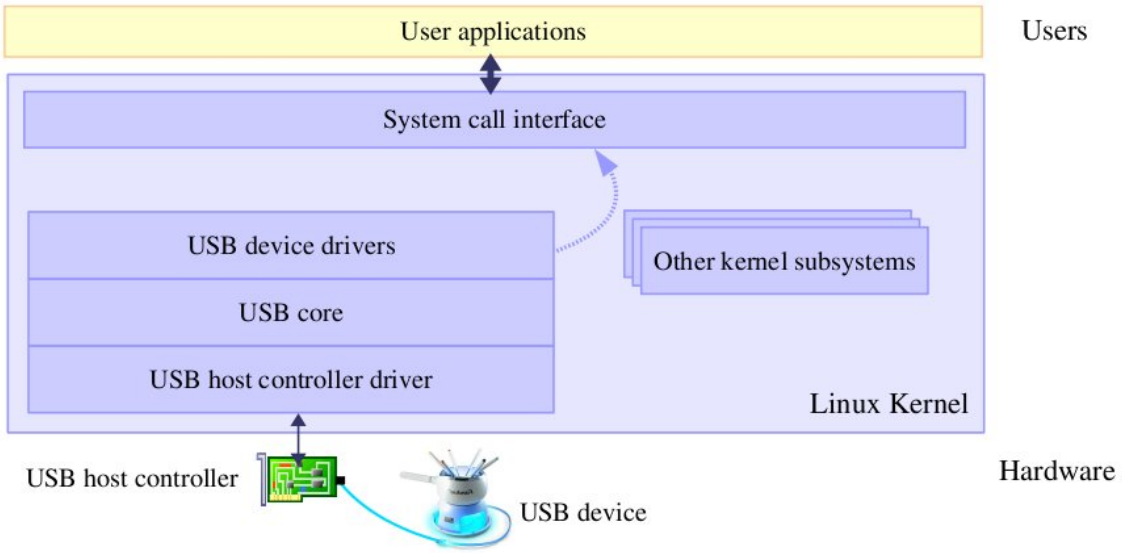
- **USB Gadget drivers** - ovladače v periferních zařízeních (obvykle nějaký vestavěný systém), která jsou připojována k hostitelskému systému

Tyto termíny vznikly pro rozlišení pojmů. V kontextu USB protokolu je pak device driver master a gadget driver je pak slave. Častá je totiž situace, kdy máme nějaké vestavěné

(embedded) zařízení, které disponuje řadičem USB Device a běží na něm embedded Linux. Zde pak existuje USB Device controller driver, což je ovladač, který obsluhuje zmíněný řadič USB Device. Zprostředkovává samotný přístup čipu ke sběrnici a je tedy platformově závislý. Poskytuje ale rozhraní zmíněné platformově nezávislé vrstvě ovladačů USB Gadget drivers. Jako příklad použití může být Storage gadget, umožující hostovi přístup jako k paměťovému zařízení. Můžeme tak např. připojit digitální kameru k PC a ta se zpřístupní jako USB storage device. Úkolem v této práci je však poskytnout podporu konkrétnímu periferním zařízení v hostitelském systému, tedy implementovat USB Device driver. Dále bude tedy popisován tento typ ovladače.

7.3.1 Struktura USB ovladačů

Struktura pro hostitelský systém je znázorněna na obrázku 7.2 (převzatého z [9]). Spodní vrstvu tvoří USB Host controller drivers, tedy platformově závislé ovladače různých hardwarových řadičů. Nad ní je vrstva USB Core, což je subsystém jádra nezávislý na architektuře, který implementuje samotnou specifikaci sběrnice USB a poskytuje rozhraní ovladačům USB zařízení pro přístup k hardwaru bez nutnosti zaobírat se nižší vrstvou. Vrchní vrstva je tvořena samotnými ovladači specifických zařízení.



Obrázek 7.2: Struktura USB ovladačů v Linuxu

7.3.2 USB v jádře Linux

Hierarchie USB zařízení device → configirations → intefraces → endpoints byla vysvětlena v sekci 3.

7.3.2.1 Koncové body (Endpoints)

V jádře jsou koncové body popsány strukturou `struct usb_host_endpoint`. V ní nalezneme strukturu `struct usb_endpoint_descriptor`, která jak už z názvu vyplývá reprezentuje deskriptor koncového bodu. Obsahuje tedy položky dané USB specifikací, které pro přehlednost nesou stejné jméno [1]. Můžeme tam najít tedy např.:

`bEndpointAddress`

Adresa specifického koncového bodu a informace o jeho směru[2].

`bmAttributes`

Určuje typ koncového bodu. Lze použít bitovou masku `USB_ENDPOINT_XFERTYPE_MASK` a určit tak, zda je o control (`USB_ENDPOINT_XFER_CONTROL`), bulk (`USB_ENDPOINT_XFER_BULK`), interrupt (`USB_ENDPOINT_XFER_INT`) či isochronous (`USB_ENDPOINT_XFER_ISOC`) endpoint.

`wMaxPacketSize`

Udavá maximální velikost paketu v bytech, se kterou může koncový bod manipulovat[3].

7.3.2.2 Rozhraní (Interfaces)

Množina koncových bodů tvoří rozhraní, které reprezentuje základní funkcionalitu zařízení. USB driver je tedy svázán s rozhraním. Jedno fyzické zařízení může poskytovat i více funkcí, z čehož vypývá, že pro jedno zařízení může být potřeba více ovladačů. Např. USB reproduktor může být složen ze dvou logických částí - ovládací tlačítka a zvukový stream. Tedy bude potřeba dvou různých ovladačů.

Jedeno rozhraní může mít více odlišných nastavení parametrů tzv. *alternate settings* s tím, že pro každé rozhraní může být v jeden okamžik aktivní vždy pouze jedno nastavení. Použití může být např. různé nastavení šířky pásma u audio zařízení. Jednotlivá nastavení jsou číslována (od 0) a jsou reprezentována strukturou `struct usb_host_interface`. Ve výchozím stavu je aktivní vždy první nastavení (s číslem 0).

USB rozhraní je v jádře reprezentováno strukturou `struct usb_interface`. Právě tuto strukturu předává USB core konkrétnímu USB ovladači, čili za tuto funkcionalitu je pak ovladač zodpovědný. Důležité části této struktury pak jsou:

`struct usb_host_interface *altsetting;`

Pole alternativních nastavení dostupných pro toto rozhraní. Struktura `struct usb_host_interface` obsahuje množinu koncových bodů (`struct usb_host_endpoint *endpoint`) pro dané nastavení.

`unsigned num_altsetting;`

[1] Tyto názvy neodpovídají schématu pojmenovávání proměnných v jádře Linux
[2] pod stejnou adresou mohou vytupovat dva různé koncové body rozslišené směrem - IN nebo OUT
[3] Data s větší velikostí jsou rozdělena a přenesena na vícekrát

Počet alternativních nastavení

`struct usb_host_interface *cur_altsetting;`

Ukazatel do pole `altsetting`, odkazující na momentálně aktivní nastavení.

`int minor;`

Minor číslo pro dané rozhraní přidělené od USB core po zavolání `usb_register_dev()`

7.3.2.3 Konfigurace (Configurations)

Rozhraní jsou svázána do konfigurací. V jádře pak USB konfiguraci popisuje struktura `struct usb_host_config`. V jeden okamžik může být aktivní pouze jedna konfigurace. Samotný USB ovladač s touto strukturou většinou neoperuje, je zde zmíněna pro úplnost.

7.3.2.4 Zařízení (Devices)

Celé USB zařízení popisuje struktura `struct usb_device`. S touto strukturou operuje USB core, které ovladači předává pouze strukturu reprezentující rozhraní. Lze však použít funkci `interface_to_usbdev()` pro získání `struct usb_device` z `struct usb_interface`.

7.3.2.5 USB Request Blocks

V jádře probíhá USB komunikace mezi ovladači a zařízeními asynchronně pomocí objektů zvaných URB (USB Request Block). URB je využíván k zaslání či přijetí dat do nebo od konkrétního USB koncového bodu. Jsou podobné paketům v síťové komunikaci. O vytvoření URB se stará ovladač. Ten může více URBs přiřadit k jednomu konkrétnímu koncovému bodu nebo může jeden URB použít opakovaně pro různé koncové body. Každý koncový bod si drží URBs ve frontě, může být proto odesláno více URBs na jeden koncový bod. Typický životní cyklus pro URB je popsán v [1] takto:

- Vytvořen ovladačem USB zařízení.
- Přiřazen konkrétnímu koncovému bodu konkrétního zažízení.
- Předán ovladačem do USB core.
- Předán USB core danému ovladači USB řadiče (USB host controller driver) konkrétního zařízení.
- Zpracován ovladačem řadiče, který provede samotný datový přenos.
- Když je dokončen, ovladač řadiče upozorní ovladač zařízení.

Upozornění spočívá v zavolání předem definované dokončovací funkce v kontextu přerušení, kde ovladač přebírá zpět kontrolu nad dokončeným URB. Ten pak může URB buď smazat, nebo ho znovu použít třeba i pro jiný koncový koncový bod.

URB je realizován strukturou `struct urb`. Jeho vytvoření se provádí zavoláním funkce `usb_alloc_urb()`. To je nutné kvůli mechanismu počítání referencí, který využívá USB core. Alokace URB tedy není staticky, ani přes `kmalloc()`. Deklarace funkce vypadá:

```
1 struct urb *usb_alloc_urb(
2         int iso_packets,
3         gfp_t mem_flags
4 );
```

2 → Počet izochronních paketů.
3 → Standardní alokační omezení (viz. kmalloc() - GFP_KERNEL, GPF_ATOMIC).

Příklad vytvoření URB pak může vypadat:

```
struct *urb u = usb_alloc_urb(0, GFP_KERNEL);
```

Smazání pak probíhá analogicky přes funkci:

```
void usb_free_urb(struct urb *urb);
```

Dále musí být URB incializován a přiřazen specifickému koncovému bodu. Toto se pak provádní přes funkce odpovídající jednotlivým typům přenosů. Např. pro *bulk* se používá funkce `usb_fill_bulk_urb()`. Deklarace funkce vypadá:

```
1 void usb_fill_bulk_urb(
2         struct urb *urb,
3         struct usb_device *dev,
4         unsigned int pipe,
5         void *transfer_buffer,
6         int buffer_length,
7         usb_complete_t complete,
8         void *context
9 );
```

2 → Ukazatel na již vytvořený URB, který chceme inicializovat.
3 → USB zařízení, kterému má být tento URB zaslán.
4 → Specifický koncový bod USB zařízení, kterému bude URB zaslán. Pro získání této hodnoty slouží specifické funkce, závisející na typu a směru komunikace. Přehled přináší obrázek 7.3 (převzatý z [9]):

5 → Ukazatel na datový buffer, který obsahuje odesílaná data v případě odesílání a přijatá data v případě příjmu. Tento buffer musí být vytvořen dynamicky pomocí `kmalloc()` nebo s předmapováním pro DMA (`usb_buffer_alloc()`), nikoliv staticky.
6 → Délka datového bufferu na který se odkazuje ukazatel `transfer_buffer`.
7 → Ukazatel na dokončovací funkci, která je volána pokud je požadavek na URB vyřízen.

Functions used to initialize the `pipe` field of the `urb` structure:

- Control pipes
  `usb_sndctrlpipe()`, `usb_rcvctrlpipe()`
- Bulk pipes
  `usb_sndbulkpipe()`, `usb_rcvbulkpipe()`
- Interrupt pipes
  `usb_sndintpipe()`, `usb_rcvintpipe()`
- Isochronous pipes
  `usb_sndisocpipe()`, `usb_rcvisocpipe()`

Prototype

send (out) receive (in)

```
unsigned int usb_[snd/rcv][ctrl/bulk/int/isoc]pipe(
     struct usb_device *dev, unsigned int endpoint);
```

Obrázek 7.3: Vytváření pipes

8 → Ukazatel na volitelnou datovou strukturu v ovladači, která může být využita v dokončovací funkci když je URB po předání do USB core předán zpět ovladači.

Inicializační funkce pro ostatní typy přenosů mají obdobnou strukturu[4], mají však navíc některé parametry odpovídající jejich typu. Zde byl zmíněn pro názornost pouze typ *bulk*, informaci o ostatní typech lze nalézt buď v [1], nebo přímo ve zdrojovém kódu jádra v souboru `/include/linux/usb.h` přístupném z [10].

Pokud je URB vytvořen a korektně inicializován, předá ho ovladač do USB core voláním funkce `usb_submit_urb()`. Funkce je deklarována následovně:

```
1 int usb_submit_urb(
2         struct urb *urb,
3         gfp_t mem_flags
4 );
```

Návratová hodnota rovna 0 znamená úspěch, záporná hodnota značí chybu.
2 → Ukazatel na cílený URB.
3 → Standardní alokační omezení (viz. kmalloc()).

K zrušení již předaného URB existují dvě funkce. Konkrétně k zrušení bez čekání slouží

```
int usb_unlink_urb(struct urb *urb);
```

a k zrušení s čekáním na potvrzení o dokončení:

```
int usb_kill_urb(struct urb *urb);
```

[4]kromě typu interrupt, pro který obdobná funkce není dostupná

kde parametr `urb` je analogicky k předchozím funkcím ukazatel na URB, který má být zrušen. Pokud zavoláme některou z těchto funkcí na URB, který nebyl předán ke zpracování, dostaneme chybovou návratovou hodnotu.

Jak již bylo zmíněno dokončovací funkce je callback volaný při dokončení URB. Tato funkce může však být vyvolána i v případě, že došlo k chybě při datovém přenosu, nebo byl potvrzený URB zrušen. Funkce dostává jako argument ukazatel `struct urb*` podle kterého se rozlišuje v kontextu kterého URB byl callback zavolán. Struktura `struct urb` obsahuje položku `int status`, která obsahuje informaci o stavu URB. Pokud je status roven 0, datový přenos byl korektně dokončen. U záporných hodnot pak mohou být rozlišeny další příčiny vyvolání dokončovací funkce. Např. `-ECONNRESET` znamená, že URB byl zrušen vyvoláním `usb_unlink_urb()`.

7.3.3 USB v GNU/Linux

Z uživatelského prostoru lze nejjednodušeji zobrazit informace o USB sběrnicích v systému a zařízeních k nim připojených zadáním příkazu `lsusb`. Příklad, jak může výpis vypadat:

```
Bus 007 Device 002: ID 03f0:171d Hewlett-Packard Wireless (Bluetooth + WLAN)
 Interface [Integrated Module]
Bus 007 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 006 Device 002: ID 08ff:2580 AuthenTec, Inc. AES2501 Fingerprint Sensor
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 003: ID 1669:1011 PiKRON Ltd. [hex]
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 002 Device 003: ID 04f2:b015 Chicony Electronics Co., Ltd VGA 24fps UVC Webcam
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Podrobný výpis informací o konkrétním zařízení lze docílit přepínačem `-v, --verbose` a specifikováním cíleného zařízení přepínačem `-s [[bus]:][devnum]` popř. `-d [vendor]:[product]`. Kontrétně tedy k zobrazení informací o přítomném převodníku (vztaženo k předešlému výpisu):

```
$ lsusb -v -s 5:3
```

popřípadě

```
$ lsusb -v -d 1669:1011
```

Jiná možnost je přístup do systému souborů *sysfs*. To je virtuální souborový systém, jehož obsah není fyzicky uložen na disku, ale je za běhu generován jádrem a zpřístupněn v uživatelském prostoru. *Sysfs* obsahuje informace o zařízeních a ovladačích a v adresářové

struktuře je přístupný přes `/sys`. Lze pomocí něj také modifikovat funkcionalitu zařízení[5], využívá jej např. správce zařízení *udev*. Informace o jednotlivých USB zařízeních se pak nacházejí ve složce `/sys/bus/usb/devices`. Tato struktura odráží příslušnost zařízení k jednotlivých sběrnicím[6], vlastní obsah však tvoří symbolické odkazy do struktury podadresářů v `/sys/devices/`, která odráží vlastní fyzické rozvržení.

V sysfs pak nalezneme jak samotné USB zařízení (reprezentováno struct usb_device), tak jednotlivá USB rozhraní (reprezentovány struct usb_interface). V případě převodníku, který obsahuje jediné USB rozhraní pak v sysfs může USB zařízení odpovídat:

```
/sys/devices/pci0000:00/0000:00:1d.0/usb5/5-2
```

a jeho USB rozhraní, se kterým je spjat ovladač:

```
/sys/devices/pci0000:00/0000:00:1d.0/usb5/5-2/5-2:1.0
```

Prvním údajem (v našem případě `usb5`) je tzv. *root hub*, tedy USB řadič. Ten má specifické číslo přidělené od USB core odpovídající tomu, jak byly jednotlivé řadiče postupně registrovány do systému. Jméno zařízení se skládá z tohoto čísla následovaného pomlčkou a číslem portu, ke kterému je dané zařízení připojeno. V našem případě má tedy převodník jako USB zařízení jméno `5-2`. Jméno jeho USB rozhraní je pak dáno jménem tohoto zařízení následovaným dvojtečkou, číslem konfigurace, tečkou a číslem rozhraní. V našem případě `5-2:1.0`, jelikož je to první konfigurace a má rozhraní s číslem 0. Podle [1] pak lze schéma pojmenování USB rozhraní zobecnit jako *root_hub-hub_port:config.interface* případně *root_hub-hub_port-hub_port:config.interface*.

Ovšem takto nelze zjistit např. nastavení koncových bodů daného rozhraní. Existuje však soubor generovaný jádrem, který obsahuje všechny tyto informace. Do verze jádra 2.6.31 existoval systém souborů *usbfs*, přístupný z adresáře `/proc/bus/usb/`. Uvedený soubor se pak nacházel v `/proc/bus/usb/devices`. Od verze jádra 2.6.31 došlo k defaultnímu zákazu *usbfs*, nicméně uvedený soubor lze nalézt v systému souborů *debugfs*, konkrétně se jedná o `/sys/kernel/debug/usb/devices`.

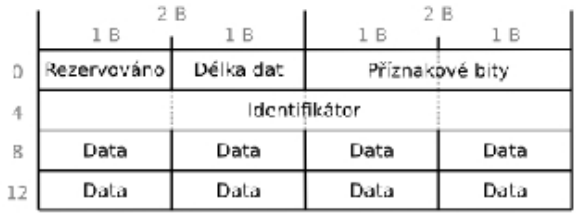
[5] například podle potřeby měnit aktivní USB konfigurace

[6] stejně tak např. pro PCI zařízení existuje adresář /sys/bus/pci/devices

7.4 Imlementace ovladače převodníku

Koncepčně vychází struktura obsluhy komunikace s převodníkem z USB implementace ovladače LinCAN. Pro datový přenos existují dva koncové body typu bulk, které se liší směrem komunikace. Jeden je přijímací (IN) a druhý odesílací (OUT). Přenos řídicích požadavků pro konfiguraci (např. nastavení komunikačních parametrů sběrnice) se pak provádí přes řídicí (control) koncový bod.

Datový přenos přes USB probíhá ve formě odeslání nebo příjmu datového bloku, který reprezentuje samotnou CAN zprávu. Struktura tohoto bloku je pevně dána, používá ji jak převodník tak LinCAN. Význam jednotlivých bytů je možné vidět na obrázku 7.4. K přenosu se používá formát little-endian, takže při přístupu do bytového pole, které tento datový blok reprezentuje, ovladač používá převodová makra. Pro ukládání např. `cpu_to_le32()` a při čtení např. `le32_to_cpu()`. Přenos řídicích požadavků probíhá také ve formě přenosu bytového pole ve formátu little-endian, ovšem zde je struktura závislá na typu požadavku.



Obrázek 7.4: Struktura datového bloku pro přenos CAN zprávy po USB

Poznámka: *V textu dále budou uváděny fragmenty zdrojového kódu ovladače pro větší názornost. Je potřeba si uvědomit, že mají mít spíše informativní význam (jako pseudokód). Neobsahují tedy nutně veškerou funkcionalitu, netestují se návratové hodnoty funkcí na zjištění chyb atd. V žádném případě tedy nejde o ekvivalent se skutečným zdrojovým kódem.*

7.4.1 Registrace

USB je specifická sběrnice z důvodu možnosti připojení zařízení za běhu systému (tzv. *hotplug*) bez nutnosti další konfigurace či restartu. Tomu musí samozřejmě odpovídat i podpora operačního systému. Ovladače jako takové mají samozřejmě nároky na systémové zdroje. Zavádět ovladače potenciálně připojitelných USB zařízení již při startu systému, když není jasné kdy a jestli vůbec bude konkrétní zařízení připojeno, by bylo značně neefektivní. V Linuxu proto existuje mechanismus registrace, kdy jaderný modul reprezentující ovladač nejprve přes speciální funkce zaregistruje do USB subsystému jím podporovaná zařízení (přes ID výrobce a produktu). Na základě těchto informací[7] je pak při fyzickém připojení zařízení USB subsystém schopen rozhodnout, který ovladač danému zařízení přísluší a zavést jej. Provedení těchto úkonů bude vysvětleno na programovém fragmentu ovladače CAN-USB převodníku:

[7] zjišťují se od zařízení při jeho enumeraci

Listing 7.1: Register to USB subsystem

```
1 #define CTU_USBCAN_VENDOR_ID      0x1669
2 #define CTU_USBCAN_PRODUCT_ID     0x1011
3
4 /* table of devices that work with this driver */
5 static struct usb_device_id ctu_usbcan_table [] = {
6         { USB_DEVICE(CTU_USBCAN_VENDOR_ID, CTU_USBCAN_PRODUCT_ID) },
7         { }      /* Terminating entry */
8 };
9
10 MODULE_DEVICE_TABLE(usb, ctu_usbcan_table);
```

1 - 2 → Definujeme ID výrobce a produktu daného USB zařízení.
5 - 8 → Informace pro subsystém popisuje struktura `struct usb_device_id`. Tyto struktury se sjednocují do pole, které má podobu viditelnou na těchto řádcích. V tomto případě převodník obsahuje tabulku s jedinou položku. K jejímu naplnění slouží makro `USB_DEVICE (vendor, product)`, které na základě zadaných parametrů vytvoří strukturu `struct usb_device_id`.
10 → Pomocí makra `MODULE_DEVICE_TABLE(type, name)` pak exportujeme údaje do uživatelského prostoru. Argumenty specifikujeme `usb` substystém[8] a tabulku s hodnotami.

Listing 7.2: Register to USB subsystem

```
11 static int ctu_usbcan_probe(struct usb_interface *intf,
12                                const struct usb_device_id *id)
13 {
14 /* later */
15 }
16
17 static void ctu_usbcan_disconnect(struct usb_interface *intf)
18 {
19 /* later */
20 }
21
22 /* usb specific object needed to register this
23    driver with the usb subsystem */
24 static struct usb_driver ctu_usbcan_driver = {
25         .name = "ctu_usbcan",
26         .id_table =     ctu_usbcan_table,
27         .probe =        ctu_usbcan_probe,
28         .disconnect =   ctu_usbcan_disconnect,
29 };
```

24 - 29 → Základní část ovladače USB zařízení je struktura `struct usb_driver`. Ovladač ji musí vytvořit a vyplnit nezbytné položky, tedy proměnné a callback funkce, které budou následně (po registraci) používány USB core. V příkladu převodníku je to:

25 → Název pod kterým bude ovladač v systému vystupovat.

26 → Ukazatel na tabulku tvořenou strukturami `struct usb_device_id`, která je definována na řádcích *5-8* a která obsahuje výčet zařízení podporovaných ovladačem.

[8] toto makro používají i jiné subsystémy, např. pci, pcmcia atd.

27 → Ukazatel na funkci `probe()`. To je callback funkce, kterou volá USB core v případě, že bylo připojeno některé USB zařízení a ovladač by měl být schopen obsloužit jeho specifické USB rozhraní předané jako argument (struktura `struct usb_interface`). Zde předáváme ukazatel na funkci jejíž definice začíná na řádku *11*, detailněji bude popsána později.

28 → Ukazatel na funkci `disconnect()`. To je callback funkce, kterou volá USB core v případě, kdy by ovladač již dále neměl mít kontrolu nad rozhraním. Např. pokud dojde k odpojení zařízení. Zde předáváme ukazatel na funkci jejíž definice začíná na řádku *17*, detailněji bude popsána později.

Listing 7.3: Register to USB subsystem

```
30 static int __init ctu_usbcan_init(void)
31 {
32         int result;
33
34         printk(KERN_INFO "CTU_USBCAN_kernel_driver_loaded\n");
35
36         /* register this driver with the USB subsystem */
37         result = usb_register(&ctu_usbcan_driver);
38         if (result)
39                 err("usb_register_failed._Error_number_%d", result);
40
41         return result;
42 }
43
44 static void __exit ctu_usbcan_exit(void)
45 {
46         printk(KERN_INFO "CTU_USBCAN_kernel_driver_unloaded\n");
47
48         /* deregister this driver with the USB subsystem */
49         usb_deregister(&ctu_usbcan_driver);
50 }
51
52 module_init(ctu_usbcan_init);
53 module_exit(ctu_usbcan_exit);
```

Tyto řádky obsahují téměř totožnou funkcionalitu, jako v případě základního jaderného modulu v sekci 7.1.

30 - 42 → Funkce, která se má zavolat po nahrání modulu do jádra.

37 → Volání funkce `usb_register()`, která registruje ovladač do USB subsystému. Jako argument předáváme ukazatel na strukturu `struct usb_driver` definovanou na řádcích *24-29*.

44 - 50 → Funkce, která se má zavolat při odstanění modulu z jádra.

49 → Volání funkce `usb_deregister()`, která odhlásí strukturu `struct usb_driver` z jádra. Tímto dochází k logickému odpojení vazeb mezi ovladačem a USB rozhraním jehož funkcionalitu zajišťuje. Tudíž je pro toho rozhraní volána funkce `disconnect()`[9], v tomto případě definovaná od řadku *17*.

[9] pokud již není zařízení fyzicky odpojeno

7.4.2 Privátní data

Jak již bylo zmíněno v sekci 7.2, v ovladačích se nepoužívají globální proměnné, ale namísto toho proměnné použitelné z více míst ovladače utvoří jednu strukturu, která pak tvoří privátní data ovladače. V impementaci převodníku vypadá tato struktura následovně:

Listing 7.4: Struktura privátních dat

```
1  /* Structure to hold all of our device specific stuff */
2  struct ctu_usbcan_usb {
3
4          /* CAN common private data */
5          struct can_priv can; /* must be the first member */
6          /* CAN hardware-dependent bit-timing constant*/
7          struct can_bittiming_const btc;
8          /* CAN system clock frequency in Hz */
9          u32 can_clock;
10
11         /* the usb device for this device */
12         struct usb_device *udev;
13         /* the net device for this device */
14         struct net_device *netdev;
15
16         /* the address of the bulk in endpoint */
17         u8 bulk_in_endpointAddr;
18         /* the address of the bulk out endpoint */
19         u8 bulk_out_endpointAddr;
20
21         /* URBs waiting for data receive */
22         struct list_head rx_pend_list;
23         /* URBs with valid received data */
24         struct list_head rx_ready_list;
25         /* URBs prepared to hold Tx messages */
26         struct list_head tx_idle_list;
27         /* URBs holding Tx messages in progress */
28         struct list_head tx_pend_list;
29         /* URBs with yet confirmed Tx messages */
30         struct list_head tx_ready_list;
31
32         /* list lock */
33         spinlock_t list_lock;
34         /* usb communication kernel thread  */
35         struct task_struct *comthread;
36
37         volatile long flags;
38 };
```

Význam jednotlivých položek by měl být srozumitelný z komentářů. Máme zde proměnné týkající se USB i síťového subsystému, adresy koncových bodů[10], spojové seznamy pro příchozí i odchozí zprávy (vysvětleny později) a proměnné týkající se samotného CAN zařízení. Ty jsou použity pro spočtení a nastavení časových parametrů sběrnice.

[10] Používáme jeden IN BULK a jeden OUT BULK koncový bod pro datové přenosy

7.4.3 Připojení a odpojení zařízení

7.4.3.1 Funkce `probe()`

Jak již bylo řečeno v předchozí sekci, `probe()` je callback funkce, kterou volá USB core v případě, že bylo připojeno některé USB zařízení a ovladač by měl být schopen obsloužit jeho specifické USB rozhraní předané jako argument (struktura `struct usb_interface`). Ukazatel na tuto funkci definovanou v ovladači se předává při registraci ovladače. Deklarace tohoto ukazatele na funkci vypadá následovně:

```
int (*probe) (struct usb_interface *intf, const struct usb_device_id *id);
```

Vidíme, že jako argument je předán i ukazatel na strukturu `struct usb_device_id`, na základě které USB core provedl volání. Pokud ovladači náleží obsluha daného USB rozhraní, vrací honotu 0. Záporná návratová hodnota značí neúspěch.

V této funkci ovladač obvykle inicializuje svoje datové struktury. Typické je zjištění adres a velikostí bufferů koncových bodů, které budou použity pro komunikaci. Tato funkce je volána v kontextu vlákna USB rozbočovače, tudíž je možné volat funkce, které můžou způsobit usnutí. Obecně se ale doporučuje provést pouze nejnutnější funkcionalitu a většinu práce provádět až při otevření zařízení uživatelem. Je to kvůli tomu, že přidávání a odebírání zařízení probíhá pouze v jednom vlákně. Mohlo by tak být způsobeno např. zpomalení detekce zařízení systémem. Zjednodušená funkce `probe()` pro CAN-USB převodník vypadá následovně:

Listing 7.5: Funkce probe()

```
1  static int ctu_usbcan_probe(struct usb_interface *intf,
2                              const struct usb_device_id *id)
3  {
4          struct net_device *netdev;
5          struct ctu_usbcan_usb *dev;
6
7          netdev = alloc_candev(sizeof(struct ctu_usbcan_usb),
8                                USBCAN_TOT_TX_URBS);
9          dev = netdev_priv(netdev);
10
11         dev->udev = interface_to_usbdev(intf);
12         dev->netdev = netdev;
13
14         netdev->netdev_ops = &ctu_usbcan_netdev_ops;
15         netdev->flags |= IFF_ECHO;
16
17         /* set up the endpoint information */
18         /* ... */
19
20         usb_set_intfdata(intf, dev);
21         SET_NETDEV_DEV(netdev, &intf->dev);
22
23         if (get_bittiming_constants(dev)){
24                 err("Could not get bittiming constants\n");
```

```
25                  goto exit;
26          }
27
28          dev->can.clock.freq = dev->can_clock;
29          dev->can.bittiming_const = &dev->btc;
30          dev->can.do_set_bittiming = ctu_usbcan_set_bittiming;
31          dev->can.do_set_mode = ctu_usbcan_set_mode;
32
33          err = register_candev(netdev);
34
35 exit:
36          return 0;
37 }
```

7 → Alokuje a nastaví strukturu reprezentující CAN jako síťové zařízení. Argumenty jsou velikost struktury pro privátní data a počet URBs pro odesílání.
9 → Získání ukazatele na strukturu privátních dat.
11 → Uložení ukazatele na USB zařízení získaného z USB rozhraní.
12 → Uložení ukazatele na síťové zařízení.
14 → Předání struktury s ukazateli na řídicí funkce síťového rozhraní.
15 → Nastavení informace, že ovladač podporuje *local loopback*.
18 → Získání informací o koncových bodech a uložení do struktury privátních dat ze struktury `struct usb_interface`
20 → Poskytnutí privátních dat USB rozhraní.
21 → Nastaví sysfs referenci fyzického zařízení síťovému logickému zařízení.
23 → Volání funkce, která má za úkol z převodníku zjistit parametry CAN řadiče pro následný výpočet časových parametrů. Bude vysvětlena později.
28 - 31 → Pokud proběhlo získání parametrů korektně (řádka *23*), dochází k jejich předání do struktury `struct can_priv` v privátních datech. Z hodnot v ní totiž vychází funkcionalita výpočtu časových parametrů CAN sběrnice, která je volána při povolování zařízení uživatelem prostřednictvím nástrojů z `iproute2`.
33 → CAN zařízení je zaregistrováno do síťového subsystému.

Po připojení zařízení a provedení funkce probe() by již v systému mělo být zařízení zobrazeno jako CAN rozhraní (např. `can0`), což je možné ověřit zavoláním:

```
$ cat /proc/net/dev
```

Zatím však zůstává neaktivní. Je nutné mu nastavit parametry a následně ho povolit.

7.4.3.2 Funkce `disconnect()`

Jak již bylo řečeno, `disconnect()` je callback funkce, kterou volá USB core v případě, kdy by ovladač již dále neměl mít kontrolu nad rozhraním (např. pokud dojde k odpojení zařízení). Ukazatel na tuto funkci definovanou v ovladači se předává při registraci ovladače. Zjednodušená funkce `disconnect()` pro CAN-USB převodník vypadá následovně:

Listing 7.6: Funkce disconnect()

```
1 static void ctu_usbcan_disconnect(struct usb_interface *intf)
2 {
3         struct ctu_usbcan_usb *dev = usb_get_intfdata(intf);
4         usb_set_intfdata(intf, NULL);
5
6         if (dev) {
7                 unregister_netdev(dev->netdev);
8                 free_candev(dev->netdev);
9         }
10 }
```

3 → Získání ukazatele na strukturu privátních dat.
4 → Nastavení ukazatele na privátní data USB zařízení na NULL k zabránění dalších (chybných) přístupů k těmato datům.
7 → Odebrání zařízení z jádra.
8 → Uvolnění z paměti (pokud existuje pouze jedna reference).

7.4.4 Funkce síťového rozhraní

Ve výpisu 7.5 funkce `probe()` byla na řádku *14* zmíněna velmi důležitá věc. Jde o vyplnění položky `netdev_ops` struktury `struct net_device`, což je ukazatel na strukturu `struct net_device_ops`. Tato struktura definuje ukazatele na řídicí funkce síťových rozhraní a musí být vyplněna před registrací zařízení do síťového subsystému. CAN zařízení obvykle implementuje pouze tři funkce, které lze vidět na kódu převodníku:

```
static const struct net_device_ops ctu_usbcan_netdev_ops = {
    .ndo_open = ctu_usbcan_open,
    .ndo_stop = ctu_usbcan_close,
    .ndo_start_xmit = ctu_usbcan_start_xmit,
};
```

První položka struktury je ukazatel na funkci `ndo_open()`, která se volá při povolování zařízení. Druhý je ukazatel na funkci `ndo_stop()`, která se volá při zakazování zařízení. Poslední je ukazatel na funkci `ndo_start_xmit()`, která se volá v případě potřeby přenosu paketu.

Packet scheduler má pro každé síťové zařízení odesílací frontu. Při aktivaci zařízení (ndo_open()) musí být tato fronta spuštěna. Poté může packet scheduler volat funkci `ndo_start_xmit()` (v kontextu softIRQ) v případě potřeby přenosu paketu [3].

7.4.5 Nastavení, povolení a zakázání CAN zařízení

CAN zařízení lze nastavovat přes netlink rozhraní pomocí programu `ip` z `iproute2`, což je kolekce nástrojů týkajících se správy síťové oblasti. Pomocí `ip` lze povolit/zakázat zařízení, nastavovat časové parametry či zobrazovat statistiky. Podrobný přehled lze nalézt v dokumentaci k SocketCAN v sekci 6.5 (soubor `Documentation/networking/can.txt`).

7.4.5.1 Nastavení

Parametry časování komunikace CAN sběrnice mohou být vždy vyjádřeny v hardwarově nezávislém formátu jak plyne ze specifikace. SocketCAN obsahuje funkcionalitu, která umožňuje tyto parametry vypočíst pro cílený CAN řadič na základě požadované komunikační rychlosti, pracovní frekvence řadiče a intervalů možných hodnot, jakými může být řadič nastaven. Nastavení CAN zařízení se záměrem komunikovat rychlostí 1 Mb/s pak může vypadat:

```
$ ip link set can0 type can bitrate 1000000
```

Zde specifikujeme požadovanou rychlost, ostatní parametry pro výpočet však musí být již k dispozici. Právě jejich získání se provádí ve funkci `probe()` popisované v sekci 7.4.3.1, konkrétně ve výpisu 7.5 na řádcích *23-31*. Zde se tedy nejprve volá funkce, která má za úkol z převodníku zjistit parametry CAN řadiče. Pokud proběhne korektně, dochází k předání parametrů do struktury `struct can_priv` v privátních datech. Konkrétně frekvence řadiče a ukazatel na strukturu `struct can_bittiming_const`, která obsahuje zmíněné intervaly možných hodnot, kterými může být řadič nastaven. Pokud pak dojde k požadavku o nastavení jako v příkladu výše, dojde k provedení funkcionality pro výpočet parametrů pro nastavení CAN řadiče a poté k zavolání funkce definované přes ukazatel na funkci s prototypem:

```
int (*do_set_bittiming)(struct net_device *dev);
```

Tedy ve výpisu 7.5 na řádku *30*. Daná funkce má pak zajistit samotné nastavení CAN řadiče převodníku podle vypočtených parametrů. Ty lze získat z privátních dat opět ze struktury `struct can_priv`, která disponuje položkou `struct can_bittiming`. Její položky obsahují právě vypočtené parametry.

7.4.5.2 Povolení

Po nastavení komunikačních parametrů lze CAN zařízení povolit. To se provádí jako povolení kteréhokoliv jiného síťového zařízení, tedy např.:

```
$ ip link set can0 up
```

Při povolení zařízení vyvolá síťový subsystém callback funkci open(), registrovanou dříve. V této funkci musí ovladač dokončit konfiguraci a po jejím provedení musí být schopen přijímat a odesílat CAN zprávy. V našem případě tato funkce zavolá funkci `open_candev()` což je obecná funkce, která je volána při povolení zařízení. Dále je třeba povolit odesílání samotných zpráv, tedy umožnit vyšším vrstvám volat odesílací funkci. To provedeme přes `netif_start_queue()`. Obě zmíněné funkce se volají s argumentem síťového zařízení, tedy s ukazatelem na strukturu `struct net_device` a ten dostává jako argument volaná funkce open(). Jejím posledním krokem je spuštění vlákna, které bude mít na starosti obsluhu veškeré komunikace.

7.4.5.3 Zakázání

Zákaz CAN zařízení lze provést jako zakázání kteréhokoliv jiného síťového zařízení, tedy např.:

```
$ ip link set can0 down
```

Při zakázání zařízení vyvolá síťový subsystém callback funkci close(), registrovanou dříve. V této funkci se provedou kroky analogické k povolení, pouze s opačným efektem. Tedy nejprve požadavek na ukončení obslužného vlákna, dále zastavení odesílací fronty přes `netif_stop_queue()` a nakonec funkce `close_candev()`.

7.4.6 Obslužné vlákno a fronty

Dále existuje oddělené vlákno, které se o průběh komunikace stará. Toto vlákno je spuštěno ve funkci `ctu_usbcan_open()` při povolení rozhraní a zastaveno ve funkci `ctu_usbcan_close()` při zakázání rozhraní. Ve funkci, kterou vlákno vykonává, dochází k alokaci a nastavení prostředků USB komunikace zvaných URB. Jejich obecné použití bylo vysvětleno v sekci 7.3.2.5. V systému se pak pracuje s komunikačními objekty, které reprezentuje struktura:

```
struct usbcan_message {
    struct urb *u;
    struct ctu_usbcan_usb *dev;
    u8 msg[USBCAN_TRANSFER_SIZE];
    u32 echo_index;
    u8 dlc;
    struct list_head list_node;
};
```

V ní je tedy odkaz na související URB a také na privátní data. Dále bytové pole pro přenášená data (`msg`), délka samotné CAN zprávy (`dlc`), pro odesílané zprávy index pro local loopback funkcionalitu (`echo_index`) a poslední je položka pro potřeby umístění objektu do front (`list_node`).

Existuje pak několik front těchto objektů, které určují jejich životní cyklus. Pro příjem jsou to:

- **rx_pend_list** - URB těchto objektů je poskytnut do USB core a čeká na příjem dat
- **rx_ready_list** - URB těchto přijímacích objektů byl vyřízen, mají k dispozici platná přijatá data

Pro odesílání jsou to:

- **tx_idle_list** - Objekty připravené k použití pro odeslání zprávy
- **tx_pend_list** - URB poskytnut do USB core k odeslání platných dat

- **tx_ready_list** - URB těchto odesílacích objektů byl vyřízen, obsahují dosud potvrzené odeslané zprávy

Z použití front je patrné, že dochází k alokaci několika těchto odesílacích a přijímacích bloků před začátkem komunikace a následně jsou pouze přemísťovány v rámci front. Jejich počet je definován na základě maker `USBCAN_TOT_RX_URBS` a `USBCAN_TOT_TX_URBS`. Tento způsob snižuje režii na alokaci struktur oproti možnému dynamickém vytváření při potřebě přenosu. Příklad alokace v následujícím fragmentu:

Listing 7.7: Alokace a nastavení komunikačních objektů

```
1  /*
2  dev -> pointer to private data
3  (struct ctu_usbcan_usb *dev)
4  */
5
6  struct usbcan_message *m;
7  struct urb *u;
8
9  /* Prepare transmit urbs  */
10 for (i=0;i<USBCAN_TOT_TX_URBS;i++){
11
12         u = usb_alloc_urb(0, GFP_KERNEL);
13         m = kzalloc(sizeof(struct usbcan_message), GFP_KERNEL);
14
15         u->dev = dev->udev;
16         m->u = u;
17         m->dev = dev;
18         m->echo_index = i;
19
20         usb_fill_bulk_urb(u, dev->udev,
21                 usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
22                 m->msg, USBCAN_TRANSFER_SIZE, ctu_usbcan_tx_callback, m);
23
24         list_add_tail(&m->list_node, &dev->tx_idle_list);
25 }
```

Po alokaci a nastavení přijímacích a odesílacích komunikačních prostředků vlákno začíná provádět nekonečnou smyčku, která vyřizuje komunikaci. Pokud neprobíhá přenos zpráv, je vlákno uspáno a k jeho opětovnému vzbuzení dochází v mechanismech dokončovacích callback funkcí po vyřízení URB požadavku. Pokud přijde požadavek na ukončení vlákna, ukončí a uvolní všechny jím alokované prostředky (komunikační objekty, URBs).

7.4.7 Příjem

Při příjmu dochází k vyvolání callback funkce definované při nastavování URB viz. sekce 7.3.2.5. Zjištění, kterého komunikačního objektu se vyvolání týka a zda byl příjem dat úspěšný, lze vidět v následujícím fragmentu kódu:

Listing 7.8: rx callback

```
1 static void ctu_usbcan_rx_callback(struct urb *urb)
```

```
2 {
3
4         struct usbcan_message *m = urb->context;
5
6         if(urb->status != 0){
7                 err("RX callback: error");
8                 return;
9         }
10
11        /* ... */
12
13 }
```

Další funkcionalitou, která je zaznačena tečkami na řádku *11* je nastavení příznaku platných přijatých dat `USBCAN_DATA_OK`, na základě kterého je vzbuzeno vlákno starající se o komunikaci a přesun komunikačního objektu z `rx_pend_list` do `rx_ready_list`.

Vlákno obsluhuje příjem tak, že testuje obsazenost právě `rx_ready_list`. Dokud není fronta prázdná, volá `usbcan_kthread_read_handler()`, kterému předává komunikační objekt z vrcholu fronty. Tato funkce představuje samotné zpracování přijaté CAN zprávy a poskytuje následné předání vyšším síťovým vrstvám v příslušné formě. Pro představu následuje fragment kódu:

Listing 7.9: Funkce vlákna obsluhující příjem

```
1  void usbcan_kthread_read_handler(struct ctu_usbcan_usb *dev,
2                                   struct usbcan_message *m)
3  {
4
5          struct can_frame *cf;
6          struct sk_buff *skb;
7          struct net_device_stats *stats = &dev->netdev->stats;
8
9          skb = alloc_can_skb(dev->netdev, &cf);
10
11         /*... to cf from m->msg ... */
12
13         netif_rx(skb);
14
15         stats->rx_packets++;
16         stats->rx_bytes += cf->can_dlc;
17
18         /*... */
19
20 }
```

5 → Struktura `struct can_frame` je základní podoba CAN rámce, se kterou SocketCAN pracuje.
6 → Struktura `struct sk_buff` popisuje socket buffer v síťovém subsystému.
9 → Alokuje socket buffer, který bude označen jako nositel CAN zprávy (viz. 6.3.3) a nastaví (`cf`) jako ukazatel do jeho datové oblasti.
11 → Naznačeno vyplnění struktury (`cf`) hodnotami z datového bufferu přenesených dat

(`m->msg`).
13 → Předání paketu vyšší vrstvě - packet scheduler (obrázek 6.1 část *b*).
15 - 16 → Úprava přijímacích statistik.
18 → Další funkcionalitou je přesun komunikačního objektu do `rx_pend_list` a opětovné poskytnutí jeho URB do USB core k přijmu další zprávy.

Takto vypadá cyklus přijetí zprávy ovladačem. Ve stručnosti zprostředkovává příjem od nižší vrstvy (USB subsystém) a předává vyšší vrstvě (síťová vrstva - SocketCAN).

7.4.8 Odesílání

Cyklus odeslání zprávy začíná zavoláním funkce `ctu_usbcan_start_xmit()` registrované dříve. To značí požadavek na odeslání CAN zprávy.

Listing 7.10: Odesílání zprávy

```
1  static netdev_tx_t ctu_usbcan_start_xmit(struct sk_buff *skb,
2                                           struct net_device *netdev)
3  {
4          struct ctu_usbcan_usb *dev = netdev_priv(netdev);
5          struct can_frame *cf = (struct can_frame *)skb->data;
6
7          struct usbcan_message *m;
8          m = list_first_entry(&dev->tx_idle_list, typeof(*m), list_node);
9
10         /* ...from cf to m->msg... */
11
12         /* put on stack - local echo */
13         can_put_echo_skb(skb, netdev, m->echo_index);
14
15         /* ... */
16
17         /* slow down tx path if needed */
18         if(list_empty(&dev->tx_idle_list))
19                 netif_stop_queue(netdev);
20
21         return NETDEV_TX_OK;
22 }
```

Vidíme, že se nám zde logicky objevují stejné struktury jako při přijmu, tedy `struct can_frame` reprezentující CAN rámec SocketCAN a `sk_buff` reprezentující socket buffer sloužící k přenosu paketů síťovou vrstvou.

5 → Datová oblast `skb->data` reprezentuje CAN rámec (`struct can_frame`)
8 → První komununikační objekt z `tx_idle_list` tedy volně použitelný k odeslání zprávy
10 → Naznačeno vyplnění hodnot datového bufferu odesílaných dat (`m->msg`) ze struktury (`cf`). Tedy analogicky opačný proces k příjmu.
13 → Buffer je uložen pro účely local loopack (viz. 6.3.2)
15 → Další funkcionalitou je přesun komunikačního objektu do `tx_pend_list` a poskytnutí jeho URB do USB core k odeslání zprávy.
18 - 19 → Pokud již není žádný volný komunikační objekt v `tx_idle_list`, musíme zastavit odesílací frontu a znemožnit tak další volání `ctu_usbcan_start_xmit()`.

Pokud došlo k úspěšnému vyřízení URB a tedy odeslání zprávy na sběrnici, dochází k vyvolání callback funkce `ctu_usbcan_tx_callback()` obdobně jako u příjimací callback funkce popsané v předchozí sekci. Struktura a funkcionalita je obdobná, pouze komunikační objekt je přesunut do fronty příslušející odesílání (`tx_ready_list`). Navíc jsou zde pouze upraveny odesílací statistiky a zavolána funkce `can_get_echo_skb()`, kdy je socket buffer vyzvednut zpět ze zásobníku pro účely local loopback a je tak vytvořeno "echo"právě odeslané zprávy.

Vlákno dokončuje odeslání tak, že testuje obsazenost `tx_ready_list`. Dokud není fronta prázdná, volá `usbcan_kthread_write_handler()`, kterému předává komunikační objekt z vrcholu fronty. Princip tedy opět analogický k příjmu. V této funkci však dochází pouze ke dvěma úkonům. Prvním je přesunutí komunikačního objektu do `tx_idle_list`, kde bude tedy opět volně použitelný pro odesílání dalších zpráv. Dále pokud je zastavena odesílací fronta, musíme ji opět spustit pro umožnění volání `ctu_usbcan_start_xmit()`. Úkony úpravy odesílacích statistik a echo by se jistě daly provádět i v této funkci. V dokončovací callback funkci jsou umístěny pouze pro logickou vazbu s upozorněním na fyzické odeslání.

Listing 7.11: Funkce vlákna dokončující cyklus odesílání

```
1 void usbcan_kthread_write_handler(struct ctu_usbcan_usb *dev,
2                                   struct usbcan_message *m)
3 {
4         usbcan_usb_message_move_list(dev, m, &dev->tx_idle_list);
5
6         if (netif_queue_stopped(dev->netdev))
7                 netif_wake_queue(dev->netdev);
8 }
```

Takto vypadá cyklus odeslání zprávy ovladačem. Ve stručnosti zprostředkovává příjem od vyšší vrstvy (síťová vrstva - SocketCAN) a předává nižší vrstvě (USB subsystém).

Kapitola 8

Testování

Vývoj i testování probíhaly s reálným využitím sběrnice CAN. Druhý prvek na sběrnici, se kterým převodník prostřednictvím CAN komunikoval, byl starší CAN-USB převodník implementovaný v roce 2008 v rámci bakalářské práce J. Kříže s názvem *Řadič sběrnice CAN připojený k PC přes sběrnici USB*[1]. Tento převodník je založen na mikrokontroléru LPC2148, obsahuje externí CAN řadič SJA1000 a podporuje ho ovladač LinCAN. Sestavu je možné vidět na obrázku 8.1. Převodník navržený v této práci (s LPC1768) byl pak připojen k notebooku s frekvencí procesoru 2GHz. Starší převodník (s LPC2148) byl pak připojen k PC s frekvencí procesoru 2,8GHz. Oba počítače mají architekturu *IA-32* a operační systém *GNU/Linux Ubuntu 10.04* se standardním jádrem *2.6.32-35*.

Jelikož komunikační protokol po USB sběrnici mezi oběma převodníky a počítačem je velmi podobný [2], podařilo se s několika drobnými úpravami ovladače založeného na SocketCAN poskytnout staršímu převodníku podporu tímto ovladačem [3]. Takto mohly být v rámci jednoho testování pokryty veškeré požadavky ze zadání. Tedy konkrétně:

- Otestovat součinnost implementace firmware převodníku s ovladačem pro USB zařízení LinCAN.
- Otestovat součinnost implementace firmware převodníku s vyvinutým ovladačem pro USB zařízení založeném na SocketCAN.
- Porovnat vlastnosti sítového ovladače založeného na SocketCAN se znakovým ovladačem LinCAN [4].

8.1 Průběh testů

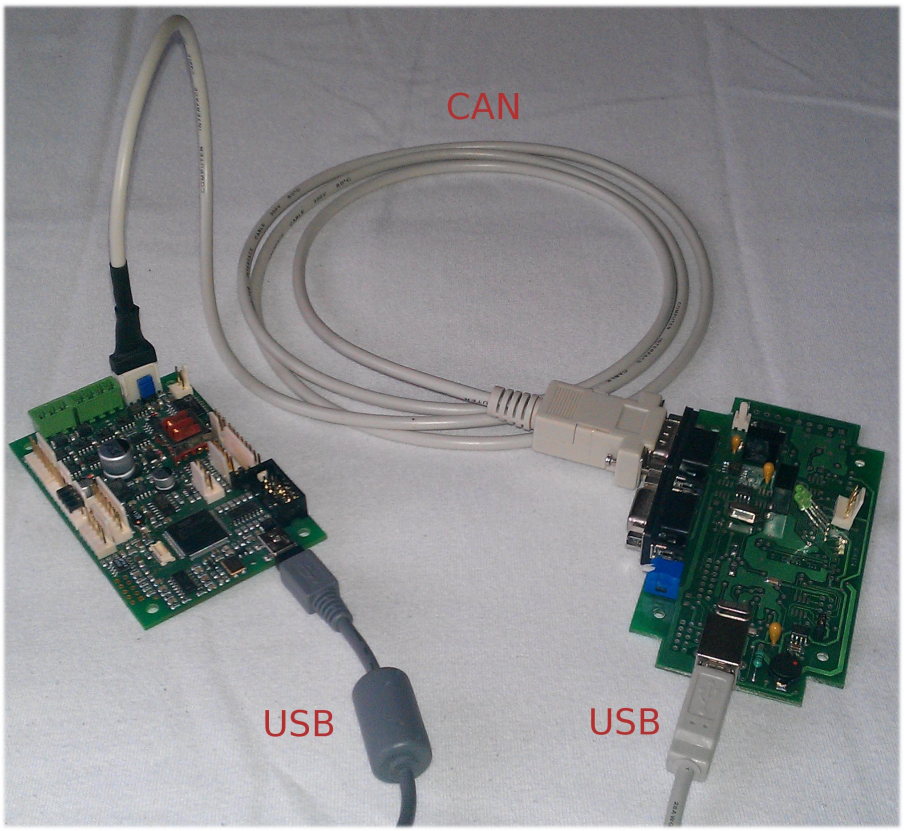
Samotné testování vychází z článku [6], který se zabývá časovou analýzou Linuxových ovladačů pro CAN. Byl měren tzv. *round trip time* tedy čas mezi odesláním zprávy druhému

[1] práce je dostupná z [8]

[2] záměrně - kvůli budoucímu sloučení funkcionality pro jednoduchou volbu podle aktuálně požadového typu hardware

[3] pouze pro účely tohoto testování, k plné podpoře se musí upravit firmware převodníku

[4] buď obě zařízení využívají LinCAN, nebo obě zařízení využívají ovladač založený na SocketCAN



Obrázek 8.1: Sestava hardware pro vývoj a testování

uzlu a jejím následném opětovném přijetí. K tomu byl využit nástroj *canping*[5]. Tato aplikace umožňuje odesílat zprávy a měřit čas, který uplyne, než dostane odpověď. Další zpráva je odeslána až po obdržení odpovědi na předchozí odeslanou zprávu. Aplikace se spouští z příkazové řádky a pomocí přepínačů lze specifikovat různá nastavení jako počet vláken, počet odesílaných zpráv, časovou prodlevu mezi zprávami atd. Naměřené časy jsou statisticky zpracovány mohou z nich být vytvořeny histrogramy. K přístupu k oboum typům ovladačů *canping* používá knihovnu *VCA* (Vitual CAN API). Ta je součástí projektu *Ort-CAN* (OCERA Real-Time CAN) a informace o ní lze nalézt na [11].

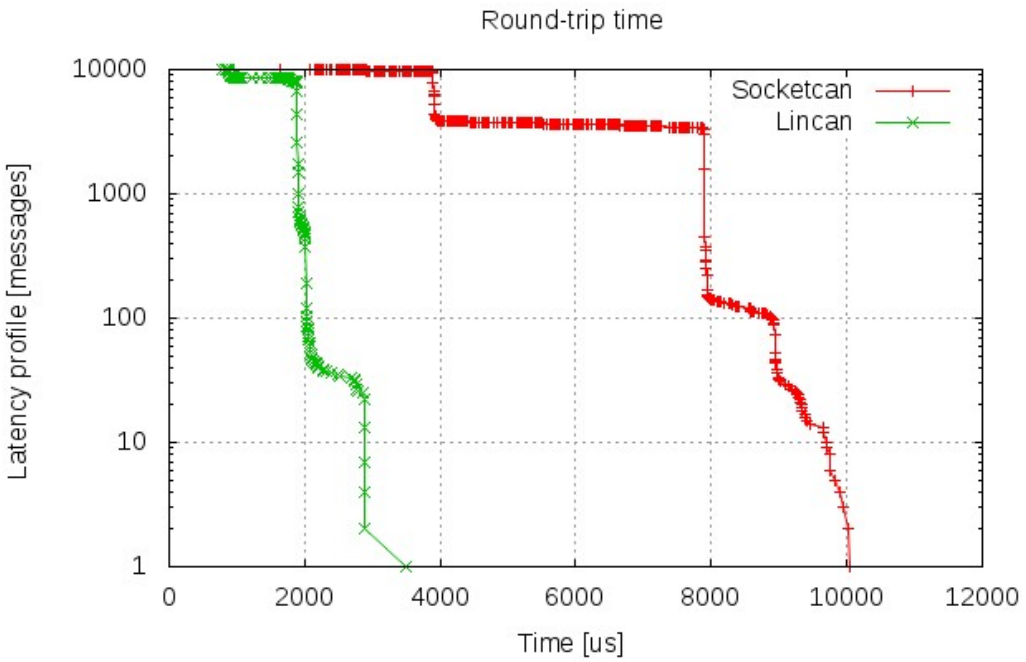
Bylo provedeno několik testů s proměnnými parametry počtu odesílacích/přijímacích vláken a časové prodlevy mezi odesíláním jednotlivých zpráv. V každém testu bylo každým vláknem odesláno 10 000 zpráv. Rychlost sběrnice (*bitrate*) byla nastavena na 1 Mb/s. Z naměřených dat byly vytvořeny grafy. Každý graf zobrazuje tzv. *latency profile*. To je kumulativní histogram s převrácenou vertikální osou v logaritmickém měřítku. Horizontální osu pak tvoří uplynulý čas. Z tohoto grafu je pak možné dobře vidět přesný počet zpráv s nejhorší latencí[6]. Tyto grafy je možné vidět na obrázcích, které budou následovat. Pouze

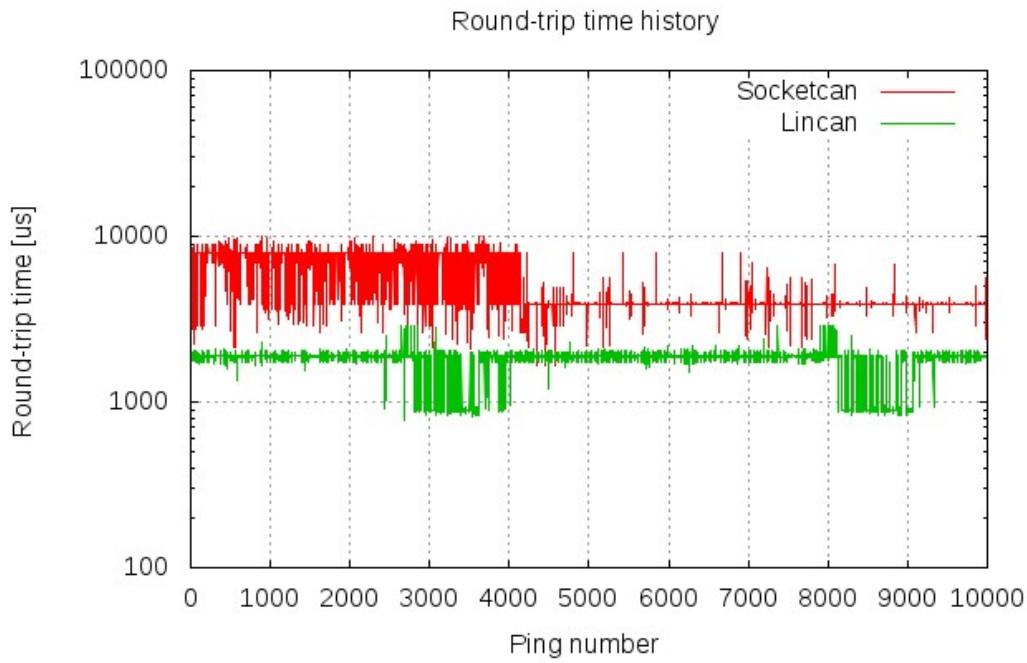
[5] http://rtime.felk.cvut.cz/gitweb/canping.git

[6] Obyčejné histogramy nejsou pro zjištění maximálního zpoždění vhodné, jelikož ho většinou reprezentouje pouze velmi malý počet zpráv. Hodnotu tohoto maxima je pak v grafu těžké odhalit

na obrázku 8.3 je pro zajímavost možné vidět přehled historie *round trip time* všech zpráv daného testu.

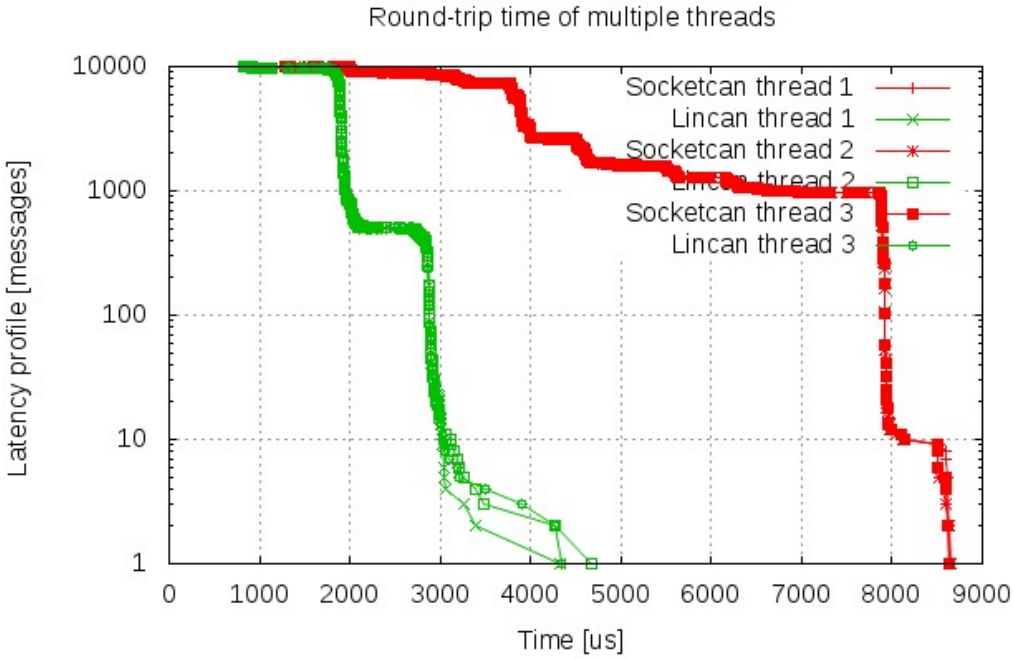
Obrázek 8.2: Round trip time - 1 vlákno, nulová prodleva

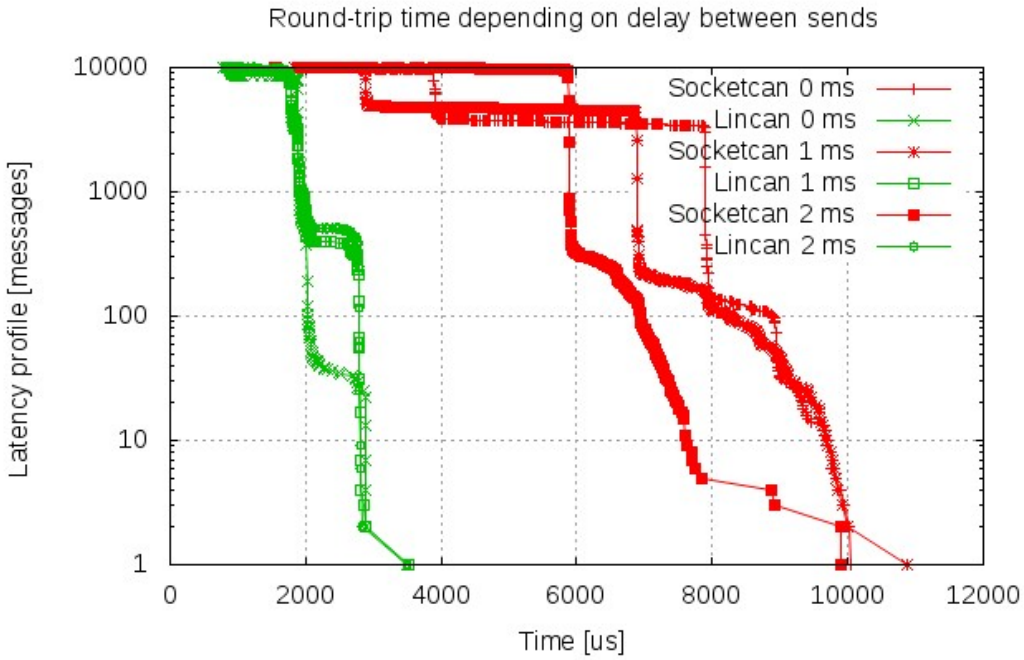




Obrázek 8.3: Round trip time historie - 1 vlákno, nulová prodleva

Obrázek 8.4: Round trip time - 3 vlákna, nulová prodleva





Obrázek 8.5: Round trip time - 1 vlákno, prodleva 0, 1 a 2 ms

Pro zajímavost je zde již pouze textově uveden výpis z testování s deseti paralelními vlákny s nulovou prodlevou. Pro SocketCAN je to:

```
$ sudo ./vca_canping -m 10 -d socketcan:can0 -c 10000 -w 0
Summary statistics:
Id 1000: count = 10000 mean = 3988.86 stddev =  611.34 min =  1911 max =  8127 [us] loss = 0% (0)
Id 1002: count = 10000 mean = 3987.05 stddev =  607.88 min =  1913 max =  8010 [us] loss = 0% (0)
Id 1004: count = 10000 mean = 3984.19 stddev =  600.74 min =  1924 max =  7961 [us] loss = 0% (0)
Id 1006: count = 10000 mean = 3989.89 stddev =  618.49 min =  1884 max =  8123 [us] loss = 0% (0)
Id 1008: count = 10000 mean = 3995.95 stddev =  637.82 min =  1952 max =  8500 [us] loss = 0% (0)
Id 1010: count = 10000 mean = 3984.37 stddev =  648.53 min =  1896 max =  8924 [us] loss = 0% (0)
Id 1012: count = 10000 mean = 4017.68 stddev =  686.59 min =  1272 max =  8997 [us] loss = 0% (0)
Id 1014: count = 10000 mean = 4009.11 stddev =  663.80 min =  1926 max =  8974 [us] loss = 0% (0)
Id 1016: count = 10000 mean = 3979.38 stddev =  632.11 min =  1869 max =  8051 [us] loss = 0% (0)
Id 1018: count = 10000 mean = 3991.75 stddev =  629.80 min =  1896 max =  7986 [us] loss = 0% (0)
```

Pro LinCAN pak výpis vypadá následovně:

```
$ sudo ./vca_canping -m 10 -d /dev/can0 -c 10000 -w 0
Summary statistics:
Id 1000: count = 10000 mean = 2784.20 stddev =  588.65 min =  1821 max =  5294 [us] loss = 0% (0)
Id 1002: count = 10000 mean = 2785.21 stddev =  575.36 min =  1863 max =  4866 [us] loss = 0% (0)
Id 1004: count = 10000 mean = 2806.85 stddev =  567.81 min =  1664 max =  6905 [us] loss = 0% (0)
Id 1006: count = 10000 mean = 2802.88 stddev =  582.11 min =  1632 max =  6390 [us] loss = 0% (0)
Id 1008: count = 10000 mean = 2796.48 stddev =  573.86 min =  1832 max =  5453 [us] loss = 0% (0)
Id 1010: count = 10000 mean = 2792.73 stddev =  582.16 min =  1781 max =  5431 [us] loss = 0% (0)
Id 1012: count = 10000 mean = 2797.77 stddev =  582.17 min =  1854 max =  7414 [us] loss = 0% (0)
Id 1014: count = 10000 mean = 2789.41 stddev =  566.51 min =   988 max =  6392 [us] loss = 0% (0)
Id 1016: count = 10000 mean = 2798.49 stddev =  594.89 min =  1862 max =  7867 [us] loss = 0% (0)
Id 1018: count = 10000 mean = 2779.17 stddev =  587.14 min =  1804 max =  6339 [us] loss = 0% (0)
```

8.2 Zhodnocení

Bylo otestováno korektní chování jak firmware převodníku, tak síťového ovladače založeného na SocketCAN. Během testování nedošlo ke ztrátě žádné zprávy. Z uvedeného srovnání je patrné, že většího zpoždění na standardním jádru dosahuje SocketCAN. To je vcelku očekávatelný výsledek, jelikož SocketCAN staví na obecné Linuxové síťové infrastruktuře, naproti tomu LinCAN je se svojí vnitřní strukturou front navržen pro specifické potřeby CAN komunikace. Další testy i s *real-time* jádry je možné najít v článku [6].

Kapitola 9

Závěr

V rámci této diplomové práce byl implentován firmware pro CAN-USB převodník s mikrokontrolérem LPC1768, který využívá zdrojových kódů projektu LinCAN. Dále byl pro tento převodník vytvořen síťový ovladač do operačního systému GNU/Linux. Tento ovladač využívá SocketCAN, což je současný CAN subsystém Linuxového jádra. V průběhu vývoje byly obě funkcionality postupně vylepšovány a testovány. To vyvrcholilo závěrečným testováním, kde byla ověřena korektnost obou řešení. Zároveň bylo provedeno základní srovnání ovladače LinCAN a vytvořeného ovladače založeného na SocketCAN.

SocketCAN obsahuje funkcionalitu pro výpočet parametrů důležitých pro nastavení časování CAN komunikace. K tomu však potřebuje znát intervaly možných hodnot těchto parametrů, kterých může řadič CAN nabývat. První interakcí mezi ovladačem a připojeným převodníkem je tak požadavek ovladače na zjištění těchto intevalů. Následně je také nutná funkcionalita pro nastavení řadiče na vypočtené hodnoty časování. Oba tyto konfigurační požadavky byly implementovány. Po tomto nastavení již může probíhat samotný přenos dat mezi ovladačem a převodníkem. Vylepšením by určitě byla implemetace některých dalších konfiguračních požadavků.

Ze srovnání v kapitole testování je patrné, že většího zpoždění na standardním jádru dosahuje SocketCAN. To je vcelku očekávatelný výsledek, jelikož SocketCAN staví na obecné Linuxové síťové infrastruktuře, naproti tomu LinCAN je se svojí vnitřní strukturou front navržen pro specifické potřeby CAN komunikace. Určitě by dále mělo smysl provést obdobné testování na *real-time* Linuxovém jádru nebo se zvýšením priorit vláken v systému.

Literatura

[1] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, Inc., 3rd edition, 2005.
`http://lwn.net/Kernel/LDD3/`.

[2] O. Hartkopp. The CAN networking subsystem of the Linux kernel. In *Proceedings of the 13th iCC*, 2012. [Online]. Available:
`http://www.can-cia.org/fileadmin/cia/files/icc/13/hartkopp.pdf`.

[3] M. Kleine-Budde. SocketCAN – The official CAN API of the Linux kernel. In *Proceedings of the 13th iCC*, 2012. [Online]. Available:
`http://www.can-cia.org/fileadmin/cia/files/icc/13/kleine-budde.pdf`.

[4] *LPC17xx User manual*. Rev. 2, NXP Semiconductors, 19 August 2010. [Online]. Available: `http://www.nxp.com/documents/user_manual/UM10360.pdf`.

[5] J. Novák. CAN bus a jeho aplikace ve vozidlech. Materiály k předmětu Sběr a přenos dat.
`http://measure.feld.cvut.cz/node/443`, stav z 3. 5. 2012.

[6] M. Sojka and P. Píša. Timing Analysis of Linux CAN Drivers. In *Proceedings of the 11th Real Time Linux Workshop*, 2009. [Online]. Available:
`http://lwn.net/images/conf/rtlws11/papers/proc/p30.pdf`.

[7] Processors – ARM.
`http://www.arm.com/products/processors`, stav z 19. 4. 2012.

[8] CAN at Real-Time Systems group, DCE FEE CTU.
`http://rtime.felk.cvut.cz/can/`, stav z 3. 5. 2012.

[9] Linux USB drivers - Free Electrons.
`http://free-electrons.com/docs/linux-usb/`, stav z 26. 4. 2012.

[10] LXR - The Linux Cross Reference.
`http://lxr.linux.no/`.

[11] Webové stránky projektu OrtCAN.
`http://ortcan.sourceforge.net/`, stav z 3. 5. 2012.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

BSD Berkeley Software Distribution

CAN Controller Area network

DSP Digital Signal Processor

FPGA Field-Programmable Gate Array

GPIO General Purpose Input/Output

IP Internet Protocol

ISA Instruction Set Architecture

MAC Media Access Control

MCU Microcontroller Unit

POSIX Portable Operating System Interface

RISC Reduced Insttruction Set Computer

TCP Transmission Control Protocol

UDP User Datagram Protocol

USB Universal Serial Bus

Příloha B

Obsah přiloženého CD

```
.
|-- docs
|   |-- lmc_cb1-mcu-core-sch.pdf    schéma zapojení HW
|   `-- vanekjir.pdf                text této diplomové práce
|-- lincan
|   |-- build-embedded.sh           inicializační skript pro překlad převodníku
|   |-- build-lincan.sh             inicializační skript pro překlad LinCAN
|   |-- embedded                    zdrojové kódy převodníku
|   |-- lincan                      zdrojové kódy LinCAN
|   `-- omk                         adresář sestavovacího nástroje OMK
`-- socketcan-devel                 zdrojové kódy projektu SocketCAN
                                    i s vytvořeným ovladačem
```