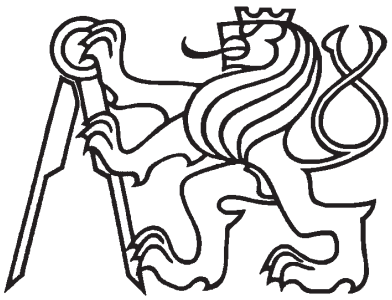
České vysoké učení technické v Praze

Fakulta elektrotechnická

Katedra řídicí techniky



# Diplomová práce

# Portování OS Linux na průmyslový mikropočítač a jeho použití v modelu vrtulníku

Květen 2006 Jiří Novotný

*České vysoké učení technické v Praze - Fakulta elektrotechnická*

***Katedra řídicí techniky*** ***Školní rok:*** 2004/2005

# ZADÁNÍ DIPLOMOVÉ PRÁCE

**Student:** Jiří Novotný

**Obor:** Technická kybernetika

**Název tématu:** Portování OS Linux na průmyslový počítač a jeho použití v modelu vrtulníku

**Zásady pro vypracování:**

1. Seznamte se s vývojovým kitem EXM32, operačním systémem Linux a sběrnicí CAN.
2. Naportujte operační systém Linux na desku EXM32 a vytvořte ovladač řadiče sběrnice CAN.
3. Vytvořte jednoduchý software určený ke sběru telemetrických dat z modelu vrtulníku a jejich odesílání na pozemní stanici.

**Seznam odborné literatury:** Dodá vedoucí práce.

**Vedoucí diplomové práce:** Ing. Ondřej Špinka

**Termín zadání diplomové práce:** zimní semestr 2004/2005 (změna zadání: 19.04.2006)

**Termín odevzdání diplomové práce:** květen 2006

prof. Ing. Michael Šebek, DrSc.
**vedoucí katedry**

L.S.

v zast.

prof. Ing. Zbyněk Škvor, CSc.
**děkan**

**V Praze dne** 25.04.2006

## 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 26. května 2006

J. Novotný

# Abstrakt

Tato práce je součástí rozsáhlého projektu, který se zabývá vývojem distribuovaného řídicího systému pro malý bezpilotní vrtulník. Cílem této práce bylo portovat operační systém Linux na průmyslový řídicí počítač, postavený na bázi procesoru Renesas SH7760. V teoretické části jsou vysvětleny vybrané partie programování aplikací pro OS Linux a popis sběrnice CAN. Praktická část popisuje portování operačního systému, tvorbu ovladače pro řadič sběrnice CAN v Linuxu a vytvořený software, sloužící ke sběru telemetrických dat.



# Abstract

This thesis is a part of the "Unmanned Automated Helicopter Project" currently being held at our department. In the course of this project, a distributed control system for a small unmanned rotorcraft is being developed. The goal of this work was to port the Linux operating system to an embedded industrial computer based on the Renesas SH7760 CPU, and to develop a simple telemetry application. In the first part of this work, the basics of Linux programming and the CAN (Controller Area Network) bus are being described, while the second part deals with the Linux porting, driver development and application programming.

## Poděkování

Na tomto místě bych rád poděkoval všem, kteří mi pomáhali a podporovali při studiu a vzniku této práce. Speciální poděkování patří vedoucímu této práce Ing. Ondřeji Špinkovi za veškeré rady a pomoc. Chtěl bych mu také poděkovat za možnost pracovat na tomto zajímavém projektu. Děkuji Ing. Pavlu Píšovi a Ing. Miroslavu Žižkovi za pomoc při řešení problémů s operačním systémem Linux a pomoc při jeho portaci na mikropočítač.

# Obsah

# Seznam obrázků

# Kapitola 1

# Úvod

Na katedře řídicí techniky je vyvíjen automaticky řízený model vrtulníku, který bude sloužit jako platforma pro demonstraci vlastností různých typů řídicích algoritmů. Projekt se zabývá návrhem a konstrukcí automaticky řízeného modelu vrtulníku. V rámci projektu je řešen jak návrh řídicího hardwaru, tak jeho základní softwarové vybavení. U tohoto projektu je kladen důraz na spolehlivost a bezpečnost. Projekt modelu vrtulníku vzniká pod vedením ing. Ondřeje Špinky. Cílem první etapy tohoto projektu bylo, aby na modelu mohla být při letu měřena telemetrická data. Byl vytvořen distribuovaný řídicí systém, který obsahuje: hlavní řídicí mikropočítač, jednotku řízení servomotorů a inerciální navigaci.

Tato práce se zabývá zprovozněním OS Linux a vývojem ovladače sběrnice CAN na hlavním řídicím mikropočítači EXM32 s procesorem SH7760.

Cílem této práce bylo vytvořit software pro komunikaci s řídicí jednotku servomotorů a inerciální navigací pomocí sběrnice CAN[1]. Takto přijatá data se odesílají bezdrátově pozemní stanici.

Práci lze rozdělit do tří částí. V první části jsou popsány základy pro práci a programovaní pro OS Linux. Tato část je zaměřena na tvorbu obecných modulů jádra a komunikaci přes počítačovou síť. Je zde popsán postup tvorby ovladače zařízení. V druhé části je popsána specifikace komunikačního protokolu sběrnice CAN a její implementace v použitém procesoru SH7760.

Poslední část pojednává o zprovoznění OS Linux na použitém mikropočítači, o překladu a ladění programů pro mikropočítače na osobním počítači. V této části je také popsán vytvořený ovladač CANu a aplikace pro sběr telemetrických dat.

[1] Controller Area Network

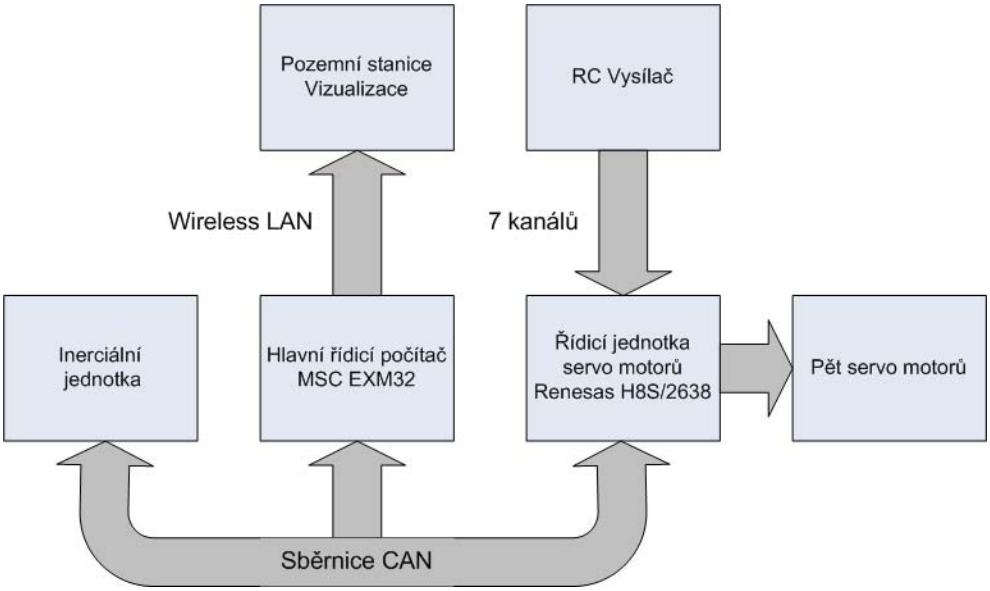
## 1.1 Popis modelu vrtulníku

Obrázek 1.1: Model vrtulníku

Základní parametry modelu vrtulníku:

- délka: 1370mm,
- šířka: 190mm,
- výška: 465mm,
- průměr rotoru: 1600mm,
- celková hmotnost: 7000g.

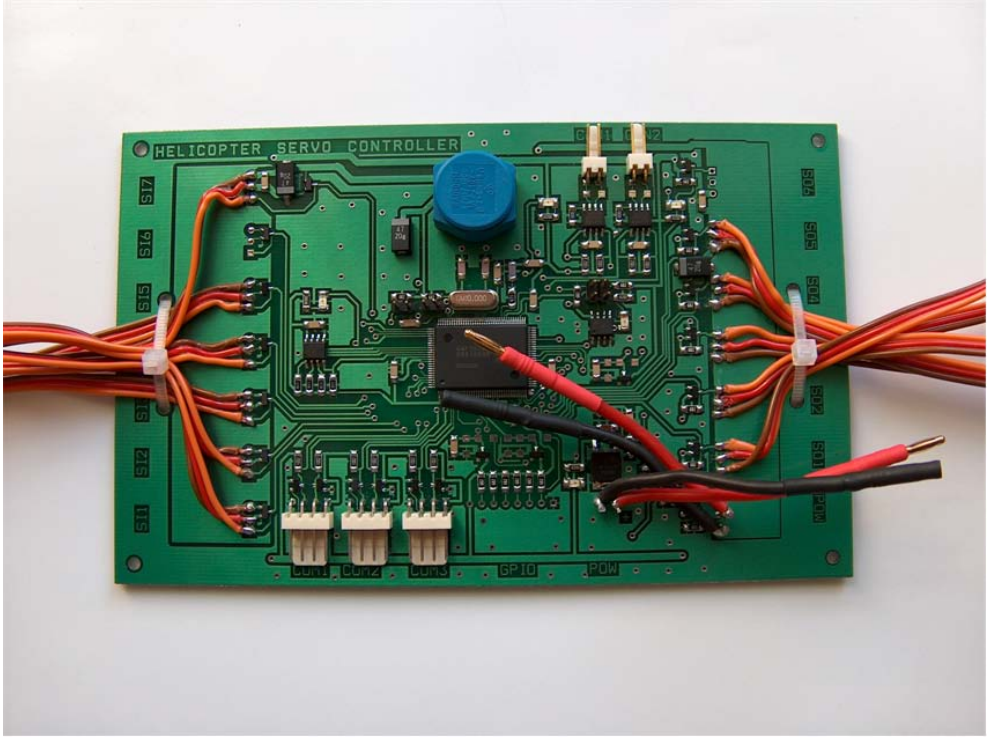
Jednotka servomotorů a inerciální navigace komunikují s hlavním počítačem pomocí sběrnice CAN. Hlavní řídicí mikropočítač posílá pozemní stanici informace pomocí počítačové sítě. Model zapojení je na obrázku 1.2.




Obrázek 1.2: Zapojení jednotek v modelu vrtulníku

### 1.1.1 Jednotka řízení servomotorů

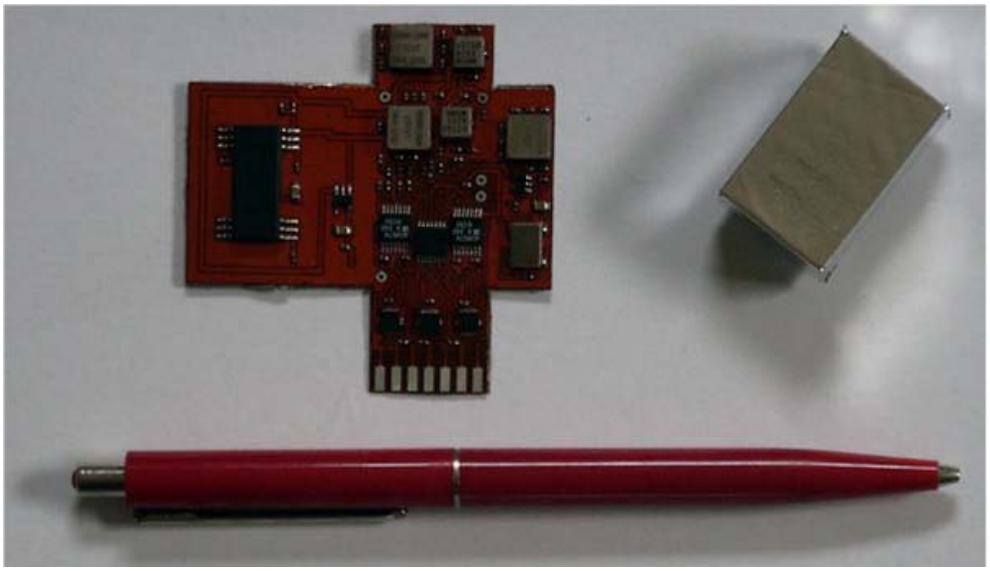
Jednotku řízení servomotorů[6] navrhl a sestavil Ota Herm v rámci diplomové práce. Obsahuje procesor H8S/2638 od firmy Renesas. Jednotka může být buď v režimu automatickém nebo manuálním. V manuálním režimu je model plně řízen RC vysílačem. V automatickém režimu mají být pomocí vysílače ovládané jen některé servomotory a polohu ostatních servomotorů určuje řídicí počítač. Výběr servomotorů, řízených přes vysílač, je dán programem mikropočítače. Režim letu je přepínán pomocí vysílače. Dále jednotka měří napětí baterií, které napájí také hlavní řídicí počítač a inerciální soustavu.




Obrázek 1.3: Jednotka řízení servomotorů

### 1.1.2 Inerciální navigace

V modelu vrtulníku byla použita inerciální navigace MICRO-ISU BP3010[5]. Inerciální navigace měří úhlové rychlosti a zrychlení ve všech osách. Data jsou měřena s frekvencí 32Hz.



Obrázek 1.4: Inerciální navigace

### 1.1.3 Vizualizační software

Po pozemní stanici je vizualizační software, který zobrazuje skutečnou polohu servo motorů, úhlové rychlosti a zrychlení. Vizualizaci na pozemní stanici vytvořil Miroslav Hájek.

# Kapitola 2

# Vybrané partie programování pro Linux

Na hlavní řídicí jednotce je použit operační systém Linux. V této diplomové práci bude v dalších kapitolách popsána tvorba ovladače sběrnice CAN a komunikačního programu. Tento program bude získávat a odesílat data přes sběrnici CAN a komunikovat s pozemní stanicí pomocí počítačové sítě. Aby bylo možné vytvořit tento software, je nutné porozumět některým programovacím technikám pro OS Linux.

V první podkapitole je popsán úvod do OS Linux: vybrané vlastnosti a adresářová stuktura[1]. V další podkapitole jsou popsány základní techniky pro komunikaci pomocí počítačové sítě. V poslední podkapitole je popsána tvorba modulů Linuxového jádra a využití jako ovladače zařízení[2].

## 2.1 Systém Linux

### 2.1.1 Popis systému Linux

První operační systém UNIX vyvinula firma Bell Laboratories. Je dán specifikací X/Open4.2. Tato specifikace je označována také někdy jako SPEC1170, určuje názvy rozhraní a chování všech funkcí operačního systému UNIX. V dnešní době je k dispozici řada distribucí: ať komerčních, jako je například Solaris od firmy SUN, nebo nekomerční Linux.

Začátkem 90. let byl v rámci GNU projektu[18], který založil Richard Stallman, vytvořen překladač, shell a další programy mimo nejnižší úrovně systému – jádra. V rámci projektu GNU mělo být vyvinuto jádro GNU Hurt, ale to se nestalo. Místo tohoto jádra bylo použito jádro, které vytvořil Linus Torvalds. Spojením několika projektů vznikl operační systém Linux[19], který

je kompatibilní s operačním systémem UNIX.

Shell je program, který funguje jako rozhraní mezi systémem Linux a uživatelem. Umožňuje uživateli zadávat příkazy, které operační systém vykoná. Například spouštět programy či skripty, kopírovat soubory, přesměrovat výstup programu na vstup druhého programu, vytvořit mezi jednotlivými programy komunikační prostředek, takzvanou rouru.

Každému programu, který je v Linuxu spuštěn, se říká proces. Každý proces se skládá z jednoho nebo více vláken. Procesy mají svůj jedinečný identifikátor PID a svůj rodičovský proces. Přepínání procesů je časově náročná operace, protože systém musí vykonat několik úkonů, například je potřeba přemapovat virtuální paměť. Paralelně běžící funkce v rámci jednoho procesu nazýváme vlákna. Velmi často je potřeba řešit současně více operací a co nejméně přepínat mezi procesy. Při běhu vláken nedochází k tak časově náročným operacím jako u přepínání procesů a vlákna sdílí téměř všechny prostředky procesů. K synchronizaci vláken se používají semafory nebo mutexy.

Jádro systému Linux, neboli kernel, má například za úkol kontrolovat jednotlivé procesy, zda nezapisují do paměti, která jim nebyla přidělena, nebo zprostředkovávat komunikaci mezi periferiemi a procesy. V případě jakékoliv chyby může jádro ukončit libovolný uživatelský proces, aby bylo zabráněno havárii celého systému. Základní funkce jádra, které jsou dány při jeho kompilaci, mohou být rozšířeny pomocí speciálních programů – modulů jádra. Velká výhoda jádra Linuxu je, že modul může být zaveden a ukončen za provozu jádra. Modul jádra může být například ovladač. Podrobnější popis a programovací techniky jádra jsou v kapitole 2.3.

### 2.1.2 Adresářová struktura systému Linux

V hlavním adresáři, neboli v kořenovém adresáři, se nachází několik důležitých adresářů.

- V adresáři **/bin** je většina základních spustitelných souborů, které zastupují základní příkazy jako například cp, mv nebo kill.
- Adresář **/boot** obsahuje jádro a soubory potřebné pro zavedení jádra při startu systému.
- V adresáři **/dev** jsou soubory pro komunikaci se zařízeními.
- Systémová nastavení jsou uložena v adresáři **/etc**.
- Adresáře uživatelů se nacházejí v **/home**.

- Sdílené knihovny jsou v adresáři **/lib**.
- Soubory, které mají porušenou konzistenci se ukládají do **/lost+found**.
- Disky, diskety a CD-ROMy se připojují do adresáře **/mnt**.
- Informace o běhu a aktuálním stavu systému lze najít v adresáři **/proc**. Mezi nejzajímavější informace patří:
  - Výpis všech aktuálně připojených zařízení, který je v souboru **/proc/devices**.
  - Počet jednotlivých přerušení od spuštění počítače je v souboru **/dev/interrupts**
  - Aktuální stav paměti je v **/proc/meminfo**.

  Modul jádra může přidat za provozu systému další soubory. Ve skutečnosti nejsou data systému uložena v souboru, ale aktuální data předává modul, který zařízení spravuje.
- Domácí adresář superuživatele je **/root**.
- Dočasné soubory jsou uloženy v **/tmp**.
- Adresář **/usr** obsahuje soubory, jenž jsou potřeba pro běžný provoz systému. Mezi nejdůležitější patří:
  - **/usr/bin** - veřejně přístupné příkazy,
  - **/usr/doc** - dokumentace,
  - **/usr/local** - konfigurační soubory,
  - **/usr/man** - manuálové stránky,
  - **/usr/tmp** - dočasné soubory programů.
- Adresář /var obsahuje datové soubory jako jsou logy, tiskové fronty nebo emaily.

Podrobné informace o systému Linux lze najít

## 2.2 Sockety

Socket je prostředek lokální nebo síťové komunikace charakterizovaný třemi atributy: doménou, typem a protokolem. Sockety mají také adresu, která se používá jako název lišící se podle domény. Socketové domény určují síťové médium, které se bude používat pro komunikaci. Nejpoužívanější socketová doména je AF_INET, která je vhodná pro síťové sockety, a AF_UNIX, která se používá pro lokální sockety implementované pomocí unixového systému souborů. Existují i další typy socketových domén, například AF_ISO nebo AF_NS. Sockety mají dva typy spojení; streamy a datagramy. Streamové a datagramové spojení jsou na sobě nezávislé.

Streamové spojení poskytuje spolehlivý, sekvenční a dvoucestný tok dat. Je zajištěno, pokud data byla odeslána bez chyby, že data nebudou ztracena, dojdou v pořadí v jakém byla poslána a síťový tok nebude přesměrován. V doméně AF_INET jsou proudové sockety implementovány typem SOCK_STREAM. Jedná se o připojení TCP/IP.

V některých případech je zbytečné a systémově nákladné udržovat neustále spojení mezi klientem a serverem. Rychlejší a systémově méně nákladnější jsou datagamy. Na rozdíl od streamového spojení není garantováno, že zpráva bude doručena na cílový počítač a že zprávy dorazí v pořadí, v jakém byly odeslány. Datagramové sockety jsou implementované v doméně AF_INET jako SOCK_DGRAM. Jedná se o připojení UDP/IP.

Programy se dělí do dvou skupin podle toho, jaké používají sockety (SOCK_STREAM). První skupinou jsou servery, což jsou programy, které vytvoří socket a čekají, až se k nim někdo připojí. Druhou skupinou jsou klienti, kteří po vytvoření socketu začnou aktivně navazovat spojení. Aby aplikace mohla poslat serveru packet, musí znát jeho IP adresu, port a typ spojení.

### 2.2.1 Socketové adresy

Adresy socketů se definují pomocí struktury sockaddr nebo pomocí structury sockaddr_in, která je podobná struktuře sockaddr, ale má položku sa_data rozdělenou na více částí.

```
struct sockaddr {
        unsigned short    sa_family;    // Socketová doména
        char              sa_data[14];  // adresa protokolu
       };
```

```
struct sockaddr_in {
        short int          sin_family;  // Socketová doména
        unsigned short int sin_port;    // Číslo portu
        struct in_addr     sin_addr;    // IP adresa
        unsigned char      sin_zero[8]; // Aby byla zachována
                                        // velikost se struct
                                        // sockaddr
       };
```

Internetová adresa (IPv4) se skládá z čtyřbajtové IP adresy, která se zapisuje 192.168.100.1, a čísla portu (např. 80 pro web server). Struktura pro IP adresu je definována jako:

```
struct in_addr {
    unsigned long s_addr;
};
```

Pro převod adresy z textového do binárního tvaru ve správném pořadí bytů se používá funkce inet_aton.

```
int inet_aton (const char *NAME, struct in\_addr *ADDR);
```

Pokud chceme převést DNS jméno počítače ( www.google.com ) na binární IP adresu, používá se funkce gethostbyname.

```
struct hostent * gethostbyname (const char *NAME);
```

### 2.2.2 Základní funkce pro práci se sockety

- **int socket(int domain, int type, int protocol);** Vytvoří socket daného typu a vrátí file-descriptor. Parametr domain je doména socketu a parametr protocol je typ spojení probraný výše. Používá server i klient.

- **int connect(int sockfd, struct sockaddr *serv_address, int addrlen);** Slouží klientovi k vytvoření spojení se serverem. Používá již vytvořený socket.

- **int bind(int sockfd, struct sockaddr *address, int addrlen);** Systémové volání bind přiřadí serveru adresu nepojmenovanému socketu. Nejčastěji se používá v kombinaci s listen() pro určení čísla portu na kterém server poslouchá. Používá jen server.

- **int listen(int sockfd, int backlog);** Volá se po bind() a slouží k nastavení socketu do stavu čekání na příchozí spojení. Používá jen server.
- **int accept(int sockfd, void *address, int *addrlen);** Je-li socket v čekacím stavu, funkce accept() čeká na příchozí spojení. Systémové volání accept se vrátí, když se klient pokusí připojit na socket určený parametrem socket.
- Pro čtení nebo zápis do socketu můžeme použít nízkoúrovňové funkce read a write nebo funkce send a recive.
  **ssize_t send( int s, const void *buf, size_t len, int flags );**

Je-li funkce write nebo send v blokovacím režimu a nemá k dispozici odpovídající velikost bufferu pro odesílání dat, zablokuje program, dokud nebudou data odeslána. Tato událost může nastat třeba z důvodů fyzického přerušení vedení. V neblokovacím režimu ( v případě, že soket nemá dostatečně velký buffer ) nejsou odeslána všechna data. Funkce send vrací počet skutečně odeslaných bytů, nebo chybu. Neblokovací režim lze nastavit buď jen pro dané odeslaní jako parametr funkce send a nebo nastavit globálně pro veškerou komunikaci pomocí socketu.

- **Recv** - v případě, že data pro čtení jsou k dispozici, chová se funkce stejně jako v blokovacím režimu. V případě, že data k dispozici nejsou, funkce vrací chybový kód.
- **Accept** - jestliže ve frontě požadavků na spojení existuje požadavek, chová se accept stejně jako v blokovacím režimu. V případě, že požadavek na spojení není k dispozici, funkce vrací chybový kód.
- **connect** - funkce connect na socket v neblokovacím režimu odešle požadavek na spojení. Spojení není ihned navázáno. Nelze socketem ihned odesílat a přijímat data.

Globální nastavení neblokovacího režimu se nastaví pomocí fcntl :

```
int oldFlag = fcntl(server_sockfd, F_GETFL, 0);
if (fcntl(server_sockfd, F_SETFL, oldFlag | O_NONBLOCK) == -1)
{
     printf("Nepodarilo se nastavit neblokovani soketu:\n");
}
```

Nastavení neblokovacího režimu pro jedno odeslání se nastaví pomocí flagu MSG_DONTWAIT.

### 2.2.3 Signály

Signál[13] je asynchronní událost, který přenáší data. POSIXová specifikace definuje operace signálů pouze na procesech a obsahuje koncept frontových realtimových signálů. Pokud proces obsahuje vlákna, tak signál bude doručen prvnímu vláknu, které nemá signál blokovaný. Jestliže vlákno signál ignoruje, ovlivní to všechna vlákna v procesu. Jestliže vlákno signál blokuje ovlivní to pouze toto vlákno.

```
void (* signal (int sig, void (*func)(int sig)))(int)
```

Prvním parametrem funkce signal je typ signálu. Druhým parametrem je obslužná funkce, která se vyvolá pomocí tohoto signálu.

### 2.2.4 Broken Pipe

Při špatné komunikaci se může uzavřít komunikační kanál a to vyvolá signál SIGPIPE. Při neobsloužení tohoto signálu je proces ukončen s chybovou zprávou Broken Pipe. Aby se tomuto zamezilo, musí se registrovat signál SIGPIPE pomoci funkce signal.

## 2.3 Tvorba modulů pro Linux

Systém Linux rozlišuje dva základní režimy: režim uživatelský a režim supervizora - jádra. Program v uživatelském režimu má některá omezení. Například může přistupovat jen do paměti, která byla programu přidělena, nemůže přímo komunikovat se zařízením nebo obsluhovat přerušení. Omezení jsou zavedena z důvodu, aby uživatelský program nemohl způsobit havárii celého systému. Správný chod programu v uživatelském prostoru je kontrolován jádrem a v případě vážné chyby může jádro program ukončit. Činnost programu jádra, neboli modulu jádra, není kontrolovaná a tím hrozí možnost zhroucení celého systému, pokud je modul špatně navržen. Modul jádra může přistupovat do celé paměti, zabrat veškerý procesorový čas. Používá se například pro obsluhu přerušení nebo jako ovladač zařízení.

Při tvorbě aplikace se musí rozhodnout, co naprogramovat v uživatelském procesu a jakou část v modulu jádra. Co největší část aplikace se má vytvořit jako uživatelský proces. Vážná chyba v uživatelské aplikaci způsobí jen výpis chybové hlášky, případně výpis paměti. Ale chyba v modulu jádra může způsobit zhroucení celého systému. Kód modulu jádra by měl být co nejméně výpočetně náročný, protože aplikace jádra může plně zabrat veškerý systémový čas počítače a tím zabránit systému v činnosti.

Jednotlivé verze Linuxového jádra se od sebe liší a není zaručena jejich vzájemná kompatibilita. Musí se určit, pro jakou verzi se modul bude navrhovat. Další dokumentace a postupy zde popsané se vztahují k verzi jádra 2.6.x. Programy byly přeloženy a vyzkoušeny ve verzi jádra 2.6.14.

Oproti starším verzím je například rozdíl ve změně překladu jádra. Podrobnější informace o rozdílu jednotlivých verzí jádra Linuxu jsou uvedeny v [20].

### 2.3.1 Výpis zpráv jádra

Pro výpis zpráv jádra nelze použít printf. Ekvivalentem v jádře je funkce printk. Na rozdíl od printf se musí printk nastavit priorita zprávy. Možné hodnoty priorit zpráv jsou definované v hlavičkovém souboru */linux/kernel.h* následovně:

```
#define KERN_EMERG     "<0>" /* systém je nepoužitelný */
#define KERN_ALERT     "<1>" /* akce se musí udělat ihned  */
#define KERN_CRIT      "<2>" /* kritické podmínky */
#define KERN_ERR       "<3>" /* chybové podmínky*/
#define KERN_WARNING   "<4>" /* varování */
#define KERN_NOTICE    "<5>" /* podstatná zpráva */
#define KERN_INFO      "<6>" /* informace */
#define KERN_DEBUG     "<7>" /* ladící zprávy */
```

Výpis kódu 2.1: Výpis zpráv jádra

Příklad zprávy jádra:

```
printk( KERN_DEBUG "Toto je ladici zprava\n" );
```

### 2.3.2 Modul Hello World

Modul se skládá minimálně ze dvou částí - inicializační a ukončovací. Pomocí makra *module_init* a *module_exit* se definuje, jaké funkce jsou volány při inicializaci nebo ukončení modulu.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init hello_init(void)
{
  printk(KERN_INFO "Hello world!\n");
  return 0;
```

```
}

static void __exit hello_exit(void)
{
  printk(KERN_INFO "Goodbye world!\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Výpis kódu 2.2: Modul Hello world

### 2.3.3 Crosscompilace modulu

Od verze linuxového jádra 2.6 se modul překládá novým způsobem, a proto pro starší verze jádra nebude fungovat. Výstupem překladu modulu je soubor modul.ko, kde modul je název daného modulu. Překlad se děje pomocí Makefile, kde se musí definovat cesta ke zdrojovým kódům jádra pro danou platformu. Pomocí symbolu *obj-m* se definuje, které moduly se mají přeložit.

```
# Cesta k jádru Linuxu
KDIR =/home/jirka/linux-2.6.14

# Název modulu, který chceme přeložit + ".o".
# Pokud se překládá najednou více modulů,
# tak se použije opakovaní symbol
# z následujícího řádku.
obj-m += hello.o

# Vlastní překlad. Definujeme pro jakou
# platformu chceme crosscompilovat,
# vyvolá se make
all:
  make -C $(KDIR) SUBDIRS=$(PWD) CROSS_COMPILE=sh4-linux-
      modules -Wall

clean:
  rm -f *.o *.ko .*.o.cmd .*.ko.cmd *.mod.c
```

Výpis kódu 2.3: Makefile modulu Hello world

### 2.3.4 Zavedení a ukončení modulu

Po přeložení modulu se pomocí příkazu *insmod* zavede modul do jádra. Modul může být zaveden pouze, je-li uživatel přihlášen jako root.

```
# insmod hello.ko
```

Pokud se neobjevila žádná chybová zpráva, modul se správně zavedl. Zavedení lze zkontrolovat příkazem *lsmod*, který zobrazí veškeré moduly v jádře. Příkaz *dmesg* vypíše všechny zprávy jádra od posledního spuštění počítače nebo vymazání zpráv. Pro vymazání zpráv slouží parametr *-c*.
Poslední zpráva by měla být Hello world.

```
# dmesg -c
Hello world
```

Modul se ukončí pomocí *rmmod*.

```
# rmmod hello.ko
```

Nyní lze pomocí lsmod a dmseg zkontrolovat ukončení modulu.

Pokud se při pokusu vložit modul do jádra zobrazí tato chybová zpráva:

```
Error inserting './hello.ko': -1 Invalid module format,
```

je třeba se podívat do systémového souboru */var/log/messages* na bližší podrobnosti. S největší pravděpodobností byl modul přeložen pro jinou verzi jádra.
Pokud v systému nejsou příkazy insmod, lsmod a rmmod je nutné doinstalovat *module-init-tools*.

Další možnost zavedení modulu do jádra systému je přidání modulu již v době kompilace jádra. Tato možnost je vhodná tehdy, když je vývoj modulu ukončen nebo je nutné modul ladit debuggerem.

### 2.3.5 Ovladače zařízení

Ovladač zprostředkovává rozhraní mezi programem a zařízením, které lze rozdělit do dvou základních tříd: znaková a bloková. K většině ovladačů zařízení se přistupuje přes systém souborů, což přináší mnohé výhody, například standardní komunikační rozhraní mezi programem a ovladačem. Soubor zařízení je speciální soubor, který se vytvoří pomocí příkazu mknod. Soubor zařízení by měl být umístěn v adresáři /dev. Není to podmínka. Příkaz mknod má jako parametr typ zařízení znakové - c, blokové - b , hlavní číslo, které udává specifikaci zařízení, a vedlejší číslo, které označuje instanci v rámci daného zařízení.

```
#  mknod /dev/hcan20 c 120 0
```

Tímto příkazem se vytvořil jen speciální soubor. Uživatelský program komunikuje se zařízením díky tomuto souboru pomocí funkce read, write, open a close.

### Ovladače zařízení - modul

Pro použití modulu jako ovladače zařízení je nutné nejdřív zařízení v jádře zaregistrovat pomocí funkce *register_chrdev*.

```
int register_chrdev(unsigned int major, const char *name,
 struct file_operations *fops);

unsigned int major je hlavní číslo zařízení
const char *name je náze zařízení
struct file_operations *fops //je struktura,
 //kde se definuje, které funkce se mají volat
 // pro  danou událost.
static struct file_operations fops = {
static int schar_open(struct inode *inode, struct file *file);
static int schar_release(struct inode *inode, struct file *file);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t,
loff_t *);
int device_ioctl(struct inode *inode, struct file *file,
 unsigned int ioctl_num, unsigned long ioctl_param);
unsigned int device_poll(struct file * file, poll_table * pt);
}
```

- Funkce **open** se volá, když některá aplikace zařízení otevře.
- Jestliže aplikace nebo její potomek uzavírá komunikaci se zařízením, volá se funkce **release**.
- Funkce **read** se volá při čtení dat z pohledu aplikace, zápis dat z pohledu modulu. Návratová hodnota udává počet zapsaných znaků, je-li návratová hodnota záporná, došlo k chybě.
- Při zápisu z pohledu aplikace se volá funkce **write**, čtení dat z pohledu modulu. Návratové hodnoty jsou ekvivalentní s funkcí read.
- Funkce **ioctl** neboli I/O control umožňuje aplikacím předávat nebo získávat aplikacím data z modulu. podrobněji v kapitole 7.2.
- Funkce pool informuje aplikaci o datových událostech.

V definici struktury se musí definovat vlastník - owner, pomocí makra THIS_MODULE.

Při programování ovladače zařízení se musí rozhodnout, zda bude zařízení otevřeno jako blokující nebo neblokující. Jestliže není k dispozici dostatek dat a čekající proces se uspí, jedná se o blokující otevření. Pokud čekajícímu procesu se vrátí informace, že data nejsou k dispozici bez uspání, jedná se o neblokující otevření. Použití blokujícího nebo neblokujícího otevření záleží na konkrétní aplikaci.

Modul se přeloží pomocí programu make. V Makefile se musí správně nastavit cesta ke zdrojovým kódům jádra. Před použitím modulu se musí na mikropočítači vytvořit zařízení.

```
#mknod /dev/zarizeni c 120 0
```

Test programu.

```
# insmod modul_zarizeni.ko
# cat /dev/zarizeni
# dmesg
Doslo k otevreni zarizeni
Funkce cteni dat neni implementovana.
```

### Komunikace modulu s uživatelským prostorem

Existují dvě možnosti předání informací mezi modulem a uživatelským prostorem. Pro menší množství dat se používájí funkce get_user a put_user. Funkce copy_to_user a copy_from_user se používají pro vetší množství dat .

```
int get_user ( void *x, const void *addr ) zkopíruje
#sizeof (addr ) bytů z adresy uživatelského
#prostoru do proměnné x.
int put_user ( void *x, const void *addr ) zkopíruje
# z proměnné x sizeof ( addr ) baytů do adresy
# uživatelského prostoru.
```

Návratová hodnota je 0, bylo-li kopírování dokončeno v pořádku.

Pokud je potřeba vypnout kontrolu práv uživatelské aplikace, existuje verze příkazů __get_user a __put_user. Návratovou hodnotu v argumentu vrací get_user_ret ( void *x, const void *addr ) a put_user_ret( void *x, const void *addr ).

Častěji je potřeba kopírovat větší množství dat, než jen jednu hodnotu a proto je efektivnější použít funkce copy_to_user a copy_from_user.

```
unsigned long copy_to_user ( void __user *to,
 const void *from, unsigned long count )
unsigned long copy_from\_user ( void *to,
 const void __user *from,  unsigned long count )
```

Pro funkce copy_to_user a copy_from_user existuje také možnost vypnutí kontroly práv uživatelské aplikace.

### 2.3.6 Obsluha přerušení

Vznikne-li na zařízení událost, je nutné informovat obsluhující proces v jádře. Špatný způsob zjišťování nových událostí je, ptá-li se proces opakovaně zařízení, zda se nevyskytla nová událost. Správný způsob je, když zařízení v případě nové události vyšle signál přerušení a na něj reaguje handler. Obsluha přerušení závisí na použité platformě. Podrobnější informace o povolených přerušení jsou v <kernel2.6.14>/arch/sh/kernel/cpu/sh4/irq_intc2.c. Další informace o přerušení na procesoru SH7760 jsou v kapitole 5 o portování Linuxu.

Když je vyvolán signál přerušení, je pozastaven chod úloh v uživatelském prostoru. Jádro pozastaví svou činnost a vyvolá příslušný handler, který má být aktivní jen nezbytně nutnou dobu. Do jeho ukončení je plně pozastavena další činnost procesoru. Handler nesmí být blokovacího typu.

Již zaregistrované handlery jsou v souboru /proc/interrupts.

### 2.3.7 Alokování handleru přerušení

Pro alokování handleru přerušení slouží funkce request_irq.

```
int request_irq ( unsigned int irq,
void ( *handler ) ( int, void *, struct pt_request ),
unsigned long irqflags,
const char *devname,
void *dev_id
)
```

Irq je číslo přerušní, které chceme zaregistrovat handler ( int irq, void *dev_id, struct pt_regs *regs ) Tato funkce je volaná, když vznikne přerušení s číslem irq. Parametr irqflags udává chování přerušení.

- **SA_SHIRQ** S tímto parametrem je umožněno sdílení irq víc než jedním zařízením.

- **SA_INTERRUPT** Není povoleno sdílení přerušení.

**devname** je název přerušení, který bude uveden v souboru /proc/interrupts

**dev_id** Při povoleném sdílení přerušení slouží k předávání parametrů výše uvedenému handleru. Ovladače se musí dohodnout na masce.

Funkce request_irq po úspěšném alokování vrací 0 jinak chybový kód. Zaregistrovaný handler přerušení obdrží při přerušení tři parametry. První je číslo irq. To se používá tehdy, pokud handler obsluhuje více přerušení. Druhý parametr je dev_id, který byl již popsán u funkce request_irq. Poslední parametr je struct pt_regs *regs, který obsahuje obraz registrů procesoru před přerušením.

Handler musí při svém ukončení vracet číslo irq, na které reagoval.

Při ukončení modulu je nutné přerušení uvolnit pomocí funkce free_irq.

```
void free\_irq ( unsigned int irq, void *dev\_id );
```

Parametry jsou stejné jako u funkce request_irq.

### 2.3.8 Ioctl

Někdy je nutné získat nebo nastavit parametry modulu jádra bez jeho ukončení. K tomu slouží ioctl. Systém Linux má 4 typy funkcí ioctl - přímý příkaz, čtení, zápis a současně čtení a zápis.

- **_IO ( base, command )** Definuje vybraný příkaz command. Do ani z aplikace nejsou přenášena prostřednictvím ioctl žádná data.

- **_IOR ( base, command, size )** Čtení dat o velikosti size z pohledu aplikace prostřednictvím ioctl.

- **_IOW ( base, command, size )** Zápis dat o velikosti size z pohledu aplikace prostřednictvím ioctl.

- **_IORW ( base, command, size )** Čtení a zápis dat o velikosti size z pohledu aplikace prostřednictvím ioctl.

Parametr base udává základní jedinečné číslo ioctl. Každá aplikace, která má otevřený přístup k zařízení, může používat ioctl.

Jedna funkce obsluhuje všechny volání ioctl a má tři parametry. První parametr base je číslo otevřeného zařízení, druhý nadefinované ioctl a poslední proměnná obsahující velikost dat.

# Kapitola 3

# Sběrnice CAN

## 3.1 Popis sběrnice CAN

Komunikační sběrnice Controller Area Network (CAN)[14], [16] byla vyvinuta firmou Bosh pro nasazení v automobilech. Jedná se o sériový typu multi-master, kde každý uzel sběrnice může být master. Při poruše jednoho uzlu může síť fungovat dál. CAN je broadcast sběrnice s prioritním přistupování k médiu. Má definovanou fyzickou a linkovou vrstvu ISO/OSI modelu. Aplikační vrstva ISO/OSI modelu je definována vzájemně nekompatibilními standardy jako například CANopen nebo DeviceNet. Přenosová rychlost může být až
1Mbit/s. Sběrnice CAN má rozsáhlou detekci chyb díky Hammingově vzdálenosti 6.

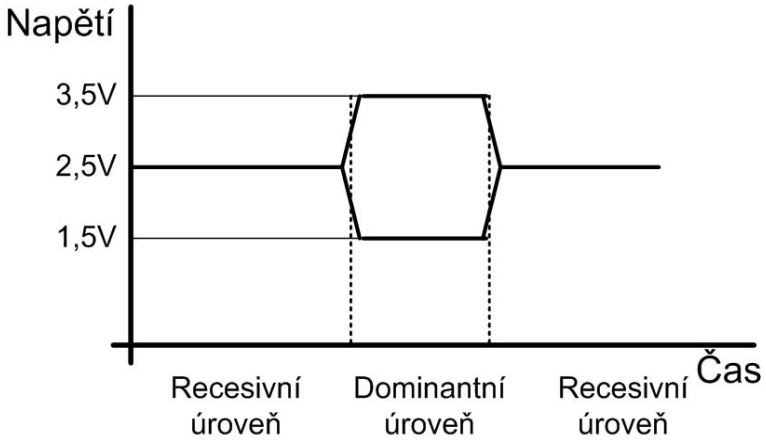
Sběrnice CAN je definována normou ISO 11898, která popisuje fyzickou vrstvu a specifikaci CAN 2.0A. Později byla vytvořena specifikace CAN 2.0B, která nově definuje standardní a rozšířený formát zprávy. Liší se v délce identifikátoru zprávy. Uzly komunikují pomocí zpráv ( datová zpráva, žádost o data ) a speciálních zpráv ( signalizace chyb, zprávy o přetížení ).

Zprávy posílané po sběrnici CAN neobsahují žádné informace o cílovém uzlu, ale mají identifikátor, kterým je určena jejich priorita. Zpráva s nižším identifikátorem má větší prioritu než zpráva s vyšším identifikátorem. Lze nastavit, aby uzel přijímal jen zprávy s požadovaným identifikátorem a ostatní nepřijal. Pro přístup k médiu je použita sběrnice s náhodným přístupem, který řeší kolize na základě priorit identifikátorů.

Sběrnice CAN používá dvou napěťových úrovní, nazývaných recesivní a dominantní. Funguje jako wired AND. Toto musí zajišťovat fyzická vrstva. Recesivní a dominantní logická úroveň si nejsou z definice rovnocenné. Jsou-li všechny vysílače v logické 1, je sběrnice CAN v logické 1 - recesivní úroveň.

Je-li libovolný vysílač v logické 0, je celá sběrnice v logické 0 - dominantní úroveň.

Recesivní úroveň je definována nulovým rozdílem potenciálu mezi oběma vodiči sběrnice CAN. Dominantní úroveň je definována napětím vodiče CAN_H 3,5 V proti zemnímu vodiči a napětím 1,5 V na vodiči CAN_L proti zemnímu vodiči.




Obrázek 3.1: Nominální napětí pro recesivní a dominantní úrovně

Pro eliminaci odrazů na vedení je na každém konci sběrnice umístěn zakončovaní rezistor o velikost 120Ω. Dle normy ISO 11898 může být na sběrnici CAN připojeno maximálně 30 uzlů. Teoreticky počet uzlů není omezen. Maximální délka sběrnice pro přenosovou rychlost 1Mbit/s je 40m. Pro jiné délky norma neuvádí maximální přenosovou rychlost. Obecně platí, že pro delší sběrnici bude přenosová rychlost menší. Maximální délka sběrnice pro jinou přenosovou rychlost je uvedena v tabulce 3.1, která má pouze informativní charakter. Maximální rychlost záleží například také na parametrech použitého kabelu.

| **Délka sběrnice** | **maximální rychlost** |
|---|---|
| m | kbit/s |
| 112 | 500 |
| 200 | 300 |
| 640 | 100 |
| 1340 | 50 |
| 2600 | 20 |
| 5200 | 10 |

Tabulka 3.1: Maximalní rychlost sběrnice CAN v závislosti na délce vedení

Linková vrstva protokolu CAN obsahuje dvě podvrstvy - Medium Access Controll ( MAC ) a Logical Link Control ( LLC ). Medium Access Control obstarává přístup k médiu. Kóduje data, vkládá bit stuffing, který je popsán v následující kapitole, detekuje chyby, rozlišuje priority zpráv a potvrzuje jejich přijetí.

Logical Link Control má za úkol filtrovat zprávy a hlásit přetížení sběrnice.

Pokud nikdo nevysílá, může libovolný uzel zahájit vysílání. Žádný z uzlů nesmí zahájit vysílání, pokud začal někdo vysílat. Jedinou výjimkou tvoří chybové rámce, které může poslat libovolný uzel, který detekuje chybu v právě přenášené zprávě.
Zahájí-li vysílání současně více uzlů, smí vysílat ten, jehož zpráva má nejvyšší prioritu ( má nejnižší identifikátor ). Každý vysílač kontroluje logickou hodnotu právě vyslaného bitu se skutečnou logickou hodnotou na sběrnici. Jestliže se hodnoty neshodují, musí okamžitě přerušit vysílání.

Pokud začnou dvě stanice současně vysílat, pozná se to při odesílání ID, které je odesláno hned po startovacím bitu. Jak bylo dříve popsáno, jsou-li všechny vysílače v recesivní úrovní, je sběrnice CAN v recesivní úrovni. Je-li libovolný vysílač v dominantní, je celá sběrnice v dominantní úrovni. Zpráva s vyšší prioritou má při odesílání první rozdílný bit ID v dominantní úrovni, a proto nebude poškozena. Tímto způsobem je zajištěno, že nedojde k poškození a přerušení již odesílané zprávy s vyšší prioritou. Stanice, která nemá právo vysílat musí počkat, až bude zpráva odeslána a potom se může opět pokusit odeslat svoji zprávu.

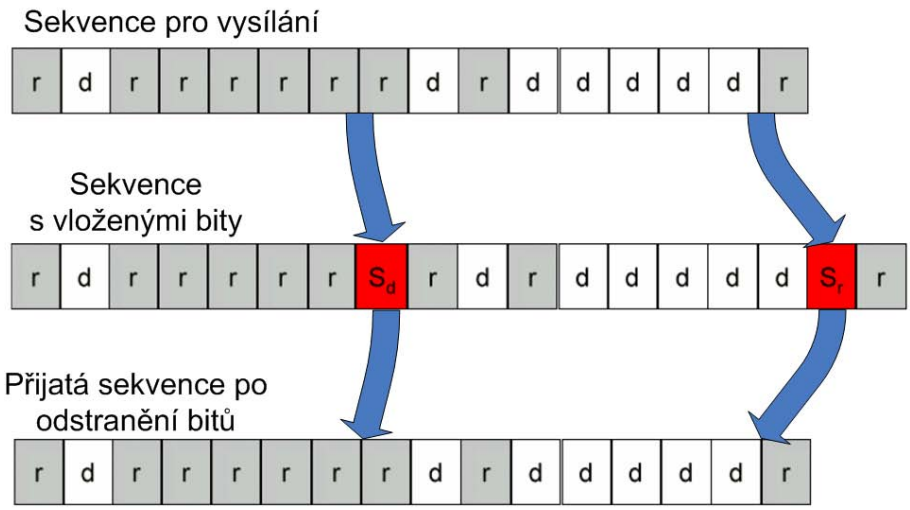
## 3.2 Zabezpečení zpráv na sběrnici CAN

Zprávy přenášené po sběrnici CAN jsou zabezpečeny několika na sobě nezávislými mechanismy, které fungují paralelně.

- **Monitoring** - Je zmiňovaná metoda, kdy vysílač kontroluje logickou hodnotu odesílaného bitu se skutečnou hodnotou logického bitu. Chyba nastane, když je na sběrnici detekována jiná logická hodnota, než která je odvysílána.

- **Cyclic Redundancy Check** ( CRC ) kód. na konci každé zprávy je uveden 15-bitový CRC kód, který je generován podle polynomu $x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$ ze všech bitů příslušné zprávy.

- **Bit-stuffing** U sběrnice CAN není garantována náběžná a sestupná hrana pro každý bit. Vzniká tu nebezpečí ztráty synchronizace mezi vysílačem a přijmačem. Proto sběrnice vkládá pomocné bity - Bit-stuffing pro odstranění toho rizika. Je-li ve zprávě minimálně pět bitů stejné úrovně, je do zprávy vložen bit opačné úrovně.




Obrázek 3.2: Bit-stuffing

- **Message Frame Check** je kontrola správného formátu dle specifikace sběrnice CAN. V případě detekování nepovolené hodnoty je vygenerována chyba.

- **Acknowlege** - Je-li zpráva přijata v pořádku libovolným uzlem, změní tento uzel jeden bit zprávy ACK, který vysílač vždy detekuje. Potvrzování zpráv je prováděno bez ohledu na filtrování příchozích zpráv.

## 3.3 Typy zpráv sběrnice CAN

Dle normy ISO 11898 pro sběrnici CAN se délka jednoho bitu skládá ze čtyř nepřekrývajících se segmentů - synchronizační segment, segment odezvy, fázový segment 1 a fázový segment 2. Každý segment je složen z celočíselného počtu časových kvant tq. Časové kvantum je výstupní signál z předděliče, do kterého jde signál z oscilátoru.

- Synchronizační segment je dlouhý 1 tq a slouží ke synchronizaci přijímače s vysílačem.
- Segment odezvy má délku jednoho až osmi časových kvant a slouží ke kompenzaci časového upoždění na sběrnici CAN.
- Fázový segment 1 je určen pro kompenzaci fázového rozdílu mezi přijímaným signálem a vnitřním časovačem. Fázový segment obsahuje jedno až osm časových kvant .
- Fázový segment 2 také slouží k synchronizaci fázového rozdílu a je roven maximální hodnotě fázového segmentu 1 a době zpracování informace. Fázový segment 2 může být zkrácen.

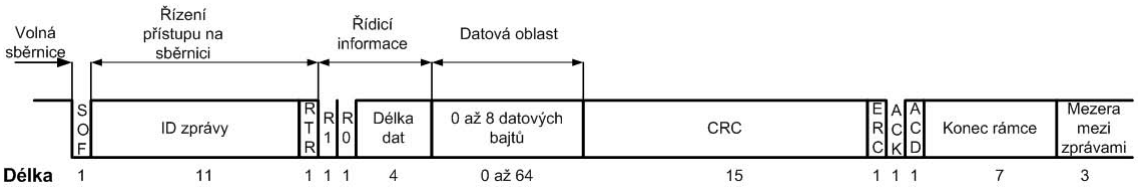
Celkový počet časových kvant je v rozsahu 8 až 25.
Přijímač je synchronizován s vysílačem na začátku vysílání zprávy. Je třeba zajistit, aby fázový rozdíl mezi interním oscilátorem a přijímaným signálem byl menší, než jedno časové kvantum tq v každém bitu zprávy.

## 3.4 Typy zpráv sběrnice CAN

Norma ISO 11898 sběrnice CAN definuje čtyři typy zpráv.

### 3.4.1 Data Frame

Datová zpráva umožňuje vyslat 0 až 8 bytů dat. Díky volitelné velikosti přenášené zprávy není zbytečně snižována propustnost sběrnice CAN. Velikost dat 0 se používá například na aktivaci příkazu zapnout nebo vypnout, kde daný příkaz je dán ID zprávy. Specifikace CAN 2.0A definuje standardní formát zpráv a CAN 2.0B rozšířený formát. CAN 2.0A má identifikátor dlouhý 11bitů a CAN 2.0B má identifikátor dlouhý 29 bitů.




Obrázek 3.3: Datová zpráva podle specifikace CAN 2.0A

Význam částí datové zprávy:

- **S**tart **O**f **F**rame je začátek zprávy, 1 bit dominant.
- **ID zprávy** - 11 bitů.
- **RTR** bit - Remote Request udává, zda se jedná o datovou zprávu nebo žádost o vyslání dat. V datové zprávě musí být tento bit dominant, v žádosti o data recessive.
- **R0**, **R1** - rezervované bity
- **Délka dat** je počet přenášených datových bajtů ve zprávě. Povolené hodnoty jsou 0 až 8.
- **Datová oblast** - datové bajty zprávy. Maximálně 8 bajtů.
- Zabezpečovací **CRC kód** a **CRC oddělovač**.
- **ACK** - bit potvrzení a **ACD** oddělovač potvrzení
- **Konec rámce**
- **Mezera mezi zprávami** - 3 bity recessive odděluje dvě zprávy

### 3.4.2 Remote Frame

Remote Frame je žádost o data, liší se od Data Frame v nastavení RTR bit do recesivní úrovně a neobsahuje datovou část.

### 3.4.3 Error frame

Chybová zpráva slouží k signalizaci chyb na sběrnici CAN. Pokud libovolný uzel detekuje chybu pomocí zabezpečovacího mechanismu, vygeneruje ihned na sběrnici CAN chybový rámec. Uzel generuje příznak chyby buď šesti aktivními nebo šesti pasivními logickými úrovněmi, podle toho, v jakém se nachází stavu. Tím je poškozena vysílaná zpráva a to zaregistrují i ostatní uzly, které také začnou vysílat chybové zprávy. Délka chybové zprávy je dlouhá minimálně 6 bitů a maximálně 12 bitů.

### 3.4.4 Overload Frame

Pokud nějaký uzel nezvládne z časového důvodu zpracovat data, která jsou vysílána na sběrnici, pošle zprávu o přetížení. Overload Frame má podobnou strukturu jako Error Frame, ale může být odvysílán až na konci zprávy.

Každý uzel má dva čítače chyb udávající počet chyb při příjmu nebo odesílání. Podle počtu chyb se uzel nachází v jednom ze tří možných stavů.

- **Aktivní stav** - uzel se aktivně podílí na komunikaci na sběrnici. V případě, že detekuje libovolnou chybu, vyšle aktivní příznak chyby Active Error flag.
- **Pasivní stav** - uzel se aktivně podílí na komunikaci, ale v případě detekce chyb posílá pasivní příznak chyb, který je nedestruktivní k právě odesílané zprávě a tím nedojde k nahlášení chyby.
- **Odpojeno** - Budiče těchto uzlů jsou vypnuty a nemají vliv na sběrnici.

| **Stav zařízení** | Hodnota čítače chyb při vysílání | Hodnota čítače chyb při příjmu |
|---|---|---|
| aktivní | $< 128$ | $< 128$ |
| pasivní | $>= 127$ | $>= 128$ |
| odpojeno | 256 | - |

Tabulka 3.2: Stav sběrnice CAN v závislosti na počtu chyb

# Kapitola 4

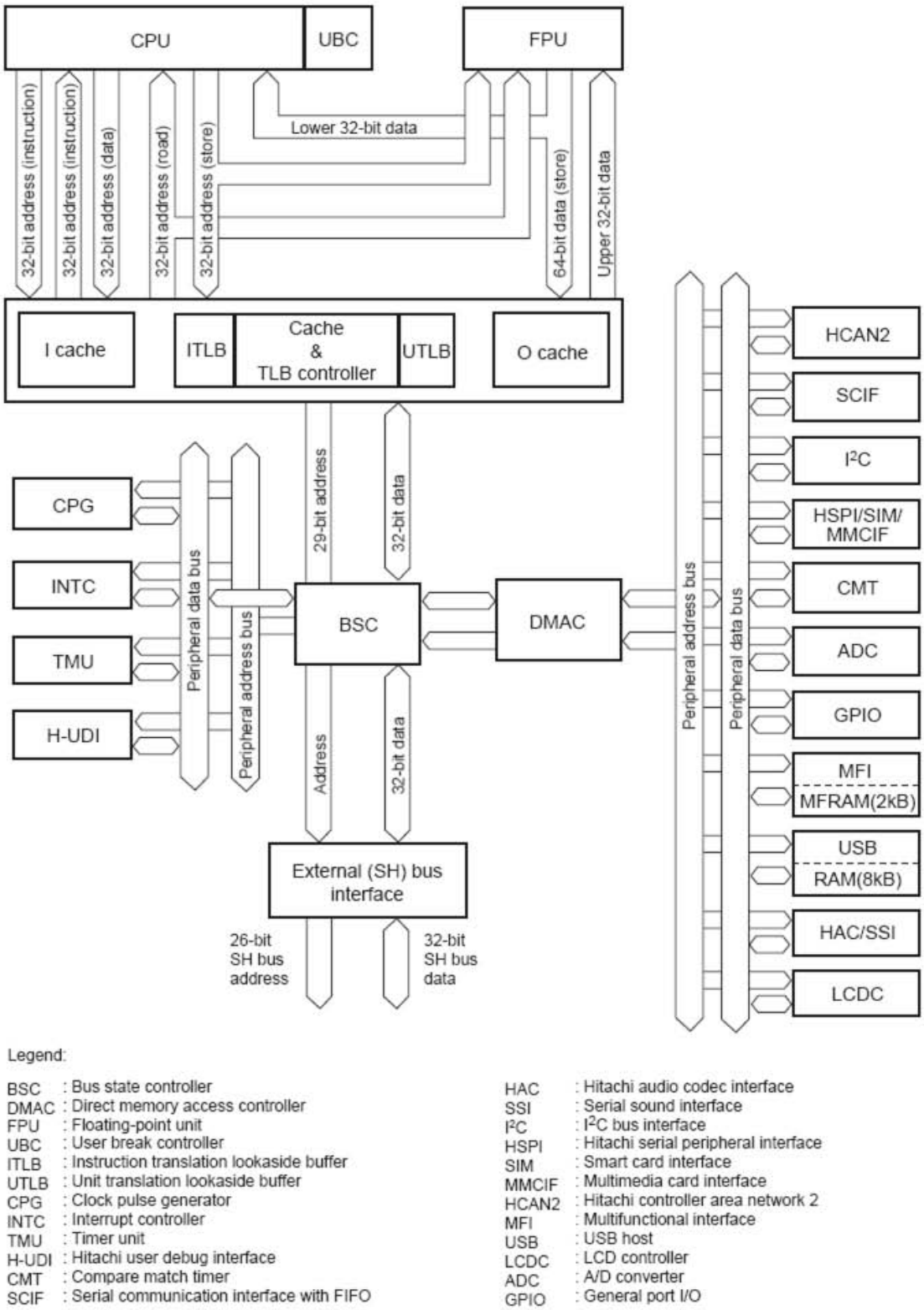
# Popis použitého hardwaru a software

Při výběru hlavního řídicího počítače byl kladen důraz na rychlost procesoru, kvůli náročnosti výpočtu akčních veličin při řízení modelu. Musel obsahovat sběrnici CAN pro komunikaci s inerciální jednotkou a řídicí jednotkou servomotorů. Důraz byl kladen na podporu ethernetu pro komunikaci s pozemní stanicí a Multimedia card pro uložení operačního systému a dat. Vybraný mikropočítač EXM32[8],[9] splňuje veškeré požadavky a obsahuje procesor SH7760[7] řady SH4. Na mikopočítači je používán operační systém SH Linux[21].

## 4.1 Procesor SH7760

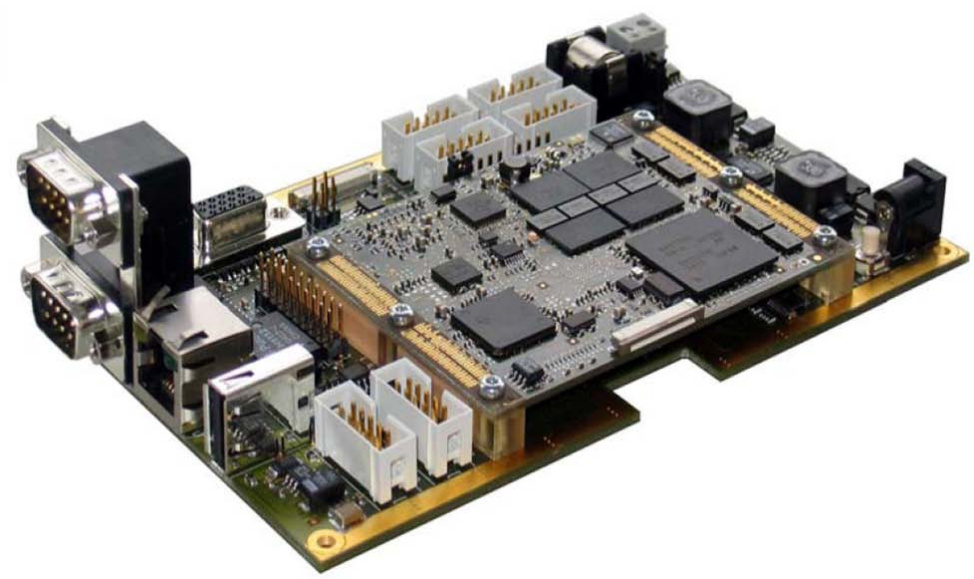
Základní vlastnosti procesoru:

- Hitachi SuperH architektura, Superscalar architektura.
- Frekvence Procesoru je 200MHz, 32-bit internal data bus.
- Šestnáct 32-bit hlavních , sedm 32-bit control, čtyři 32-bit systémové registry.
- Na čipu je floating-point koprocessor.
- Děvet nezávislých extérních přerušeních, 15 úrovní signalizace,
- Serial communication interface, Multimedia card interface, Hitachi controller area network 2, $I^2C$ bus interface, USB host, LCD controller, A/D converter




Obrázek 4.1: Procesor SH7760 - blokové schéma

## 4.2 Popis mikropočítače EXM32

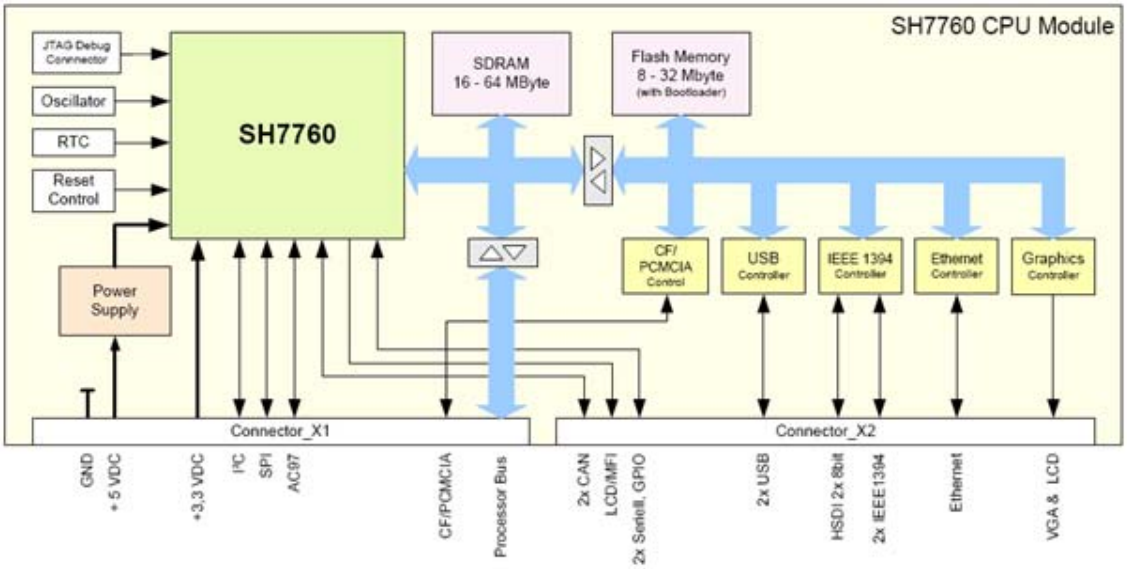


Obrázek 4.2: Mikropočítač EXM32

Základní vlastnosti mikropočítače EXM32:

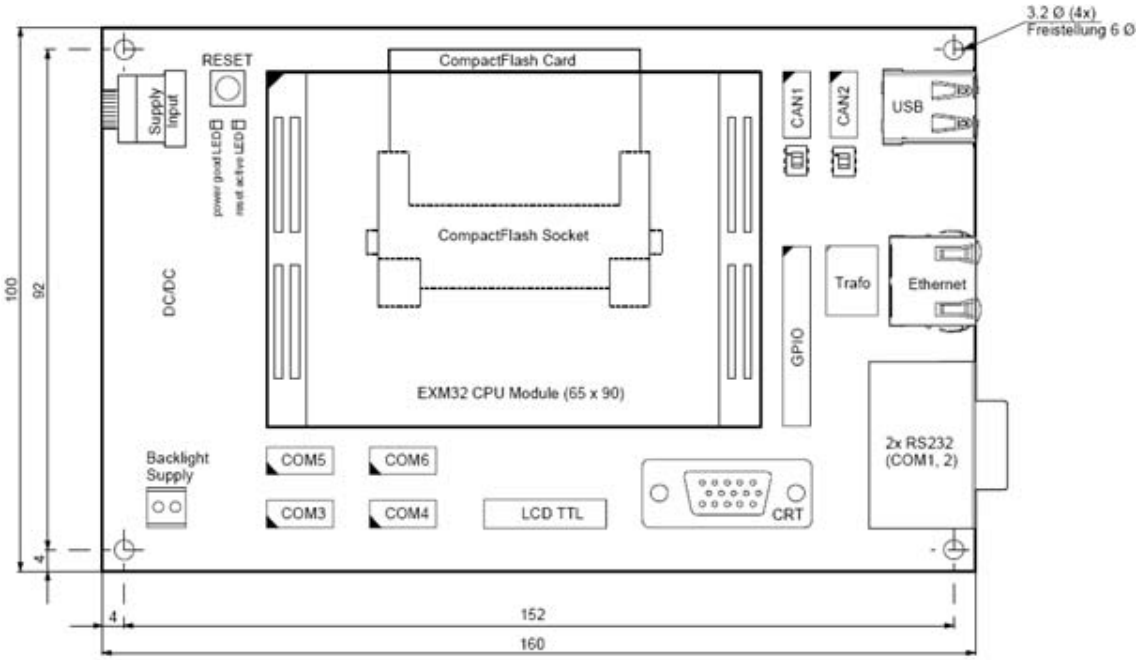
- Procesor Rensesas SH7760 32-bit RISC CPU 200MHz, 64MB SDRAM ( 32 bit ), 32MB Flash ( 16 bit ),
- 2x COM ( RS232 ),
- 4x COM ( TTL ),
- CompactFlash slot,
- 2x CAN,
- Ethernet controler,
- VGA a LCD,
- IEE 1394 OHCI Controller,

- USB.




Obrázek 4.3: Mikropočítač EXM32 - blokové schéma

Na motherboardu jsou umístěný dva COM RS232 nad sebou, kde spodní RS232 je COM1 a horní je COM2. CompactFlash disk se zasune lícní stranou ( obvykle je to strana, kde je jméno výrobce) dolů a sériovým číslem nahoru.




Obrázek 4.4: Mikropočítač EXM32 - vývod konektorů

## 4.3 Hitachi Controller Area Network 2

V kapitole 3 byla popsána sběrnice CAN a specifikace. Interface s programem této sběrnice na daném řadiči je dán jejím výrobcem. To znamená, že je na výrobci například způsob inicializace, počet mailboxů, kde v paměti budou uložena došlá data, přerušení a mnoho dalších parametrů. V této kapitole bude podrobně popsána architektura a práce s řadičem CAN, který byl implementován firmou Hitachi v procesoru SH7760. Budou v ní vysvětleny a popsány základní operace, jako je například inicializace, příjem a odesílání zpráv.

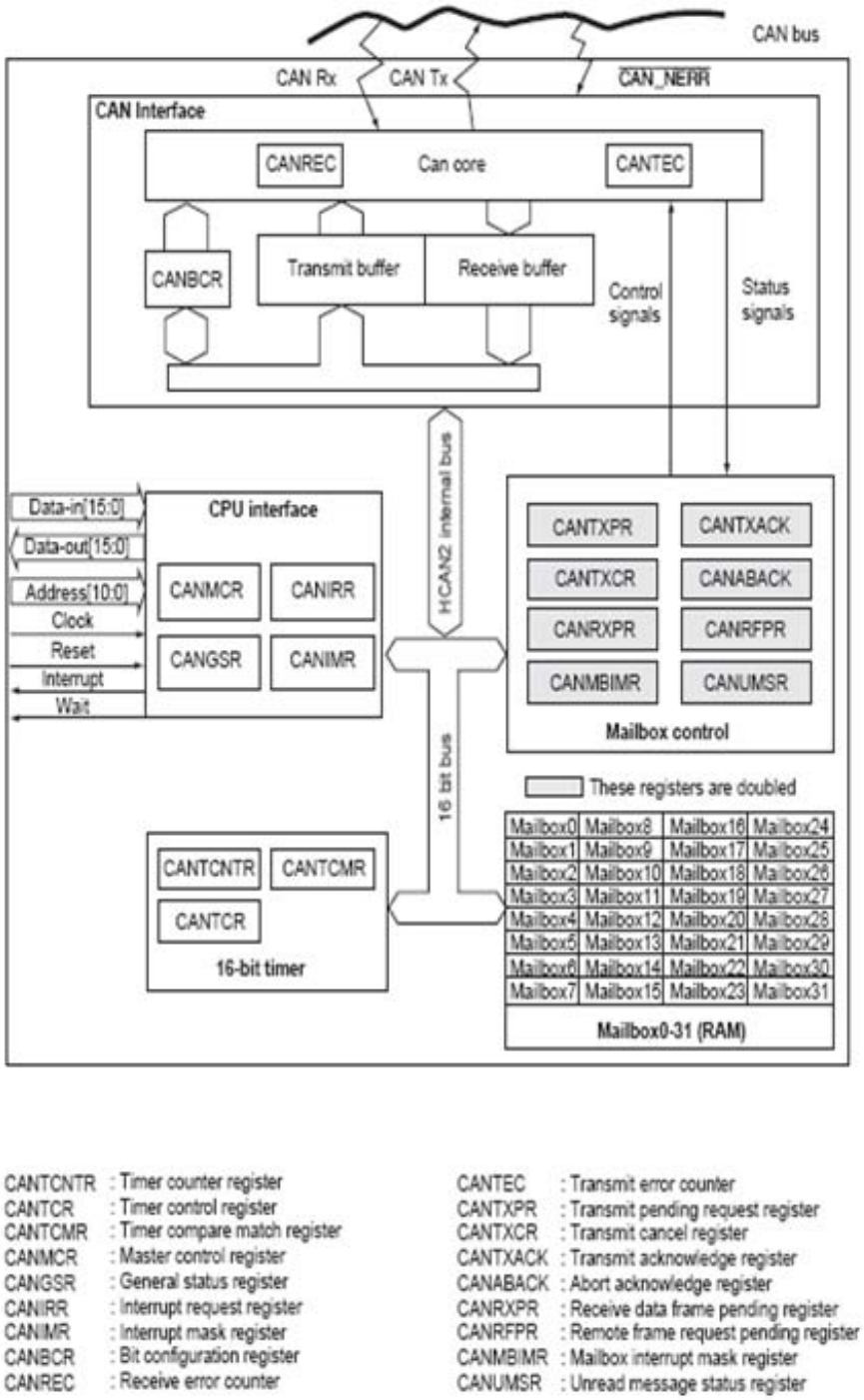
### 4.3.1 Obecný popis

Periférie procesoru SH7760 pro ovládání Controller Area Network se jmenuje Hitachi Controller Area Network 2 (HCAN2). Procesor obsahuje dva řadiče CAN - HCAN20 a HCAN21.
Mezi hlavní vlastnosti HCAN2 patří:

- Podpora specifikace normy ISO 11898 - 2.0A a 2.0B.

- Periférie HCAN2 obsahuje 31 programovatelných mailboxů na příjem a vysílání a jeden mailbox pouze na vysílaní.
- Podporuje režim Sleep.
- Obsahuje programovatelný filtr masek ( standardní a prodloužený identifikátor ) pro všechny mailboxy.
- Podporuje flexibilní Time stemp pro přenos i příjem zprávy

Periférie HCAN2 se skládá z pěti bloků - Micro Processor Interface (MPCI), Mailbox, MailboxControl, Timer a CAN interface. Umístění těchto částí je zobrazeno na obrázku 4.5




Obrázek 4.5: Blokové schéma HCAN2

### Micro Processor Interface

Umožňuje komunikaci mezi procesorem, registry a mailboxy CANu. Povoluje nebo zakazuje přerušení pro události CANu a nastaví komunikační rychlost. Pomocí MPCI může být aktivován režim Halt nebo Sleep.
Registry Master control register, Interrupt request register, Interrupt mask register a General status register patří do tohoto bloku.

### Mailboxy

HCAN2 obsahuje 32 **Mailboxů**.

| Adresa | Data bus | | | | | | | | | | | | | | | | Velikost | Název |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
| H'100 + N*32 | 0 | STDID[10:0] | | | | | | | | | | | RTR | IDE | EXTID [17:16] | | 16 bitů | Control |
| H'102 + N*32 | EXTID[15:0] | | | | | | | | | | | | | | | | 16 bitů | |
| H'104 + N*32 | | | NMC | ATX | DART | MBC[2:0] | | | 0 | | CBE | | DLC[3:0] | | | | 8/16 bitů | |
| H'106 + N*32 | TimeStamp[15:0] | | | | | | | | | | | | | | | | 16 bitů | Time stamp |
| H'108 + N*32 | MSG_DATA_0 (first Rx/Tx byte) | | | | | | | | MSG_DATA_1 | | | | | | | | 8/16 bitů | Data |
| H'10A + N*32 | MSG_DATA_2 | | | | | | | | MSG_DATA_3 | | | | | | | | 8/16 bitů | |
| H'10C + N*32 | MSG_DATA_4 | | | | | | | | MSG_DATA_5 | | | | | | | | 8/16 bitů | |
| H'10E + N*32 | MSG_DATA_6 | | | | | | | | MSG_DATA_7 | | | | | | | | 8/16 bitů | |
| H'110 + N*32 | Local acceptance filter mask 0 (LAFM0) | | | | | | | | | | | | | | | | 16 bitů | LAFM |
| H'112 + N*32 | Local acceptance filter mask 1 (LAFM1) | | | | | | | | | | | | | | | | 16 bitů | |

Obrázek 4.6: Mailboxy HCAN2

První část Control slouží k nastavení parametrů a konfiguraci daného mailboxu. STDID nebo EXTID udává ID přijaté zprávy, respektive se zde nastavuje ID odesílané zprávy. Pomocí MBC nastavujeme režim čtení a zápis mailboxu. DLC slouží k nastavení velikosti odesílané zprávy. LAFM představuje masku pro ID doručených zprávy.

### Mailbox Control

Pro odesílání a ukládání přijatých zpráv do paměti RAM slouží Mailbox Control. Stará se o arbitráž pro přístup na sběrnici CAN, dle priorit nebo čísla mailboxu. Nastavuje registry pro identifikaci stavu CANu.

Mailbox control má následující registry: Transmit pending register, Transmit cancel register, Transmit acknowledge register, Abort acknowledge, Receive data frame pending, Remote frame request pending a Mailbox interrupt mask.

### Timer

Má za úkol přidat do mailboxu Time Stamp přijaté nebo odeslané zprávy. Registry Timer counter, Timer control a Timer compare match jsou registry, které patří do tohoto bloku.

### CAN Interface

Tento blok je dán specigikací CAN2.0 a má například za úkol sledovat chyby na sběrnici a jejich počet ukládat do registrů pro odesílané a příchozí zprávy. Registry CAN Interface jsou Receive Error Counter, Transmit Error Counter, the Bit Configuration.

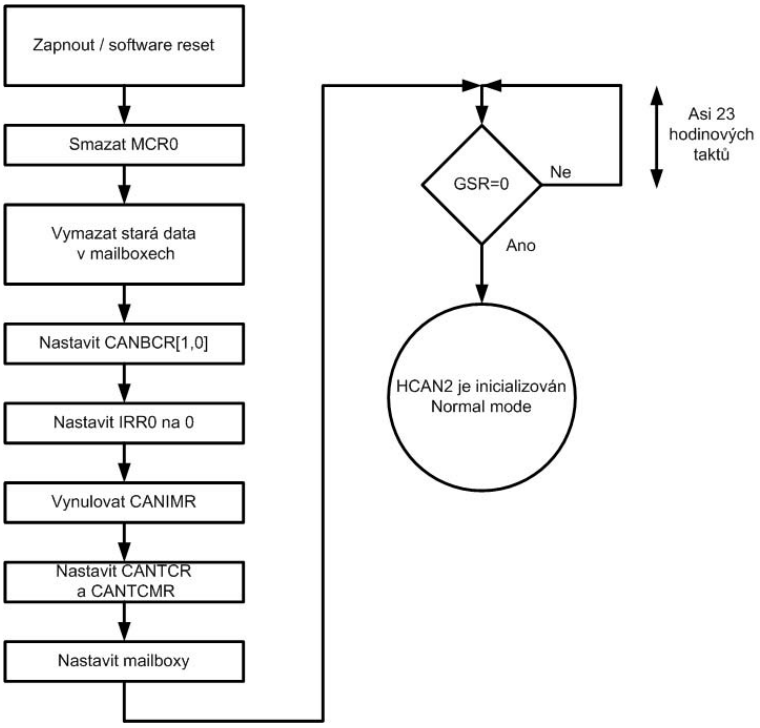
### 4.3.2 Práce s CANem

#### Inicializace CANu

Pro správné fungování je nutné CAN nejdříve inicializovat podle přesně dané sekvence.

Před inicializací se musí znát komunikační rychlost sběrnice CAN. Rychlost se nastavuje pomocí registrů BCR0 a BCR1. Konfiguraci jednotlivých mailboxů je možné nastavit také za provozu sběrnice.

Nejdříve se smažou staré flagy a data z mailboxu. Potom se inicializuje IRR0 a nastaví rychlost. Nakonec se povolí přerušení, které se bude v modulu obsluhovat a počká se asi 23 taktů na inicializaci CANu.

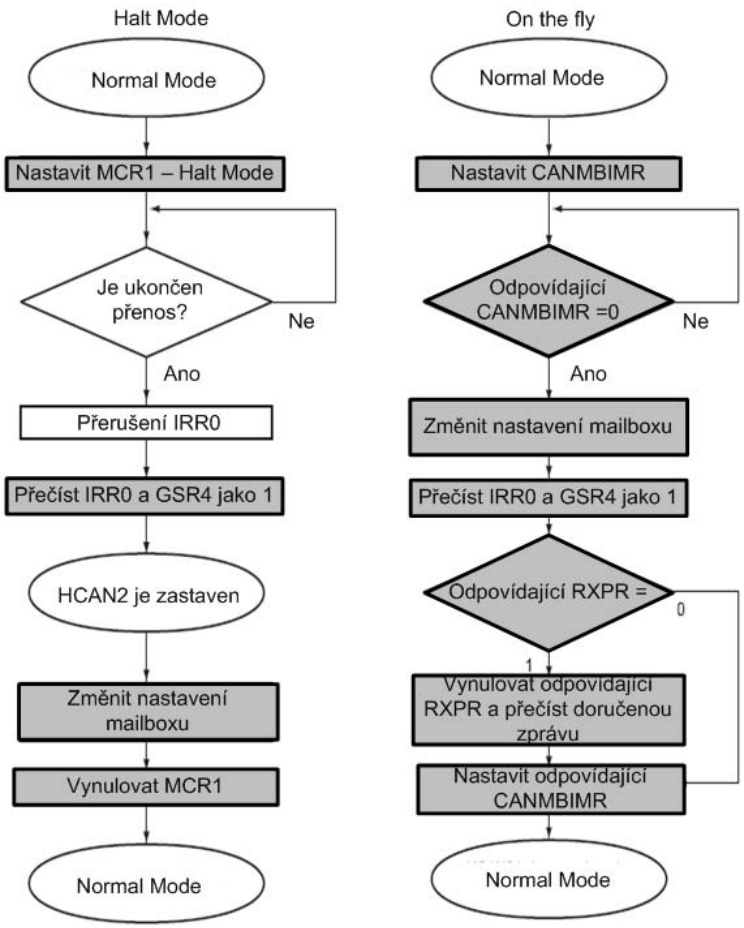



Obrázek 4.7: Inicializace CANu na procesoru SH7760

Funkce *void CAN_Init( int BCR0, int BCR1 )* v modulu jádra hcan20.c inicializuje mailbox dle sekvence, která je na obrázku 4.7.

### Změna parametrů mailboxu CANu za provozu

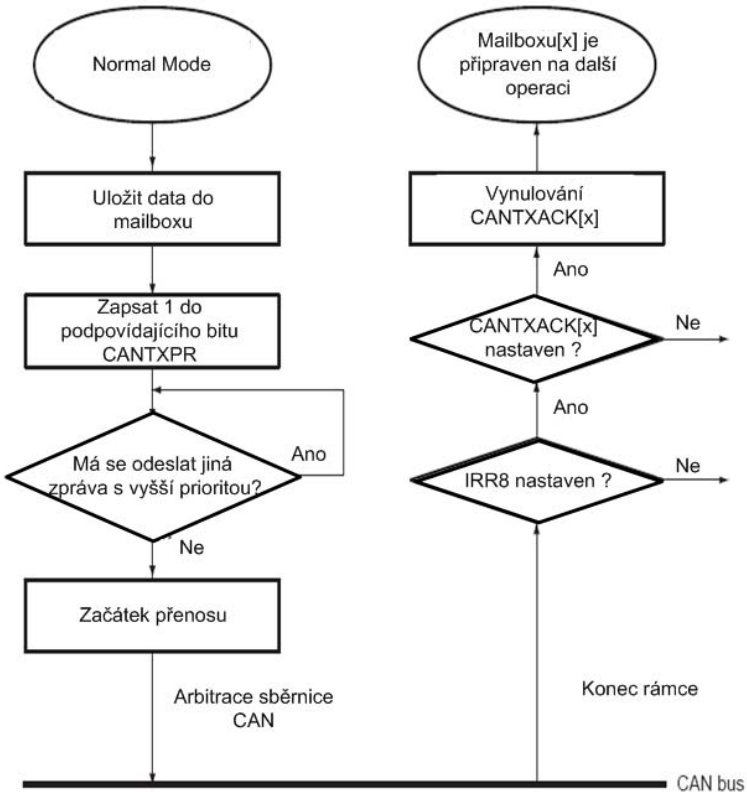
Existují dvě možnosti, jak nastavit parametry Mailboxu za provozu.
První možnost je zastavit CAN, nastavit mailbox a opět ho spustit (**Halt Mode**). Při této možnosti nedojde ke ztrátě dat před pozastavením. Činnost CANu je pozastavena až po ukončení všech operací, jako je příjem a odesílání dat. Během pozastavení může dojít k nepřijetí zpráv.
Druhá možnost je změnit nastavení za běhu (**on-the-fly**). Výhodou této metody je, že změna probíhá jen na jednom mailboxu a ostatní mailboxy normálně fungují. Nevýhoda je, že může dojít k nepřijetí a nebo neodeslání poslední zprávy před rekonfigurací.
Algoritmy jsou na obrázku 4.8.




Obrázek 4.8: Změna nastavení mailboxů na procesoru SH7760

### Odesílání zpráv

Z pohledu modulu jádra se zpráva po sběrnici CAN odesílá jednoduše. Nejdříve se zapíší data a ID zprávy do příslušného mailboxu, z kterého budou data odesána. Potom se v registru Transmit pending request nastaví bit odpovídající příslušnému mailboxu. Nyní Mailbox Contol rozhodne odešle zprávu. Po úspěšném odeslání je vyvoláno přerušení a nastaven bit odpovídající číslu mailboxu v Transmit acknowledge registru. Analogicky je možné odeslat Remote Frame.
Přesný algoritmus odesílání zpráv je na obrázku 4.9.



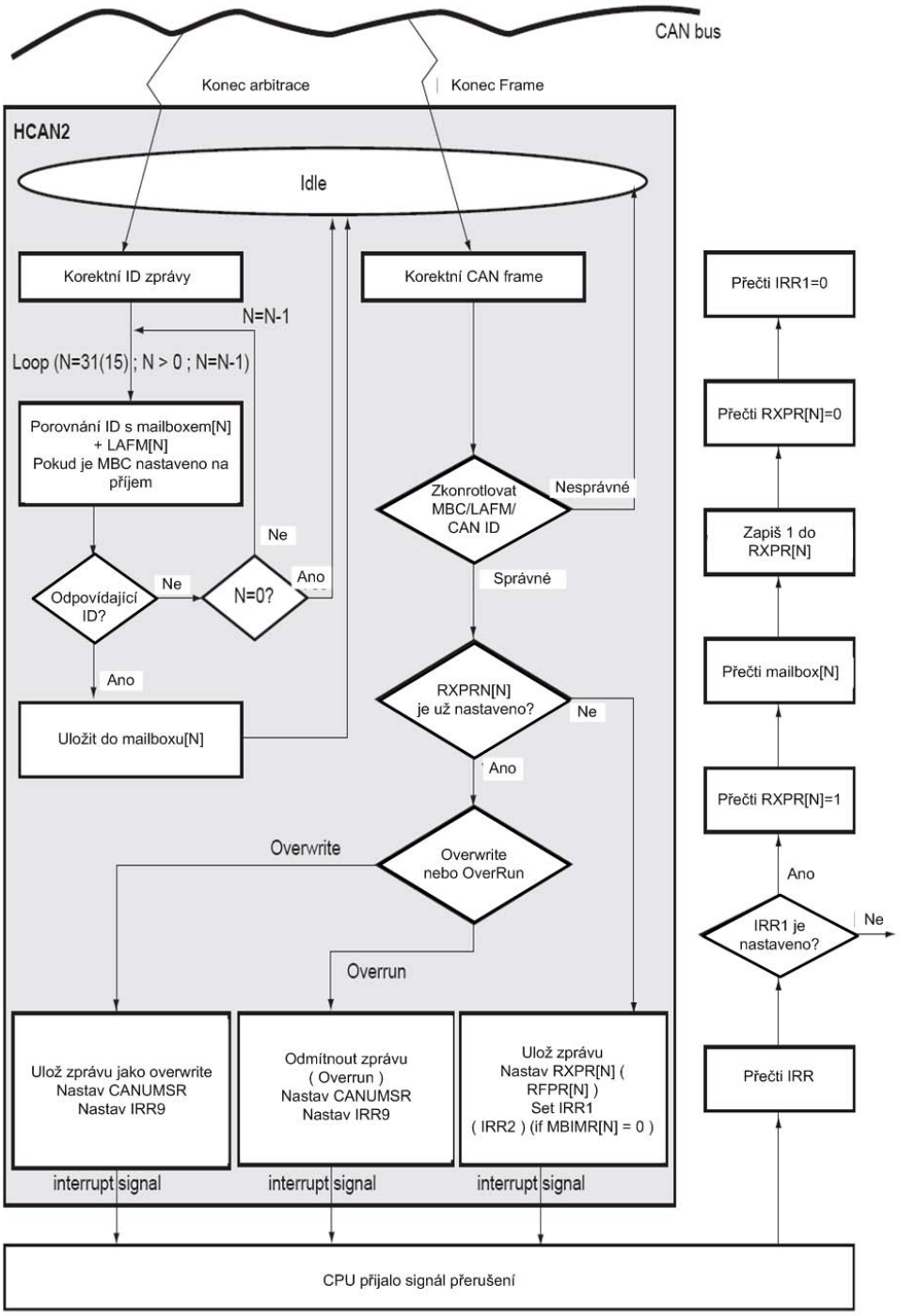

Obrázek 4.9: Odeslání zprávy na procesoru SH7760

### Příjem zpráv

Pro informaci o příchozí zprávě je nutné mít povolené přerušení příchozích zpráv a současně i přerušení pro daný mailbox.

U doručené zprávy se porovnává ID zprávy s LAFM mailboxů. Pokud $(LAFM\&ID) = ID$, je došlá zpráva uložena do prvního mailboxu s touto maskou. Do STDID mailboxu je uloženo skutečné ID zprávy. Mailboxy se kontrolují sestupně ke svému číslu. Pokud není nalezena shoda s maskou, tak zpráva není uložena.
Po zapsání zprávy do mailboxu je vyvoláno přerušení odpovídající Data Frame nebo Remote Frame a zapsán bit do registru Receive data frame pending pro došlý mailbox. Pomocí přerušení a tohoto registru, zjistí modul jádra, že došla nová zpráva a do kterého mailboxu byla doručena.

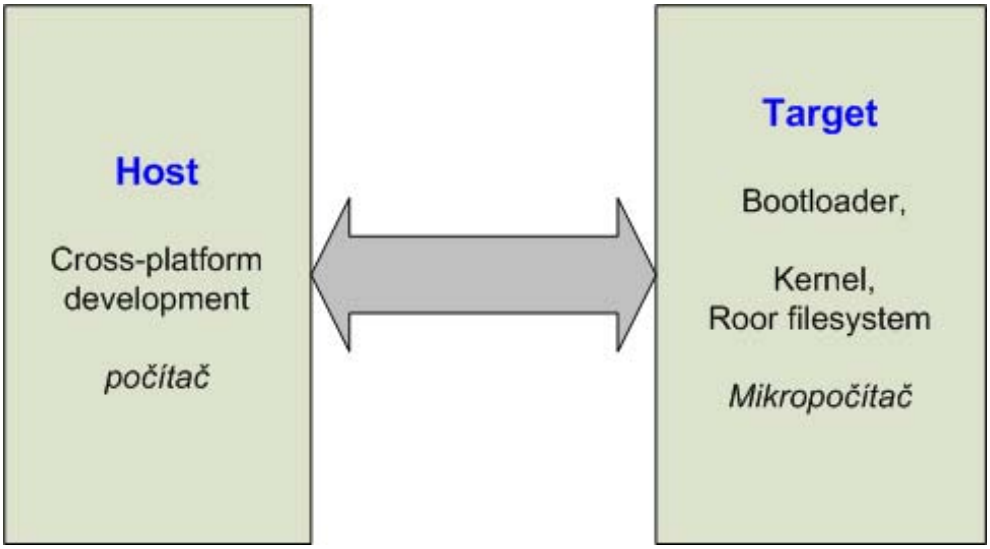
Podrobný algoritmus příjmu zpráv je na obrázku 4.10.




Obrázek 4.10: Příjem zprávy na procesoru SH7760

## 4.4 Linux a vývojový software

Aplikace pro mikropočítač se vyvíjejí na osobním počítači. Vývojový prostředek na počítači byl zvolen překladač gcc, který patří do GNU projektu. Aby bylo možné překládat (crosscompilovat) programy pro použitý mikopočítač, musí se nainstalovat překladač pro architekturu SH4, knihovny funkcí pro jádro systému a další knihovny. V kapitole A je popsán postup a pořadí instalace vývojových nástrojů.




Obrázek 4.11: Překlad aplikací pro mikropočítač

Na mikropočítači je použit SH Linux 2002. Jedná se o plnohodnotnou distribuci v textovém režimu. Jádro OS Linux je verze 2.6.14. Do systému je možné instalovat software pomocí balíčků nebo překladu.

V této diplomové práci byl použit tento software na osobním počítači:

- **gcc-sh-linux-3.4.4-1.i386** - **G**NU **C**ompiler **C**ollection, překladač programovacího jazyka C,
- **gdb-sh-linux-20001217-4.i386** - **G**NU **D**eb**ug**ger, ladící nástroj
- **glibc-sh-linux-2.3.3-27.12.noarch** - GNU project's C standard library, knihovna funkcí pro architekturu SH,

- **glibc-sh4-linux-2.3.3-27.12.noarch** - GNU project's C standard library, knihovna funkcí pro architekturu SH4,
- **Insight 6.0** - grafický ladící nástroj, http://sources.redhat.com/insight,
- **SH Linux 2002** - operační systém,
- **Software pro EXM32** - autor: ing Miroslav Žižka,
- **zlib-sh-linux-1.1.3-5.noarch** - cross-platform data compression library,
- **zlib-sh4-linux-1.1.3-5.noarch** - knihovna pro sh4.

Použitý hardware:

- **EXM_DEBUGADAPTOR** - Propojení mezi mikropočítačem a JTAGem,
- **JTAG debbuger** - **J**oint **T**est **A**ction **G**roup, dle IEEE 1149.1, standartni port pro přístup k procesoru a paměti. Byl použit od firmy Lauterbach,
- **MSC EXM32** - Hlavní řídicí počítač.

Důležité internetové adresy:

- http://www.sh-linux.org/ - na těchto stránkách jsou instalační balíčky pro SH Linux a tvorbu aplikací pro platfomu SH3 a SH4,
- http://www.kernel.org/ - jádra OS Linux jsou k dispozici na těchto stánkách,
- http://www.ms-n.org/SH-Linux/ Stránky obsahují jádro a instalalační balíčky pro OS Linux,
- http://www.shlinux.com/ - stránky věnované SH Linuxu pro mikropočítače Edosk. Některé postupy je možné použít por libovolný mikropočítač architektury SH.

## 4.5 Ochrana proti zápisu na mikopočítači

Linux sh2002 může být spuštěn na mikropočítači ve dvou režimech. Při prvním režimu je zakázán zápis na CompactFlash disk z důvodu, že v modelu vrtulníku se systém neukončuje pomocí příkazu poweroff, ale je vypnuto napětí. Kvůli tomu by mohlo dojít k poškození souborů a také by při startu systému byl kontrolován CompactFlash disk. Kontrola CompactFlash disku zbytečně zpomaluje start systému.

Aby mohl systém Linux bez problému fungovat při zakázaném zápisu na CompactFlash disk, musí se vždy při startu část systému nakopírovat do paměti RAM. Přesněji, původní adresář /etc se přejmenuje na /etc-cf a v paměti se vytvoří nový adresář /etc, který bude nahrán z /etc-cf. Rozhodující je, že do adresáře /etc je možno zapisovat data. Pokud uživatel cokoliv změní nebo smaže v adresáři /etc, po novém startu systému se tyto změny neprojeví. Podobné operace se udělají i s adresářem /var. Skript, který se používá pro režim zakázaného zápisu, je /etc/rc.d/rc.cfroot. Autorem skriptu je Ing. Pavel Píša. Pokud by uživatel chtěl změnit nějaké nastavení v adresáři /etc nebo /var musí nejdříve povolit zápis na CompactFlash disk pomocí příkazu *cfrootrw*.

```
#cfrootrw
```

Při povoleném zápisu se může zapisovat kamkoliv na CompactFlash disk, ale obsahy adresářů /etc a /var jsou v paměti RAM. Může se změnit nastavení, ale ne v adresáři /etc, ale v adresáři /etc-cf. Po změnách se zakáže zapis na CompactFlash disk pomocí příkazu *cfrootro*.

```
#cfrootro
```

Během zakázaného zápisu na CompactFlash disk je možné uložit dočasné soubory do adresáře /tmp, který je namapován do paměti RAM.

V některých případech je potřeba, aby systém mohl zcela zapisovat na CompactFlash disk, proto je nutné vypnout ochranu zápisu. Vypnutí ochrany se docílí za komentováním řádku /etc/rc./rc.cfroot v souboru /etc-cf/rc.d/rc.sysinit a resetem systému. Nyní je třeba systém správně ukončit pomocí příkazu poweroff nebo reboot, kdy se provede reset systému. Pokud dojde k havárii systému, bude při dalším startu kontrolována integrita CompactFlash disku.

# Kapitola 5

# Portování Linuxu

Aby bylo možné provozovat Linux na mikropočítači EXM32, musí správně proběhnout několik inicializačních částí[3].

Základním úkolem Initial Program Load (IPL) je inicializovat zařízení v minimální konfiguraci, aby bylo možné spustit operační systém.

Linux loader (LILO) zavede a spustí jádro systému Linux. Aby jádro OS Linux správně fungovalo, musí být přeložené pro použitý procesor a obsahovat ovladače mikropočítače.

## 5.1 Initial Program Load

Jsou dva typy IPL.

- **Cold-start IPL**, cílový počítač nemá BIOS. IPL musí být nahráno na adresu vectoru reset. Toto je například mikropočítač EXM32.
- **Warm-start IPL** Počítač má BIOS a IPL je jednoduché rozšíření IPL.

Dále bude popsán typ Cold-start IPL.

Program IPL musí být nahrán na adresu Reset vectoru. Následuje pořadí základních operací, které vykonává IPL.

1. Inicializovat procesor. Inicializační část je psána v assembleru a pro jejím provedení je vyvolána funkce main programu IPL, psaná již v jazyku C.
2. V závislosti, kde je operační systém uložen, musí IPL namapovat paměť a nebo zařízení. Namapování paměti dělíme na **Linearly mapped**, například paměť ROM nebo FLASH, a **Bank-switched**, například

disk, PC-Card nebo síť.

3. Nakopírovat startovací program do paměti RAM.
4. Spustit tento program a předat mu řízení.

Při tvorbě IPL se musí nadefinovat, že se jedná o procesor rodiny SH4, little endian a je potřeba inicializovat IDE, aby mohl být spuštěn Linux z CF.

## 5.2 Linux loader

Program Linux Loader přijme kontrolu od IPL a předá ji Linuxovému jádru. Není vyvíjen pro specifický file systém a může zavést libovolný operační systém z disku, diskety nebo CompactFlash disku. U mikropočítače EXM32 je použita verze lilo-21.7.3. Pro vytvoření bootovacího CompactFlash disku nebo přeložení nové verze jádra je nutné vytvořit na CF lilo. Lilo se vytvoří pomocí osobního počítače a verze lilo na PC musí být stejná jako je na EXM32. Ačkoliv verze lilo 21.7.3 je provozována s Linuxovým jádrem 2.6, je nutné mít pro překlad zdrojové kódy jádra 2.4.
Pokud by bylo nutné použít novější verzi lilo, musí se crosscompilovat a nakopírovat do adresáře /boot na CF.
Konfigurace, jak se má lilo nastavit je v souboru lilo.conf. Lilo se na CF vytvoří:

```
# ./lilo -r /mnt/cf -C /lilo.conf
```

Soubor *lilo.conf* je umístěn v kořenovém adresáři na CompactFlash disku.

## 5.3 Přerušení

V dokumentaci k procesoru SH7760 se neudává číslo přerušení, ale INTEVT code, z kterého se dá vypočítat číslo přerušení dle následujícího vzorce IRQNUMBER = ( INTEVT >> 5 ) - 16. ovladače Číslo přerušení sběrnice CAN je HCAN20 = 56 a HCAN21 = 57.

Během programování ovladače CANu nefungovalo korektně přerušení. Po bližším prozkoumání jádra bylo zjištěno, že obsahuje pro procesor SH7760 chyby. Přerušení nebylo správně aktivováno. Chyba se opravila v souboru /arch/sh/kernel/cpu/sh4/irq_intc2.c ve strukture struct intc2_init

Struktura ukládá data, s kterými jádro nastavuje přerušení pomocí funkce

```
void make_intc2_irq(unsigned int irq,
    unsigned int ipr_offset, unsigned int ipr_shift,
    unsigned int msk_offset, unsigned int msk_shift,
    unsigned int priority).
```

Parametr irq je číslo přerušení, ipr_offset je offset ipr, který je pro HCAN20 28 až 32, proto ipr_shift = 28 a ipr_offset=4. msk_offset = 0, číslo bitu Interrupt Request Sources pro HCAN20 je 25. Poslední parametr je priorita přerušení.

```
Správné nastavení:
{52,  8, 16, 0, 11,  3},/* SCIF0_ERI_IRQ */
{53,  8, 16, 0, 10,  3},/* SCIF0_RXI_IRQ */
{54,  8, 16, 0,  9,  3},/* SCIF0_BRI_IRQ */
{55,  8, 16, 0,  8,  3},/* SCIF0_TXI_IRQ */
{72,  8, 12, 0,  7,  3},/* SCIF1_ERI_IRQ */
{73,  8, 12, 0,  6,  3},/* SCIF1_RXI_IRQ */
{74,  8, 12, 0,  5,  3},/* SCIF1_BRI_IRQ */
{75,  8, 12, 0,  4,  3},/* SCIF1_TXI_IRQ */
{76,  8,  8, 0,  3,  3},/* SCIF2_ERI_IRQ */
{77,  8,  8, 0,  2,  3},/* SCIF2_RXI_IRQ */
{78,  8,  8, 0,  1,  3},/* SCIF2_BRI_IRQ */
{79,  8,  8, 0,  0,  3},/* SCIF2_TXI_IRQ */
{64,  8, 28, 0, 17,  3},/* USBHI_IRQ */
{68,  8, 20, 0, 14, 13},/* DMABRGI0_IRQ */
{69,  8, 20, 0, 13, 13},/* DMABRGI1_IRQ */
{70,  8, 20, 0, 12, 13},/* DMABRGI2_IRQ */

{56,  4, 28, 0, 25,  4},/* HCAN20 */
{57,  4, 24, 0, 24,  4},/* HCAN21 */
```

Je to pouze základní nastavení přerušení pro účel použití v modelu vrtulníku. Další přerušení lze dodefinovat pomocí datasheetu nebo získat s novou verzi jádra.

V době psaní této diplomové práce je v Linuxovém jádře 2.6.16 plná podpora pro procesor SH7760 a přerušení by mělo fungovat bez problémů. Pro implementaci nového Linuxového jádra se musí přidat podpora pro EXM32, kterou vytvořil Ing. Miroslav Žižka, a jádro přeložit. Podpora mikropočítače se přidá pomocí souboru:
< cesta k jádru >/arsh/sh/Kconfig. Nastavení několika souborů Kconfig je analogické s nastavením pro verzí 2.6.14. Konfigurace pro platformu sh je vyvolána pomocí

```
# make xconfig ARCH=sh
```

Nové jádro nebylo implementováno z důvodu data vydání, které bylo před ukončením diplomové práce.

# Kapitola 6

# Ladění programů na mikropočítači EXM32

Při tvorbě softwaru dochází ke vzniku chyb. Překladač program úspěšně přeloží, ale program nefunguje tak, jak by si jeho autor přál. Pro ladění aplikací na mikropočítači EXM32 existuje několik možností[18]. Záleží, zda se bude ladit běžná aplikace, modul jádra, celé jádro nebo modul a aplikace součastně.

## 6.1 JTAG debugger

Pokud je k dispozici JTAG - Debuger a EXM DebugAdapter může se pomocí aplikace TRACE32 ladit program. V programu TRACE32 je k dispozici standardní prostředí pro ladění programů. JTAG Debuger a EXM DebugAdapter se připojí dle obrázku A.1.

## 6.2 Insight, DDD

Grafická možnost ladění aplikací nebo jádra systému na osobním počítači přinášejí programy Insight nebo DDD. Mikropočítač MSC EXM32 je připojen k osobnímu počítači pomocí sériového kabelu. Kabel je připojen na MSC EXM32 na stejný port, kde se nachází konzole.

Program IPL byl modifikován tak, že jsou-li všechny vypínače v poloze nula, automaticky se nastartuje systém Linux. Pokud je libovolný přepínač v poloze jedna, IPL se zeptá uživatele, jestli si přeje spustit systém Linux, nebo debuggovací mód. Při zvolení debuggovacího režimu se musí na os-

obním počítači vypnout konzole a teprve potom pustit Insight nebo DDD. Samozřejmě je možné také spustit na počítači textový GDB.

Aby bylo možné ladit programy tímto způsobem, je nutné nastavit startovací adresu programu na 0x08d000000. Na startovací adrese se totiž v tomto módu nachází debuger, který nedovolí svoje přepsání. Následuje výpis makefile.

```
all: program
SRC = program.c
CC = sh4-linux-gcc
LD = sh4-linux-ld

INCLUDE = .
CFLAGS = -g -Wall -c
POSUN = -Ttext=0x08d000000

all: program.o
$(LD) $(TARGET).o -o $(TARGET) $(POSUN)

program.o:
$(CC) $(CFLAGS) $(TARGET).c

clean:
rm -f *.o *~ core
```

makefile

## 6.3 Ladění modulu jádra

Modul lze přilinkovat k jádru a to celé ladit například pomocí Insight. Nevýhoda je, že překlad jádra je časově náročný. Další možnost je používat co nejvíce výpisů a použít program rdwrmem od Ing. Pavla Píši. Tento program umí vypisovat a zapisovat do paměti a registrů. Jedná o aktuální výpis paměti, která v dalším okamžiku může být jiná. Na zjištění některých informací to postačuje. Pro účely ladění přerušení je vhodné sledovat soubor /proc/interrupts, kde je výpis počtu přerušení od spuštění počítače.

Nejobtížnější je ladit současně uživatelskou aplikaci a modul jádra, které vzájemně mezi sebou komunikují. Zde je jen možnost ladit pomocí výpisu proměnných a sledování stavu paměti.

# Kapitola 7

# Modul jádra Linuxu pro obsluhu CANu

Aby bylo možné zpracovávat odesílat a přijímat zprávy po sběrnici CAN je nutné vytvořit ovladač. Tento ovladač musí být maximálně univerzální, aby bylo možné nastavit parametry, jak požaduje aplikace. Mezi tyto parametry patří například komunikační rychlost a nastavení zpráv, které se mají přijmout.

Jednou z částí této diplomové práce bylo vytvoření ovladače CANu pro Linux, který byl vytvořen jako modul jádra. Procesor SH7760 obsahuje dva CANy. Bylo zvoleno řešení, že každý CAN má svůj vlastní modul. Toto řešení má výhodu, že ovladače jsou na sobě nezávislé. Nevýhoda je existence dvou modulů.

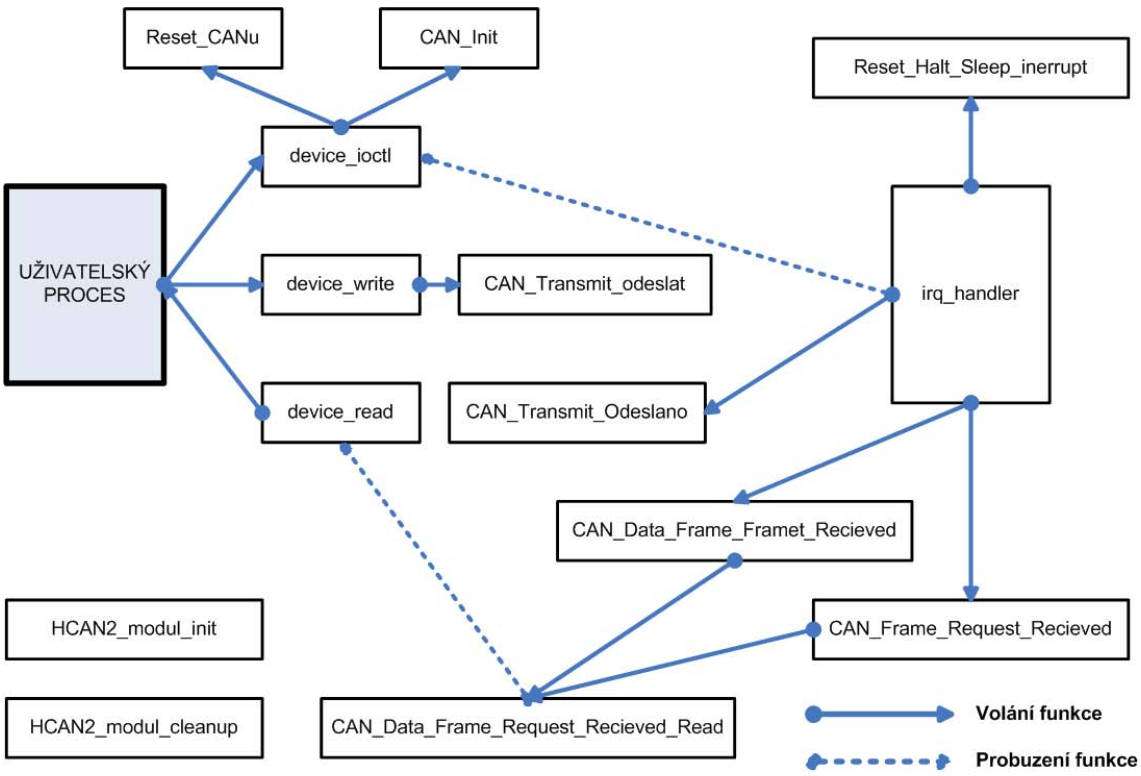
Při překladu modulu se zadá adresový prostor CANu a modul nastaví další konfiguraci sám při inicializaci.

Ovladač se jmenuje HCAN2 a skládá se z několika částí. První část je inicializace modulu a zařízení. Druhá část dělá konfiguraci a reaguje na požadavky uživatelské aplikace přes ioctl. Třetí část obsluhuje přerušení, které se například vyvolá v případě došlé nebo odeslané zprávy a v případě chyby. Přerušení je možné zakázat. Tento ovladač reaguje na většinu přerušení a v případě chyby pomocí ioctl předá informace o chybě uživatelské aplikaci. Poslední důležitá část předává nebo získává z aplikace data. Modul samozřejmě obsahuje ukončovací část. Modul je navržen jako blokovací. Blokovací režim je podrobněji popsán v kapitole 2.3.5. Algoritmy pro inicializaci CANu, obsluhu přerušení, odesílání a příjem zpráv jsou použity dle [10].

Číslo zařízení bylo zvoleno 120 pro první a 121 pro druhý CAN.

Závislost a volání jednotlivých funkcí je na obrázku 7.1. Podrobný popis funkcí je funkcí je v příloze C.1.

**Závislost funkcí v modul HCAN2**




Obrázek 7.1: Závislost funkcí v modul HCAN2

## 7.1 Inicializace modulu

Při inicializaci modulu je zaregistrováno zařízení hcan20 nebo hcan21 v adresáři /dev a přerušení odpovídající CANu. Další nastavení se nastaví pomocí ioctl.

## 7.2 ioctl

Konfigurační informace jsou do modulu přenášeny pomocí ioctl. Bylo vytvořeno celkem sedm funkcí ioctl.
Pro správný chod je nutné mít v uživatelském programu přidán header $/include/konfigurace.h$, který se nachází v adresáři se zdrojovým kódem.

První ioctl má název KONFIGURACNI_CISLO_MAILBOX, slouží k nastavení parametrů mailboxu a je definované jako _IOW. Z uživatelské aplikace je volána jako

```
$ioctl( int fd, IOCTL_SET_CAN20, struct &can_konfigurace_mailbox );
```

Parametr mailbox je struktura, kde jednotlivé položky odpovídají položkám mailboxu. Při nastavení mailboxu se automaticky nastaví přerušení pro daný mailbox.

```
struct can_konfigurace_mailbox {
int cislo_mailboxu;
int MBC;
int id;
int lafm;
}
```

CAN se inicializuje, pokud je volané ioctl KONFIGURACNI_CISLO_BCR a je _IOW. Z uživatelské aplikace voláno

```
ioctl( int fd, IOCTL_BCR_CAN20, struct &struct ) ;

struct can_BCR {
int BCR0;
int BCR1;
}
```

Pro zjištění chyb na sběrnici CAN slouží ioctl INFO_CHYBY_NA_CANU, které je _IO. Vrací číslo přerušení, které je svázáno s chybou, a je to ioctl blokující.

Při ladění modulu je užitečné si nechat vypisovat události, které nastaly, data, která se přijala a další důležité informace. Aby se v modulu nemusely funkce na výpis informace zakomentovávat a odkomentovávat, lze použít ioctl DEBUGGER. Parametr je int, kde je-li nastaven na 1, dochází k výpisu zpráv. V modulu je k výpisu zpráv použito makro

```
#define MSG(string, args...)  if ( zobrazovat_debuger_info )
printk("info z modulu hcan20 : " string, ##args)
```

Reset CANu se udělá pomocí ioctl _IO RESET_CANu.
Vypnout přerušení pro daný mailbox lze pomocí ioctl VYPNOUT_PRERUSENI_PRO_MAILBOX, kde parametr je číslo mailboxu.

Zapnout lze opětovnou inicializací.

Přednastavit mailbox v režimu Halt mode lze pomocí KONFIGURACE_MAILBOX_PROVOZ. Parametr je struktura can_konfigurace_mailbox.

## 7.3 Posílání a příjem zpráv

Po inicializaci CANu a nastavení mailboxů lze odesílat a přijímat data. Uživatelská aplikace s jádrem komunikuje pomocí funkcí copy_from_user a copy_to_user. Parametr při předání je na datovou strukturu, která je použita v LinCanu.

```
struct canmsg_t {
short flags;
int cob;
canmsg_id_t id;
unsigned long timestamp;
unsigned int length;
unsigned char data[CAN_MSG_LENGTH];
}
```

Modul obsahuje buffer pro 100 přijatých zpráv. Při překročení tohoto limitu budou staré zprávy smazány.

## 7.4 Ukončení modulu

Při ukončování modulu se zruší zařízení hcan20 a uvolí se přerušení. Činnost řadiče CANu je pozastavena.

# Kapitola 8

# Sběr telemetrických dat

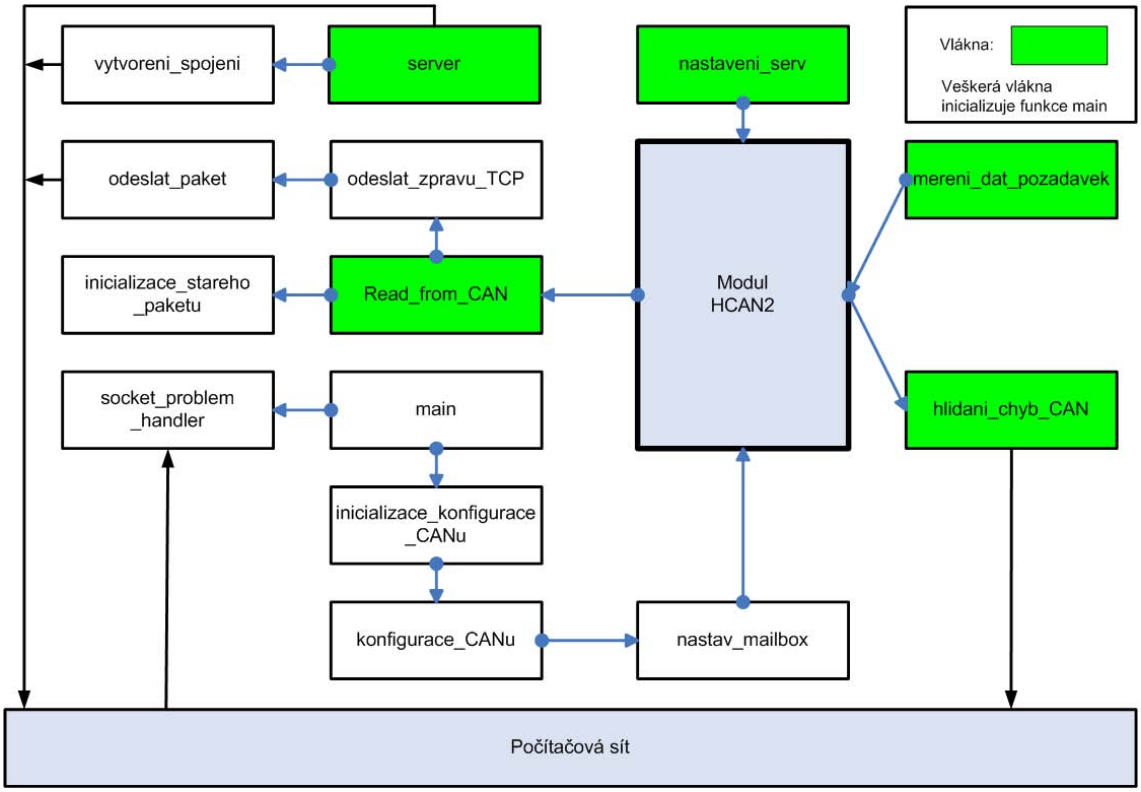
Mikropočítač EXM32 komunikuje pomocí CANu s řídící jednotkou servomotorů a inerciální navigací. Zprávy doručené po sběrnici CAN přijme modul jádra HCAN2 a uloží je v paměti. Bylo by velmi neefektivní, aby modul jakkoliv zpracovával, nebo dokonce posílal přes počítačovou sít došlé zprávy. Toto má mít za úkol aplikace v uživatelské části paměti.

V rámci diplomové práce byl napsán program, který přijme došlé zprávy po CANu, složí je do paketu a odešle po TCP/IP. Tento program se jmenuje vrtulnik a je přiložen v příloze na CD.

Program se skládá z několika vláken. Jedno z hlavních vláken se jmenuje server a má za úkol evidenci klientů, kterým se mají posílat data. Klient pouze otevře spojení a čeká na data. Pokud klient uzavře komunikaci, server to pozná pomocí funkce server. Druhé stěžejní vlákno je Read_from_CAN, které pomocí modulu HCAN2 získává informace o přijatých zprávách pomocí sběrnice CAN. Po sběrnici se posílají telemetrická data a chybové zprávy. V závislosti na významu zpráv se řadí do paketu, nebo se ihned posílají klientům. Vlákno Nastaveni_serv generuje výstupní signál pro řídicí jednotku servomotorů. V tomto vlákně lze nastavit polohu jednotlivých servomotorů a určit, která jsou v automatickém režimu řízena pomocí tohoto vlákna a která pomocí RC vysílače. Další důležité vlákno se jmenuje hlidani_chyb a získává z modulu HCAN2 chybové zprávy, například přepsání zprávy v mailboxu.

Na obrázku 8.1 je znázorněna závislost a volání jednotlivých vláken v tomto programu. Všechny vlákna jsou vytvořena ve funkci main. Pro správnou funkci programu je nutné mít zavedený modul jádra HCAN2. Podrobný popis jednotlivých funkcí je v příloze C.2.

### 8.0.1 Závislost funkcí v programu vrtulnik




Obrázek 8.1: Závislost funkcí v programu vrtulník

## 8.1 Inicializace programu

Funkce main nejdříve zaregistruje funkci pro signal SIGPIPE, potom inicializuje CAN pomocí funkce $inicializace\_konfigurace\_CANu()$;. Pak se v závislosti na konfigurací vytvoří několik vláken. Mezi nejdůležitější vlákna patří server, čtení dat z CANu, hlídání chyb pomocí ioctl, generování testovacího signálu pro servomotor. Je možnost spustit vlákno, které generuje příkaz k měření dat z ID 20. Po zapojení nové inerciální soustavy je toto vlákno nepoužíváno. Je zachováno z důvodu, kdyby bylo potřeba testovat kontrolér servomotorů.

## 8.2 TCP/IP komunikace

Cílem vlákna server je přidávat a odebírat klienty, kterým jsou zasílané poskládané pakety z CANu. Server TCP/IP je vytvořen na portu 9735. Klient otevře socketové spojení a serveru neposílá žádné data.
Toto vlákno neodesílá žádná data, jen udržuje seznam aktivních klientů.

Zprávy jsou odesílané z několika vláken.
Každá odcházející zpráva začíná znakem @ a tento znak je započten do celkové délky zprávy. Po úvodním znaku @ následují dva znaky představující ID, podle kterého se dá zjistit velikost celého paketu. Nakonec následuje datová část. Zpráva je odeslána jako string.

| **Typ zprávy** | ID | Celková velikost |
|---|---|---|
| Chyba na jednotce řízení servomotorů | 05 | 19 |
| Přepnutí režimu vrtulníku | 10 | 5 |
| Chyba na inerciální soustavě | 0f | 19 |
| Chyba na řídícím mikropočítači - CANu | 20 | 19 |
| Telemetrická data | 25 | 63 |

Tabulka 8.1: Odchozí zprávy TCP/IP

Obsah a význam datové části paketů *Chyba na řídicí jednotce servomotorů* a *Chyba na inerciální soustavě* je dán odesílající jednotkou. Program ji bezezměny ihned přepošle.
Režim vrtulníku může být buď automatický nebo manuální. Přepínání režimů má na starost řídicí jednotka servomotorů, ktera tuto skutečnost oznámí pomocí CANu. Datová část odcházející zprávy po TCP/IP je buď **00**, což znamená manuální režim a **FF** znamenající automatický režim.
Řídící počítač definuje dvě skupiny chyb, které se staly při práci CANu. Jedna skupina je typ chyb, na které reagovalo přerušení a modul ji obsloužil. Druhý typ chyb ukazuje data, která nedošla včas, více v kapitole 8.3
Chyby řídícího mikropočítače jsou uvedeny v tabulce 8.2
Velikost datové části je dána velikostí zprávy CAN.

| **Datová část** | typ chyby |
|---|---|
| 3000000000000000 | Receive Overload Warning Interrupt Flag |
| 4000000000000000 | Receive Overload Warning Interrupt Flag |
| 5000000000000000 | Error Passive Interrupt Flag |
| 6000000000000000 | Bus Off Interrupt Flag |
| 7000000000000000 | Overload Frame |
| 9000000000000000 | Message /OverRun/Overwrite Interrupt Flag |
| 0000000000000001 | Nedošla zpráva serva_poloha1 |
| 0000000000000002 | Nedošla zpráva serva_poloha2 |
| 0000000000000004 | Nedošla zpráva o napětí baterek |
| 0000000000000008 | Nedošla zpráva o uhlové_rychlosti |
| 0000000000000016 | Nedošla zpráva o zrychlení |

Tabulka 8.2: Chyby řídícího mikropočítače

Výsledná datová část zprávy z ID 25 je složena s datových zpráv CAN v tomto pořadí:
serva_poloha1, serva_poloha2, napeti_baterek, uhlove_rychlosti a zrychleni. Velikost jednotlivých bloků zprávy je popsána v následující kapitole 8.3.

## 8.3 Čtení dat z ovladače sběrnice CAN

Zprávy, které přišly po CANu, získá z modulu toto vlákno a skládá z nich pakety.
Okamžik měření dat pro jedotku řízení servomotorů a inerciální soustavu je zpráva s ID 20. V tento okamžik dojde ke změření nových hodnot a odeslání řídicímu počítači. Perioda měření dat je 32Hz.
Pokud do nového začátku měření nedojdou všechna data, řídící počítač je nahradí starými daty a navíc odešle pomocí TCP/IP chybovou zprávu, která odpovídá tabulce 8.2. Pokud po CANu dorazí chybová zpráva nebo informace o přepnutí režimu vrtulníku, je tato informace okamžitě odeslána.

| ID | **Maska serv** | | **Servo 1** | | **Servo 2** | | **Servo 3** | |
|---|---|---|---|---|---|---|---|---|
| **35** | **Maska byte** | **0 byte** | **Hi byte** | **Lo byte** | **Hi byte** | **Lo byte** | **Hi byte** | **Lo byte** |

Tabulka 8.3: Nastavení servomotorů - zpráva 35

| ID | Servo 4 | | Servo 5 | | - | | - | |
|---|---|---|---|---|---|---|---|---|
| **36** | **Hi byte** | **Lo byte** | **Hi byte** | **Lo byte** | **0** | **0** | **0** | **0** |

Tabulka 8.4: Nastavení servomotorů - zpráva 36

| ID | Úhlová rychlost x | | Úhlová rychlost y | | Úhlová rychlost z | |
|---|---|---|---|---|---|---|
| **40** | **Hi byte** | **Lo byte** | **Hi byte** | **Lo byte** | **Hi byte** | **Lo byte** |

Tabulka 8.5: Uhlové rychlosti klonění

| ID | Zrychlení x | | Zrychlení y | | Zrychlení z | |
|---|---|---|---|---|---|---|
| **40** | **Hi byte** | **Lo byte** | **Hi byte** | **Lo byte** | **Hi byte** | **Lo byte** |

Tabulka 8.6: Zrychlení ve směru

| ID | Napětí | |
|---|---|---|
| **50** | **1** | **2** |

Tabulka 8.7: Test rychlosti programu

## 8.4 Kontrola chyb modulu

Vlákno *void *hlidani_chyb_CAN( void *ptr );* komunikuje pomocí ioct s modulem a příjme od něho kódy chyb. Chyby jsou poslány pomocí TCP/IP klientům.

## 8.5 Odesílané zprávy po sběrnici CAN

Vlákno se jmenuje
*void *nastaveni_serv( void *ptr );*
Pro nastavení servomotorů slouží zprávy s ID 25 a 26. Nuly ve zprávě 26 jsou z tohot důvodu, že řídící jednotka servomotorů má výstup na 6 servomotorů.

| ID | Maska servomotorů | | Servo 1 | | Servo 2 | | Servo 3 | |
|---|---|---|---|---|---|---|---|---|
| **25** | **Maska byte** | **0 byte** | **Hi byte** | **Lo byte** | **Hi byte** | **Lo byte** | **Hi byte** | **Lo byte** |

Tabulka 8.8: Nastavení servomotorů - zpráva 25

| ID | **Servo 4** | | **Servo 5** | | **-** | | **-** | |
|---|---|---|---|---|---|---|---|---|
| **26** | **Hi byte** | **Lo byte** | **Hi byte** | **Lo byte** | **0** | **0** | **0** | **0** |

Tabulka 8.9: Nastavení servomotorů - zpráva 26

V první verzi se předpokládalo, že řídící jednotka bude posílat zprávu s ID 20 bez datové části, což je příkazem k měření dat. Po zapojení nové inerciální soustavy se změnila filozofie a tuto zprávu posílá inerciální navigace. Toto vlákno je neaktivní.

## 8.6 Režimy programu

Program umožňuje kromě běžného režimu také čtyři testovací režimy. Při spojování jednotlivých zařízení po sběrnici CAN docházelo k problémům a bylo potřeba ladit spojení zvlášť s řídicí jednotkou servomotorů, zvlášť s inerciální soustavou a testovat vizualizační software na pozemní stanici. Režim se nastaví v headru vrtulnik.h pomocí proměnné STATUS_PRIPOJENYCH_ZARIZENI .

| **Číslo režimu** | popis |
|---|---|
| 1 | Test komunikace se řídicí jednotkou servomotorů |
| 2 | Test komunikace s inerciální soustavou |
| 3 | Běžný provoz |
| 4 | Kalibrace TCP/IP klienta grafické vizualizace |

Tabulka 8.10: Režimy programu

# Kapitola 9

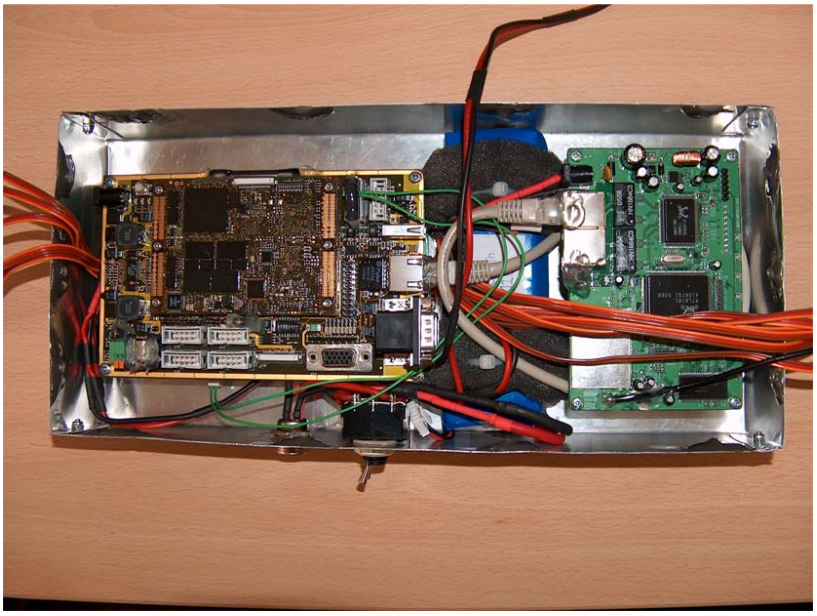
# Ověření funkce celého systému

Celý projekt kladl velký důraz na bezpečnost a spolehlivost každé části distribuovaného řídicího systému modelu vrtulníku.

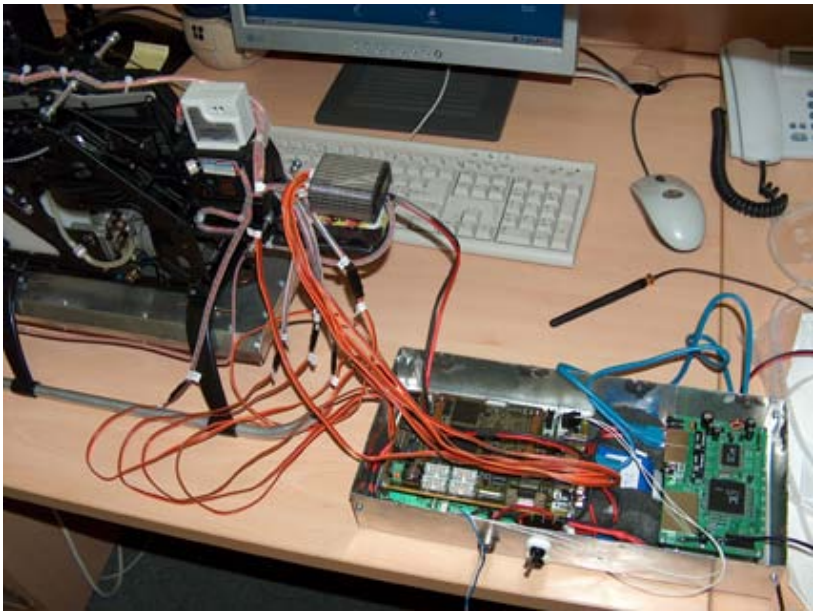
Při letu modelu vznikají silné vibrace a bylo třeba spojit jednotlivé části systému tak, aby nemohlo dojít k rozpojení kabelů. Byl sestaven speciální box, do kterého se zabudovali jednotlivé části distribuovaného řídicího systému. Box je na obrázku 9.1. Jednotka řízení servomotorů se nachází pod hlavním řídicím počítačem, který je vlevo. Vpravo je WiFi access point.

Před prvním letem modelu byly provedeny důkladné zkoušky celého systému. Zkoušky proběhly formou dlouhodobého testování reakcí jednotek a měřením telemetrických dat. V první etapě se testovaly samostatné jednotky a postupně byly odstraněny všechny chyby. V druhé etapě se testoval cělý systém současně. Hlavní řídicí počítač generoval zkušební signál pro jeden servomotor. Ostatní servomotory byly ovládané pomocí RC soupravy. Při testech byla telemetrická data přenášena pomocí bezdrátové počítačové sítě. Pokud by selhala jednotka řízení servomotorů, nebylo by možné nastavit polohu pro servomotory, které by nereagovaly na signál. V případě poruchy hlavního řídicího mikropočítače by se zastavil pohyb jednoho servomotoru a pozemní stanice by nedostávala žádné telemetrické data. Žádná z poruch nenastala a všechny prvky celého systému fungovaly bezchybně. Tím byla ověřena funkce celého řídicího systému, včetně výsledků této diplomové práce. Pokud by se vyskytla jakákoliv chyba nesměl by model vzlétnout.

Po úspěšných testech proběhlo několik zkušebních letů. Ani při těchto letech nebyla jištěna žádná chyba celého systému. Na obrázku 9.5 jsou naměřená telemetrická data jednoho letu.



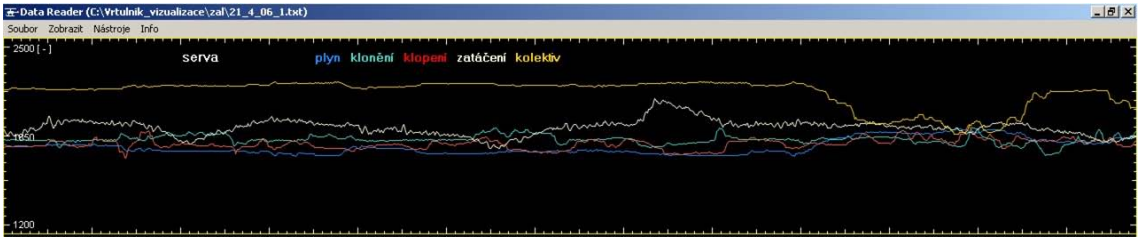
Obrázek 9.1: Úložný box pro jednotky v modelu vrtulníku



Obrázek 9.2: Testování celého řídicího distribuovaného systému

Obrázek 9.3: Zkušební let

Obrázek 9.4: Zkušební let




Obrázek 9.5: Vizualizace naměřených telemetrických dat

# Kapitola 10

# Závěr

Cílem této práce bylo vytvořit software pro komunikaci hlavního řídicího mikropočítače s jednotkou řízení servomotorů a inerciální navigací pomocí sběrnice CAN. Přijatá data mikropočítač odesílá, pomocí bezdrátové počítačové sítě, pozemní stanici.

Aby bylo možné vytvořit požadovaný software, bylo prvním úkolem zprovoznit na mikropočítači operační systém Linux. K tomu byl použit upravený software, který původně vytvořil Ing. Miroslav Žižka. Ve druhém úkolu této práce byl vytvořen ovladač sběrnice CAN v podobě modulu Linuxového jádra. K tomuto ovladači byla naprogramována aplikace, která odesílá telemetrická data a chyby systému pomocí počítačové sítě.

Před zkušebním letem modelu vrtulníku byl výsledný distribuovaný řídicí systém, včetně této práce, důkladně testován. Při těchto testech nebyly zjištěny žádné závažné chyby v celém řídicím systému. Po testech proběhlo několik zkušebních letů. Během letu byla na pozemní stanici vykreslována telemetrická data pomocí vizualizačního softwaru. Díky zkušebním letům byl celý řídicí systém vyzkoušen v reálných podmínkách a byla změřena skutečná telemetrická data. Tím byla také ověřena funkčnost této diplomové práce.

# Literatura


[1] Stones, S., Matthew, N. *Linux začínáme programovat.* Praha : Computer Press, 2000. 898 s. ISBN 80-7226-307-2.

[2] Corbet, J., Rubini, A., Kroah-Hartman, G. *Linux device drivers, third edition* O'Reilly, 2005. 616 s. ISBN 0-596-00590-3.

[3] Yaghmour, K. *Building Embedded Linux Systems* O'Reilly, 2003. 416 s. ISBN 0-596-00222-X.

[4] Herout, P. *Učebnice jazka C, III. upravené vydání* České Budějovice : KOPP, 1994. 272 s. ISBN 80-85828-21-9.

[5] *Výrobce ineriální navigace MICRO-ISU BP3010* [online]. <http://www.bec-nav.de/>

[6] Herm, O. *Návrh a realizace řídicí jednotky servomotorů pro model vrtulníku* Praha : České Vysoké Učení Technické, Fakulta elektrotechnická, Katedra řidicí techniky, 2006.

[7] *Hitachi SuperH RISC engine SH7760 Hardware Manual* Firemní literatura Hitachi, 2003-02-28. 1345 s.

[8] *MSC EXM32 Starter kit, HW Getting Started* Firemní literatura MSC, Revision 0.1. 2005 17 s.

[9] *MSC EXM32-MB_LITE User's Manual* Firemní literatura MSC, Revision 0.1. 2005. 48 s.

[10] *MSC EXM32-SH7760 CPU Module User's Manual* Firemní literatura MSC, Revision 0.3. 2005. 94 s.

[11] Klement, D. *Řízení pohonu řezacího stroje pro tvarové dělení materiál* Praha : České Vysoké Učení Technické, Fakulta elektrotechnická, Katedra řidicí techniky, 2005.

[12] Ditrich, M. *Řešení komunikace mezi systémem NX5030 firmy INTRONIX a sériově vyráběnými automaty PLC.* Praha : České Vysoké Učení Technické, Fakulta elektrotechnická, Katedra řidicí techniky, 2004.

[13] Vacek, F., Sojka, M. *Počítačové systémy.* [online]. Poslední revize 2006-03-01 [cit. 22. 5. 2006]. <http://dce.felk.cvut.cz/pos>

[14] Hanzálek, Z. *Distribuované řídicí systémy - přednášky.* [online]. [cit. 22. 5. 2006]. <http://dce.felk.cvut.cz/hanzalek/prednasky/>

[15] Salzman, P. J., Burian, M., Pomerantz, O. *The Linux Kernel Module Programming Guide* [online]. Poslední revize 2005-12-31 ver 2.6.3 [cit. 22. 5. 2006]. <http://www.tldp.org/LDP/lkmpg/2.6/html/>

[16] Spurný, F. *Úvod do problematiky sběrnice CAN* [online]. Poslední revize 2005-12-31 ver 2.6.3 [cit. 22. 5. 2006]. <http://fieldbus.feld.cvut.cz/can/>

[17] *wikipedia* [online]. <http://www.wikipedia.org/>

[18] *Projekt GNU* [online]. Poslední revize 2006-05-11 <http://www.gnu.org/>

[19] *Projekt GNU* [online]. Poslední revize 2006 <http://www.linux.org/>

[20] *API changes in the 2.6 kernel series* [online]. Poslední revize 2006-5-11 [cit. 22. 5. 2006]. <http://lwn.net/Articles/2.6-kernel-api/>

[21] *Domovské stránky projektu SHLinux* [online]. [cit. 22. 5. 2006]. <http://www.sh-linux.org/>

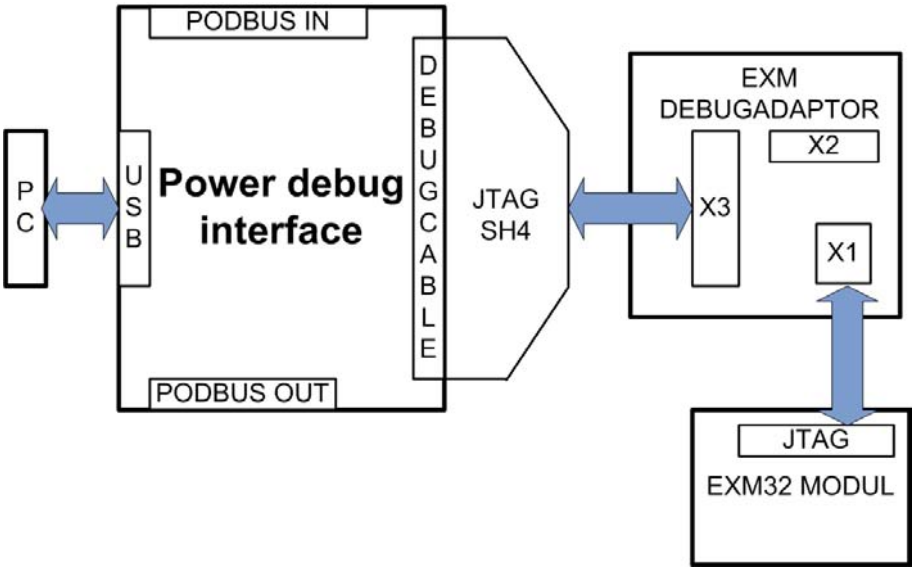
# Dodatek A

# Zprovoznění mikropočítače EXM32

V této kapitole jsou popsané postupy instalace software a nahrání OS Linux na mikropočítač EXM32.

## A.1 Nahrání IPL na mikropočítač EXM32

Mikropočítač EXM32 je standardně dodáván s operačním systémem QNX nebo MS Windows CE. Pro provozování OS Linux na mikropočítači je nutné vytvořit boot CompactFlash se systémem Linux a nahrát do něj IPL. K nahrání IPL je potřeba JTAG debugger a Power debug interface, adaptér EXM_DEBUGADAPTOR a program Trace32.




Obrázek A.1: Zapojení JTAG s mikropočítačem EXM32

Na obrázku A.1 je schéma zapojení. JTAG se zapojuje do EXM_DEBUGADAPTOR na port X3 a deska EXM32 je zapojena na port X1.

Program Trace32 je dodáván jak pro Linux, tak pro MS Windows. Po instalaci programu Trace32 je nutné program správně nakonfigurovat pro danou architekturu, procesor a připojení přes USB kabel.

Je potřeba do mikropočítače EXM32 nahrát IPL, které se nachází na přiloženém CD v adresáři IPL. Program IPL pro mikropočítač EXM32 vytvořil ing. Miroslav Žižka. V programu Trace32 v menu File dáme volbu Open BatchFile a otevřeme soubor EXM32-TRACE32.CMM. Soubor obsahuje informace o typu procesoru, vymazání paměti na desce EXM32 a také příkaz, pomocí kterého se nahraje soubor na desku EXM32. Nejdůležitější informace se nacházejí v druhé půlce BatchFile.

Mezi nejdůležitější nastavení patří:

```
MAP.BOnchip 0x0++0x01ffffff      ; Specifikuje, kde je FLASH/ROM
SYStem.CPU SH7760                 ; Typ procesoru
SYSTEM.JTAGCLOCK 20.0Mhz         ; Rychlost JTAGCLOCK
system.OPTION SLOWRESET ON
system.OPTION LITTLEEND ON        ; Procesor je litlle endian
SYStem.Up
```

Následuje sekce příkazů, která odemkne a vymaže FLASH, aktivuje nahrávání

nového programu a zkontroluje nahrávání. Smazání paměti trvá delší dobu.

```
FLASH.RESet
FLASH.CREATE D:0x0--0x01ffffff 20000 I28F200J3 Word
flash.UNLOCK ALL
FLASH.Erase ALL

FLASH.Program ALL

DATA.LOAD.BINARY C:\sh-stub.bin D:0x0 /VERIFY
; umístění programu, který se nahraje do FLASH na desce EXM32

FLASH.Program  ;nahraje program
```

Program sh-stub.bin vznikne překladem IPL, který je na přiloženém CD. Původní program vytvořil ing. Miroslav Žižka. Program IPL byl upraven, aby se automaticky spustil OS Linux, není-li ani jeden přepínač na EXM32 ve stavu sepnuto. Pokud by byl libovolný přepínač sepnut, program IPL se zeptá, jestli má spustit Linux, nebo debuggovací režim.

Druhou možností je nahrání IPL pomocí Trace32 ručním nastavením parametrů. Parametry jsou totožné s batchfile.

Pokud se podařilo úspěšně nahrát IPL do FLASH na mikropočítači je třeba nahrát na CompactFlash Linux.

## A.2 Tvorba CompactFlash disku s distribucí Linux

1. Vytvoří se filesystém na CampactFlash ( CF ) disku.

2. Naformátuje se CF

   ```
   # MKFS.EXT3 /dev/sda1
   ```

3. Připojíme CF

   ```
   # mount /dev/sda1 /mnt/cf
   ```

4. Nakopíruje se sh-linux na CF. Poznámka: Kopírovat MUSÍME pod právy ROOT a včetně linků.

   ```
   # cp /home/jirka/sh-linux /mnt/cf -l.
   ```

5. Zkopíruje se přeložené jádro

```
# cp /home/jirka/kernel/arch/sh/boot/zImage /mnt/cf/boot
```

6. Vytvoří se LILO[1]

```
# ./lilo -r /mnt/cf -C /lilo.conf
```

7. Odpojí se CF

```
# umount /mnt/cf
```

Nyní se může CF vložit do mikropočítače a spustit. Pokud vše proběhlo správně měl by se spustit Linux.

## A.3 Instalace balíčků na PC

Aby se mohli vyvíjet aplikace, překládat jádro Linuxu pro použitý procesor, je nutné nainstalovat ve správném pořadí některé balíčky.

1. Binutils

```
# rpm -i binutils-sh-linux-XXXXXXX-1.i386.rpm
```

2. gcc

```
# rpm -i gcc-sh-linux- XXXXXXX .i386.rpm
```

3. glibc

```
# rpm -i glibc-sh-linux-XXXXXXX.noarch.rpm
# rpm -i glibc-sh4-linux-XXXXXXX.noarch.rpm
```

4. gdb

```
# rpm -i gdb-sh-linux- XXXXXXX .i386.rpm
```

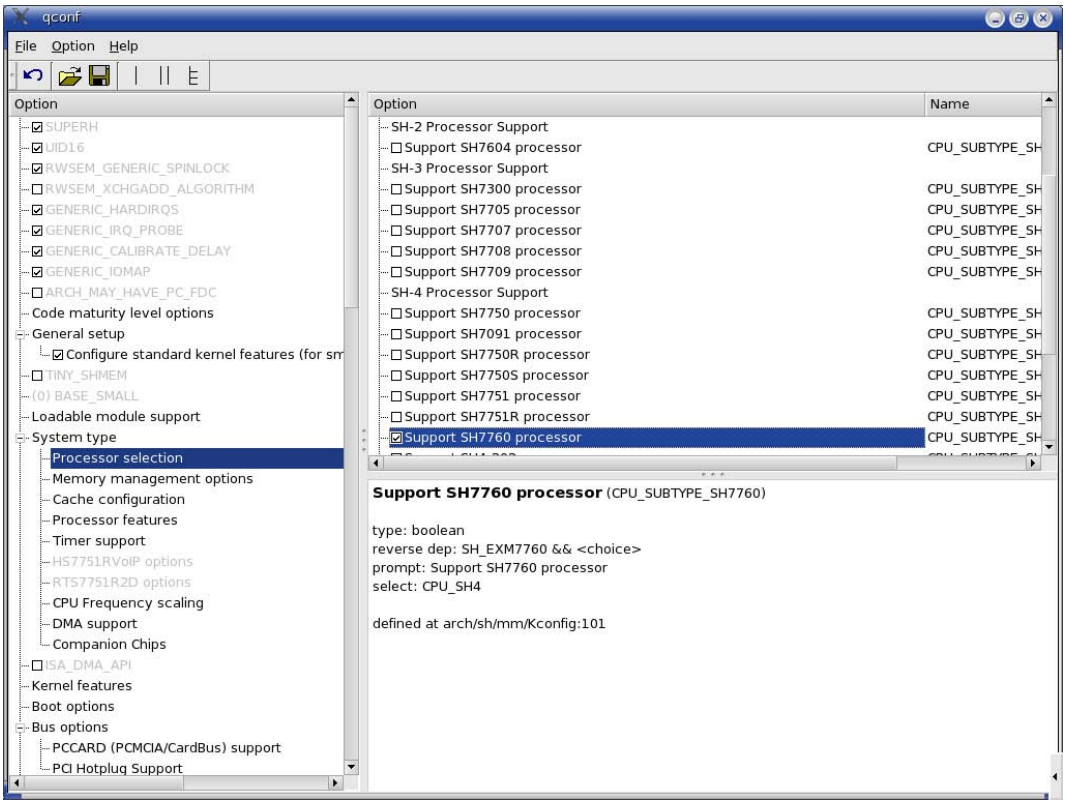
[1]Program lilo musí být ve verzi 21.7.3, protože konfigurace desky očekává tuto verzi. Pokud by bylo nutné použít novější verzi lilo, musí se crosscompilovat a nakopírovat do adresáře /boot na CF. Program lilo 21.7.3 je na přiloženém CD.

Nyní se může přeložit program nebo jádro Linuxu. Program se překládá pro použitou platformu tak, že místo gcc je použit sh4-linux-gcc. Příklad překladu:

```
# sh4-linux-gcc hello_world.c -o hello_world
```

Překlad jádra Linuxu

Někdy je potřeba změnit parametry jádra a opět přeložit jádro. Konfigurační skript se volá pomocí *make xconfig*. Je možné použít *make menuconfig* nebo*make config*. Rozdíly jsou v prostředí konfigurace.




Obrázek A.2: Překlad jádra OS Linux

Příklad překladu jádra

```
# cd kernel_2.6.14
# make xconfig
# make
```

Výsledek překladu je soubor zImage, který je v adresáři <adresář s jádrem>/arch/sh/boot. Soubor zImage je komprimované jádro Linuxu pro použitou platformu.

## A.4 Přenos souborů na mikropočítač EXM32

Aplikace nebo moduly lze na mikropočítač nahrát několika způsoby. Pokud byl správně zprovozněn Linux na mikropočítači a počítače jsou připojeny do počítačové sítě, je na mikropočítači spuštěna Samba a SSH. V terminálu pomocí *ifconfig* se zjistí IP adresa . Pomocí programu SAMBA lze se připojit CompactFlash kartu jako sítový disk. Například v aplikaci Konqueror se zadá *smb://192.168.136.151/homes*. Další možností je použít SSH. Pro kopírování souborů existuje příkaz scp.

```
# scp program root@192.168.136.151:/tmp
```

Po tomto příkazu, pokud nejsou vyměněné klíče SSH, může být uživatel požádán o zadání hesla. SSH se dá samozřejmě použít i k vytvoření virtuální konzole na mikropočítači.

```
# ssh  root@192.168.136.151
```

Je možné nechat si přidělit IP adresu mikropočítače pomocí DHCP serveru nebo nastavit pevnou IP adresu. Specifikace zadání této práce požaduje, aby mikropočítač zapojený v modelu vrtulníku měl IP adresu 192.168.100.1. Při vývoji aplikací byl vrtulník zapojen v počítačové síti, kde IP adresy přiděloval DHCP server. Pro nastavení IP adresy slouží konfigurační soubor /etc/sysconfig/networking/ifcfg-eth0. Jednotlivé položky jsou zřejmé z jejich názvu.

```
DEVICE=eth0
BOOTPROTO=dhcp  #Pokud se zakomentuje tento řádek,
#přidelí se pevná IP adresa,
#která je nastavena v tomto souboru.
ONBOOT=yes
BROADCAST=192.168.100.255
NETWORK=192.168.100.0
NETMASK=255.255.255.0
IPADDR=192.168.100.1
USERCTL=no
```

## A.5 Instalace debugeru INSIGHT

Přeloklad programu pro platformu PC.

```
# mkdir pc
# cd pc
# ../configure
# ..make
# su
# make install
```

Pro platformu sh4 se překlad provede následovně.

```
# mkdir sh4
# cd sh4
# ../configure --program-prefix=sh --target=sh
# make
# su
# make install
```

# Dodatek B

# Jednoduchý modul zařízení

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>


int zarizeni_init_module(void);
void zarizeni_cleanup_module(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t
    *);
static ssize_t device_write(struct file *, const char *, size_t,
    loff_t *);

#define DEVICE_NAME "modul_zarizeni"

static int Major;
static int Device_Open = 0;

static struct file_operations fops = {
  .read = device_read,
  .write = device_write,
  .open = device_open,
  .release = device_release
};

/*
 * Tato funkce je volána, když se modul inicializuje
 */
int zarizeni_init_module(void)
{
        Major = register_chrdev(120, DEVICE_NAME, &fops);
```

```
  if (Major < 0) {
    printk(KERN_ALERT "Nastala chyba pri registraci zarizeni %d\
        n", Major);
    return Major;
  }

  return 0;
}

/*
 * Tato funkce je volána, kdyz se modul ukoneuje
 */
void zarizeni_cleanup_module(void)
{

  int ret = unregister_chrdev(120, DEVICE_NAME);
  if (ret < 0)
    printk(KERN_ALERT "Doslo k chybe pri ukonceni modulu: %d\n,
        asi je zarizeni stale pouzivano", ret);
}


/*
 * Tato funkce se vola, kdyz uzivatelska aplikace otvira
     zarizeni
 */
static int device_open(struct inode *inode, struct file *file)
{
  printk(KERN_INFO "Doslo k otevreni zarizeni\n");

  if (Device_Open)
    return -EBUSY;

  Device_Open++;

  try_module_get(THIS_MODULE);

  return 0;
}

/*
 * Tato funkce se vola, kdyz uzivatelska aplikace zavira
     zarizeni
 */
static int device_release(struct inode *inode, struct file *file
    )
{
```

```
  Device_Open--;      /* Pocet pripojenych zarizeni zmensim o
      jeden */
  module_put(THIS_MODULE);

  return 0;
}


static ssize_t device_read(struct file *filp, char *buffer,
    size_t length, loff_t * offset)
{
  printk(KERN_INFO "Funkce cteni dat neni implementovana.\n");
  return 0;

}


static ssize_t device_write(struct file *filp, const char *
    buffer, size_t len, loff_t * off)
{
  printk(KERN_INFO "Funkce zapis dat neni implementovana.\n");
  return 0;
}

MODULE_LICENSE( "GPL" );
module_init( zarizeni_init_module );
module_exit( zarizeni_cleanup_module );
```

Výpis kódu B.1: modul - zařízení

# Dodatek C

# Popis vytvořeného softwaru

## C.1 Popis funkcí modulu

### C.1.1 CAN_Data_Frame_Received

```
void CAN_Data_Frame_Received();
```

Tato funkce je vyvolána handlerem přerušení, zjistí, do kterého mailboxu přišla data a vyvolá funkci

```
void CAN_Data_Frame_Received_Read( int cislo_dosleho_mailboxu );
```

### C.1.2 CAN_Data_Frame_Received_Read

```
void CAN_Data_Frame_Received_Read (int mail);
```

Přečte došlou zprávu a uloží do bufferu. Probudí funkci na odeslání dat aplikaci

```
static ssize_t device_read( struct file *filp,
char *buffer, size_t length, loff_t * offset );
```

Vstupním parametrem je číslo mailboxu, kam došla zpráva.

### C.1.3 CAN_Frame_Request_Received

```
void CAN_Frame_Request_Received();
```

Funkce je vyvolána handlerem přerušení, zjistí, do kterého mailboxu přišla data a vyvolá funkci

```
void CAN_Data_Frame_Received_Read( int cislo_dosleho_mailboxu );
```

### C.1.4 CAN_Init

```
void CAN_Init( int BCR0, int BCR1 );
```

Ioctl vyvolá tuto funkci pro nastavení přenosové rychlosti a inicializaci CANu. Funkce smaže stará data ve všech mailboxech.

### C.1.5 CAN_Transmit_odeslano

```
void CAN_Transmit_odeslano();
```

Úspěšné odeslání dat je potvrzeno přerušením. Tato funkce je volaná obslužnou funkcí přerušení.

### C.1.6 CAN_Transmit_odeslat

```
void CAN_Transmit_odeslat( struct canmsg_t sendmsg );
```

Uloží do mailboxu data a připraví je k odeslání.

### C.1.7 device_ioctl

```
int device_ioctl( struct inode *inode, struct file *file,
unsigned int ioctl_num, unsigned long ioctl_param );
```

Při volání ioctl je volána tato funkce. Jsou zde reakce na jednotlivé ioctl, které byly popsány v kapitole 7.2. Návratová hodnota odpovídá jednotlivým ioctl.

### C.1.8 device_read

```
static ssize_t device_read( struct file *filp, char *buffer,
 size_t length, loff_t * offset );
```

Jestliže uživatelská aplikace chce číst data, je volána tato funkce. Tato funkce je blokovací a návratová hodnota je velikost odesílaných dat aplikaci.

### C.1.9 device_write

```
static ssize_t device_write( struct file *filp, const char *buffer,
 size_t len, loff_t * off );
```

Chce-li poslat uživatelský program data po CANu, je vyvolána tato funkce v modulu. Návratová hodnota je velikost přijatých dat.

### C.1.10 HCAN2_modul_cleanup

```
void HCAN2_modul_cleanup();
```

Ukončení modulu.

### C.1.11 HCAN2_modul_init

```
int HCAN2\_modul\_init();
```

Inicializace modulu.

### C.1.12 irq_handler_HCAN2

```
irqreturn_t irq_handler_HCAN2( int irq, void *dev_id,
struct pt_regs *regs );
```

Handler pro všechna přerušení přerušení CANu. Volá obslužné funkce pro dané přerušení.

### C.1.13 Reset_CANu

```
void Reset_CANu();
```

Vyvolá se reset řadiče CANu.

### C.1.14 Reset_Halt_Sleep_Interrupt_CANu

```
void Reset\_Halt_Sleep_Interrupt_CANu( );
```

Je-li vyvoláno přerušení, značící reset, zastavení nebo režim sleep, obslužná funkce přerušení vyvolá tuto funkci.

## C.2 Popis programu vrtulník

### C.2.1 hlidani_chyb

```
void *hlidani_chyb_CAN( void *ptr );
```

Pokud nastanou chyby v modulu jádra nebo na sběrnici, aplikace se to dozví díky tomuto vláknu.

### C.2.2 inicializace_konfigurace_CANu

```
int inicializace_konfigurace_CANu();
```

Nastaví přenosovou rychlost CANu.

### C.2.3 konfigurace_CANu

```
void konfigurace_CANu ( int fd, int cislo_CANu);
```

Tato funkce nastaví mailboxy na příjem nebo vysílání.

### C.2.4 main

```
int main();
```

Vyvolá funkci pro nastavení CANu a vytvoří vlákna.

### C.2.5 mereni_dat_pozadavek

```
void *mereni_dat_pozadavek(void *arg);
```

Toto vlákno posílá v pravidelném časovém intervalu zprávu po CANU s ID 20. V poslední verzi není voláno.

### C.2.6 nastav_mailbox

```
void nastav_mailbox ( int fd, int cislo_mailboxu, int MBC,
 int id, int lafm, int cislo_CANu );
```

Jeden mailbox pomocí funkce ioctl se nastaví pomocí této funkce.

### C.2.7 nastaveni_serv

```
void *nastaveni_serv( void *ptr );
```

Generuje signál typu pila a posílá ji jednotce serv.

### C.2.8 odeslat_zpravu_TCP

```
void odeslat_zpravu_TCP(  struct canmsg_t msg,
 struct zprava_pro_server *paket,
 struct zprava_pro_server *stary_paket, int cislo_CANu);
```

Vytvoří a odešle pakety TCP/IP ze zpráv, které došly. Generuje chyby, pokud nedošla včas požadovaná zpráva.

### C.2.9 read_from_CAN

```
void *read_from_CAN( void *ptr );
```

Vlákno čte data z CANu 0 nebo 1.

### C.2.10 server

```
void *server( void *ptr )
```

Vlákno vytvoří server a aktualizuje seznam připojených klientů.

### C.2.11 socket_problem_handler

```
void socket_problem_handler(int sig);
```

Obsluha signálu v BROKENPIPE.

### C.2.12 vytvoreni_spojeni

```
int vytvoreni_spojeni( int port);
```

Vytvoří socketové spojení.

### C.2.13 zprava_pro_server

```
void inicializace_stareho_paketu ( struct zprava_pro_server *stary_paket );
```

Nastaví hodnoty packetu na 0, aby když nebudou přijata žádná data, existovala nulová data.

# Dodatek D

# Obsah přiloženého CD

Přiložené CD obsahuje následující adresáře:

- **DP** Obsahuje kopii ve formátu pdf této diplomové práce.
- **HCAN2** V tomto adresáři je software vytvořený v rámci této diplomové práce.
- **Insight** Obsahuje ladící software.
- **IPL** Obsahuje program Initial Program Load, který se nahrává do mikropočítače.
- **Kernel** Obsahuje Linuxové jádro nastavené pro mikropočítač EXM32.
- **LILO** Program pro vytvoření ComactFalsh disku, z kterého je možno bootovat Linux
- **SHLinux_CF** V tomto adresáři je kopie distribuce SHLinux pro účely vytvoření ComactFalsh disku. Tato kopi již nakonfigurována pro EXM32.