České vysoké učení technické v Praze
Fakulta elektrotechnická - Katedra řídící techniky

Bakalářská práce

Platforma pro simulaci hardwaru ve smyčce

Lukáš Hamáček

2007

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

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 10.8.2007 Lukáš Hamáček

Tímto bych rád poděkoval ing. Liboru Waszniowskému za odborné vedení a pomoc při vypracování bakalářské práce.

České vysoké učení technické v Praze, fakulta elektrotechnická

Katedra řídicí techniky | Školní rok: 2006/2007

Zadání bakalářské práce

Student: Lukáš Hamáček

Obor: Kybernetika a měření

Název tématu: Platforma pro simulaci s hardware ve smyčce

Zásady pro vypracování:

1. Seznamte se s principy hardware in the loop simulace.
2. Navrhněte a implementujte podporu simulace linearniho systemu v reálném čase, na vestavěném počítači a s podporou komunikace CANopen.
3. Demonstrujte použití na vhodné aplikaci

Seznam odborné literatury: Dodá vedoucí práce

Vedoucí bakalářské práce: Ing. Libor Waszniowski

Datum zadání bakalářské práce: zimní semestr 2006/07

Termín odevzdání bakalářské práce: 15. 8. 2007

Prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

v zast.

Prof. Ing. Zbyněk Škvor, CSc.
děkan

V Praze, dne 28. 2. 2007

Anotace

Tento dokument popisuje návrh platformy pro simulaci hardwaru v uzavřené smyčce. Jedná se o systém, který používá jednodeskový počítač BOA5200 pro vykonávání vyvinutého programu simulátoru. Program na počítači běží pod operačním systémem Linux. Komunikace s operátorskou aplikací je zprostředkována pomocí spojení TCP. S testovaným zařízením si simulátor vyměňuje data přes sběrnici CAN s protokolem CANopen.

Implementovaná platforma je v současnosti používána pro testování řídícího systému tlumiče stranových kmitů letounu ve firmě Aero Vodochody a.s. Zde je nasazena zjednodušená modifikace univerzální platformy, která usnadňuje uživateli konfiguraci a práci se simulátorem.

Annotation

This document describes design of a platform for Hardware-In-the-Loop simulation. This system is based on industrial computer BOA5200 running operating system Linux and the developed simulator software. The simulator is controlled by an operator application via TCP connection. The data exchange with tested hardware is performed via CANopen protocol and industrial bus CAN.

The implemented platform is nowadays used for testing of the side vibration damper control system of planes in Aero Vodochody a.s company. This is a simplified modification of the universal platform which makes the configuration and usage of the simulator easier.

Obsah

1 Úvod ... 1

1.1 Princip simulace hardwaru ve smyčce ... 1

1.2 Platforma pro simulaci HW ve smyčce ... 1

2 Použité technologie ... 3

2.1 Jednodeskový počítač BOA5200 ... 3

2.1.1 Parametry základní desky ... 3

2.1.2 Periferie ... 3

2.1.3 Parametry MPC5200 ... 3

2.2 Operační systém Linux ... 3

2.3 Sběrnice CAN ... 4

2.3.1 Ovladač SocketCan ... 4

2.4 Protokol CANopen ... 4

2.4.1 *Process Data Object* (PDO) ... 5

2.4.2 Service Data Object (SDO) ... 5

2.4.3 *Network Management* (NMT) ... 5

2.4.4 *Synchronization Object* (SYNC) ... 5

2.4.5 *Emergency Object* (EMCY) ... 6

2.4.6 Ovladač CanFestival ... 6

3 Program LTI simulátoru ... 7

3.1 Architektura programu ... 7

3.1.1 Vícevláknový program ... 7

3.1.2 Stavy systému ... 8

3.2 Pomocné části programu ... 8

3.2.1 Reprezentace matic ... 8

3.2.2 Zásobníky ... 9

3.3 LTI model ... 9

3.4 Časování simulace ... 9

3.5 Vstupní a výstupní vektory ... 10

3.6 Vstupní a výstupní zařízení ... 11

3.7 TCP server ... 11

3.8 *Target-Host* (TH) protokol ... 12

3.8.1 *State message* (stavová zpráva) ... 12

3.8.2 Reset konfigurace ... 13

3.8.3 Autodetekce *endianity* ... 13

3.9 Monitory ... 13

3.10 *Direct vector* (Ud) ... 13

3.11 Logování událostí ... 14

3.11.1 Úrovně logování ... 14

3.12 Zdrojové kódy a překlad programu ... 14

3.13 Dokumentace ... 15

4 Závěr ... 16

5 Literatura a odkazy ... 17

Příloha A - Modifikace pro Aero Vodochody **18**
A.1 Vstupní a výstupní zařízení 18
A.2 Mapování vstupů a výstupů 18
Příloha B - How to build LTI simulator **20**
Příloha C - How to download software to BOA **21**
C.1 Building Linux kernel and file system 21
C.2 Setting up the Redboot 21
C.3 Downloading Linux to BOA 22
Příloha D - How to use CanFestival **23**
D.1 Obtaining and building CanFestival 23
D.2 Setting up local CANopen node 23
D.3 Initializing and starting CanFestival 23
D.4 Stopping CanFestival 24
D.5 Writing to Object dictionary 24
D.5.1 Writing to local OD 24
D.5.2 Writing to network OD 25
D.6 Using PDO service 26
D.7 Generating SYNC message 26
D.8 SDO and PDO example 27
D.9 Emergency message 28
Příloha E - TH protocol example **29**
E.1 Simulation task 29
E.2 System configuration 30
E.3 Running simulation 32
Příloha F - Obsah přiloženého CD **35**

1 Úvod

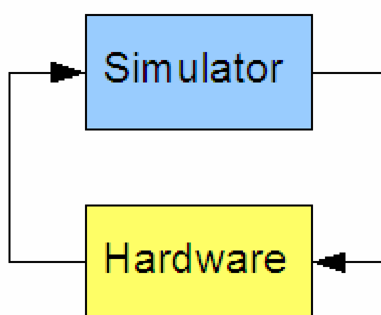
Cílem této práce je navrhnout univerzální platformu pro realizaci simulace hardwaru ve smyčce. Důraz bude kladen na to, aby jednotlivé části systému mohly být využity i v jiných projektech na katedře řídící techniky. Například pro řídící počítač vybereme takový hardware, který bude splňovat požadavky většiny projektů. Pro tento hardware potom budou tvořit pracovníci katedry sadu nástrojů pro překlad operačního systému a jiného softwaru, ovladače a návody. Tím vznikne univerzální platforma, podporující komunikaci pomocí protokolu CANopen, která bude moct být nasazena ve většině vyvíjených řídících systémech.

V první části tohoto dokumentu jsou popsány použité technologie, jejich vlastnosti, výhody a nevýhody a také důvod, proč jsme konkrétní produkty vybrali pro náš projekt. Dále se dokument věnuje především popisu softwaru LTI simulátoru a jednotlivých jeho částí. Cílem je vysvětlit především myšlenky použité při vývoji programu. Větší rozsah než samotný dokument mají přílohy, které obsahují návody k používání simulátoru, jeho překladu a nahrání do paměti počítače. Jedna kapitola je také věnována vysvětlení způsobu řízení simulátoru pomocí speciálního protokolu. Většina příloh je psaná anglicky a to z toho důvodu, že i programová dokumentace je v angličtině a tyto návody budou jejím doplňkem.

Samostatná kapitola je věnována používání ovladače CanFestival pro síť CANopen. Tento ovladač byl totiž vybrán jako hlavní ovladač pro CANopen, který bude používaný na katedře. Je tedy nutné vytvořit kvalitní podporu a usnadnit tak ostatním jeho použití.

1.1 Princip simulace hardwaru ve smyčce

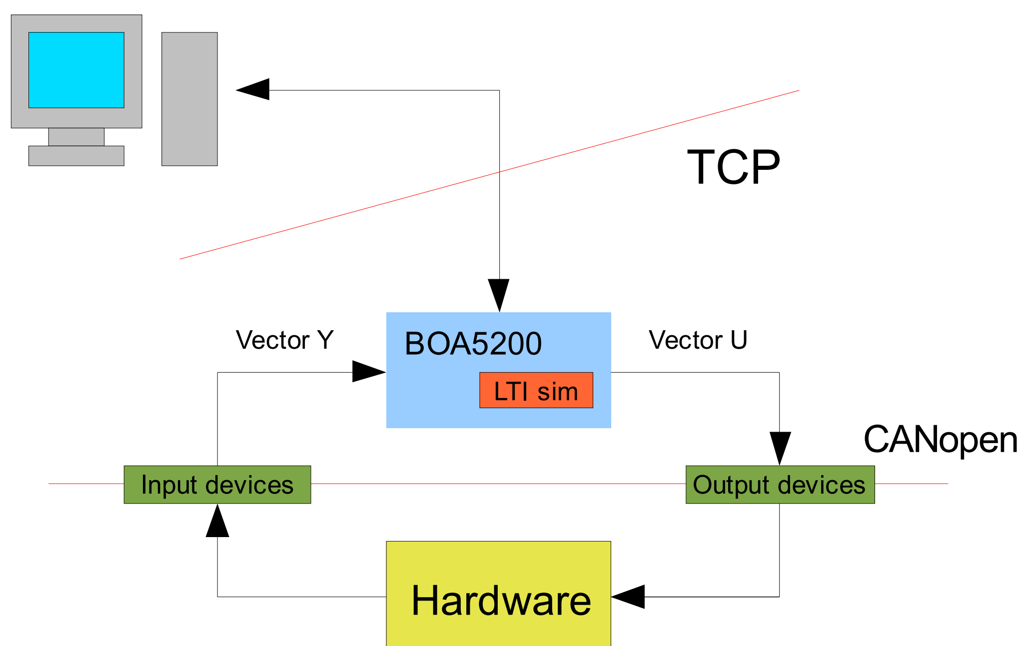
Metoda simulace hardwaru ve smyčce (anglicky *Hardware-In-the-Loop simulation*) [1] se používá pro testování libovolného řídícího systému. Místo toho, abychom připojili vyvíjený řídící systém k reálné soustavě, k jejímuž řízení je navrhován, vytvoříme matematický model této soustavy a k regulátoru připojíme pouze simulátor provádějící tento model (viz. Obrázek 1). Na tomto způsobu testování hardwaru je výhodné zejména to, že při testech nepoužíváme reálnou soustavu a nemůžeme jí tedy poškodit. Další velkou výhodou jsou výrazně širší možnosti monitorování a analýzy průběhu simulace. Navíc je možné simulovat i stavy nedosažitelné v reálné soustavě a testování lze provádět ještě před jejím dokončením.



Obrázek 1 - Simulace hardwaru ve smyčce

1.2 Platforma pro simulaci HW ve smyčce

Naším cílem je vytvořit univerzální platformu pro simulaci hardwaru ve smyčce, která bude ovládána pomocí aplikace s grafickým uživatelským rozhranním. Samotná simulace bude prováděna v simulačním počítači, který bude připojen k operátorskému počítači přes místní síť pomocí protokolu TCP. Vstupy a výstupy budou tvořit distribuovaná zařízení připojená k simulačnímu počítači přes průmyslovou sběrnici CAN a komunikující pomocí protokolu vyšší úrovně CANopen (viz. Obrázek 2).



Obrázek 2 - Platforma pro simulaci hardwaru ve smyčce

Jako příklad konkrétní aplikace platformy uvedu simulátor, který dodáváme do společnosti Aero Vodochody. Pro testování funkčnosti autopilota je vytvářen simulátor, který simuluje reakci letadla na natočení směrového kormidla. Vstupem soustavy je tedy úhel natočení kormidla měřený senzorem úhlového natočení Sendix 5858 připojeným pomocí sítě CANopen. Výstupem jsou otáčky motoru, který otáčí gyroskopem autopilota a simuluje tak otáčení letadla. Dynamika letadla je popsána LTI modelem.

Já se na platformě podílím vývojem softwaru simulátoru. Simulátor je program psaný v jazyce C, který poběží na jednodeskovém počítači BOA5200 v Linuxu s jádrem 2.6. Součástí jádra, které bylo pro desku přizpůsobeno na katedře, je i ovladač pro sběrnici CAN. Mým úkolem bylo implementovat volně konfigurovatelný software vykonávající simulaci LTI modelu a komunikující s operátorskou aplikací prostřednictvím TCP a s periferními zařízeními prostřednictvím sítě CANopen, jejíž ovladač jsem integroval do systému. Výsledkem mé práce je tedy spustitelný soubor simulátoru uložený v systému souborů Linuxu v paměti flash na desce BOA. Po připojení napájení desky se spustí nejprve Linux a potom také simulátor, který čeká na připojení operátorské aplikace. Po obdržení konfigurace a požadavku na start simulace z operátorské aplikace jsou periodicky sbírána měřená data z CANopen zařízení, vypočítáván krok simulace zadaného LTI modelu a vypočtená data jsou odesílána na CANopen zařízení. Všechna data jsou průběžně odesílána do operátorské aplikace.

2 Použité technologie

V této kapitole popíšu technologie a prostředky použité při návrhu platformy pro simulaci hardwaru ve smyčce.

2.1 Jednodeskový počítač BOA5200

Nejdůležitější částí platformy pro simulaci hardware ve smyčce je simulační počítač, na kterém se v reálném čase vykonává matematický model simulované soustavy. Pro tento účel jsme vybrali jednodeskový počítač BOA5200 kvůli dostatečnému výpočetnímu výkonu, podpoře OS Linux, komunikačním možnostem a provedení umožňující snadné vestavění do průmyslových aplikací.

Zařízení se skládá ze základní desky (*Main board*) osazené procesorem PowerPC MPC5200 a paměťmi, která se připojuje k tzv. *Carrier board*. Tato deska obsahuje konektory pro periferie vyvedené ze základní desky.

2.1.1 Parametry základní desky

- rozměry: 62x62 mm
- procesor: MPC5200
- SDRAM: 64MB, 32bitová
- FLASH: 16MB, 16bitová

2.1.2 Periferie

- Ethernet 10/100Mbps
- 2x CAN
- RS232
- SPI
- Adresová a datová sběrnice
- GPIO s funkcí *timeru* a *interruptu*

2.1.3 Parametry MPC5200

- Jádro MPC603e, superskalární architektura
- Frekvence jádra 400MHz
- FPU (floating point unit)
- MMU (memory management unit) – procesy s odděleným paměťovým prostorem – větší bezpečnost programování

2.2 Operační systém Linux

Simulátor LTI modelu běží na desce BOA5200 v operačním systému Linux. Na katedře byla vytvořena úprava jádra Linuxu verze 2.6.18 pro tuto desku. Zdrojové kódy jádra, sada nástrojů pro překlad a konfiguraci spolu s podrobným popisem je k dispozici na webové stránce [2].

V první verzi používáme standardní jádro Linuxu bez podpory pro *real-time*. I když navrhujeme aplikaci, která vyžaduje pro své správné fungování *real-time* chování, můžeme si dovolit spouštět simulátor z klasického Linuxu, protože výpočetní výkon použitého počítače je dostatečně velký. Přesto ale plánujeme úpravu operačního systému na *real-time* formu. Tím zajistíme spolehlivé fungování i při velmi krátkém simulačním kroku (4 ms).

2.3 Sběrnice CAN

Controller Area Network (CAN) [3] je sériový komunikační protokol, vyvinutý v 80. letech dvacátého století firmou Bosch pro nasazení v automobilovém průmyslu. CAN byl navržen tak, aby umožnil provádět distribuované řízení systému v reálném čase s přenosovou rychlostí do 1Mbit/s a vysokým stupněm zabezpečení přenosu proti chybám.

Jedná se o protokol typu *multi-master*, kde každý uzel sběrnice může být *master* a řídit tak chování jiných uzlů. Není tedy nutné řídit celou síť z jednoho "nadřazeného" uzlu, což přináší zjednodušení řízení a zvyšuje spolehlivost (při poruše jednoho uzlu může zbytek sítě pracovat dál). Pro řízení přístupu k médiu je použita sběrnice s náhodným přístupem, která řeší kolize na základě prioritního rozhodování. Po sběrnici probíhá komunikace mezi dvěma uzly pomocí zpráv (datová zpráva a žádost o data), a management sítě (signalizace chyb, pozastavení komunikace) je zajištěn pomocí dvou speciálních zpráv (chybové zprávy a zprávy o přetížení). Zprávy vysílané po sběrnici protokolem CAN neobsahují žádnou informaci o cílovém uzlu, kterému jsou určeny, a jsou přijímány všemi ostatními uzly připojenými ke sběrnici. Každá zpráva je uvozena identifikátorem, který udává význam přenášené zprávy a její prioritu, nejvyšší prioritu má zpráva s identifikátorem 0. Protokol CAN zajišťuje, aby zpráva s vyšší prioritou byla v případě kolize dvou zpráv doručena přednostně a dále je možné na základě identifikátoru zajistit, aby uzel přijímal pouze ty zprávy, které se ho týkají (*Acceptance Filtering*).

2.3.1 Ovladač SocketCan

Linuxový ovladač pro sběrnici CAN s názvem SocketCan [4] vytváří API pro přístup ke sběrnici s využitím standardního komunikačního prostředku Linuxu – *socketů*. Je definována speciální třída (*protocol family*) PF_CAN, která je analogií PF_INET používané při komunikaci pomocí protokolů TCP/IP. K ovladači SocketCan existuje i jeho *Real-Time* (XENOMAI) implementace. Ta využívá stejné API jako základní verze.

Ovladač SocketCan má několik výhod oproti ostatním ovladačům sběrnice CAN, které nás přesvědčili o jeho vhodnosti pro náš projekt:

- Volně šiřitelný v rámci *GNU Public License*
- Vytvořena verze pro jádro Linuxu 2.4 i 2.6
- Jednoduché použití založené na Linuxových *socketech*

Důležité je také to, že je SocketCan v současnosti hojně využívaný na Katedře řídící techniky, což zajišťuje dobrou podporu a dostatek informací a zkušeností s jeho používáním.

2.4 Protokol CANopen

CANopen [5] je protokol vyšší úrovně využívající ke komunikaci mezi zařízeními sběrnici CAN. V počátcích byl vyvíjen autorem sběrnice CAN, firmou Bosch, ale poté přešel pod správu organizace CiA (*CAN in Automation*). CANopen specifikuje aplikační vrstvu k protokolu CAN a komunikační profily (CiA DS 301).

Protokol CANopen nahlíží na CAN ne jako na sběrnici, ale jako na síť. Každé zařízení v této síti tvoří uzel sítě a má přidělen unikátní identifikátor (adresu). Jednotlivé zprávy tak nejsou odesílány jako *broadcast*, ale jsou adresovány konkrétnímu zařízení. Každé zařízení v síti má slovník (*Object Dictionary* – OD), který obsahuje kompletní nastavení komunikačních parametrů i samotného zařízení, aktuální hodnoty vstupů a výstupů a stavové a chybové registry. Jednotlivá zařízení jsou potom konfigurována a řízena zápisem do jejich *Object Dictionary* přes síť.

Pro komunikaci a správu sítě je definováno několik služeb.

2.4.1 *Process Data Object* (PDO)

PDO je jednoduchý komunikační prostředek sloužící pro přenos dat mezi zařízeními. Pokud chceme přenést nějaká data z jednoho zařízení na druhé, musíme nastavit jednak odeslání PDO s určitým ID na zdrojovém zařízení a jednak příjem PDO se stejným ID na zařízení cílovém. Přenášená data jsou potom opět registry v *Object Dictionary*, které jsou na konkrétní PDO mapovány. Při příjmu PDO jsou data opět rozdělena a zapsána do OD cílového zařízení v závislosti na nastavení mapování. Příjem PDO zprávy není nijak potvrzován.

Služba PDO definuje několik způsobů odesílání zprávy (*Transmission types*):

- Po události v zařízení (vypršení časovače, změna dat)
- Po příjmu synchronizační zprávy ze sítě
- Na požádání cílového zařízení

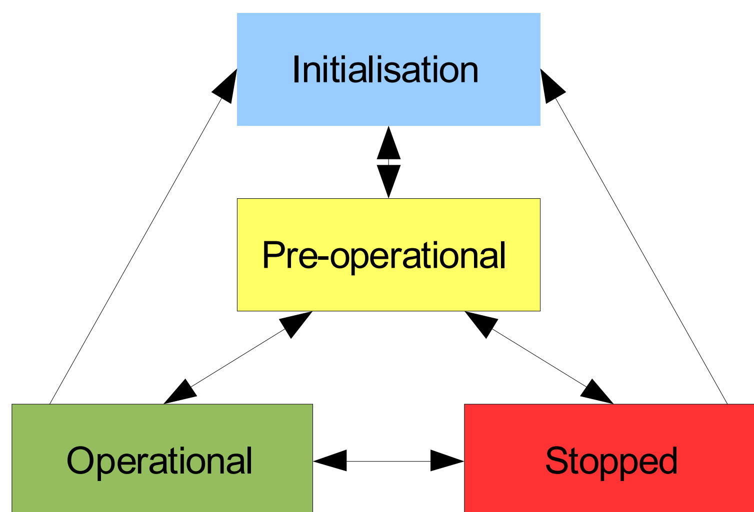
2.4.2 Service Data Object (SDO)

Služba SDO slouží ke konfiguraci síťového zařízení zápisem hodnot do jeho OD. Tento zápis je potvrzovaný. Komunikace probíhá podle modelu server-klient. Přenos dat iniciuje vždy klient požadavkem na zápis. Standard CANopen specifikuje, že každé zařízení musí mít alespoň jeden SDO server, aby bylo možné ho konfigurovat. Adresace zařízení je realizována opět podle ID zpráv jako u ostatních služeb.

Každá SDO zpráva obsahuje adresu registru v cílovém OD, kam mají být přenášená data zapsána. Server po příchodu SDO zprávy odpoví zprávou s výsledkem zápisu.

2.4.3 *Network Management* (NMT)

Každé CANopen zařízení se může nacházet v jednom ze čtyřech stavů (viz. Obrázek 3). Ve stavu *Operational* zařízení pracuje a provádí všechny nastavené operace. Přechody jednotlivých zařízení mezi stavy, tj. jejich zapínání a vypínání, je řízeno jediným zařízením, které v síti figuruje jako *master* této služby, pomocí NMT zpráv. NMT zprávy mají ID rovno nule, což odpovídá maximální prioritě na sběrnici CAN.



Obrázek 3 - Diagram přechodů zařízení mezi režimy

2.4.4 *Synchronization Object* (SYNC)

Jedno zařízení v síti může fungovat jako producent synchronizačních zpráv, ostatní zařízení tyto zprávy pouze přijímají. Příjem synchronizační zprávy může sloužit jako podnět k odeslání definovaných PDO. Zajišťuje také synchronizaci sběru a aplikace dat, což je důležité pro synchronnost vzorkování.

2.4.5 *Emergency Object* (EMCY)

Emergency zprávy slouží zařízením k oznamování chybových stavů. Zpráva obsahuje chybový kód, jehož význam se liší podle konkrétních zařízení. Kterým zařízením byla zpráva odeslána, poznáme opět podle jejího ID.

Protokol CANopen specifikuje ještě několik dalších služeb pro zprávu a detekci chyb sítě. V našem projektu je ale nepoužíváme, proto je nebudeme zmiňovat.

2.4.6 Ovladač CanFestival

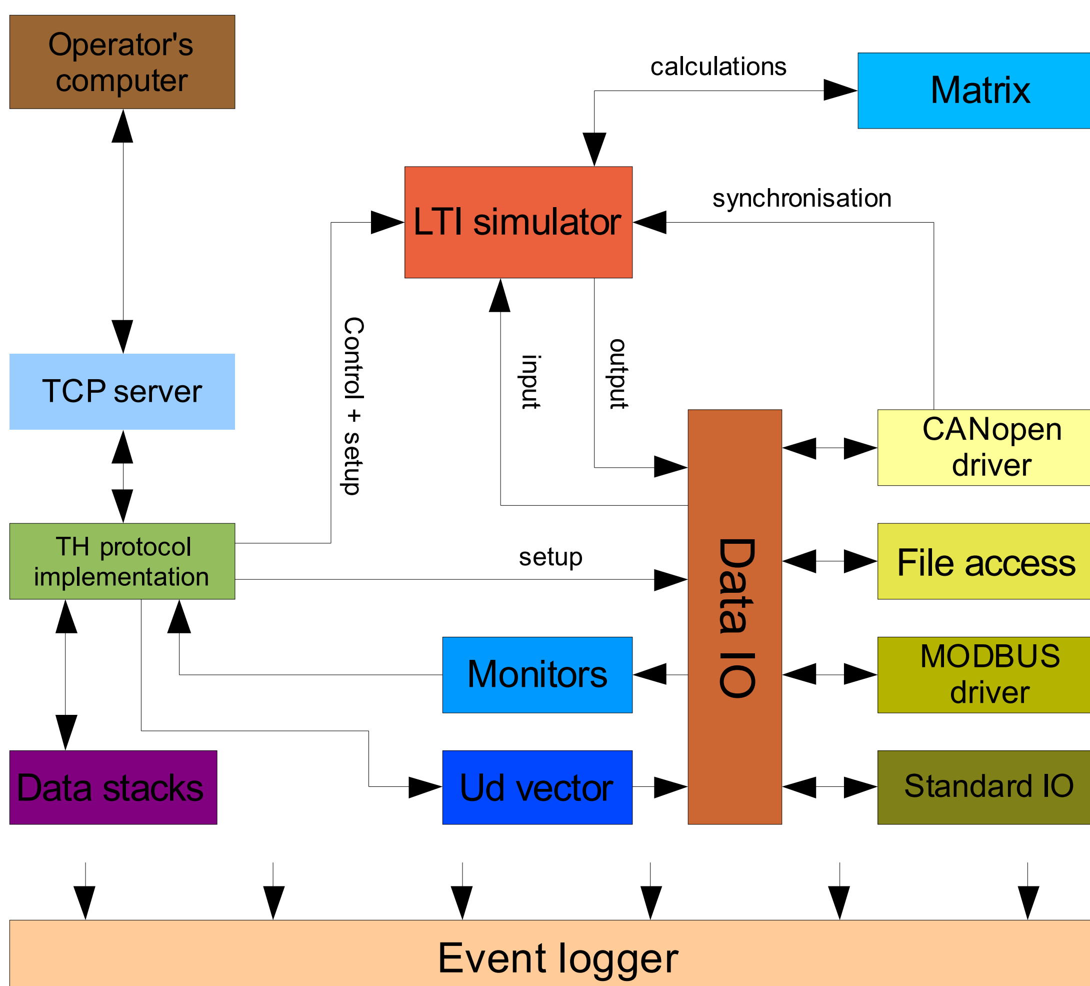
CanFestival [6] je sada nástrojů, která implementuje nejdůležitější části protokolu CANopen, je volně šiřitelná v rámci GPL a v nejnovější verzi umožňuje i spolupráci s ovladačem SocketCan, který budeme používat. Vzhledem k tomu je pro nás CanFestival ideální volbou.

S CanFestivalem se pracuje tak, že se nejprve pomocí grafického nástroje, který je součástí distribuce CanFestivalu, vytvoří konfigurace jednoho uzlu (node) vytvářené sítě. Z této konfigurace je automaticky vygenerován kód v jazyce C. Ten je nutné přidat ke kódu aplikace, která na tomto uzlu sítě poběží, a dopsat funkce, které obsluhují jednotlivé události sítě, jako je například příjem zprávy. Nakonec se celý program přeloží a sloučí s knihovnami CanFestivalu. Vše je nutné přeložit pro procesor PowerPC, aby bylo možné program spustit na desce BOA.

3 Program LTI simulátoru

3.1 Architektura programu

Program lze rozdělit do několika logických celků podle funkčnosti (viz. Obrázek 4). Komunikaci s operátorským počítačem obstarává TCP server a *parser* zpráv podle TH protokolu (*Target-Host protocol*). Zprávami z operátorského počítače je provedeno nastavení LTI modelu, jeho vstupů a výstupů a vstupních a výstupních zařízení. Řídícími zprávami je potom spouštěna a přerušována simulace. Monitory slouží aplikaci operátorského počítače k průběžnému sledování průběhu simulace. Pro přenos simulačních dat z operátorského počítače do simulátoru je definován tzv. Ud (*direct*) vektor, jehož vzorky jsou přenášeny opět zprávami TH protokolu.



Obrázek 4 - Architektura programu

3.1.1 Vícevláknový program

Program běží v několika vláknech, která jsou vytvářena a rušena podle aktuální potřeby. Po spuštění programu je ihned nastartováno vlákno obsluhující TCP server, který čeká na připojení klienta. Po odpojení klienta nebo při ztrátě rámcování zpráv je vlákno restartováno a tím uveden server opět do stavu čekání na příchozí připojení.

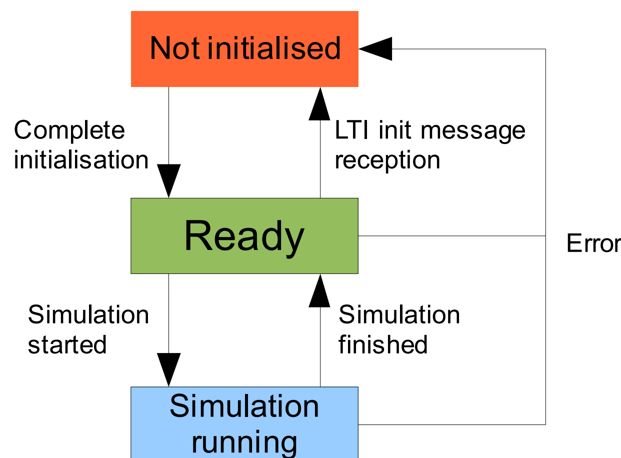
Vlákna jsou využita i při čtení a zápisu do IO zařízení. Každá vstupní nebo výstupní relace je spuštěna v samostatném vlákně, aby čekání na data z jednoho zařízení nebrzdilo celý proces. Zápis do zařízení připojených přes CANopen je řízen ovladačem CanFestival a z pohledu simulátoru jsou hodnoty

pouze zapisovány do obyčejných proměnných. U kopírování proměnné nehrozí žádná blokace a režie vytváření vlákna by byla mnohonásobně vyšší než samotný zápis, proto v případě CANopen zařízení se vlákna nepoužívají.

Žádná jiná vlákna už aplikací vytvářena nejsou a tak jsou v podstatě všechny funkce volány z vlákna TCP serveru. Jedná se ale o časově nenáročné funkce, které provádějí nějaké nastavení. Simulace je spuštěna voláním inicializační funkce po příchodu příslušené zprávy TH protokolu. Tato funkce spustí odesílání synchronizačních zpráv v ovladači CanFestival a skončí. Simulační krok a výpočet stavu je potom realizován jako *callback* funkce po synchronizační zprávě. Na provedení simulace tedy není potřeba vytvářet zvláštní vlákno. Ve skutečnosti je vlákno pro obsluhu časovačů vytvořeno přímo CanFestivalem.

3.1.2 Stavy systému

Simulátor se během svého běhu může nacházet v jednom ze tří stavů – *Not initialized, Ready* nebo *Simulation running* (viz. Obrázek 5). Po zapnutí do ukončení nastavení z klientské aplikace se nachází ve stavu *Not initialized*. V tomto stavu nemůže být spuštěna simulace. Po dokončení nastavení přejde simulátor do stavu *Ready* a očekává příkaz ke startu simulace. Během simulace se systém nachází ve stavu *Simulation* running. Po korektním ukončení simulace přejde simulátor opět do stavu *Ready* a simulace může být spuštěn znovu. Pokud nastane chyba v jakémkoliv stavu, je systém navrácen do stavu *Not initialized* a nastavení musí být provedeno znovu.



Obrázek 5 - Diagram přechodů mezi stavy systému

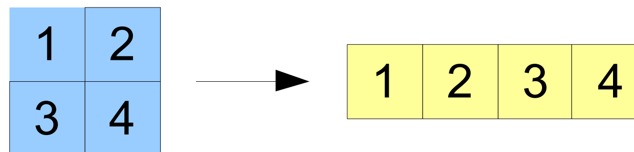
Pokud chceme provést nějaké změny v nastavení simulátoru mezi dvěmi simulacemi, je nutné zrušit minulé nastavení, aby se systém nemohl dostat do nekonzistentního stavu. Proto jsme zavedli, že při příchodu zprávy TH protokolu s nastavením LTI modelu je smazána dosavadní konfigurace simulátoru. Program tak přejde do stavu *Not initialized*.

3.2 Pomocné části programu

Některé části programu neplní přímo funkce, které odpovídají logické podstatě simulátoru LTI modelu. Proto je popíšu v samostatné kapitole.

3.2.1 Reprezentace matic

LTI model je zadáván stavovým popisem, který je složený ze čtyřech matic. Výpočet nového stavu potom vyžaduje násobení a sčítání těchto matic. Matice jsou v programu ukládány jako jednorozměrné pole proměnných typu *double*. Hodnoty jsou do pole řazeny po řádcích (viz. Obrázek 6). Ke každé matici je tedy nutné udržovat ještě informace o počtu řádkou a sloupců.



Obrázek 6 - Řazení hodnot matic do pole

V programu jsou implementovány funkce pro maticové násobení a sčítání s maticemi v této formě zápisu.

3.2.2 Zásobníky

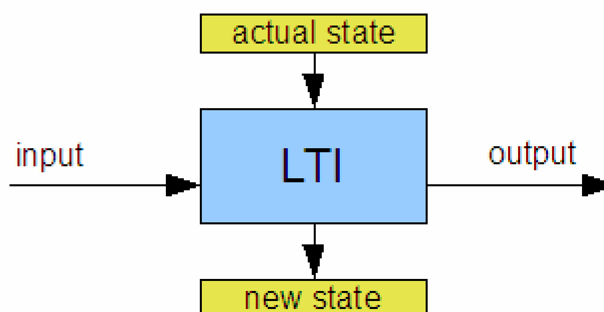
V programu je implementován zásobník typu fronta pro datový typ *char* (8 bitů). Samotný zásobník je reprezentován strukturou s polem hodnot a pomocnými proměnnými a funkcemi pro vytvoření a odstranění zásobníku a vkládání a vybírání prvku. Fronta je realizována jako kruhový zásobník [8] s pevnou velikostí. Velikost je určena při vytváření a nelze měnit v době existence zásobníku.

Zásobníky jsou využity jako *buffery* pro data přijímaná a odesílaná přes TCP. Jednotlivé zprávy TH protokolu jsou ukládány v zásobnících po bajtech. Zásobník je také vytvořen pro ukládání dat vektoru Ud, které jsou přijímány z operátorského počítače a čekají na využití v simulačním kroku. V tomto případě jsou ukládaná data typu *double*. I tak jsou ale v zásobníku udržovány jako pole hodnot typu *char* a po vyzvednutí přetypovány.

3.3 LTI model

Nejdůležitější částí simulátoru je samotný LTI model (*Linear Time Invariant system*) [9]. Ten je v programu implementovaný jako struktura uchovávající informace o systému. LTI systém je popsaný čtyřmi maticemi stavového popisu, které jsou u diskrétních systémů označovány jako M, N, C a D. Ve struktuře jsou dále uloženy údaje o řádu systému, počtu vstupů a výstupů a aktuální hodnoty vnitřních proměnných.

Součástí modulu LTI je sada funkcí počítajících následující stav systému v závislosti na aktuálním stavu a hodnotách vstupů (viz. Obrázek 7). Jediným úkolem LTI modulu je matematicky správný a co nejrychlejší výpočet. Časování simulace je řízeno jinou částí programu.



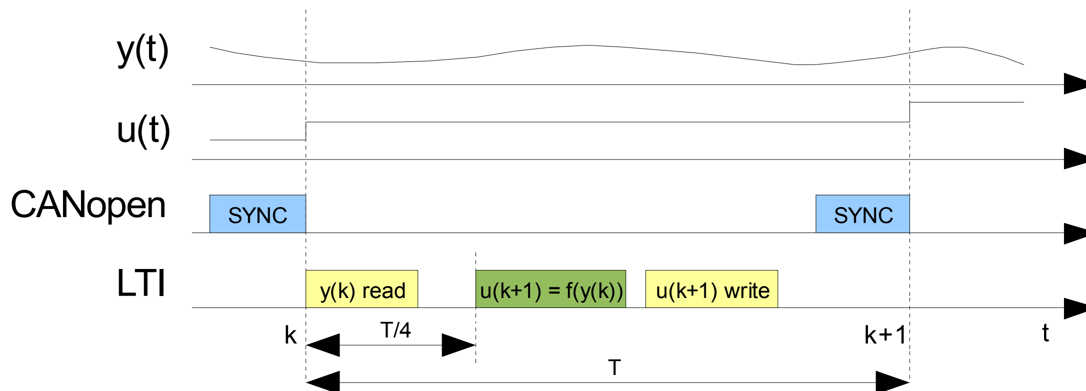
Obrázek 7 - Výpočet následujícího stavu LTI systému

3.4 Časování simulace

Po příchodu zprávy *start simulace* jsou nastaveny parametry simulace a v ovladači CanFestival je aktivováno odesílání synchronizačních zpráv s periodou stejnou jako je požadovaná perioda simulace. Všechna zařízení v síti CANopen jsou nastavena tak, že odesílají data po příjmu synchronizační zprávy. V simulátoru je nastavena *callback* funkce reagující na odeslání synchronizační zprávy, která spustí časovač na čtvrtinu periody simulace. Jako *handler* signálu tohoto časovače je přiřazena funkce provádějící výpočty simulačního kroku. Toto zpoždění od synchronizace je zavedeno z toho důvodu,

aby bylo zajištěno, že v době provádění výpočtů budou již přečteny hodnoty ze všech vstupních zařízení.

Po výpočtu nového stavu LTI modelu a nového výstupního vektoru jsou výstupní hodnoty zapsány do zařízení sloužících jako výstup. Tyto hodnoty jsou ale zařízeními využity až po příjmu další synchronizační zprávy (viz. Obrázek 8).



Obrázek 8 - Časový průběh jednoho simulačního cyklu

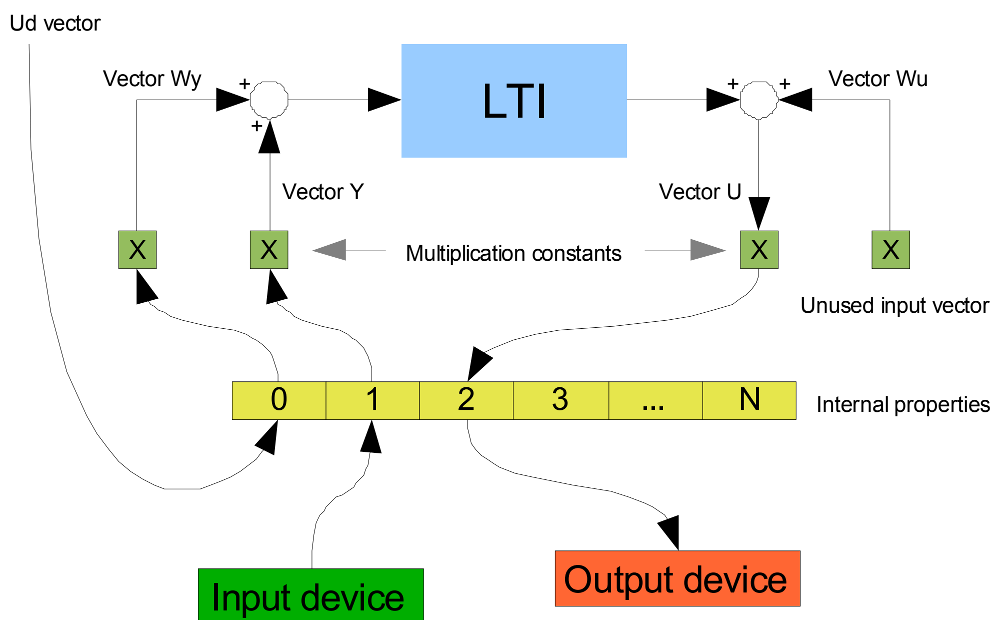
Vzhledem k tomu, že výstupní hodnoty vypočítané na základě vstupních hodnot v čase *k* se objeví na výstupu až v čase *k+1*, zavádí simulátor do systému jednotkové zpoždění. Je tedy nutné počítat s tím, že k přenosu F(z), který zadáme LTI modelu pro simulaci je přidáno ještě toto zpoždění.

$$G(z) = F(z).\frac{1}{z}$$

3.5 Vstupní a výstupní vektory

K popisu vstupů a výstupů LTI modelu používáme označení vektory (*IO vectors*), viz. Obrázek 9. Každý vektor má určitou šířku. Ta určuje kolik hodnot je přenášeno vektorem paralelně. V systému je vytvořeno pole vnitřních proměnných (*internal properties*), které slouží k předávání dat mezi vektory a zařízeními. Jednotlivé vnitřní proměnné jsou identifikovány pomocí jejich indexu v poli (0 až N).

Aby byl simulátor univerzální, lze vstupy a výstupy LTI modelu konfigurovat následujícím způsobem. Ke vstupnímu (Y) a výstupnímu (U) vektoru LTI modelu se přičítají ještě dva přídavné vektory (Wy a Wu). Každá proměnná každého z těchto vektorů může být navíc násobena libovolnou konstantou. Díky této struktuře je možné vytvořit libovolnou dvojici regulátor-soustava, ať už představuje LTI simulátor regulátor nebo soustavu. Logické vektory LTI modelu, respektive jejich proměnné, jsou mapovány na vnitřní proměnné simulátoru. Na stejné proměnné jsou potom mapovány vstupní a výstupní zařízení a tím je realizován přenos dat. Je možné mapovat i více vektorů na jednu proměnnou a tím například uzavřít zpětnou vazbu z výstupu na vstup modelu přímo ve struktuře simulátoru. Pokud jeden z takto spojených vektorů ještě vynásobíme konstantou -1, vytvoříme zápornou zpětnou vazbu.



Obrázek 9 - Příklad mapování IO vektorů na zařízení

3.6 Vstupní a výstupní zařízení

V simulátoru je možné definovat libovolné množství vstupních a výstupních zařízení (viz. Obrázek 9). Ke každému z nich je přiřazena vnitřní proměnná, jejíž hodnoty dané zařízení zapisuje nebo čte. Stejně jako vstupní a výstupní vektory mohou být i zařízení mapována na vnitřní proměnné v podstatě libovolně. Je možné nastavit jedné výstupní proměnné dvě výstupní zařízení a tak například vypisovat hodnoty odesílané do CANopen zařízení na konzoli nebo ukládat do souboru. Pokud bychom přiřadili jedné proměnné jedno vstupní a jedno výstupní zařízení, budou vlastně hodnoty přeposílány z jednoho zařízení na druhé beze změny, pouze se zpožděním jednoho simulačního kroku.

Lze používat zařízení několika typů:

- Zařízení v síti CANopen
- Zařízení připojené pomocí RS232 a komunikující pomocí protokolu MODBUS
- Soubor v lokálním souborovém systému Linuxu
- Standardní vstup a výstup

Primární typ zařízení je CANopen, ostatní typy budou využívány jen výjimečně, proto se podrobněji budeme zabývat pouze komunikací po síti CANopen.

V současnosti jsou podporována tato CANopen zařízení:

- Senzor úhlového natočení Kübler Sendix Absolut 5858
- Řídící jednotka střídavého motoru CPD170, řízení otáček (zápis i čtení)

Čtení ze vstupních zařízení je vyžádáno na začátku zpracování simulačního kroku. Procedury provádějící čtení se liší podle typu zařízení. Každé čtení (kromě zařízení CANopen) je spuštěno v samostatném vlákně. Po přečtení ze vstupních zařízení jsou hodnoty zapsány do proměnných, na které byla zařízení namapována. Odtud jsou předány v podobě IO vektorů LTI modelu. Po výpočtu nových výstupních hodnot LTI modelem jsou stejným způsobem hodnoty zapsány do příslušných zařízení.

3.7 TCP server

Komunikace s operátorským počítačem je realizována přes místní síť pomocí protokolu TCP. Po spuštění programu je v samostatném vlákně spuštěn TCP server. Ten otevře *socket* a poslouchá na zadaném portu (přednastaveno 5555) a čeká na připojení aplikace z operátorského počítače. Po navázání spojení jsou vytvořeny dva zásobníky, jeden na příchozí, jeden na odchozí data. Všechny

příchozí data jsou potom čteny po bajtech a ukládány do zásobníku. Ve chvíli, kdy nejsou k dispozici žádná data ke čtení nebo je zásobník plný, je zavolán parser TH protokolu, který data v zásobníku zpracuje.

Zápis je realizován funkcí, která není součástí vlákna. Ta, pokud je zavolána, postupně odešle všechna data z odchozího zásobníku. Používají ji funkce, které vytvářejí zprávy TH protokolu a ukládají je do zásobníku. Pokud je detekována chyba spojení nebo odpojení klienta, je server restartován a znovu uveden do stavu čekání na spojení.

3.8 *Target-Host* (TH) protokol

Pro komunikaci s aplikací operátorského počítače jsme navrhli vlastní komunikační protokol s názvem *Target-Host protocol* [10]. Jedná se o jednoduchý protokol založený na zprávách. Každá zpráva začíná dvoubajtovým identifikátorem. První bajt určuje třídu zpráv (řídící, inicializační, atd.), druhý bajt potom představuje konkrétní typ zprávy. Pak už následují data v pořadí a množství definovaném protokolem pro konkrétní ty zprávy.

Ve zprávách není přenášena jejich délka. U některých zpráv je délka pevná. U jiných se může měnit, ale vždy, tak že se dá jednoznačně určit buď z konfigurace simulátoru nebo z prvních několika bajtů zprávy. Příkladem může být zpráva nastavující LTI model. V ní jsou přenášeny hodnoty všech matic. Počet hodnot se liší podle velikosti matic, ale ta může být spočtena z údajů o počtu vstupů a výstupů systému a jeho řádu. Tato tři čísla jsou přenášena na začátku této zprávy.

Problém nastane, pokud klient odešle zprávu s nesprávnou délkou. Parser potom ztratí přehled o tom, kde zprávy začínají a končí, což vyústí v chybu *neznámé ID zprávy*. Ta je oznámena chybovým hlášením klientovi a server je restartován, protože není možné zaručit spolehlivou detekci začátku další zprávy. Tato vlastnost by byla zcela nepřípustná u serverů s širším využitím, kde není možné se spolehnout na správné fungování klienta. My ale předpokládáme využití pouze jednoho klienta vyvinutého a odladěného v rámci projektu a proto si takovéto zjednodušení můžeme dovolit. Ztráta dat během přenosu nastat nemůže, aniž by nedošlo ke ztrátě spojení, protože protokol TCP je sám o sobě potvrzovaný a spolehlivý.

Ne všechny zprávy jsou relevantní ve všech stavech systému. Řídící zpráva *zastavit simulaci* nemá, například, žádný smysl v době, kdy simulace neběží. Podobně nové nastavení LTI modelu není možné v průběhu simulace. Parser tyto případy detekuje, zprávu ignoruje a informuje o tom klienta varováním.

3.8.1 *State message* (stavová zpráva)

Podrobný popis všech zpráv je uveden v dokumentaci k TH protokolu [10], proto jej zde nebudu uvádět. Popíšu fungování zprávy, jejíž využití je komplexnější než u ostatních zpráv. Stavová zpráva (viz. Tabulka 1) slouží k oznámení o přechodu systému z jednoho stavu na druhý a také důležitých informací, varování a hlavně chyb klientské aplikaci. Její definici převzatou z TH protokolu znázorňuje obrázek.

State message - ID = 3		direction S->C		Message with the state of the system.	
Field name	ID	State	Mess.type	Message	Additional info
Field position	0-1	2	3	4	5-12
Field value	0x0103				
Data type	16-bit integer	char	char	char	
Field Length [B]	2	1	1	1	8

Tabulka 1 - Definice stavové zprávy TH protokolu

Po identifikátoru zprávy následuje jeden bajt s aktuálním stavem systému. Pokud je zpráva pouze odpovědí na vyžádání informace o stavu od klienta, jsou následující bajty nulové, tzn. typ zprávy říká, že žádná zpráva není přenášena. Pokud je ale zpráva odeslána simulátorem bez výzvy, obsahuje

vždy nějakou další informaci. Typ zprávy může nabývat hodnot *INFO, WARNING* nebo *ERROR*. V dalším Bajtu je potom číselný kód zprávy a v poli *additional* info další informace, které je třeba předat klientské aplikaci. Při události je odeslána tato zpráva s informací o tom, co se stalo a se stavem, do kterého systém díky této události přešel.

3.8.2 Reset konfigurace

Při inicializaci simulátoru nezáleží na pořadí v jakém jsou jednotlivé zprávy s nastavením odesílány. Jako první ale vždy musí být odeslána zpráva s nastavením LTI modelu a to ze dvou důvodů. Prvním z nich je ten, že je systém nastaven tak, že při příchodu této zprávy smaže aktuální konfiguraci simulátoru. To proto, aby nedocházelo k míchání staré a nové konfigurace.

3.8.3 Autodetekce *endianity*

Problém *endianity* [11] vychází z rozdílné reprezentace vícebajtových čísel různými systémy. Pokud zapisujeme vícebajtové číslo do paměti s šířkou 8 bitů, je jasné, že číslo musí být rozděleno po bajtech a zapsáno postupně. Rozlišujeme hlavně dvě možné varianty zápisu. Když je v buňce s nejnižší adresou zapsán bajt s nejvyšším významem (MSB – *Most Significant Byte*), hovoříme o metodě *Big-endian*. Pokud jsou bajty uspořádány obráceně, tj. na nejnižší adrese je nejméně významný bajt (LSB – *Least Significant Byte*), jedná se o *Little-endian*.

Ve zprávách TH protokolu jsou také přenášena vícebajtová čísla. Například hodnoty matic LTI modelu jsou čísla typu *double*. Vzhledem k tomu, že předpokládáme komunikaci pouze mezi dvěma systémy, které se nebudou měnit, nebyl by velký problém napevno nastavit v jakém pořadí mají být bajty přenášeny. Mnohem efektivnější řešení ale je, zjišťovat použitou *endianitu* při začátku komunikace. Potom je tento problém kompletně vyřešen na straně serveru.

V praxi je autodetekce *endianity* provedena tak, že hned za identifikátorem zprávy s konfigurací LTI modelu je přenášeno dekadické číslo 123 v datovém typu *unsigned integer*. Server toto číslo přečte a tak pozná jestli klientská aplikace používá *Big* nebo *Little-endian*. Podle toho je potom nastaven příjem a odesílání všech vícebajtových proměnných po celou dobu komunikace. Autodetekce *endianity* je tedy dalším důvodem, proč je nutné přenášet zprávu s konfigurací LTI modelu jako první.

3.9 Monitory

Aby mohl uživatel sledovat a vyhodnocovat průběh prováděné simulace, je nutné umožnit klientské aplikaci sledovat hodnoty vstupů a výstupů během simulace. K tomu jsem zavedl systém monitorů. Ty se aktivují příslušným údajem ve zprávě *start simulace*. Pokud jsou aktivní, je po každém simulačním kroku odesláno celé pole vnitřních proměnných simulátoru ve zprávě monitoru klientské aplikaci. Klient tedy musí udržovat informaci o tom, jak byly namapovány vektory a zařízení na vnitřní proměnné, aby mohl určit, která proměnná zprávy reprezentuje který vstup nebo výstup.

Je nutné aby klientská aplikace četla data ze *socketu* už v průběhu simulace, protože hrozí zaplnění zásobníku příchozích dat a tím zablokování komunikace.

3.10 *Direct vector* (Ud)

Obráceně než monitory funguje tzv. *direct vector* (Ud), který přenáší data z klientské aplikace a podle toho, jak je nastaveno jeho mapování, ukládá hodnoty do vnitřních proměnných systému. Odtud mohou být využity jako vstupní proměnné LTI modelu nebo odesílány nějakému zařízení. V logické struktuře simulátoru se Ud vektor chová jako vstupně výstupní zařízení, ale jeho konfigurace se provádí stejnou zprávou jako konfigurace vektorů LTI modelu. Ud vektor může mít libovolnou šířku, může jím být tedy přenášeno libovolné množství hodnot. Každá hodnota je potom, stejně jako u ostatních vektorů, mapována samostatně na vnitřní proměnnou a je násobena vlastní konstantou. Ačkoliv je tedy Ud vektor jenom jeden, je možné přenášet jím současně více dat a rozdělit ho na různá zařízení a vektory.

V programu je Ud vektor realizován zásobníkem, do kterého jsou všechny hodnoty uloženy po příjmu zprávy. V simulačním kroku jsou potom hodnoty odpovídající jednomu časovému vzorku vyjmuty a

podle nastavení mapování zkopírovány do příslušných vnitřních proměnných. Pokud je zásobník prázdný je zachována minulá hodnota proměnných. Není tedy nutné posílat novou zprávu před každým simulačním krokem, pokud nechceme měnit hodnoty proměnných Ud vektoru. Pro realizaci jednotkového skoku tedy například stačí dvě zprávy. Jedna s hodnotou nula na začátku simulace a druhá s hodnotou jedna v čase jednotkového skoku.

Důležitým faktem, se kterým je nutné počítat, je, že je zásobník Ud vektoru vyprázdněn při resetu konfigurace, tedy po příchodu zprávy s konfigurací nového LTI modelu.

3.11 Logování událostí

Pro diagnostiku chování simulátoru je vytvořen *logger*, který zapisuje důležité události, varování a chyby jednak do souboru a jednak na obrazovku. Každý záznam začíná informací o tom, jestli se jedná a běžnou událost, varování nebo chybu. Potom následuje text sdělení. Záznamy v logovacím souboru jsou navíc opatřeny aktuálním datem a časem události.

Logovací soubor je vytvořen při startu programu při první logované události a jeho název je složen z data a času vytvoření. Tím je zajištěno, že nedojde ke konfliktu názvů souborů. Pro každé spuštění programu je vytvořen nový soubor.

Zápis do souboru i výpis textu na konzoli jsou poměrně pomalé procesy. Proto není dobré zapínat logování v běžném provozu, ale využívat ho jenom v době ladění. Program je nastaven tak, že při spuštění bez parametrů jsou výpisy i logování vypnuté. Pokud chceme logování nebo výpisy aktivovat, musíme to specifikovat spouštěcími parametry.

3.11.1 Úrovně logování

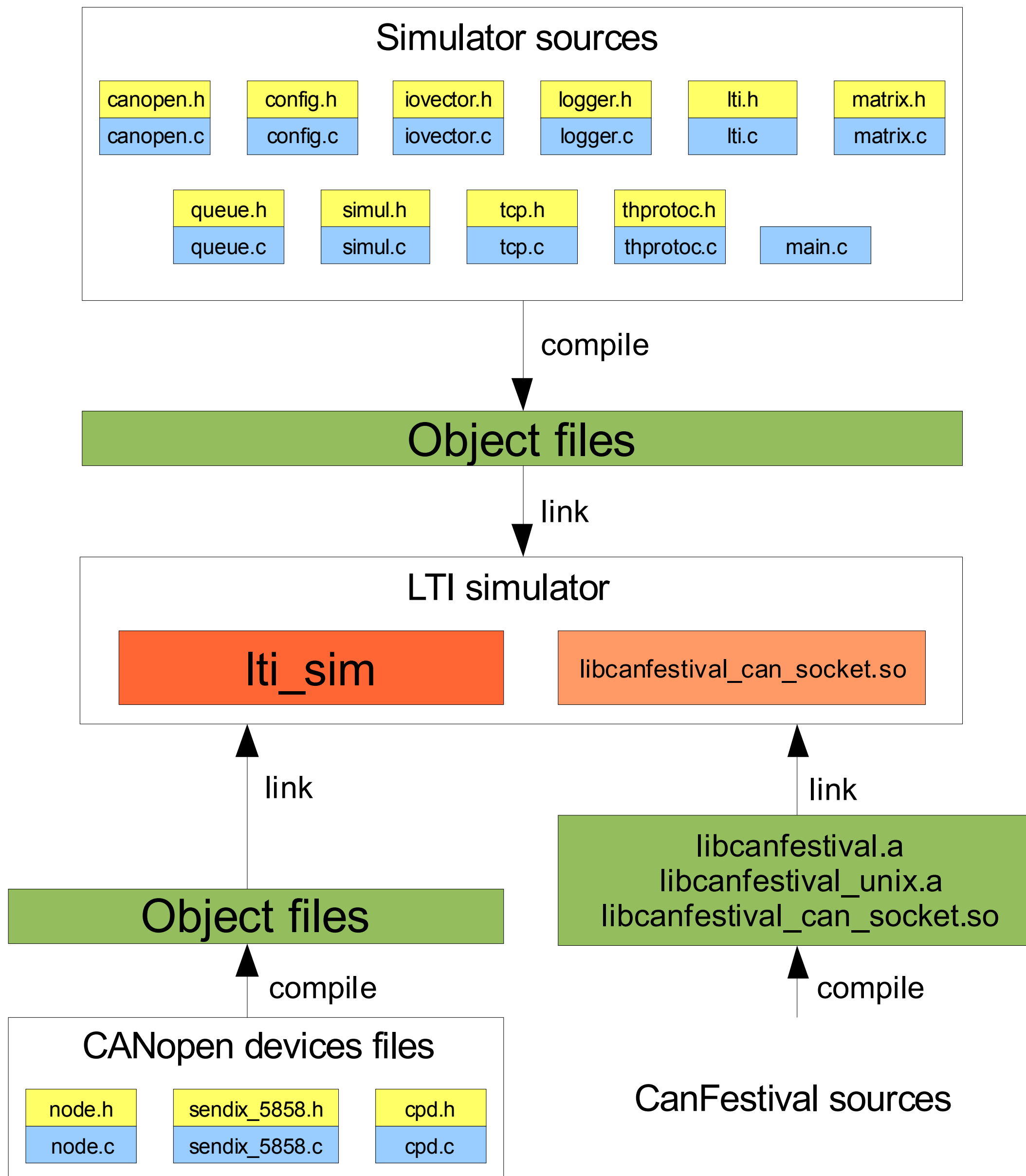
Pro logování i výpis na obrazovku se dají nastavit tři úrovně podle množství logovaných informací – *LOW, MEDIUM* a *HIGH*. Výpisy na obrazovku a logování se nastavují zvlášť. Úrovně se nastavují tak, že se zadají jako parametry při spouštění programu. Parametr –v určuje úroveň vypisování a parametr –l úroveň logování. Hodnota se potom udává písmeny L (*LOW*), M (*MEDIUM*) nebo H (*HIGH*). Pro příklad uvádím příkaz, který spustí simulátor se střední úrovní logování do souboru a nízkou úrovní výpisu na obrazovku.

```
$ lti_sim –l M –v L
```

Pokud některý parametr (nebo oba) není uvedený, je příslušný typ výpisu vypnut.

3.12 Zdrojové kódy a překlad programu

Jak je obvyklé při programování v jazyce C, je zdrojový kód programu rozdělen do souborů podle tématicky podobných částí. Hlavičkové soubory s příponou .h jsou v adresáři *include*, zdrojové soubory s příponou .c v adresáři *src*. Zdrojové soubory jsou doplněny souborem *Makefile* pro program *make* [7], který provede překlad a linkování celého programu (viz. Obrázek 10). Během překladu je příkaz *make* aplikován i na zdrojové soubory ovladače CanFestival, jehož překladem vzniknou knihovny, které jsou použity simulátorem. Výsledkem překladu je spustitelný soubor simulátoru *lti_sim* a dynamická knihovna CanFestivalu *libcanfestival_can_socket*.so s prostředky pro komunikaci s ovladačem pro sběrnici CAN.



Obrázek 10 - Překlad a linkování programu

3.13 Dokumentace

Zdrojový kód programu je okomentovaný tak, aby bylo možné z těchto komentářů automaticky vygenerovat programovou dokumentaci. Stránky HTML vytvořené pomocí programu Doxygen jsou k dispozici na [12].

4 Závěr

V rámci této práce jsem vytvořil platformu pro simulaci hardwaru v uzavřené smyčce. Jejím jádrem je jednodeskový počítač BOA5200 osazený procesorem PowerPC MPC5200. Tento hardware jsme vybrali zejména kvůli jeho dostatečnému výkonu a velké rozšířenosti a z ní plynoucí podpoře. Na počítači běží operační systém Linux s jádrem verze 2.6. Distribuce Linuxu včetně nástrojů pro jeho překlad byla upravena pro fungování na desce BOA na katedře řídící techniky. Já tento Linux, který je doplněn i programy pro správu a obsluhu jako je Telnet server, pouze využívám a celý systém dále rozšiřuji o integraci poprotokolu CANopen.

Samotný simulátor je program psaný v jazyce C. Při vývoji byl kladen důraz na snížení časové náročnosti jednotlivých operací, protože je nutné, aby simulace probíhala v reálném čase. K tomu nám také pomáhá vysoký výkon počítače. Programová dokumentace je tvořena HTML stránkami vygenerovanými z komentářů kódu pomocí programu Doxygen. Dokumentace je dále doplněna přílohami tohoto dokumentu, které na příkladech popisují překlad a používání programu a komunikaci se simulátorem.

Simulátor je spuštěn po připojení napájení počítače a *nabootování* Linuxu. K němu se přes TCP spojení připojí operátorská aplikace, která jeho řízení a konfiguraci provádí pomocí zpráv TH protokolu, který jsme pro simulátor navrhli. Stejným způsobem jsou i odesílána data sloužící k monitorování průběhu simulace operátorskou aplikací.

Výměna dat s testovaným hardwarem je realizována přes sběrnici CAN a protokol vyšší úrovně CANopen. V dnešní době je tato kombinace hojně využívána a na trhu je k dispozici velké množství senzorů a akčních prvků, které podporují přímo CANopen. Pro každé nové zařízení je ale nejprve potřeba v programu vytvořit podporu. V současné době je podporován senzor úhlového natočení Sendix 5858 a řídící jednotka střídavého motoru CPD170. Jako ovladač protokolu CANopen používáme produkt s názvem CanFestival. Ten byl vybrán jako hlavní ovladač CANopen na katedře řídící techniky a proto jsem pro něj vytvořil podrobný návod k použití, který je k dispozici v příloze.

Ačkoliv je platforma velmi univerzální je potřeba pro každého uživatele vytvořit snadněji ovladatelnou a jednodušší verzi splňující jeho konkrétní požadavky. Pro společnost Aero Vodochody jsme vytvořili modifikaci, která nevyžaduje konfiguraci vstupních a výstupních zařízení a výrazně tak usnadňuje operátorovi práci se simulátorem. V současné době je jeden prototyp platformy umístěn již u zákazníka, kde je postupně odlaďován a poté bude zapojen do běžného provozu.

5 Literatura a odkazy

[1] http://en.wikipedia.org/wiki/Hardware_in_the_loop

[2] http://rtime.felk.cvut.cz/hw/index.php/Boa5200

[3] http://www.semiconductors.bosch.de/en/20/can/index.asp

[4] http://openfacts.berlios.de/index-en.phtml?title=Socket-CAN, http://www.can-cia.org/can/

[5] http://www.can-cia.org/canopen/,
http://www.nikhef.nl/pub/departments/ct/po/doc/CANopenCiA.pdf,
http://www.nikhef.nl/pub/departments/ct/po/doc/CANopen30.pdf,
http://www.datamicro.ru/download/301_v04000201.pdf

[6] http://www.canfestival.org/

[7] http://www.gnu.org/software/make/

[8] http://en.wikipedia.org/wiki/Circular_buffer

[9] http://en.wikipedia.org/wiki/LTI_system_theory

[10] Target-Host protocol

[11] http://en.wikipedia.org/wiki/Endianness

[12] http://lukas-hamacek.wz.cz/hil/doc/html/index.html

[13] http://rtime.felk.cvut.cz/hw/index.php/HOWTO#Updating_RedBoot

[14] http://ecos.sourceware.org/ecos/docs-latest/redboot/persistent-state-flash.html

[15] http://rtime.felk.cvut.cz/hw/index.php/HOWTO#Building_kernel

[16] http://lukas-hamacek.wz.cz/hil/MPC5200_root_flash.tar.gz

[17] http://rtime.felk.cvut.cz/hw/index.php/HOWTO#Tool_chain

[18] http://lukas-hamacek.wz.cz/hil/lti_sim.zip

Příloha A - Modifikace pro Aero Vodochody

Platforma pro simulaci hardwaru ve smyčce je vyvíjena jako univerzální nástroj na realizaci této metody testování hardware. Způsob konfigurace je navržen tak, že téměř veškeré nastavení je prováděno pomocí zpráv TH protokolu v době běhu a do programu simulátoru není nutné zasahovat. Široké možnosti konfigurace, ale na druhou stranu přinášejí i nevýhody, nebo spíše komplikace. V první řadě je nutné obsluhovat simulátor z operátorské aplikace, která umožňuje uživateli veškeré nastavení provádět a to pokud možno jednoduchou a přehlednou formou. Potom je také nutné, aby tento uživatel byl detailně seznámen se způsobem nastavení a práce se simulátorem. Pro správné nastavení periferií je nutné mít i základní přehled o protokolu CANopen a umět nastavit distribuovaná zařízení. Takové požadavky na uživatele jsou v praxi samozřejmě nepřiměřené.

Každý zákazník bude využívat simulátor k testování jen určitého druhu zařízení, komunikace bude realizována stále stejnými vstupně výstupními zařízeními a vektory budou mapovány stále stejně nebo maximálně v několika variantách. Proto se předpokládá, že pro každého zákazníka se vytvoří přednastavený systém spolu se zjednodušenou operátorskou aplikací. Uživatel potom bude muset provádět pouze nastavení, která odliší různé způsoby testování v rámci jeho pohledu na problém a nebude se muset zabývat například konfigurací vstupně výstupních zařízení.

Prvním zákazníkem je společnost Aero Vodochody, kde bude simulátor sloužit v první fázi projektu pro simulování chování letadla v závislosti na natočení klapky. Pro tuto aplikaci je vytvořeno fixní nastavení systému a pouze minimální nutná část konfigurace je ponechána na uživatelské aplikaci.

A.1 Vstupní a výstupní zařízení

Simulátor dodaný do společnosti Aero Vodochody používá dvě zařízení připojená pomocí sítě CANopen. Jedná se o senzor úhlového natočení Kübler Sendix 5858 absolut a řídící jednotku střídavého motoru CPD170.

Konfigurace sběrnice je následující:

- Rychlost sběrnice CAN je 1Mb/s
- Zakončovací odpor sběrnice je zapnut v senzoru Sendix

Konfiguraci zpráv sítě CANopen znázorňuje Tabulka 2.

		SDO		PDO			
		BOA -> DEV	DEV -> BOA	BOA -> DEV		DEV -> BOA	
Device	Node ID	COB_IB	COB_IB	COB_IB	Type	COB_IB	Type
Sendix 5858	2					0x382	0x01
CPD	8	0x608	0x588	0x508	0x01	0x488	0x01

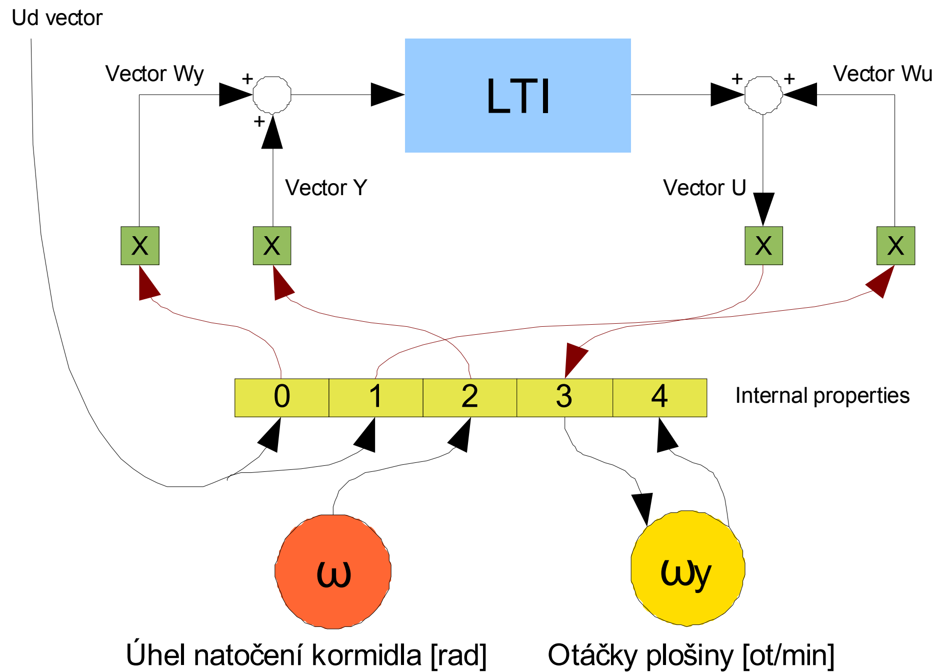
Tabulka 2 - Tabulka používaných ID zpráv na síti CANopen

Dále jsou v síti používány zprávy jako například NMT nebo Emergency. Jejich identifikátory, ale vycházejí přímo ze specifikace CANopen a jsou nastaveny napevno v zařízeních a ovladači CanFestival.

A.2 Mapování vstupů a výstupů

V Aeru Vodochody je využíván LTI model s jedním vstupem a jedním výstupem (viz. Obrázek 11). To znamená, že šířka všech vstupních a výstupních vektorů (U, Y, Wu, Wy) bude 1. Navíc je napevno vytvořen Ud vektor šířky 2. Je vidět, že systém využívá 5 vnitřních proměnných. Na první dvě proměnné (0 a 1) jsou mapovány dvě proměnné Ud vektoru. Hodnota natočení kormidla měřená senzorem Sendix je mapována na proměnnou 2. Pro nastavení otáček motoru řízeného jednotkou

CPD je použita hodnota proměnné 3. Na proměnnou 4 je mapován výstup řídící jednotky, který udává aktuální otáčky motoru. Mapování vektorů LTI modelu (U, Y, Wu, Wy) včetně násobících konstant není předdefinováno a musí být provedeno při konfiguraci pomocí příslušných zpráv TH protokolu. Tím je umožněno uživateli připojovat jednotlivá zařízení a kanály Ud vektoru na požadované vektory modelu. Příklad nastavení včetně mapování vektorů je na obrázku. Mapování, které provádí uživatel při konfiguraci je naznačeno červenými šipkami a jedná se pouze o jednu z mnoha možností propojení zařízení na vektory.



Obrázek 11 - Fixní mapování IO v simulátoru

Příloha B - How to build LTI simulator

First of all download PowerPC MPC5200 tool chain from [17] and install it. Then download CanFestival driver source from [6]. The primary way how to do it is to check out it from CanFestival CVS by these steps:

```
cvs -d:pserver:anonymous@lolitech.dyndns.org:/canfestival login
(type return, without entering a password)
The system will respond:
Logging in to :pserver:anonymous@lolitech.dyndns.org:2401/canfestival
Then, enter:
cvs -z3 -d:pserver:anonymous@lolitech.dyndns.org:/canfestival co -P
CanFestival-3
```

Then insert CanFestival folder and run this commands:

```
./configure --timers=unix --can=socket --cc=powerpc-603e-linux-gnu-gcc --
arch=ppc --os=linux --target=unix
make
make install
```

Building of CanFestival will produce some libraries and copy them to */usr/powerpc-603e-linux-gnu/lib* folder. Then download simulator source from [18] and decompress it. Insert simulator directory and run *make* command without any parameters. It builds the simulator and if the paths in *src/Makefile* are correctly set it copies it into prepared file system (only if it is present of course).

The default TCP port which is used to connect of the operator is 5555. If you want to change it do it in file *include/config.h* before compilation.

Příloha C - How to download software to BOA

This chapter describes step by step the way how to run the simulator at the BOA5200 computer. First of all it is necessary to compile Linux kernel and create the file system which includes the simulator software. Linux has to be set to run the simulator after startup. Then it has to be downloaded to the BOA's *flash* and the boot program has to be configured to start Linux.

C.1 Building Linux kernel and file system

Visit [15] and download Linux kernel version 2.6. The proceed building according to the given instructions. No module installation is necessary and you do not have to run *menuconfig* before building. At the end copy the file *zImage.elf* into the directory of your TFPT server.

The prepared file system can be downloaded from [16]. To insert simulator to the file system copy file *lti_sim* to the *MPC5200_root_flash/usr/lti_sim/* directory and library *libcanfestival_can_socket.so* to *MPC5200_root_flash/usr/lib*. Then build the file system to get its JFFS2 form. To do this you have to have *mkfs.jffs2* from the BOA computer CD installed. Run this command:

```
mkfs.jffs2 -r <directory>/MPC5200_root_flash -o <tftp directory>/myfs.jffs2 -b
-e 0x10000
```

It makes the file system and stores it into your TFTP directory.

C.2 Setting up the Redboot

Redboot is program which is stored in BOA's *flash* and is started after power supply switch on. It activates computer peripherals such as Ethernet connection and run the script which id defined by user. It serves to start software like operating system.

For running the Linux we use it is necessary to have Redboot release at least 2007_02_01. If there is older version of Redboot at your BOA computer, you have to update it according to [13].

For the first time you have to connect to BOA using RS232 link. If no boot script is defined the Redboot console starts at the serial. Redboot configuration is performed by *fconfig* command [14]. Recommended configuration to use BOA to run the simulator:

```
Redboot> fconfig
Run script at boot: true
Boot script:
 fis load Linux
 exec -c "root=/dev/mtdblock2 rw rootfstype=jffs2 ip=dhcp"
Boot script timeout (1000ms resolution): 5
Use BOOTP for network configuration: true
Default server IP address: IP address of computer with TFTP serve
FEC Network hardware address [MAC]: 0x00:0x00:0x00:0x00:0x00:0x02
GDB connection port: 9000
Force console for special debug messages: false
Network debug at boot time: false
Update RedBoot non-volatile configuration - continue (y/n)? y
Redboot> reset
```

This configuration expects usage of DHCP server in the network which it is connected to. The operator's application connects to the simulator and has to know its IP address. So that DHCP server has to assign the permanent IP address to BOA. All DHCP servers allow the administrator to bind IP address with MAC address (00:00:00:00:00:02 in this case) or with the port of the service (5555 in this case).

C.3 Downloading Linux to BOA

This supposes previous two chapters having done and Ethernet network to be connected. You have to connect to BOA using RS232 and let Redboot boot. Break startup of the boot script by pressing *CTRL+C*. To download prepared Linux to BOA's flash follow these steps:

```
Redboot> fis init       // format the flash
Redboot> mfill -b 0x100000 -l 0x800000 -p 0xffffffff
Redboot> load -r -v -b 0x100000 /myfs.jffs2
Redboot> fi cr JFFS2 -l 0x800000
Redboot> load -v -b 0x800000 zImage.elf
Redboot> fi cr Linux
Redboot> reset
```

Now BOA should start, run the boot script and boot Linux. After startup Linux starts LTI simulator and Telnet server is activated as well, so that it is possible to connect to BOA using Telnet instead of RS232 connection.

Příloha D - How to use CanFestival

D.1 Obtaining and building CanFestival

CanFestival driver source can be downloaded from [6]. The primary way how to do it is to check out the source directory from CanFestival CVS by these steps:

```
cvs -d:pserver:anonymous@lolitech.dyndns.org:/canfestival login
(type return, without entering a password)
The system will respond:
Logging in to :pserver:anonymous@lolitech.dyndns.org:2401/canfestival
Then, enter:
cvs -z3 -d:pserver:anonymous@lolitech.dyndns.org:/canfestival co -P
CanFestival-3
```

Then insert CanFestival folder and run this commands:

```
./configure --timers=unix --can=socket --cc=powerpc-603e-linux-gnu-gcc --
arch=ppc --os=linux --target=unix
make
make install
```

Building of CanFestival will produce some libraries and copy them to */usr/powerpc-603e-linux-gnu/lib* folder.

D.2 Setting up local CANopen node

CanFestival source includes tool called *objdictedit*. This program has graphical user interface and serves to creating configuration of the CANopen node of application we develop. It is started by command *objdictedit*. Using this program you can create the object dictionary for the CanFestival. Mainly it is necessary to set up all SDO and PDO, the node has to send and receive. To each PDO you have to define more things - PDO receive or transmit, variable to be mapped and variable mapping. The variable will then be used in application code to exchange data between CanFestival driver and application. If you set up PDOs here they will be sent and received automatically and the values will be stored into mapped variable. Setting up SDO means that you can use it in your application, but if you want to send it, it has to be done in code.

After creating this basic configuration you have to build the dictionary which results into to files with extensions .h and .c. You have to link these files with your application.

D.3 Initializing and starting CanFestival

To explain how to use CanFestival I include some code examples. It supposes node configuration generated by *objdictedit* which name is *canopen*. It is then used as prefix of automatically generated functions and variables (*canopen_Data* is for example object dictionary used by CanFestival).

Header *canfestival.h* has to be included. For start using CanFestival it is necessary to load dynamic library with CAN bus driver API created while compiling CanFestival source. In the example SocketCan driver is used to access CAN bus.

```
if(!LoadCanDriver("libcanfestival_can_socket.so")) {
   exit -1;
}
```

Then you have to open CAN device. Parameters of the communication have to be prepared in structure of type *s_Board*. In this case CAN is set to use device can1 and baudrate 1Mbps. Handler of the bus is then stored to variable of type *CAN_HANDLE*.

```
/**
 * CAN board definition.
 * Device name can1, baudrate 1M
 */
s_BOARD canopen_board = {"1", "1000"};

CAN_HANDLE canopen_handle;

if(!(canopen_handle = canOpen(&canopen_board, &canopen_Data))) {
  exit -1;
  }
```

CanFestival implements callbacks to some events like change of state or message reception. There are pointers to callback functions in *canopen_Data* structure. You can assign your own functions to these pointers or leave them unused.

```
canopen_Data.initialisation = canopen_initialisation;
canopen_Data.preOperational = canopen_preOperational;
canopen_Data.operational = canopen_operational;
canopen_Data.post_sync = canopen_post_sync;
canopen_Data.post_TPDO = canopen_post_TPDO;
canopen_Data.stopped = canopen_stopped;
```

After setting up these few things you have to call function *StartTimerLoop* which initializes CanFestival timers. As an argument you have to insert pointer to the first function you want to proceed after starting the timers.

```
StartTimerLoop(&initNode);
```

Here is an example of such a function. It sets local node ID to 1 and bring the node to *operation* state. Callback functions are called after changing state if used.

```
void initNode(CO_Data * d, UNS32 id)
{
  setNodeId(&canopen_Data, 0x01);
  setState(&canopen_Data, Initialisation);
  setState(&canopen_Data, Operational);
}
```

D.4 Stopping CanFestival

After finishing work with CANopen it is good to stop the communication and timers and close the device. It is done by these few functions.

```
setState(&canopen_Data, Stopped);
StopTimerLoop();
canClose(&canopen_Data);
canopen_handle = NULL;
```

D.5 Writing to Object dictionary

CANopen devices and all network services of each node are configured by writing to device object dictionary (OD). It is different while writing to local OD or OD of some network device.

D.5.1 Writing to local OD

Function *WriteLocalDict* is used for writing to OD of local node managed by CanFestival.

```
UNS32 writeLocalDict( CO_Data* d,
          UNS16 wIndex,
          UNS8 bSubindex,
          void * pSourceData,
          UNS8 * pExpectedSize,
```

```
            UNS8 checkAccess);
```

Meaning of function arguments:

- **d** - pointer to local object dictionary structure
- **wIndex** - the index in the object dictionary where you want to write an entry
- **bSubindex** - the subindex of the Index. e.g. mostly subindex 0 is used to tell you how many valid entries you can find in this index. Look at the canopen standard for further information
- **pbSourceData** - pointer to the variable that holds the value that should be copied into the object dictionary
- **pExpectedSize** - pointer to variable with size of the data to be written
- **CheckAccess** - if other than 0, do not read if the data is Read Only or Constant

D.5.2 Writing to network OD

It is necessary to use SDO service for writing data to OD of some network device. Each CANopen device has at least one SDO server. It means that it is able to receive SDOs with one ID and answer by SDOs with another ID. It is usual that the server listen for SDOs with ID 0x600 + ID of the node and transmits SDOs of ID 0x580 + node ID. Each SDO we want to use has to be defined while creating node configuration by *objdictedit*. Then the SDO is sent by calling appropriate function. The best is to use function *k* which syntax is shown below.

```
UNS8 writeNetworkDictCallBack (CO_Data* d,
                 UNS8 nodeId,
                 UNS16 index,
                 UNS8 subIndex,
                 UNS8 count,
                 UNS8 dataType,
                 void *data,
                 SDOCallback_t Callback);
```

Meaning of function arguments:

- **d** - pointer to local object dictionary structure
- **nodeId** - ID of the device we want to send SDO
- **index** - the index in the object dictionary where you want to write an entry
- **subIndex** - the subindex of the Index. e.g. mostly subindex 0 is used to tell you how many valid entries you can find in this index. Look at the canopen standard for further information
- **count** - number of bytes to be written
- **dataType** - type of the data, use 0 for integers, real numbers and other values
- **data** - pointer to the data
- **Callback** - pointer to a function which is called after finishing the transfer

The SDO service is by definition confirmed. It means that SDO server sends response with result of the transfer after each SDO reception. This response has to be read by client to be sure that the data were written correctly. The result should be read in your function which pointer is given to the function *writeNetworkDictCallBack* as parameter *callback*. The result of SDO transfer in such a function is read by CanFestival function *getWriteResultNetworkDict*. This function has to be called after each SDO transmission because it releases line used to transfer. Header of this function is:

```
UNS8 getWriteResultNetworkDict (CO_Data* d, UNS8 nodeId, UNS32 * abortCode);
```

And meaning of function arguments is:

- **d** - pointer to local object dictionary structure
- **nodeId** - ID of the device we have sent SDO
- **abortCode** - pointer to the variable where error code is written in case of error

The function can return one of these values according to the SDO transfer state:

- SDO_FINISHED - data is available
- SDO_ABORTED_RCV - Transfer failed. (abort SDO received)
- SDO_ABORTED_INTERNAL - Transfer failed. Internal abort.
- SDO_DOWNLOAD_IN_PROGRESS - Data not yet available

The function can be used in cycle waiting while return value is SDO_DOWNLOAD_IN_PROGRESS. But the better solution is calling it in callback function from *writeNetworkDictCallBack*. The callback is call after finishing the transfer.

D.6 Using PDO service

Using PDO service for data exchange is very easy in CanFestival. The most usual transmission type of PDOs is that they are transmitted after SYNC message. In this case it just has to be set up in *objdictedit* configuration and then it works, reads and stores values into mapped variables. After each PDO reception or transmission *post_TPDO* callback is called if used.

Sometimes it is necessary to send PDO from code without SYNC message reception. This is very inconsistent in CanFestival version 3. There are some functions which should solve this situation but they are commented and some other solution is promised to be done in the future. For now I have done it by using function *sendPDO* which syntax is:

```
UNS8 sendPDO (CO_Data* d, s_PDO pdo, UNS8 request);
```

And meaning of function arguments is:

- **d** - pointer to local object dictionary structure
- **pdo** - structure with PDO message
- **request** - REQUEST (request for sending PDO) or NOT_A_REQUEST (normal PDO)

PDO message has to be created using structure s_PDO which meaning is clear:

```
typedef struct struct_s_PDO {
   UNS32 cobId;         /* COB-ID */
   UNS8 len;            /* Number of data transmitted (in data[]) */
   UNS8 data[8];  /* Contain the data */
} s_PDO;
```

D.7 Generating SYNC message

If you want to set up local node to become the SYNC server you have to write some information to local OD. Here is an example of function which set up the SYNC messages generation:

```
/**
 * This function initializes local canopen node to send SYNC message
 * with given period.
 * @param d - local object dictionary
 * @param period - period of the SYNC message in us
 */
void synchro_setup(CO_Data *d, unsigned long period)
{
   UNS32 SYNC_COBID = 0x40000080;
   UNS32 SYNC_INTER = period;
   UNS8 size = sizeof(UNS32);
   writeLocalDict(d, 0x1006, 0x0, &SYNC_INTER, &size, RW);
   writeLocalDict(d, 0x1005, 0x0, &SYNC_COBID, &size, RW);
}
```

According to CANopen specification ID of the SYNC message is 0x80.

After the setup it can be started by command *startSYNC(CO_Data *d)* and stopped by command *stopSYNC(CO_Data *d)*.

D.8 SDO and PDO example

At the end I attach example of a function (*cpd_start()*) which starts some device communicating by CANopen. The ID of the device is given by the parameter. The other function (*cpd_checkSDO_start()*) is callback used to handle SDO transfer result. It is necessary to wait until one SDO transfer is finished before starting some other. Global variable *cpd_init_step* is used for counting actual step of SDO transfer. If some SDO transfer fails it is set to -1 and the startup process is terminated.

```
/**
 * This function is given to the writeNetworkDict function as a callback.
 * It checks the result of SDO transmition and after finishing the transmition
 * it closes the transfer.
 * @param d - local object disctionary
 * @param nodeID - ID of the node which the local node is communicating with
 */
void cpd_checkSDO_start(CO_Data* d, UNS8 nodeId)
{
   UNS32 abortCode;
   if(getWriteResultNetworkDict (d, nodeId, &abortCode) != SDO_FINISHED) {
      cpd_init_step = -1;
   }
   closeSDOtransfer(&canopen_Data, nodeId, SDO_CLIENT);
   cpd_start(&canopen_Data, nodeId);
}

/**
 * This function switch canopen device into operational state.
 * @param d - local object dictionary
 * @param nodeId - ID of the device
 * return 0 - ok, -1 - failed
 */
int cpd_start(CO_Data *d, UNS8 nodeId)
{
   UNS8 data8;
   UNS16 data16;
   UNS32 data32;
   switch(cpd_init_step++) {
      case 0:
         masterSendNMTstateChange(d, nodeId, NMT_Start_Node);
         s_PDO pdo;
         cpd_controlword = 0;
         pdo.cobId = 0x200 + nodeId;
         int i = 0;
         for(i = 0; i < 8; i++) {
            pdo.data[i] = 0;
         }
         pdo.data[0] = 0;
         pdo.data[1] = 0;
         pdo.len = sizeof(pdo.data);
         sendPDO(d, pdo, NOT_A_REQUEST);
         cpd_controlword = 6;
         pdo.cobId = 0x200 + nodeId;
         for(i = 0; i < 8; i++) {
            pdo.data[i] = 0;
         }
         pdo.data[0] = 7;
         pdo.data[1] = 0;
         pdo.len = sizeof(pdo.data);
         sendPDO(d, pdo, NOT_A_REQUEST);
         cpd_controlword = 0x0F;
         pdo.cobId = 0x200 + nodeId;
         for(i = 0; i < 8; i++) {
```

```
            pdo.data[i] = 0;
         }
         pdo.data[0] = 0xF;
         pdo.data[1] = 0;
         pdo.len = sizeof(pdo.data);
         sendPDO(d, pdo, NOT_A_REQUEST);

         // Start operational mode
         data8 = 0xFC;
         writeNetworkDictCallBack(d, nodeId, 0x6060, 0x00, 1, 0, &data8,
cpd_checkSDO_start);
         break;
      case 1:
         // Check operation status
         readNetworkDictCallback(d, nodeId, 0x6061, 0x00, 0,
cpd_checkSDO_start);
         break;
      case 2:
         // Setpoint specification
         data16 = 0x0002;
         writeNetworkDictCallBack(d, nodeId, 0x301B, 0x11, 2, 0, &data16,
cpd_checkSDO_start);
         break;
      case 3:
         canopen_init_result = 0;
         cpd_init_step = 0;
         break;
      case -1:
         canopen_init_result = -1;
         cpd_init_step = 0;
         break;
   }
   return 0;
}
```

D.9 Emergency message

CanFestival version 3 does not support reception of emergency messages. I have worked it around by adding callback into CanFestival source just into function reading data from CAN bus. Function *canReceive_driver* is placed in file *<canfestival_directory>/drivers/can_socket/can_socket.c* (for SocketCan driver only!). I have created pointer to a function which is set by application.

```
void (* emcy_callback)(Message *);
```

This code filters messages being received and all messages which ID agrees with EMCY service (0x80; 0x180) are sent to application function set as callback.

```
if(m->cob_id.w >= 0x81 && m->cob_id.w < 0x180) (*emcy_callback)(m);
```

Příloha E - TH protocol example

The simulator is completely controlled via TCP connection using TH protocol. The specification of all messages is in document [10]. For easy understanding I have created an example which describes setting up the simulator and starting simulation step by step, message by message.

E.1 Simulation task

Imagine that the aim is to simulate system which state space representation of mathematical description is:

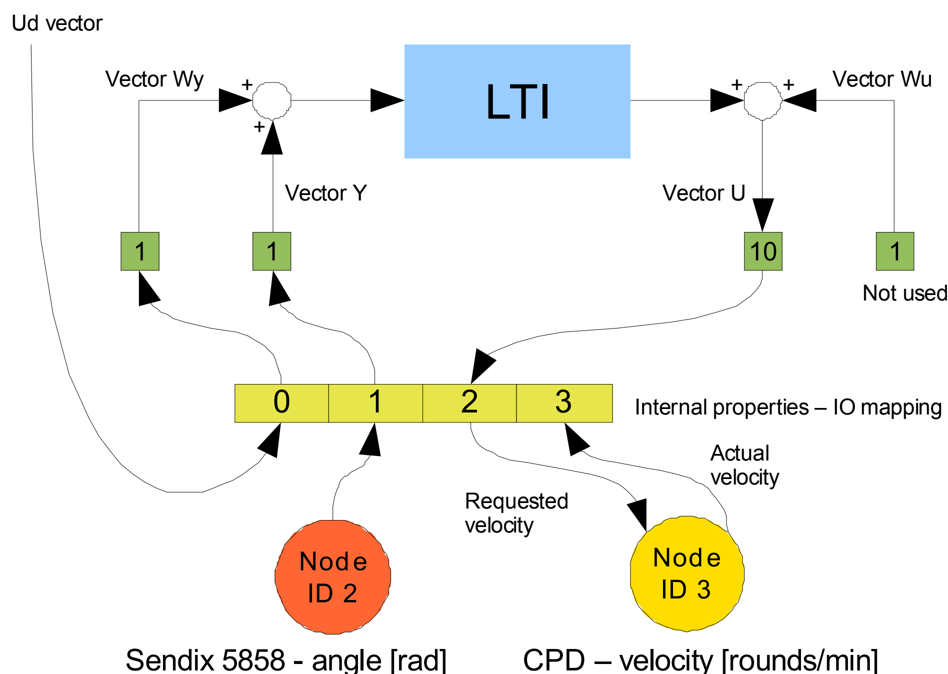
$$M = \begin{bmatrix} 0.8 & 0 \\ 0 & -0.9 \end{bmatrix}$$

$$N = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

$$D = \begin{bmatrix} 0 \end{bmatrix}$$

Its input will be angle of sensor Sendix 5858 connected via CANopen. ID of this device in CANopen network is set to 2. Output of the model is representing velocity of a motor and will be send to CPD control unit connected again via CANopen. Its ID is set to 3. The value of the output has to be multiplied by 10 for some reason. We want to know the real value of motor velocity in all the steps of simulation. To do this we have to set CPD as input device as well. To simulate some error behavior we want to allow changing input value by some constant in the real time. To do this we need to define Ud vector and set it up so that its value will be added to LTI input vector. Requested configuration is shown in the picture.



Obrázek 12 - Simulator configuration used in example

E.2 System configuration

After connecting to the simulator the TCP connection is open and simulator is awaiting TH messages. First configuration message we sent has to be message with LTI model definition. In this message just after ID is transmitted decimal number 123 which is used to endian detection.

Field name	Data type	Field length [B]	Field value
ID 1st byte	char	1	2
ID 2nd byte	char	1	1
Endian setup	32-bit integer	4	123
Order	char	1	1
Inputs number	char	1	1
Outputs number	char	1	1
Matrix M (1, 1)	double	8	0.8
Matrix M (1, 2)	double	8	0
Matrix M (2, 1)	double	8	0
Matrix M (1, 2)	double	8	-0.9
Matrix N (1, 1)	double	8	1
Matrix N (2, 1)	double	8	0
Matrix C (1, 1)	double	8	1
Matrix C (1, 2)	double	8	0
Matrix D (1, 1)	double	8	0
Initial state (1)	double	8	0
Initial state (2)	double	8	0
Total length		**97**	**bytes**

Tabulka 3 - LTI configuration message

After transmission of LTI model setup we have to configure IO devices and IO vectors of the model, but it does not matter in which order. Let's start with vector initialization. Vector structure is fixed and shown in the picture. In configuration we can assign each vector an internal variable. The same variable will be then assigned to the device we want to connect to this vector.

Setup of all vectors is performed by the same message. Concrete vector identification is done by setting vector type field. Vector type 2 identifies vector Y. It is mapped onto variable 1.

Field name	Data type	Field length [B]	Field value
ID 1st byte	char	1	3
ID 2nd byte	char	1	1
Vector type	char	1	2
Vector length	char	1	1
Variable	char	1	1
Multipl. const.	double	8	1
Total length		**13**	**bytes**

Tabulka 4 - Vector Y initialization

Vector U is mapped onto variable 2 and is multiplied by 10.

Field name	Data type	Field length [B]	Field value
ID 1st byte	char	1	3
ID 2nd byte	char	1	1
Vector type	char	1	0

Vector length	char	1	1
Variable	char	1	2
Multipl. const.	double	8	10
Total length		**13**	**bytes**

Tabulka 5 - Vector U initialisation

Vector Wy is mapped onto variable 0 where it will be bind with Ud vector.

Field name	**Data type**	**Field length [B]**	**Field value**
ID 1st byte	Char	1	3
ID 2nd byte	char	1	1
Vector type	char	1	3
Vector length	char	1	1
Variable	char	1	0
Multipl. const.	double	8	1
Total length		**13**	**bytes**

Tabulka 6 - Vector Wy initialisation

The fourth vector - Wu is not used so that it can be left not configured. Vector Ud is not vector in the same meaning as the other vectors because it behaves more like input device then logical vector. It serves to sending data from operator’s application to the model. In this case it is mapped onto variable 0 and its data will be used by vector Wy.

Field name	**Data type**	**Field length [B]**	**Field value**
ID 1st byte	char	1	3
ID 2nd byte	char	1	1
Vector type	char	1	4
Vector length	char	1	1
Variable	char	1	0
Multipl. const.	double	8	1
Total length		**13**	**bytes**

Tabulka 7 - Ud vector initialisation

Now the vectors are initialized and the last thing we have to set up are IO devices. Device configuration is done by two messages, one for input devices and one for output devices. Let’s start with input devices. We use sensor of angle and as the second input device is used CPD which measures real velocity of the motor.

Field name	**Data type**	**Field length [B]**	**Field value**
ID 1st byte	char	1	3
ID 2nd byte	char	1	3
Device number	char	1	2
Variable	char	1	1
Device type	char	1	0x20
Additional info (1)	char	1	1
Additional info (2)	char	1	2
Additional info (3)	char	1	0
Additional info (4)	char	1	0
Additional info (5)	char	1	0

Additional info (6)	char	1	0
Additional info (7)	char	1	0
Additional info (8)	char	1	0
Variable	char	1	3
Device type	char	1	0x20
Additional info (1)	char	1	2
Additional info (2)	char	1	3
Additional info (3)	char	1	0
Additional info (4)	char	1	0
Additional info (5)	char	1	0
Additional info (6)	char	1	0
Additional info (7)	char	1	0
Additional info (8)	char	1	0
Total length		**23**	**bytes**

Tabulka 8 - Input devices setup

In the same way output device (CPD) will be configured.

Field name	Data type	Field length [B]	Field value
ID 1st byte	char	1	3
ID 2nd byte	char	1	4
Device number	char	1	1
Variable	char	1	2
Device type	char	1	0x20
Additional info (1)	char	1	2
Additional info (2)	char	1	3
Additional info (3)	char	1	0
Additional info (4)	char	1	0
Additional info (5)	char	1	0
Additional info (6)	char	1	0
Additional info (7)	char	1	0
Additional info (8)	char	1	0
Total length		**13**	**bytes**

Tabulka 9 - Output devices setup

E.3 Running simulation

Now the system is completely configured and waiting for the simulation start message. Let's start the simulation with the period of 10 millisecond and duration 10 second.

Field name	Data type	Field length [B]	Field value
ID 1st byte	char	1	1
ID 2nd byte	char	1	1
Time [s]	double	8	10
Period [s]	double	8	0.010
Monitors [on/off]	char	1	1
Total length		**19**	**bytes**

Tabulka 10 - Simulation start

The last field activates the monitors so that in each simulation step array of internal variables will be sent to the operator application. After receiving start message the simulation starts and state message is sent to operator. According to the definition it says that the state is *simulation_in_progress* and info message means *simulation_started*.

Field name	Data type	Field length [B]	Field value
ID 1st byte	char	1	1
ID 2nd byte	char	1	3
State	char	1	2
Mess. Type	char	1	1
Message	char	1	2
Additional info (1)	char	1	0
Additional info (2)	char	1	0
Additional info (3)	char	1	0
Additional info (4)	char	1	0
Additional info (5)	char	1	0
Additional info (6)	char	1	0
Additional info (7)	char	1	0
Additional info (8)	char	1	0
Total length		**13**	**bytes**

Tabulka 11 - State message, simulation started

Because no message with Ud vector value has been received yet, vector Wy is kept in 0. To change its value it is necessary to send message with requested value. Below is an example of the message with value -1.56.

Field name	Data type	Field length [B]	Field value
ID 1st byte	char	1	3
ID 2nd byte	char	1	2
Ud vector value	double	8	-1.56
Total length		**10**	**bytes**

Tabulka 12 - Ud vector value message

One monitor message with all internal variables is sent in each simulation step. Order of the values transmitted is the same as they were mapped onto internal properties. Below is an example of one monitor message.

Field name	Data type	Field length [B]	Field value
ID 1st byte	char	1	4
ID 2nd byte	char	1	1
property 0	double	8	-1.56
property 1	double	8	3.14
property 2	double	8	500
property 3	double	8	501
Total length		**34**	**bytes**

Tabulka 13 - Monitor message

According to vector mapping made in configuration phase the properties have these meanings: 0 - value of Ud vector and Wy vector as well (-1.56), 1 - value of sensor of angle and Y vector (3.14), 2 - output value (vector U) used to control motor velocity (500 turns per minute), 3 - real velocity of the

motor measured by CPD (501). The message length agrees with number of internal properties. Number of internal properties is counted by the system as maximal mapped variable number used in configuration increased by one. In our case we used variables 0, 1, 2 and 3 so that length of internal properties array is 4 and there are 4 values transmitted in monitor message.

After 10 second the simulation is finished. Monitor message transmitting is stopped and state message with information about simulation finish is sent.

Field name	**Data type**	**Field length [B]**	**Field value**
ID 1st byte	char	1	1
ID 2nd byte	char	1	3
State	char	1	1
Mess. Type	char	1	1
Message	char	1	3
Additional info (1)	char	1	1
Additional info (2)	char	1	0
Additional info (3)	char	1	0
Additional info (4)	char	1	0
Additional info (5)	char	1	0
Additional info (6)	char	1	0
Additional info (7)	char	1	0
Additional info (8)	char	1	0
Total length		**13**	**bytes**

Tabulka 14 - State message, simulation finished

Now the message meaning (simulation finished) is supplied with number 1 in first byte of additional info. It means that the simulation finished successfully.

Příloha F - Obsah přiloženého CD

Na přiloženém CD je vytvořena tato adresářová struktura:

/bd/	- tento dokument ve formátu *doc* a *pdf*
/lti_sim/	- zdrojové kódy simulátoru
/Canfestival-3/	- zdrojové kódy použitého ovladače Canfestival
/BOA/	- přeložené jádro Linuxu a systém souborů se simulátorem pro desku BOA
/doc/	- dokumentace k simulátoru a ke všem použitým prostředkům