ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA ELEKTROTECHNICKÁ

Katedra aplikované elektroniky a telekomunikací

DIPLOMOVÁ PRÁCE

MicroCANopen na platformě STM32

Autor práce: Ondřej Herink

Vedoucí práce: Ing. Petr Krist, Ph.D.

Plzeň 2014

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta elektrotechnická
Akademický rok: **2013/2014**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Ondřej HERINK**
Osobní číslo:	**E12N0146P**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Elektronika a aplikovaná informatika**
Název tématu:	**MicroCANopen na platformě STM32**
Zadávající katedra:	**Katedra aplikované elektroniky a telekomunikací**

Zásady pro vypracování:

Pro mikrokontroléry řady STM32F20x na jádře ARM Cortex-M3 a STM32F40x na jádře ARM Cortex-M4 implementujte komunikační protokol MicroCANopen, včetně jeho proprietárních rozšíření a diagnostických služeb, pro řídicí systém zadavatele ZAT a.s. Plzeň. Implementaci realizujte přímo na řídicím hardwarovém modulu firmy ZAT a.s. Plzeň.

1. Seznamte se s architekturou a vývojovými prostředky mikrokontrolérů STM32F20x a STM32F40x.
2. Prostudujte specifikaci komunikačního profilu CANopen DS-301.
3. Definujte datová rozhraní a konfigurační a stavové struktury komunikačního driveru MicroCANopen, v souladu s pravidly systémové exekutivy ZAT.
4. Do systémové exekutivy ZAT implementujte základní funkcionalitu protokolového stacku MicroCANopen, jeho proprietárních rozšíření a diagnostických služeb na základě požadavků řídicího systému ZAT. Projekt důsledně rozdělte do dvou vrstev - hardwarově závislý ovladač CAN řadiče mikrokontroléru a vlastní vrstvu, implementující MicroCANopen protokolové služby. Program realizujte v jazyce C, případně C++.
5. Za účelem integrace do systémové exekutivy ZAT řešte implementaci protokolového stacku MicroCANopen formou periodicky volaného obecného stavového automatu s minimálními časovými prodlevami.

Rozsah grafických prací: **podle doporučení vedoucího**

Rozsah pracovní zprávy: **30 - 40 stran**

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

Seznam odborné literatury:

Student si vhodnou literaturu vyhledá v dostupných pramenech podle doporučení vedoucího práce.

Vedoucí diplomové práce: **Ing. Petr Krist, Ph.D.**
Katedra aplikované elektroniky a telekomunikací

Datum zadání diplomové práce: **14. října 2013**

Termín odevzdání diplomové práce: **12. května 2014**

L.S.

Doc. Ing. Jiří Hammerbauer, Ph.D.
děkan

Doc. Dr. Ing. Vjačeslav Georgiev
vedoucí katedry

V Plzni dne 14. října 2013

Abstrakt

Herink Ondřej. MicroCANopen na platformě STM32. Katedra aplikované elektroniky a telekomunikací, Západočeská univerzita v Plzni – Fakulta elektrotechnická, 2014, vedoucí: Ing. Petr Krist, Ph.D.

Předkládaná diplomová práce se zabývá specifikacemi komunikačního protokolu CAN, popisuje princip sběrnice CAN a zmiňuje se i o elektromagnetické kompatibilitě sběrnice. Druhá část práce je zaměřena na použitý hardware pro komunikační driver MicroCANopen a použití nástrojů pro ladění driveru. Poslední část diplomové práce řeší implementaci komunikačního protokolu MicroCANopen do mikrokontroléru STM32F407.

Klíčová slova

MicroCANopen, sběrnice CAN, komunikační protokol, typy zpráv, napěťové úrovně, budič sběrnice, EMC, analyzátor, mikrokontrolér STM32F407, STM32F4-Discovery kit.

Abstract

Herink Ondřej. MicroCANopen on the STM32 Platform. Department of applied electronics and telecommunications, University of West Bohemia in Pilsen – Faculty of electrical engineering, 2014, head: Ing. Petr Krist, Ph.D.

This thesis deals with the specification of the communication CAN protocol, describes the principle of CAN bus and mentions the EMC. The second part focuses on the hardware used for the communication driver MicroCANopen and using tools for debugging the driver. The last part of the thesis solves implementation of the protocol MicroCANopen to STM32F407 microcontroller.

Key words

MicroCANopen, CAN bus, communication protocol, message types, voltage levels, bus driver, EMC, analyzer, microcontroller STM32F407, STM32F4-Discovery kit.

Prohlášení

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

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně, s použitím odborné literatury a pramenů uvedených v seznamu, který je součástí této diplomové práce.

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

V Plzni dne 19.5.2014 Jméno příjmení

……..

Poděkování

Tímto bych rád poděkoval vedoucímu diplomové práce Ing. Petru Kristovi, Ph.D. za vedení práce a za cenné profesionální rady a připomínky.

Obsah

OBSAH 8

ÚVOD 10

SEZNAM SYMBOLŮ A ZKRATEK 11

1 CAN 13

1.1 PRINCIP ČINNOSTI SBĚRNICE CAN 13
1.2 REFERENČNÍ MODEL ISO/OSI A STANDARD CAN 13
1.3 PRINCIP PŘIPOJENÍ UZLU NA SBĚRNICI 14
1.4 ARBITRÁŽ 15
1.5 BIT STUFFING 16
1.6 DOBA PŘENOSU JEDNOHO BITU 16
1.7 KOMUNIKAČNÍ PROTOKOL 17
1.7.1 Standardní formát CAN 2.0A 18
1.7.2 Rozšířený formát CAN 2.0B 19
1.8 ZABEZPEČENÍ PŘENOSU ZPRÁVY 19
1.9 TYPY ZPRÁV 20
1.10 ČÍTAČE CHYBOVÝCH STAVŮ 21
1.11 PROPOJENÍ SBĚRNICE CAN 22
1.12 NAPĚŤOVÉ ÚROVNĚ SBĚRNICE CAN 23
1.12.1 Napěťové úrovně high-speed 23
1.12.2 Napěťové úrovně low-speed 24
1.13 CAN KONEKTOR 24

2 ELEKTROMAGNETICKÁ KOMPATIBILITA SBĚRNICE CAN 25

2.1 PŮSOBENÍ EMI NA SBĚRNICI CAN 25
2.2 ZAKONČENÍ SBĚRNICE 25
2.3 PŘENOSOVÁ RYCHLOST VS. DÉLKA SBĚRNICE 27
2.4 BUDIČE ODOLNÉ VŮČI EMI A ESD 27
2.5 PROVEDENÍ KABELŮ 27

3 VÝVOJOVÉ A PODPŮRNÉ PROSTŘEDKY 28

3.1 STM32F4-DISCOVERY KIT 28
3.2 BUDIČ SN65HVD230DR 29
3.3 ANALYZÁTOR USB-CAN ADAPTER TRIPLE DRIVERS V4.2 31

4 SOFTWARE 34

4.1 Zdrojový modul mcohw.c 34
4.2 Zdrojový modul can.c 35
4.3 Zdrojový modul mco.c 36
4.4 Zdrojový modul LCD16x2lbr.c 36
4.5 Zdrojový modul stm32f4xx_it.c 37
4.6 Zdrojový modul main.c 37
ZÁVĚR 39
SEZNAM OBRÁZKŮ 40
SEZNAM TABULEK 40
SEZNAM LITERATURY A INFORMAČNÍCH ZDROJŮ 41
PŘÍLOHY 43

Úvod

Předkládaná diplomová práce byla navrhnuta pro potřeby firmy ZAT a.s. Plzeň a popisuje implementaci komunikačního protokolu MicroCANopen do hardwarové platformy Cortex-M4. Práce nejprve vysvětluje základní vlastnosti komunikačního protokolu CAN a objasňuje, jak je to s jeho elektromagnetickou kompatibilitou. Další část práce popisuje vývojový hardware, samotný mikrokontrolér STM32F407 a podpůrné nástroje umožňující ladění a vývoj. V poslední části jsou rozebrány funkce zdrojového kódu a nutné funkce pro implementaci MicroCANopen. V příloze této práce je uveden celý zdrojový kód.

Seznam symbolů a zkratek

ACK	Acknowledge – potvrzovací bit bezchybného přijetí zprávy
CAN	Controller Area Network – sériová datová sběrnice
CRC	Cyclic Redundancy Check – cyklický redundantní součet, jehož funkce se používá k detekci chyb během přenosu
CSMA/AMP	Carrier Sense Multiple Access/Arbitration on Message Priority – metoda přístupu na sběrnici pomocí priority identifikátoru
CSMA/CD	Carrier Sense Multiple Access/Colission Detection – metoda detekce kolizí na sběrnici
DLC	Data Length Code – informuje o délce datového pole v CAN zprávě
DPS	Deska Plošných Spojů
EMC	ElectroMagnetic Compatibility – elektromagnetická kompatibilita
EMI	ElectroMagnetic Interference – elektromagnetická interference (rušení)
EOF	End Of Frame – ukončovací rámec zprávy CAN obsahující sedm bitů
ERC	End of CRC – bit, který ukončuje CRC pole v CAN zprávě
ESD	ElectroStatic Discharge – elektrostatický výboj
FIFO	First In, First Out – (první dovnitř, první ven), paměť fronty
GND	GrouND – zemnící potenciál, referenční bod
ID	IDentifier – identifikátor rámce CAN zprávy
IDE	IDentifier Extention – bit v rámci CAN zprávy určující formát protokolu
INT	INTermission – tří bitová pauza na sběrnici před vysíláním další zprávy
ISO/OSI	International Organization for Standardization/Open Systems Interconnection – referenční model, kde jednotlivé vrstvy jsou nezávislé a snadno nahraditelné
LCD	Liquid Crystal Display – displej z tekutých krystalů
LED	Light-emitting diode – polovodičová součástka vyzařující světlo
LSB	Least Significant Bit – nejméně významný bit
MEMS	Micro-Electro-Mechanical Systems – elektronické mikro-mechanické prvky
MSB	Most Significant Bit – nejvýznamnější bit
OTG	On-The-Go – pracuje jako hostitel/periferie nebo obráceně
RTR	Remote Transmision Request – bit určující, zda CAN správa obsahuje data, nebo o ně žádá

Rx	Receive – označení signálu pro příjem
SJW	Synchronization Jump Width – programovatelná šířka synchronizačního skoku pro prodloužení či zkrácení šířky jednoho bitu
SWD	Serial Wire Debug – ladící rozhraní
SOF	Start Of Frame – první bit označující začátek rámce CAN zprávy
I^2C	Inter-Integrated Circuit – multi-masterová počítačová sériová sběrnice
SRR	Substitute Remote Request – bit, který nahrazuje RTR bit v původním standardním CAN 2.0A rámci
TTL	Transistor-Transistor Logic – tranzistorově-tranzistorová logika, standard vycházející z technologie bipolárních tranzistorů a jejich napájení
Tx	Transmit – označení signálu pro vysílání
USB	Universal Serial Bus – univerzální sériová sběrnice

1 CAN

Sériová datová sběrnice CAN (Controller Area Network) byla vyvinuta koncem osmdesátých let firmou Robert Bosch GmbH. Původně byla určena pro automobilový průmysl, kde komunikace probíhala mezi řídícími jednotkami, senzory a výkonovými prvky. Velký důraz byl kladen na malý počet vodičů (Sběrnice využívá pouze dva vodiče, zemnící potenciál je zapojen na kostru automobilu.) a odolnost proti rušení. Sběrnice se začala stále více rozšiřovat do dalších systémů, a tak získala svoji vlastní mezinárodní normu ISO 11898. Dnes se sběrnice CAN používá nejen v automobilovém průmyslu, ale uplatňuje se také v průmyslových aplikacích, zabezpečovacích nebo distribuovaných systémech.

1.1 Princip činnosti sběrnice CAN

Na jedné sběrnici může být připojeno několik uzlů (jednotek). V této topologii se uzly nerozdělují na master a slave jednotky, ale všechny uzly na sběrnici jsou si rovné. Neexistuje adresování zprávy pro určitý uzel. Libovolný uzel může vyslat zprávu – multimaster režim. Vysílanou zprávu přijme každý uzel na sběrnici. Jedná se o tzv. Broadcast zprávy. Přijatá zpráva pak musí projít přes přijímací filtr, který rozhodne, zda zpráva bude dále zpracována. Filtr každého uzlu může být nastaven pro jiné identifikátory. Na základě identifikátoru se určuje nejen obsah zprávy, ale hlavně její priorita. Identifikátor má vyšší prioritu, čím nižší má binární hodnotu. Uzel dokáže detekovat chybu při odesílání zprávy a pokusí se o její znovu vysílání. Poškozený uzel se dokáže sám odpojit od sběrnice.

1.2 Referenční model ISO/OSI a standard CAN

Obr. 1: Referenční model ISO/OSI

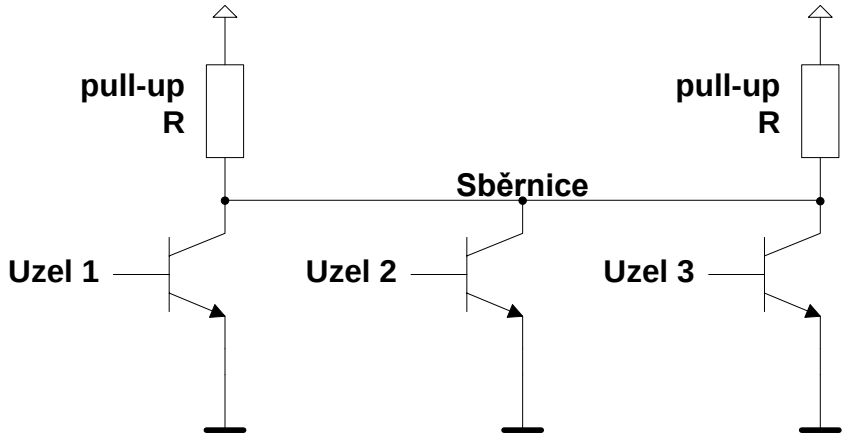
Na obr. 1 je obecný referenční model ISO/OSI. Komunikační standard CAN implementuje jen první dvě vrstvy a to jsou fyzická vrstva a linková vrstva. Nad těmito vrstvami je už jen aplikační rozhraní.

Konkrétně u standardu CAN fyzická vrstva definuje přenosové médium. Nejčastěji se používá kroucená dvojlinka, ale samozřejmě nic nebrání tomu, aby standard CAN používal jako přenosové médium optické vlákno nebo bezdrátový přenos. V každém případě přenosové médium musí být schopné přenášet dva logické stavy a to dominantní (log. „0“) a recesivní (log. „1“). Dále fyzická vrstva definuje bitové kódování a dekódování signálu, časování a s tím spojenou synchronizaci, elektrické úrovně signálu a konektory.

Linková vrstva se stará o řízení přístupu na sběrnici, vytváření a dekódování rámců pro data, zabezpečení dat, řízení rámce, filtrování příchozích zpráv, potvrzování bezchybného příjmu zpráv, detekci a signalizaci chyb. Všechny definované položky standardu CAN budou dále popsány a vysvětleny. [13]

1.3 Princip připojení uzlu na sběrnici

Všechny uzly si jsou na sběrnici rovnocenné. Pokud se na sběrnici nevysílá, tak v libovolný okamžik může jakýkoliv uzel začít vysílat. Každý uzel je na sběrnici připojen metodou otevřeného kolektoru. Na obr. 2 je znázorněno blokové schéma propojení uzlů se sběrnicí. Jednotlivé uzly zde principielně představují tranzistory.

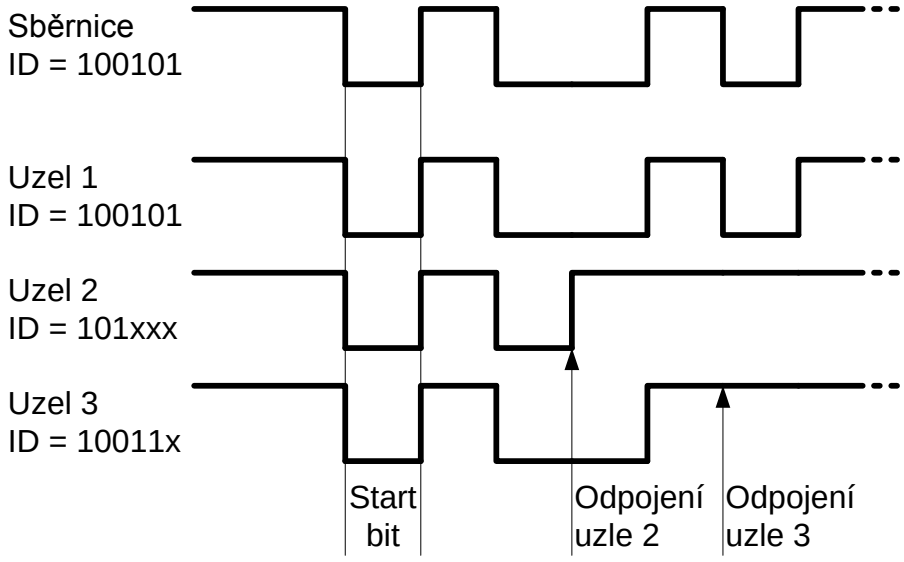


Obr. 2: Připojení uzlů na sběrnici

Z principu otevřeného kolektoru vyplývá, že pokud není žádný tranzistor otevřený (tzn., žádný uzel se nesnaží vysílat a sběrnice je v klidovém stavu), tak na sběrnici je logická „1“ neboli recesivní stav a to díky pull-up odporu. Stačí, aby se jeden tranzistor otevřel (uzel začne vysílat) a na sběrnici se objeví logická „0“ neboli dominantní stav. Takže jakýkoliv uzel je schopen na sběrnici vyvolat dominantní stav. [12]

1.4 Arbitráž

Jako přístupová metoda arbitráže se využívá CSMA/AMP (Carrier Sense Multiple Access/ Arbitration On Message Priority). Díky této metodě se nemůže stát, že by dva uzly vysílaly data na jedné sběrnici současně. Vždy vysílá data jen jeden uzel a to takový, jehož identifikátor má vyšší prioritu neboli nižší binární hodnotu. Každý uzel, respektive jeho řadič ví, jaký bit odeslal na sběrnici a současně má schopnost detekovat aktuální stav sběrnice. Takže když na sběrnici pošle recesivní bit, ale ve skutečnosti je na sběrnici dominantní stav, okamžitě přestane vysílat a „odpojí se“ od sběrnice. Této použité metodě se říká CSMA/CD (Carrier Sense Multiple Access/ Colission Detection). Po nezdařilém pokusu o vysílání se uzel po určité (náhodné) době pokusí vysílat znovu. Nevýhodou je, že se nedá zjistit doba zpoždění přenosu zprávy.



Obr. 3: Řešení konfliktů na sběrnici

Na obr. 3 je znázorněno řešení konfliktu. Uzel 1 má potřebu vyslat zprávu. Nejprve tedy detekuje klidový stav na sběrnici a poté začíná vysílat. První tedy odešle start bit, následuje identifikátor. V ten samý okamžik však začaly vysílat uzel 2 a uzel 3. Tím vznikl konflikt na sběrnici. První dva bity identifikátoru mají všechny tři uzly stejný. Třetí bit identifikátoru uzlu 2 se mění do recesivního stavu, na sběrnici však zůstává dominantní stav. Nyní uzel 2 zjistil, že současně s ním vysílá jiný uzel s vyšší prioritou, tím pádem se musí odpojit. Nyní jsou ke sběrnici připojené už jen uzly 1 a 3, konflikt ale stále pokračuje. Uzel 3 chce jako pátý bit identifikátoru odeslat v recesivním stavu, na sběrnici se ale objeví dominantní stav. Nezbývá mu nic jiného, než se odpojit od sběrnice. Uzel 1 má z těchto tří uzlů nejvyšší prioritu a tak může odvysílat svou zprávu. Z logiky a výsledného stavu sběrnice vyplývá, že arbitráž

splňuje funkci logického součinu vysílaných signálů. [12][14]

1.5 Bit stuffing

Sběrnice CAN nevyužívá k synchronizaci hodinový signál, jako je to například u sběrnice I^2C, ale využívá tzv. bitovou synchronizaci. Přijímač uzlu je vybaven fázovým detektorem, který po přijetí několika bitů dokáže zjistit komunikační rychlost a bitově synchronizovat komunikaci. Správná funkce fázového detektoru je podmíněná tím, že logické stavy bitů se musejí neustále měnit, aby se fázový závěs mohl „dolaďovat“. To při komunikaci nelze zaručit. Může nastat situace, kdy se má ve zprávě odeslat sekvence několika stejných bitů. Fázový závěs by se mohl rozladit. Proto se do sekvence stejných bitů zavádí bit stuffing. Situaci naznačuje obr. 4.

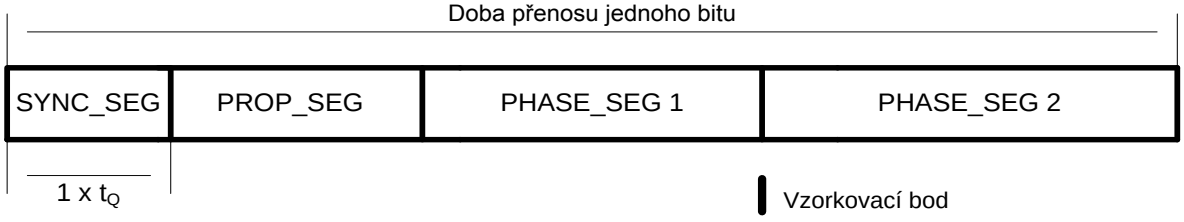
Vysílaná sekvence stejných bitů	0	1	1	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0	0	1	0	1
Vyslaná sekvence s bit stuffingem	0	1	1	0	1	1	1	1	**1**	**0**	1	0	0	1	0	0	0	0	0	**0**	**1**	0	1	0	1
									bit stuffing											bit stuffing		

Obr. 4: Bit stuffing

V první sekvence stejných bitů v obrázku je tvořena log. „1“. Po pátém bitu sekvence řadič vysílacího uzlu automaticky dodá opačný bit (log. „0“). Řadič přijímače odpočítává pět stejných bitů a šestý dodaný bit vyjme. Tímto se neovlivní data a fázový závěs má možnost se doladit. [12]

1.6 Doba přenosu jednoho bitu

Klade se velký důraz na časování sběrnice, protože jak už víme z kapitoly 1.4 o arbitráži, rozhodování o přidělení na sběrnici probíhá v reálném čase bit po bitu. Každý uzel tedy musí být nastaven na správnou přenosovou rychlost a jednotlivé bity uzlů musejí být dokonale synchronizovány. Ideální synchronismus neexistuje, protože sem zasahují vlivy zpoždění signálu, jitter, nebo reálné vlastnosti vedení. Z toho důvodu jsou vylepšeny možnosti konfigurace časování přenosu jednoho bitu.



Obr. 5: Časové segmenty jednoho bitu

Na obr. 5 je znázorněna časová šířka jednoho bitu. Tato šířka je rozdělena na 4 časové segmenty a ty se dále dělí na časová kvanta t_Q. Časové kvantum je odvozeno od kmitočtu oscilátoru hodin řadiče uzlu. Časové segmenty se nastavují podle celých násobků časového kvanta a jednotlivé segmenty mají následující význam:

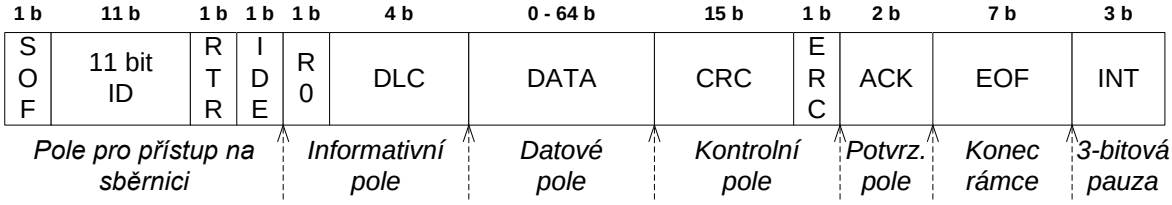
- SYNC_SEG (Synchronization Delay Segment) – Synchronizační zpožďovací segment je vždy roven jednomu časovému kvantu t_Q. V tomto intervalu se očekává hrana signálu.
- PROP_SEG (Propagation Delay Segment) – Zpožďovací segment kompenzuje dobu šíření signálu po sběrnici. Když se sečte dvojnásobný čas potřebný k průchodu signálu sběrnicí s bobami průchodů přes vstupní a výstupní obvody uzlů, získáme zpoždění, které je kompenzováno tímto segmentem.
- PHASE_SEG 1,2 (Phase segment 1, 2) – Mezi PHASE_SEG 1 a PHASE_SEG 2 nastává okamžik vzorkování stavu sběrnice. Fázové segmenty 1,2 se podle potřeby fázového závěsu buď zkracují, nebo prodlužují a to v případě, že se hrana signálu objevila mimo SYNC_SEG. Zkrácení nebo prodloužení se mění o programovatelný počet časových kvant SJW (Synchronization Jump Width). Pokud hrana bitu přijde déle, než přijímač očekává, PHASE_SEG 1 je prodloužen o SJW. Pokud hrana bitu přijde dříve, než přijímač očekává, PHASE_SEG 2 je zkrácen o SJW.

Někdy při nastavování přenosové rychlosti se PROP_SEG vůbec neuvádí a je zahrnut v PHASE_SEG 1. V podstatě se dá nastavit libovolná přenosová rychlost, ovšem maximálně do 1Mbit/s kvůli přísným požadavkům na časování signálů. Nejčastěji se používají přenosové rychlosti: 5; 10; 20; 50; 100; 125; 250; 500; 1000 kbit/s. [12]

1.7 Komunikační protokol

Podle specifikace ISO 11898 CAN používá dva komunikační protokoly v high-speed režimu, které se liší pouze v bitové délce identifikátorů. Standardní formát CAN 2.0A má délku identifikátoru 11 bitů a rozšířený formát CAN 2.0B má identifikátor dlouhý 29 bitů. Podle bitové délky identifikátoru se na sběrnici může přenášet 211 nebo 229 objektů. Na jedné sběrnici se smí používat oba dva formáty zpráv, avšak vyšší prioritu má formát CAN 2.0A. [12]

1.7.1 Standardní formát CAN 2.0A



Obr. 6: Formát standardního rámce CAN 2.0A

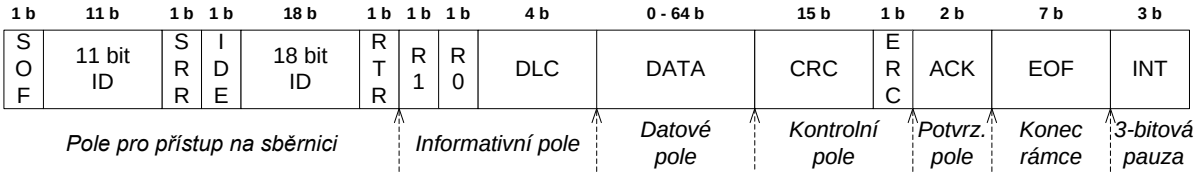
Na obr. 6 je znázorněn standardní formát rámce CAN 2.0A. Jednotlivé pole znamenají:

- **SOF** Start Of Frame – Pokud je sběrnice volná, vyšle se první dominantní bit, který označuje začátek rámce vysílané zprávy. Tento bit slouží k synchronizaci všech uzlů na sběrnici.
- **ID** IDentifier – Identifikátor, který určuje prioritu zprávy. Čím je jeho binární hodnota nižší, tím má vyšší prioritu. To je dáno tím, že log. 0 má dominantní stav. První bit identifikátoru je vždy jako nejvíce významový bit (MSB) a poslední je nejméně významový bit (LSB). V tomto případě je identifikátor 11-ti bitový.
- **RTR** Remote Transmision Request – Tento bit dává informaci o tom, zda zpráva obsahuje data, nebo žádná data neobsahuje a žádá o ně. Pokud se jedná o datový rámec, tento bit je v dominantním stavu (log. 0). V opačném případě (log. 1) zpráva vyžaduje data od jiného uzlu. Od kterého uzlu jsou data vyžadována, to je dáno právě identifikátorem této zprávy.
- **IDE** IDentifier Extention – Bit, který identifikuje formát protokolu. Když je tento bit v dominantním stavu, jedná se o standardní formát CAN 2.0A, naopak rozšířený formát CAN 2.0B indikuje recesivní stav.
- **R0** Reserved – Jedno bitová rezerva pro budoucí využití.
- **DLC** Data Length Code – Toto 4bitové pole informuje o počtu datových Bytů ve zprávě. Maximální počet Bytů obsažených ve zprávě je 8 a minimální počet je 0.
- **DATA** Datové pole může obsahovat 0 - 64 bitů. Záleží na hodnotě v DLC poli.
- **CRC** Cyclic Redundancy Check – Cyklický zabezpečovací 15-ti bitový kód, užitý k detekci chyb při přenosu zprávy. Provádí se kontrolní součet dat pomocí následujícího generujícího polynomu: $G(x) = x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$.
- **ERC** End of CRC – Bit v recesivním stavu, který ukončuje CRC pole.
- **ACK** ACKnowledge – Pole dvou bitů. První bit je vyhrazen pro přijímače a slouží k ověření správnosti přijaté zprávy. Sběrnice se nachází v recesivním stavu, a pokud

se zatím úspěšně přijala zpráva jakýmkoliv uzlem, přepíše se na dominantní stav. Druhý bit odděluje potvrzovací bit a mění sběrnici zpět do recesivního stavu.

- **EOF** End Of Frame – Ukončovací rámec obsahuje sedmi bitové pole a v tomto poli se vysílají samé recesivní bity. Bit-stuffing je z důvodu možnosti hlášení chyby v tomto poli zakázán. V době vysílání ukončovacího rámce mohou přijímače nahlásit chybu tak, že recesivní stav na sběrnici se změní na dominantní. Chyba se týká špatného přijetí zprávy nebo rozdílného kontrolního součtu dat a CRC.
- **INT** INTermission – Před dalším vysíláním zprávy musí být na sběrnici minimálně tří bitová pauza, která zajistí čas pro přijímač právě zpracovávající zprávu. Toto tří bitové pole INT je v recesivním stavu. [12]

1.7.2 Rozšířený formát CAN 2.0B



Obr. 7: Formát rozšířeného rámce CAN 2.0B

Na obr. 7 je znázorněn rozšířený formát rámce CAN 2.0B. Rámec má stejný význam jako standardní rámec, jen se liší o tři rozšiřující pole, které jsou popsány níže.

- **ID** IDentifier – V rozšířeném rámci se nacházejí dvě rozdělené pole pro jeden 29-ti bitový identifikátor. První pole je jedenácti bitové a druhé pole je osmnácti bitové.
- **SRR** Substitute Remote Request – Tento bit je vždy v recesivním stavu, protože standardní rámec má vyšší prioritu před rozšířeným. Standardní rámec na této bitové pozici informuje, zda zpráva obsahuje data (dominantní stav), nebo o data žádá (recesivní stav). O tom, že se jedná o rozšířený rámec, informuje následující bit recesivním stavem.
- **R1** Reserved – Další jedno bitová rezerva pro budoucí využití. [12]

1.8 Zabezpečení přenosu zprávy

Jak již bylo uvedeno výše, zabezpečení datového pole provádí CRC generující polynom $G(x) = x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$. Před odesláním zprávy se spočte polynom a jeho hodnota se uloží do rámce. Při přijímání té samé zprávy se zároveň provádí kontrolní součet datového pole a následně se porovná s přijatým CRC. Pokud je shoda ve výsledku polynomu před odesláním i po příjmu zprávy, zpráva se doručila v pořádku. Pokud se však polynomy

neshodují, přijímač o tom informuje v poli EOF.

O další zabezpečení se stará ACK pole. Sběrnice se v tom okamžiku nachází v recesivním stavu a jakýkoliv přijímač, který přijal zprávu bez chyby, změní stav sběrnice na dominantní. Změna stavu na sběrnici informuje vysílač o tom, že alespoň jedna přijímací stanice přijala jeho zprávu. Pokud se změna stavu na sběrnici neprovede, vysílač se automaticky pokusí vyslat zprávu znovu.

1.9 Typy zpráv

Jak již bylo zmíněno v kapitole 1.7.1, zpráva může obsahovat data (takovéto zprávě se říká Data frame), nebo naopak zpráva o data může žádat (této zprávě se říká Remote frame). Datová zpráva může obsahovat 1-8 Bytů. Kolik přesně Bytů nese zpráva, se dá zjistit z informativního pole DLC, jehož binární hodnota, po převedení na dekadickou, udává přímo počet Bytů. Datovou zprávu si může přečíst kterýkoliv uzel. RTR bit je u datové zprávy v dominantním stavu.

Zpráva žádající o data se pozná tak, že RTR bit je v recesivním stavu, binární hodnota informativního pole DLC je nulová a zpráva neobsahuje žádná data. Jinak struktura zprávy se už v ničem neliší od datové zprávy. Tento typ zprávy žádá data od jiného uzlu. Od kterého uzlu se data vyžadují, je dáno identifikátorem této zprávy. Remote frame zpráva má tedy RTR bit v recesivním stavu a z řešení arbitráže vyplývá, že datová zpráva má větší prioritu, než zpráva, co o data žádá. Nutno ještě dodat, že je to pro případ, kdy se řeší arbitráž dvou uzlů, které ve stejný okamžik vysílají se stejným identifikátorem. Jinak stále platí, že priorita se určuje ze všeho nejdřív podle identifikátoru.

Dalším typem CAN zprávy je zpráva signalizující přetížení a nazývá se Overload frame. Tato zpráva má stejnou strukturu jako chybová zpráva a informuje vysílací uzel o tom, že některý z přijímacích uzlů nestihl zpracovat předchozí zprávu a je tedy přetížen. Zpráva signalizující přetížení však nezpůsobí opakované vysílání nezpracované zprávy. Tento typ zprávy je také vyslán v případě, když ve tříbitovém poli INT (kapitola 1.7.1) se objeví dominantní bit.

Posledním typem zprávy je chybová zpráva nazývaná Error frame. Tato zpráva je odeslána vždy přijímacím uzlem, který při příjmu detekoval nějakou chybu. Může se jednat o chyby jako nepotvrzení ACK, chybný výsledek CRC, neprovedení bit-stuffingu po sekvenci pěti totožných bitů, stav na sběrnici se nerovná aktuálně vysílanému stavu (s výjimkou u pole pro přístup na sběrnici a potvrzovacího ACK pole), je nesprávná polarita polí apod. O diagnostiku chyb se stará komunikační řadič každého uzlu. Error frame je generován

přijímacím uzlem, který detekoval během vysílání chybu CRC a to právě v okamžiku, kdy vysílací uzel vysílá EOF pole. Error frame může být také generován přijímacím uzlem hned po zjištění chyby. Rozlišují se dva chybové rámce:

- Active error frame – formát chybového rámce je šest za sebou jdoucích dominantních bitů (Error flag), následuje osm za sebou jdoucích recesivních bitů (Error delimiter). Tento formát na sběrnici poruší bit-stuffing šesti dominantními bity. Vysílací uzel tedy ví, že některý přijímací uzel detekoval chybu a tak se pokusí zprávu odeslat znovu. Při opakovaném vysílání zprávy může dojít na arbitráž mezi vysílacími uzly a tak by mohla být zpráva opět odeslána až za delší dobu.
- Passive error frame – formát chybového rámce je šest za sebou jdoucích recesivních bitů (Error flag), následuje osm za sebou jdoucích recesivních bitů (Error delimiter). Jelikož se rámec skládá jen z recesivních bitů a ty nepřebijí dominantní bity, tak při odesílání chybového rámce přijímacím uzlem vysílací uzel ani nezaregistruje chybu. Zpráva se už znovu neodešle na rozdíl od předchozího případu. Vysílací uzel, který odešle Passive error frame vyvolá porušení bit-stuffingu a přijímací uzly tak zjistí chybu a odešlou Active error frame.

Vzhledem k tomu, že Error frame může být vyslán jakýmkoliv uzlem a přijme jej každý uzel, tak při některých druzích poruch by mohl na sběrnici vzniknout kolaps a narušila by se komunikace. Proto komunikační řadič musí mít nástroj, který takovéto situace vyřeší. Jakým způsobem funguje takovýto nástroj, je popsáno v následující kapitole. [12]

1.10 Čítače chybových stavů

Chybové stavy jsou zaznamenávány čítačem chyb tzv. Error counter. Každý uzel má dva své čítače chyb pro chyby vzniklé při odesílání zpráv a pro chyby vzniklé při příjmu zpráv. Jeden nebo druhý čítač se při dané chybě inkrementuje o 8 a při správném příjmu nebo odvysílání se dekrementuje pouze o 1. Jeden chybný stav se napraví osmi bezchybnými stavy. Z hlediska hlášení chyb se stavy uzlů rozlišují na:

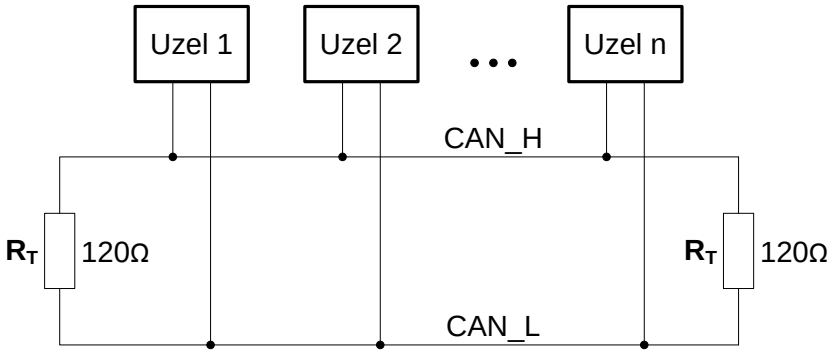
- Aktivní (Active error) – Uzel je po inicializaci v Active error stavu a má právo vysílat chybové zprávy Active error frame. Aktivně se tak podílí na hlášení chyby při její detekci.
- Pasivní (Pasive error) – Pokud čítač chyb uzlu přesáhne hodnotu 127, uzel přejde do Passive error stavu. V tomto stavu smí vysílat jen Passive error frame chybové zprávy, které jsou znevýhodněné, jak bylo popsáno v předchozí kapitole. Active error frame chybové zprávy smí opět vysílat jen za té podmínky, že jeho čítač chyb se dostane pod

hodnotu 128.

- Odpojené (Bus-off) – Pokud čítač chyb přesáhne hodnotu 255, uzel nesmí dále ovlivňovat činnost na sběrnici. Proto řadič uzel odpojí od sběrnice a ten se tak dostane do Bus-off stavu. Aby se uzel mohl znovu připojit na sběrnici, musí projít zotavovací sekvencí. To znamená, že musí reinicializovat svůj řadič, vynulovat chybový čítač a setrvat 128 stavů v nečinnosti. Jeden časový stav odpovídá šířce 11 bitů, takže musí počkat celkem 1408 bitových dob a pak se může znovu připojit na sběrnici.

1.11 Propojení sběrnice CAN

Sběrnice CAN se dá provozovat s jedním přenosovým vodičem, jako je to znázorněno na obr. 2, ale rozhodně častěji se používají dva vodiče. Fyzické propojení uzlů se sběrnicí CAN předvádí obr. 8. Kruhová topologie sběrnice má dva diferenční kroucené vodiče CAN_H a CAN_L. Tyto dva vodiče jsou vůči zemnícímu potenciálu nesymetrické, ale jsou symetrické navzájem a tím dokážou definovat diferenční napětí. Hlavní výhoda diferenčních vodičů je v tom, že pokud působí na sběrnici elektromagnetické rušení, tak se indukuje na oba vodiče stejně. Diferenční napětí se ale při rušení ve výsledku nemění.



Obr. 8: Obecné uspořádání sběrnice CAN

Vodiče CAN_H a CAN_L jsou na obou koncích zakončeny odpory o hodnotě 120Ω, jimž se říká terminátory. Spolu s dvěma vodiči je také vhodné vést referenční vodič (na obr. 8 není zakreslen), od kterého se odvíjí zemnící potenciál budičů uzlů. Specifikace ISO 11898 říká, že maximální počet uzlů připojených na sběrnici je 30. Některé budiče umožňují připojit i více uzlů. Specifikace ISO 11898 dále říká, že při maximální přenosové rychlosti 1 Mbit/s může být délka sběrnice jen 40 m a odbočka od sběrnice k uzlu smí být dlouhá do 30 cm. Obecně platí, že čím pomalejší je přenosová rychlost, tím může být délka sběrnice i odbočky delší. Pro lepší přehlednost jsou v tab. 1 uvedeny přenosové rychlosti v závislosti na délce sběrnice. [12]

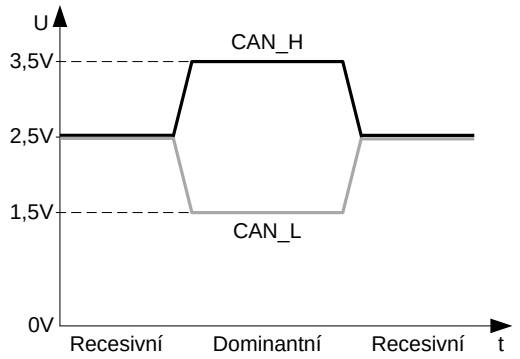
Přenosová rychlost	Maximální délka sběrnice
5 kBit/s	10 km
10 kBit/s	6,7 km
20 kBit/s	3,3 km
50 kBit/s	1,3 km
100 kBit/s	620 m
125 kBit/s	530 m
250 kBit/s	270 m
500 kBit/s	130 m
1 MBit/s	40 m

Tab. 1: Přehled přenosové rychlosti v závislosti na délce sběrnice

1.12 Napěťové úrovně sběrnice CAN

Z CAN řadiče mikroprocesoru jsou vyvedeny dva signály Tx (pro vysílání) a Rx (pro příjem), které jsou ovládány TTL logikou. O převod TTL úrovně signálu na CAN úroveň se stará budič. Budič je tedy mimo mikroprocesor a musí být externě napájen. Podle standardu se napěťové úrovně dělí na high-speed CAN pro vysokorychlostní přenos a low-speed CAN pro pomalejší přenos. Některé budiče zvládnou pracovat s oběma standardy napěťových úrovní. High-speed a low-speed napěťové úrovně jsou mezi sebou nekompatibilní, takže na jedné sběrnici funguje vždy jen jeden druh napěťových úrovní.

1.12.1 Napěťové úrovně high-speed

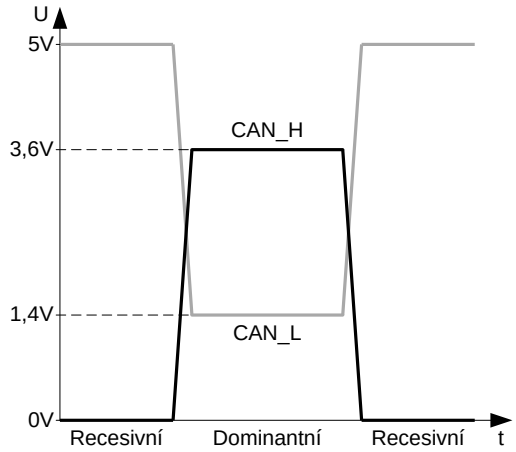


Obr. 9: Napěťové úrovně na sběrnici CAN – high-speed

Obr. 9 ukazuje graf napěťových úrovní v recesivním nebo dominantním stavu při high-speed komunikační rychlosti. Když je na sběrnici recesivní stav, úroveň napětí je na obou vodičích CAN_H a CAN_L stejná. Činí 2,5V a diference mezi vodiči je tedy nulová. Jestliže je na sběrnici dominantní stav, velikost napětí na CAN_H vodiči je 3,5V a na CAN_L vodiči

1,5V. Diferenční napětí je v tomto případě 2V.

1.12.2 Napěťové úrovně low-speed

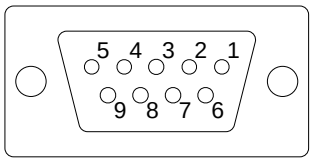


Obr. 10: Napěťové úrovně na sběrnici CAN – low-speed

Obr. 10 popisuje stejný graf jako předchozí obrázek, ovšem pro low-speed komunikační rychlost. Na první pohled je patrné, že průběh napětí je naprosto odlišný. Pokud je sběrnice v recesivním stavu, na vodiči CAN_H je nulové napětí a na vodiči CAN_L je napětí 5V. Nastane-li dominantní stav na sběrnici, vodič CAN_H bude mít v tu chvíli 3,6V proti zemi a vodič CAN_L bude mít 1,4V proti zemi. Diference bude 2,2V.

1.13 CAN konektor

Konektor DE-9 je typický pro sběrnici CAN. Na obr. 11 je znázorněno, jak jsou v konektoru umístěné jeho piny. Female („samice“) konektor je součástí jednotky nebo uzlu a opačný druh konektoru male („samec“) společně s kabelem tvoří sběrnici nebo odbočku na sběrnici. V tab. 2 je popsán význam jednotlivých pinů.



Obr. 11: Konektor DE-9 – male

Pin	Název pinu	Význam pinu
1	–	Rezervovaný
2	CAN_L	vodič CAN_L - dominantní úroveň low
3	CAN_GND	Zemnící potenciál
4	–	Rezervovaný
5	CAN_SHLD	Ochranné stínění
6	GND POWER	Zemnění externího napájení
7	CAN_H	vodič CAN_H - dominantní úroveň high
8	–	Rezervovaný
9	CAN_V+	Externí napájení

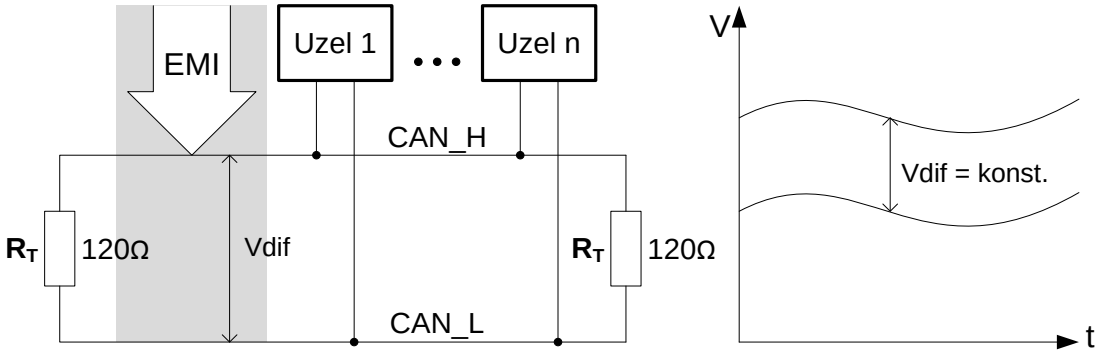
Tab. 2: Popis pinů CAN konektoru DE-9 [1]

2 Elektromagnetická kompatibilita sběrnice CAN

Sběrnice CAN se původně vyvinula pro automobilový průmysl. Jeden z požadavků byl takový, aby se vytvořila sériová sběrnice, která by byla odolná proti elektromagnetickému rušení (EMI) v automobilu. Obecně lze říci, že sběrnice CAN se vyznačuje vysokou provozní spolehlivostí a odolností.

2.1 Působení EMI na sběrnici CAN

Pokud začne na CAN_H a CAN_L vodiče působit elektromagnetické rušení, jako je to znázorněno v obr. 12 a), úroveň napětí může trochu kolísat. Diferenční napětí se však nemění a zůstává stále stejné. Na obr. 12 b) je graf, který předstírá působení EMI a říká, jak se v čase mění úroveň napětí na CAN_H a CAN_L vodičích v dominantním stavu. Přitom diferenční napětí je stále konstantní.

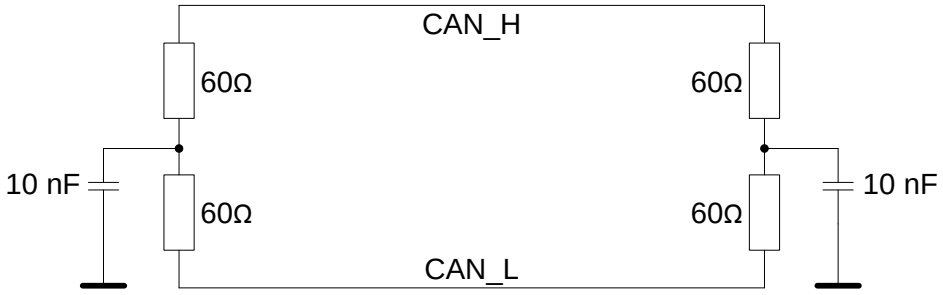


Obr. 12: a) Působení EMI na sběrnici, b) Změna napětí při působení EMI

2.2 Zakončení sběrnice

Elektromagnetická kompatibilita je závislá na správném zakončení sběrnice terminátory. Jsou známé dva základní způsoby ukončení sběrnice. Jeden způsob se používá

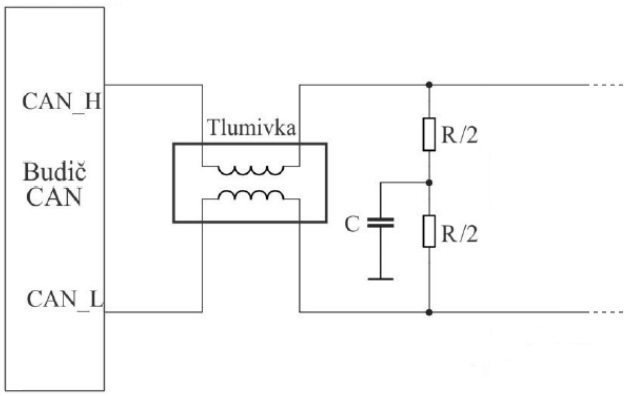
při ne příliš silném EMI a vlastní propojení vypadá jako na obr. 8. Oba konce dvou sběrnicových vodičů jsou spojeny zakončovacími rezistory o hodnotě 120Ω . Nebo je možné 120-ti ohmové rezistory rozdělit na další dva 60-ti ohmové a doplnit mezi ně paralelně kondenzátor, jehož druhý konec se přivede na GND. Hodnota kondenzátoru se volí 10 – 100 nF. Toto zapojení je znázorněno na obr. 13 a je to vlastně druhý způsob ukončení sběrnice, který se používá při silnějším působení EMI.



Obr. 13: Ukončení sběrnice s rozděleným zakončovacím terminátorem

Také odbočovací uzly musejí být zakončené podobným způsobem jako u obr. 13, jen se používají jiné hodnoty zakončovacích rezistorů. Hodnoty se odvíjejí od počtu odbočovacích uzlů, avšak celkové zatížení sběrnice musí splnit rozmezí 50 - 65 Ω. Počet připojených jednotek na sběrnici je pro každou reálnou aplikaci daný, ale platí, že čím méně jednotek je na sběrnici připojeno, tím méně je zdrojů rušení.

Někdy je elektromagnetické rušení tak silné, že se musí udělat ještě větší opatření. Uzly mohou být zdrojem šumu nebo nesymetrií signálu a to přispívá k tlumení společného signálu. Hned za vývody CAN budiče se vloží tlumivka, jak je znázorněno na obr. 14. Vytvoří se tak vysoká impedance mezi vodiči pro společné signály a nízká impedance mezi vodiči pro diferenciální signály. Tlumivka sníží šum a zlepší nesymetrii. Hodnota tlumivky se volí do 0,5 µH.



Obr. 14: Zařazení tlumivky mezi sběrnicové vodiče

2.3 Přenosová rychlost vs. délka sběrnice

Jak již bylo zmíněno, přenosová rychlost může být různá, avšak maximálně do 1 Mbit/s. Čím větší bude délka vodiče, tím bude sběrnice náchylnější na EMI. Snaha je o co nejkratší možnou délku kabelu, aby vyzáření EMI do sběrnice bylo co nejmenší. Pokud by to bylo možné, přenosová rychlost by se měla snížit na co nejmíň, aby vlivem EMI nevznikaly falešné impulsy na sběrnici.

2.4 Budiče odolné vůči EMI a ESD

Volba budiče je také velice důležitá z hlediska elektromagnetické kompatibility (EMC). Kvalitní budič má menší elektromagnetické vyzařování, je odolný vůči elektrostatickým výbojům (ESD) a přechodným jevům, disponuje proudovou a tepelnou ochranou, zabrání výskytu trvalému stavu logické nuly na sběrnici a umí přepínat mezi High-speed, Slope control a Standby režimy. U High-speed režimu se jako přenosové médium doporučuje použít kvalitní stíněný kabel. U Slope control režimu se může využít méně kvalitního kabelu např. nestíněné kroucené dvojlinky.

2.5 Provedení kabelů

Nejčastěji se pro sběrnici CAN používají tři druhy kabelů: Silný kabel, tenký kabel a plochý kabel. Vlastnosti každého kabelu jsou odlišné. V tab. 3 jsou pro každý kabel vypsané specifické vlastnosti. Útlum signálu u silného kabelu je menší, než u tenkého kabelu. Silný kabel je nejvíce odolný vůči EMI. Přenosová rychlost u tenkého kabelu nezáleží na délce vodiče jako u kabelu silného, ale na celkové délce přípojky uzlu. Přenosová rychlost u plochého kabelu se odvíjí od maximální délky kabelu, avšak v porovnání se silným kabelem je maximální délka vodiče menší při stejné přenosové rychlosti.

	Silný kabel	**Tenký kabel**	**Plochý kabel**
Velikost jádra [mm^2]	0,823	0,205	1,31
Průměr izolace [mm]	3,81	1,96	2,79
Zkrut páru na 1 m	10 otáček	16 otáček	0 otáček
Stínění páru	hliníková fólie	hliníková fólie	žádné
Tloušťka stínění [mm]	0,05	0,025	0
Impedance při 1 MHz	120 Ω	120 Ω	120 Ω při 500 kHz
Max. zpoždění šíření signálu na 1 m	4,46 ns	4,46 ns	5,25 ns
Kapacitní reaktance mezi vodiči na 1 m při 1 kHz	39 pF	39 pF	48,2 pF
Max. stejnosměrný odpor při 20 °C na 1 km	23 Ω	92 Ω	16,1 Ω

Tab. 3: Vlastnosti kabelů používaných pro sběrnici CAN [2]

3 Vývojové a podpůrné prostředky

Komunikační driver MicroCANopen měl být implementován do systémové exekutivy ZAT. Rozhodlo se, že se driver nejprve bude vyvíjet pomocí STM32F4-Discovery kitu, který obsahuje právě mikrokontrolér STM32F407.

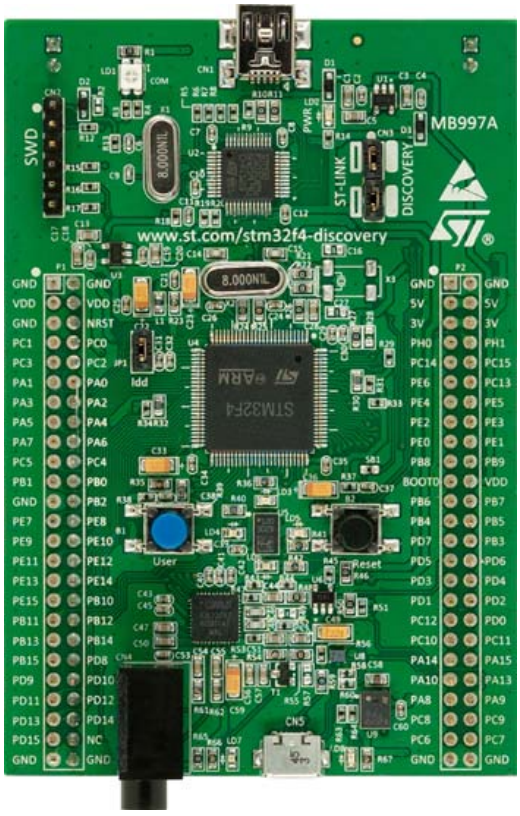
Další pomocný nástroj je dvouřádkový šestnáct-ti znakový (16x2) LCD displej ATM1602B, pomocí něhož se zobrazují data z přijaté zprávy. Výhoda tohoto displeje je, že na jednom řádku zobrazí až 16 znaků. Jeden nibl (4 bity) se v hexadecimálním čísle (0-F) dá vyjádřit jedním znakem. Displej tak na jednom řádku zobrazí hexadecimálně celé datové pole.

Discovery kit postrádá budič a konektor pro CAN sběrnici, takže se musel vyrobit modul, který tento deficit nahradil.

Pro nastavování filtru, vysílání a odesílání zpráv byl použit analyzátor USB-CAN Adapter TRIPLE drivers V4.2, který bude popsán v jedné z následujících kapitol. V dalších kapitolách bude také popsán STM32F4-Discovery kit, mikrokontrolér řady STM32F40x a budič SN65HVD230DR.

3.1 STM32F4-Discovery kit

Pro skutečné ověření funkce vytvořeného zdrojového kódu byl využit STM32F4-Discovery kit od firmy STMicroelectronics, který je vyobrazen na obr. 15.



Obr. 15: STM32F4-Discovery kit [5]

Na jedné DPS se nachází jak vlastní mikrokontrolér STM32F407VGT6, tak i programátor ST-Link/V2. Čip, který se stará o programování, má označení STM32F103. Díky přítomnosti SWD konektoru se může programovat nebo ladit externí mikrokontrolér. Discovery kit je napájen 5V z USB konektoru. Dále se na kitu nachází 3-osý MEMS akcelerometr LIS302DL, MEMS audio senzor MP45DT02, což je vlastně digitální mikrofon, audio digitálně analogový převodník CS43L22, což je audio zesilovač ve třídě D, 8 LED diod, z toho 4 diody jsou uživatelské, 2 tlačítka (User, Reset), další USB OTG konektor, dvě dvouřadé lišty 2x25 pro vstupy/výstupy mikrokontroléru. [8]

Na piny PD0 (Rx) a PD1 (Tx) Discovery kitu se připojuje modul budiče. Na tyto piny se totiž dá naportovat alternativní funkce CAN 1. LCD displej je připojen na piny PE0 – PE3 (datové vstupy displeje), PB8 (zapsání instrukce nebo dat), PB9 (povolení zápisu) a na napájecí piny 5V a GND. Pro účely ladění přijmu zprávy Discovery kitem a pro nepřijetí zprávy z důvodu přeplnění FIFO paměti byly použity dvě LED diody. Zelená LED dioda signalizuje příjem zprávy tak, že při každém přijetí zprávy krátce blikne a červená LED dioda se rozsvítí, když je přeplněná paměť FIFO a další příchozí zpráva tak nemůže být zpracována. Červená dioda zhasne jen tehdy, když se zmáčkne tlačítko User na Discovery kitu.

Mikrokontrolér STM32F407

Hlavní součástkou na Discoverx kitu je samozřejmě 32-bitový mikrokontrolér STM32F407 na jádře ARM Cortex-M4 taktéž od firmy STMicroelectronics. V tab. 4. jsou popsány jeho hlavní parametry. Více informací o mikrokontroléru se dá dozvědět z online literatury [6].

Flash paměť	1 MB	**USART**	4x
SRAM	192 kB	**UART**	2x
Prac. frekvence	168 MHz	**USB OTG**	2x
16-bit časovač	12x	**CAN**	2x
32-bit časovač	2x	**Ethernet MAC 10/100**	1x
12-bit A/D převodník	16x	**SDIO**	1x
12-bit D/A převodník	2x	**Porty 16 bit**	5x
I2C	2x	**Ucc**	1,8-3,6 V
I2S	2x	**Pouzdro**	LQFP 100
SPI	3x		

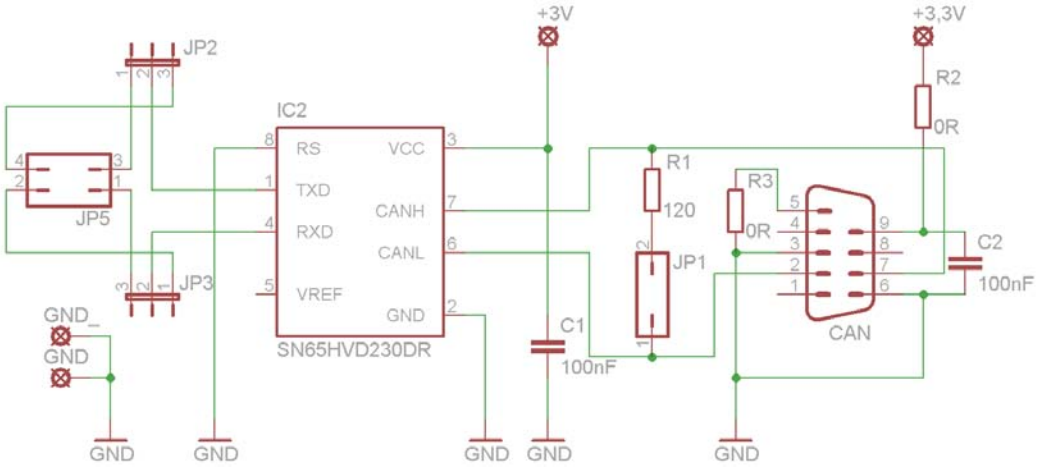
Tab. 4: Parametry mikrokontroléru STM32F407 [7]

3.2 Budič SN65HVD230DR

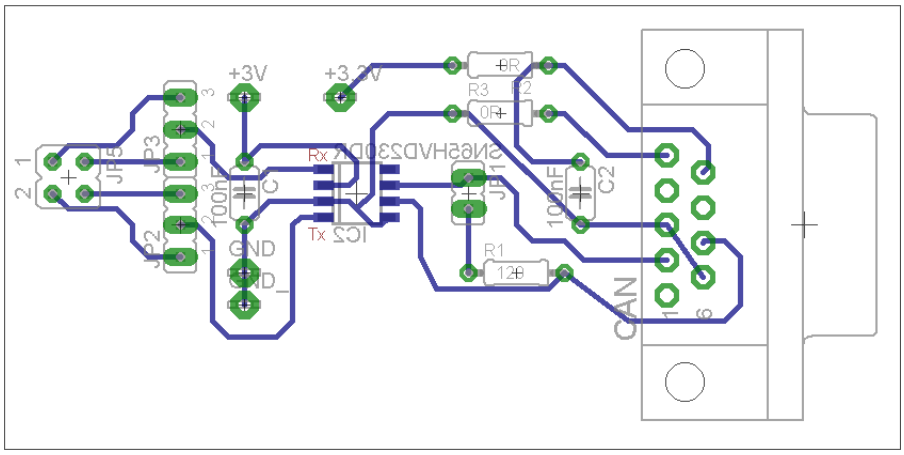
Jelikož Discovery kit nemá v sobě implementovaný budič CAN, musel být vyroben modul s budičem. Modul určený přímo pro Discovery kit se sice dá koupit, ale výroba

modulu vyšla mnohem levněji. K Discovery kitu se dá tento modul externě připojit. Budič je vlastně převodník mezi TTL úrovní signálů Rx a Tx a CAN úrovněmi CAN_H a CAN_L. Discovery kit pracuje s 3V TTL logikou, takže budič musí být kompatibilní s touto logikou. Byl vybrán budič SN65HVD230DR od firmy Texas Instruments, protože se zdál jako nejvhodnější a napěťové úrovně TTL logiky odpovídaly specifikaci Discovery kitu. Navíc Firma Texas Instruments nabízí zdarma až tři vzorky tohoto budiče, takže to hrálo ve výběru také roli.

Maximální napájecí napětí pro budič je 6V, budič je ale napájen 3V z Discovery kitu. Budič podporuje High-speed, Slope control a Standby režimy. High-speed režim umožňuje vysokorychlostní přenos po sběrnici. Slope control režim se používá pro Low-speed přenosovou rychlost. Standby režim se využije v případě low power zařízení. V tomto režimu se vypne vysílač budiče a tím se sníží příkon. Při potřebě vysílání se ze Standby režimu přepne na jiný režim. Tento režim má ještě jedno chytré využití. Když se řadič CAN dané řídící jednotky vymkne kontrole, budič přejde do Standby režimu, tím se zablokuje komunikace a jednotka se vyřadí z provozu. Budič v modulu je nastavený na High-speed režim a změna na jiný režim není možná. Na obr. 16 je schéma modulu s budičem. Vycházelo se ze základního zapojení z datasheetu pro budič SN65HVD230DR [4]. Modul je doplněn o zkratovací jumper JP1, který při zkratování připojí ke sběrnici zakončovací terminátor 120Ω, jak bylo popsáno v kapitole 1.11. Výhodou tohoto modulu je, že piny Tx a Rx se dají jakkoliv přepinovat pomocí pinů JP2 a JP3 ve schématu, takže se v případě potřeby mohou na Discovery kitu použít i jiné porty. JP5 se připojí přímo na piny Discovery kitu. Deska plošných spojů (obr. 17) byla vytvořena v návrhovém prostředí Eagle. [3]



Obr. 16: Schéma modulu budiče



Obr. 17: Deska plošných spojů modulu budiče

3.3 Analyzátor USB-CAN Adapter TRIPLE drivers V4.2

Pomocí analyzátoru USB-CAN Adapter TRIPLE drivers V4.2 od české firmy IMFsoft, s.r.o. bylo možné otestovat příjem i vysílání CAN zpráv. „USB-CAN převodník je zařízení určené zejména pro snadné dynamické ladění CAN aplikací a pro okamžitou a přehlednou diagnostiku CAN linky. Převodník je řízen prostřednictvím sběrnice USB z aplikace USB-CAN adapter, z vlastní uživatelské aplikace vytvořené modifikací aplikace CAN Start ve vývojovém prostředí Delphi nebo s použitím Dll knihovny.“[9] Na obr. 18 je ukázka analyzátoru s CD, na kterém je uložena aplikaci USB-CAN Adapter V4.11, ovladače pro analyzátor a jiné dokumenty.

Obr. 18: Analyzátor USB-CAN Adapter TRIPLE drivers V4.2 s instalačním CD [10]

Dále jsou popsány parametry analyzátoru:

- „Budiče kompatibilní s High speed, Low speed a One wire CAN v jednom převodníku
- Zasílání rámců CAN 2.0A a CAN 2.0B
- K dispozici 15 nezávislých Message Center
- Komunikační rychlost 10kbps až 1Mbps
- Dynamický příjem a zobrazení CAN zpráv (implementovaná vyrovnávací paměť 256B)
- Zobrazení reálného času příjmu zprávy s rozlišením 1ms a výpočet průměrné periody příjmu
- Okamžité, opožděné nebo periodické vysílání až 8 zpráv současně (1ms až 65,5s)
- Zobrazení celkové počtu přijatých zpráv, periody zpráv, zatížení linky a chyb CAN linky
- Přepočet zpráv na skutečné hodnoty s možností grafického zobrazení v reálném Trendu
- Dlouhodobý záznam zpráv nebo přepočtených hodnot do souboru
- Příjem zpráv bez potvrzení (ACK) tzv. Listening Mode
- Rozšířené vyhledávání v seznamu přijatých zpráv
- Vysílání a příjem zpráv REMOTE FRAME
- Automatické vkládání popisu zpráv
- Uložení uživatelských nastavení
- Podpora protokolu CANopen CiA DS 301
- Možnost připojení více převodníku k jedinému počítači
- Signalizace napájení a inicializace LED (červená/zelená)
- Standardní zapojení CANNON konektoru
- Napájení ze sběrnice USB
- Ochrana proti přepětí a přepólování (Transil)“[9]

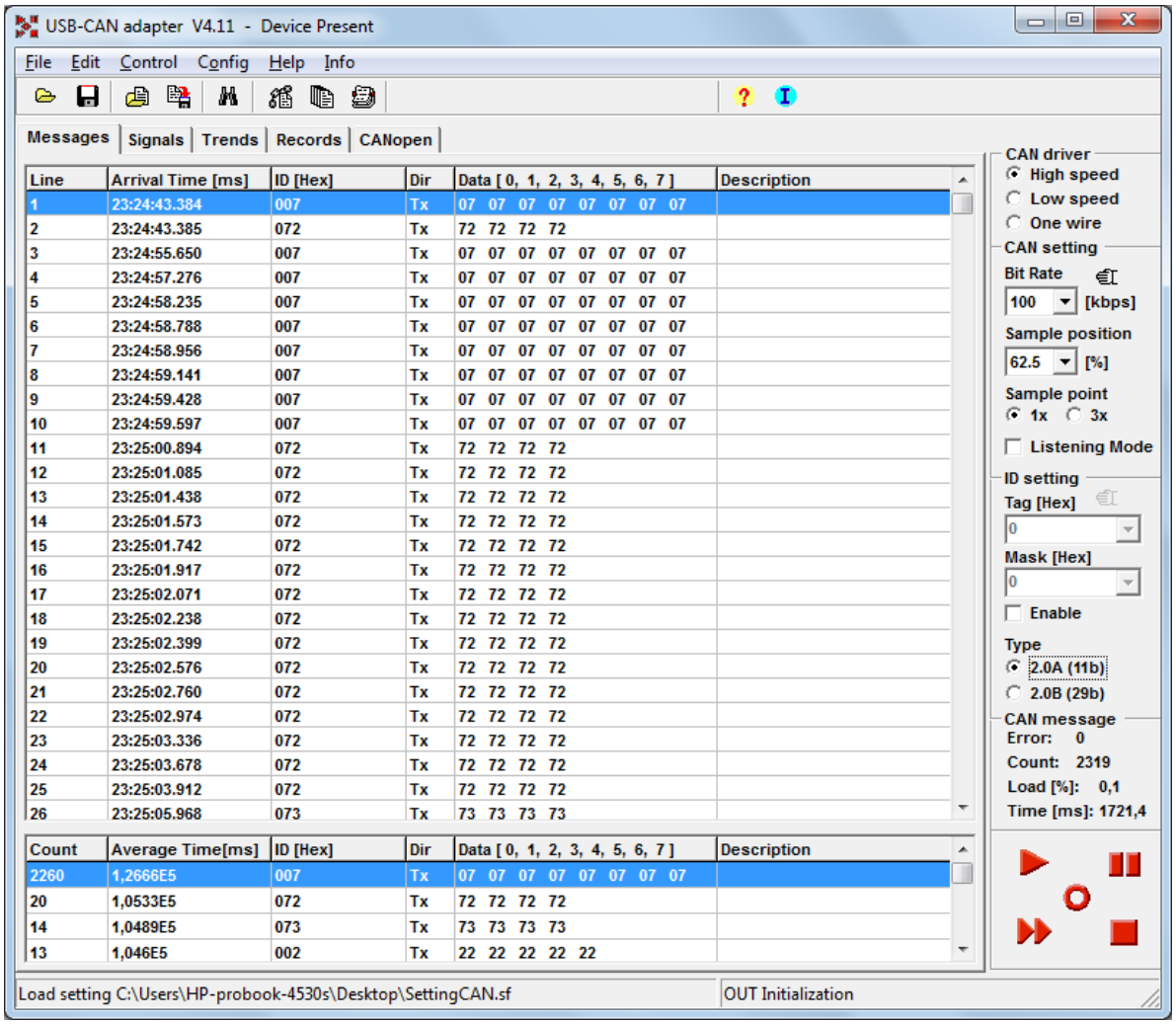
Aplikace USB-CAN Adapter

Aplikace USB-CAN Adapter je uživatelsky velmi přívětivá a snadná na ovládání. Po instalaci aplikace a připojení analyzátoru na sběrnici stačí jen nastavit přenosovou rychlost sběrnice CAN, inicializovat adapter a v tento okamžik se už monitoruje činnost na sběrnici. Hlavní okno aplikace obsahuje prvky pro nastavení požadovaných parametrů sběrnice CAN. Toto okno umožňuje přepínání mezi pěti záložkami, jejichž funkce je následující:

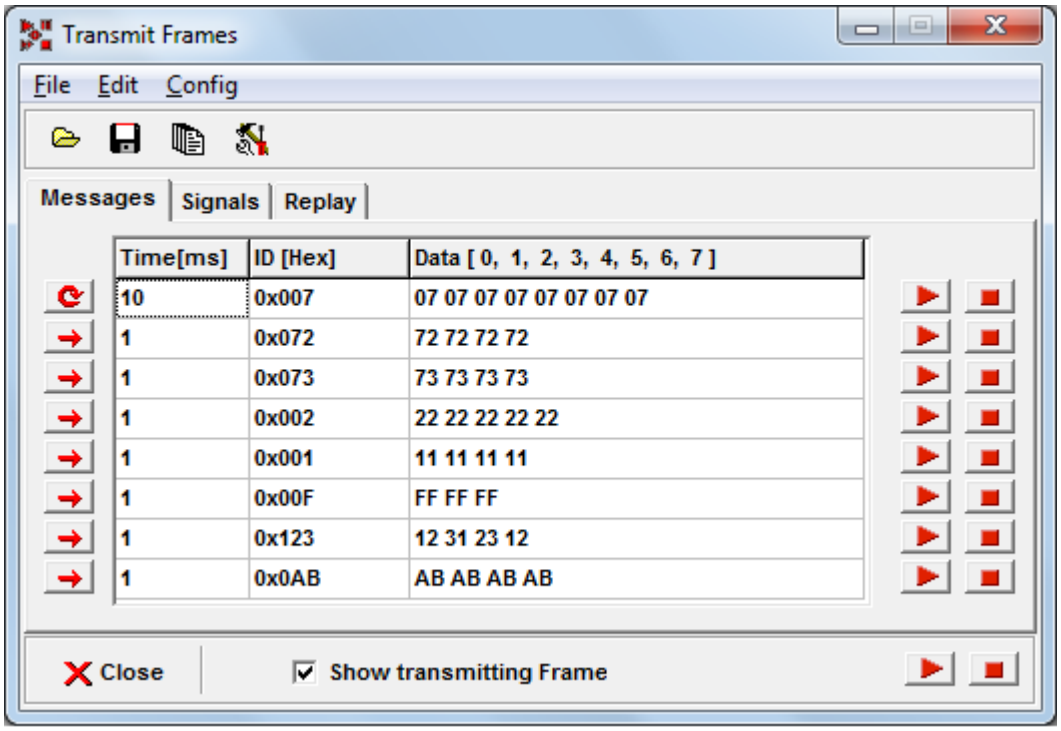
- Záložka Messages – okamžité zobrazení CAN zpráv s možností vkládání popisu o významu zpráv
- Záložka Signals – přepočet a zobrazení CAN dat ve formátu skutečných veličin

- Záložka Trends – poskytuje názorné zobrazení až 15 veličin v reálném trendu
- Záložka Records – průběžné ukládání přijatých zpráv nebo veličin do *.txt souboru
- Záložka CANopen – nástroje pro řízení jednoho Slave zařízení komunikujícího dle standardu CANopen, vysílání, příjem a zpracování zpráv

O nastavení a vysílání CAN zpráv se stará vedlejší okno Transmit Frames. V tomto okně se dá nastavit až 8 různých CAN rámců. U každého rámce se nastaví identifikátor a datové pole. U každého rámce se dá také nastavit, jestli se má odeslat jen jednou nebo jestli se má odesílat cyklicky podle nastavitelného intervalu. Rámce se mohou odesílat jednotlivě nebo postupně všechny za sebou. Odeslané rámce se samozřejmě zobrazují v záložce Messages. Na obr. 19 je ukázka uživatelského prostředí hlavního okna se záložkou Messages a na obr. 20 je vedlejší okno Transmit Frames. [9]



Obr. 19: Hlavní okno záložky Messages



Obr. 20: Vedlejší okno Transmit Frames pro nastavení a vysílání CAN zpráv

4 Software

Program byl realizován v programovacím jazyce C a jako programovací editor byl použit Keil uVision4. Nejen, že tento programátorský nástroj dokáže zkompilovat zdrojový kód a nahrát ho do mikrokontroléru, dokonce umí pomocí nástroje debugger program ladit.

Firma STMicroelectronics nabízí zdarma ke stažení na svých stránkách [11] knihovnu určenou mikrokontroléru řady STM32F40x, díky níž se dají ovládat všechny periférie mikrokontroléru. Knihovna se jmenuje STM32F4xx_StdPeriph_Driver a některé její funkce jsou také využity v tomto projektu.

V následujících podkapitolách budou popsány zdrojové moduly *main.c, mcohw.c, can.c, mco.c, LCD16x2lbr.c a stm32f4xx_it.c*, kde bude také vysvětlena činnost jejich funkcí. Hlavní zdrojový modul bude popsán jako poslední, protože obsahuje funkce z jiných zdrojových modulů. Pro lepší pochopení funkcionality budou tedy nejprve vysvětleny funkce ze zdrojových modulů *mcohw.c, can.c, mco.c, LCD16x2lbr.c a stm32f4xx_it.c.*

4.1 Zdrojový modul mcohw.c

Hlavním úkolem hardwarového ovladače je ovládání hardwarového rozhraní CAN, implementace časovače s tikem jedné milisekundy a uložení obdržené zprávy do fronty. Tato metoda může být implementována do všech CAN řadičů bez ohledu na to, jak propracované

je jejich rozhraní. Hardwarově závislý ovladač definuje následující povinné funkce:

- *uint16_t MCOHW_Init(uint16_t BaudRate)* – Tato funkce inicializuje rozhraní a připouští přenosové rychlosti 1000, 800, 500, 250, 125, 50, 25 nebo 10 kbit/s. Používá k tomu funkci *uint16_t CAN_Initialization(uint16_t Bitrate, uint32_t IfaceTransferTout)* definovanou v *can.c* zdrojovém modulu.
- *uint16_t MCOHW_SetCANFilter(uint16_t CANID)* – Funkce se stará o nastavení CAN filtru a při každém jejím volání nastaví jeden identifikátor. Využívá k tomu *uint16_t CAN_SetFilter(uint32_t FilterBank, uint16_t CANID, uint16_t MaskOrList, uint16_t Fifo)* funkci použitou z knihovny STM32F4xx_StdPeriph_Driver. Dokáže nastavit až 28 filtračních bank.
- *uint16_t MCOHW_PushMessage(CAN_MSG *pTransmitBuf)* – Hlavním úkolem funkce je vysílání zpráv. Každým voláním této funkce se zpráva uloží do paměti fronty pro odesílání zpráv. Používá k tomu funkci *uint8_t CAN_Transmit(CAN_TypeDef* CANx, CanTxMsg* TxMessage)* užitou z knihovny STM32F4xx_StdPeriph_Driver.
- *uint16_t MCOHW_PullMessage(CAN_MSG *pReceiveBuf)* – Funkce se stará o příjem zpráv. S každým voláním funkce se vyčte zpráva z paměti fronty. Samozřejmě, že zpráva musí projít nejdřív přes přijímací filtr CAN. Využívá k tomu *void CAN_Receive(CAN_TypeDef* CANx, uint8_t FIFONumber, CanRxMsg* RxMessage)* funkci opět z knihovny STM32F4xx_StdPeriph_Driver.
- *uint32_t MCOHW_GetTime(void)* – Funkce vrací aktuální hodnotu 32-bitové proměnné *TimCnt1ms*, která je inkrementována každou milisekundu při vyvolání přerušení čítače.
- *uint8_t MCOHW_IsTimeExpired(uint32_t timestamp)* – Funkce porovnává čas daný parametrem funkce (v milisekundách) s vnitřním čítačem a říká, zda čas daný parametrem už vypršel, či nikoliv. Maximální porovnávaná hodnota je 32 sekund.

4.2 Zdrojový modul can.c

Zdrojový modul *can.c* obsahuje celkem tři funkce. Funkce *uint16_t CAN_BaudRate(uint16_t BaudRate)* umí nastavit přenosovou rychlost sběrnice. Na výběr je z devíti přenosových rychlostí (1000, 800, 500, 250, 125, 50, 25, 10 kbit/s) a další dvě přenosové rychlosti (33,3 a 83,3 kbit/s) jsou „zakomentovány“, ale připraveny k použití. Nastavuje se zde doba přenosu jednoho bitu, jak bylo popsáno v kapitole 1.6.

Funkce *uint16_t CAN_Initialization(uint16_t Bitrate, uint32_t IfaceTransferTout)*

nastavuje a inicializuje CAN1 rozhraní. Nejprve spouští hodiny pro CAN1 a pro port, na který se bude připojovat budič CAN. Nastavuje alternativní funkci CAN1 pro určené piny, nastavuje definovanou přenosovou rychlost a povoluje interrupt pro příjem zpráv.

Poslední funkce *uint16_t CAN_SetFilter(uint32_t FilterBank, uint16_t CANID, uint16_t MaskOrList, uint16_t Fifo)* nastavuje filtr pro příjem zpráv a do jedné filtrační banky dokáže uložit až 4 standardní identifikátory. Filtr umí zadávat jen standardní identifikátory.

4.3 Zdrojový modul mco.c

Tento zdrojový modul spadá pod vlastní vrstvu implementující MicroCANopen protokol. Modul inkluduje hlavičkový soubory *mco.h* a *mcohw.h*. Modul zatím obsahuje jen jedinou funkci *MCO_Init(uint16_t bitrate, uint8_t Node_ID, uint32_t heartbeat),* která pomocí funkce *uint16_t MCOHW_Init(uint16_t BaudRate)* nastavuje přenosovou rychlost sběrnice. Funkce *MCO_Init()* definuje i parametry *Node_ID a heartbeat,* ale ty zatím nejsou použité. Pro napsání zbylých funkcí vlastní vrstvy implementující MicroCANopen protokol už nebyl dostatečný časový prostor.

4.4 Zdrojový modul LCD16x2lbr.c

Modul *LCD16x2lbr.c* je vlastně vytvořená knihovna pro dvouřádkový 16-ti znakový LCD displej. Na začátku se definuje příkaz *PULS_ENABLE* pro pin, který je propojen s LCD displejem a vlastně o to, že vznikne jeden puls, který umožní zapsat připravená data na displej. Další příkaz *SET_DATA* definuje pro jeden pin, který je také propojen s displejem a říká, jestli se na displej zapíšou data nebo příkaz. Funkce *void delay_ms(unsigned int cas)* a *void delay_us(unsigned int cas)* jsou jen čekací funkce. Funkce *void write_data(unsigned char b)* zapisuje na displej jenom data, naopak funkce *void write_cmd(unsigned char b)* zapisuje na displej jenom příkazy. Funkce *void write_lcd(unsigned char b)* dělá to samé jako funkce *write_data().* Pro mazání displeje je tu funkce *void clear_lcd(void),* která odešle speciální příkaz pro mazání. Na displej se můžou psát řetězce pomocí funkce *void write_string(char *msg).* Funkce *void GoToXY(char sloupec, char radek)* dokáže umístit kurzor na kterékoliv pole displeje a odtud pak začít vypisovat znaky. Pro inicializaci a nastavení displeje poslouží funkce *void init_lcd(void).* Funkce *void hexa_cisla(char cislo)* umí desítkové číslo převést na hexadecimální a pak ho zobrazit na displeji. Funkce *void zobrazLCD_hexa(unsigned long long hex)* je speciálně vytvořená funkce pro ladící účely přijmu zpráv, protože dokáže na displeji zobrazit celé datové pole CAN zprávy v hexadecimálním tvaru.

4.5 Zdrojový modul stm32f4xx_it.c

Tento zdrojový modul je speciálně vytvořený soubor, který obsahuje jen funkce, na které dojde při nahození interruptu. Soubor obsahuje dvě funkce. Funkce *void CAN1_RX0_IRQHandler(void)* se vyvolá při přerušení, když přijde CAN zpráva. Obecně by funkce vyvolané přerušením neměly být nějak obzvlášť dlouho trvající, aby při častém přerušení neohrozily správný chod programu. V této funkci se možná až zbytečně řeší rozsvícení LED diod a vypisování na displej, ale pro diagnostiku je to přínosné. Druhá funkce *void TIM2_IRQHandler(void)* tohoto souboru řeší obsluhu při přetečení čítače. Každým vyvoláním této funkce se inkrementuje proměnná TimCnt1ms.

4.6 Zdrojový modul main.c

Zdrojový kód modulu *main.c* se nachází v příloze 1. Obsahuje hlavní *int main()* funkci a *while* smyčku, ve kterém neustále běží hlavní program. Hned na začátku se „nainkludují“ nezbytné hlavičkové soubory *stm32f4xx.h, LCD16x2lbr.h, can.h, mco.h, mcohw.h* a pak se definují datové struktury:

```
GPIO_InitTypeDef GPIO_InitStructure;
CAN_FilterInitTypeDef CAN_FilterInitStructure;
CanTxMsg TxMessageStructure;
CAN_MSG TxMessage;
CanRxMsg RxMessageStructure;
CAN_MSG RxMessage;
NVIC_InitTypeDef NVIC_InitStructure;
TIM_TimeBaseInitTypeDef TIM_InitStructure;
```

Hlavní funkce *int main(void)* nastavuje vstupní/výstupní porty a jejich piny pro diody, tlačítko a LCD displej. Nejprve se povolí hodiny pro daný port funkcí *RCC_AHB1PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState),* poté se port „vyresetuje“ funkcí *GPIO_DeInit(GPIO_TypeDef* GPIOx),* pak se pomocí struktury zvolí konkrétní pin, nastaví se, zda jde o vstup nebo výstup a nakonec se funkcí *GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)* port inicializuje. Zmíněné funkce jsou použity z STM32F4xx_StdPeriph_Driver knihovny.

Následuje nastavení filtrů funkcí *uint16_t MCOHW_SetCANFilter(uint16_t CANID)* z *mcohw.h* hardwarově závislého ovladače.

Znovu se použijí funkce z knihovny STM32F4xx_StdPeriph_Driver tentokrát pro nastavení čítače TIM2 a interrupt (přerušení) tohoto čítače. Pro čítač se povolí hodiny a pak se

„vyresetuje“, obdobně jako to bylo u portů. U čítače se dále nastavuje předdělič, časová perioda a inkrementace. Inicializace tohoto čítače nastavila jeho přetečení každou milisekundu. To znamená, že pokud se povolí interrupt tohoto čítače, každou milisekundu se vyvolá přerušení a jeho obslužný program.

Dále proběhne inicializace LCD displeje funkcí *void init_lcd()* a na jeho prvním řádku se vypíšou samé nuly s funkcí *zobrazLCD_hexa(unsigned long long hex)* a na druhém řádku se vypíše nápis HEXA s funkcí *write_string(char *msg).*

Ve *while* smyčce se testuje zmáčknutí tlačítka a tím se ovládají LED diody. Když není zmáčknuté tlačítko, tak zhasne zelená dioda (pokud tedy byla předtím rozsvícená kvůli přijetí zprávy). Když se stiskne tlačítko, tak se rozsvítí modrá dioda a pokud svítila červená dioda (signalizace plné paměti FIFO a nedoručení zprávy), tak zhasne.

Dále je ve *while* smyčce „zakomentován“ kód pro odesílání zpráv. Nastavuje se zde identifikátor, délka datového pole a samotné datové pole. Takto nastavená zpráva by se odeslala funkcí *uint16_t MCOHW_PushMessage(CAN_MSG *pTransmitBuf)* z *mcohw.h* hardwarově závislého ovladače. „Odkomentování“ kódu a následné nahrání zdrojového kódu do mikrokontroléru by zapříčinilo cyklické odesílání zprávy uzlem.

Na konci *while* smyčky jsou funkce signalizující přijetí zprávy rozsvícením nebo zhasnutím zelené LED diody.

Pak se program vrací znovu na začátek *while* smyčky a tento cyklus se opakuje pořád dokola.

Závěr

Prostudoval jsem architekturu mikrokontroléru STM32F40x a seznámil jsem se s vývojovými nástroji. Popsal jsem detailně specifikaci komunikačního profilu CAN. Okrajově se zabýval problematikou elektromagnetické kompatibility sběrnice CAN, což bylo nad rámec této práce. Snažil jsem se naprogramovat komunikační rozhraní a konfigurační a stavové struktury komunikačního driveru MicroCANopen.

Cílem práce bylo naprogramovat a implementovat komunikační protokol MicroCANopen pro mikrokontrolér řady STM32F40x na jádře ARM Cortex M4 v řídicím systému zadavatele ZAT a.s. Plzeň a to včetně jeho proprietárních rozšíření a diagnostických služeb. Na naprogramování vlastní vrstvy implementující MicroCANopen protokolové služby z časových důvodů bohužel nedošlo, nicméně funkce hardwarově závislého ovladače CAN řadiče mikrokontroléru jsou z větší části hotové. Discovery kit, tvářící se jako uzel, je schopný vysílat CAN zprávy, filtrovat příchozí zprávy podle nastaveného identifikátoru a pokud zpráva projde filtrem, tak jí dokáže i přijmout.

Pokud se na projektu bude dále pracovat, rozhodně bude nutné naprogramovat všechny chybějící funkce. Poté by se protokolový stack MicroCANopen řešil formou periodicky volaného obecného stavového automatu s minimálními časovými prodlevami. Jako konečnou fází by byla implementace protokolového stacku MicroCANopen do systémové exekutivy ZAT.

Seznam obrázků

Obr. 1: Referenční model ISO/OSI....13
Obr. 2: Připojení uzlů na sběrnici....14
Obr. 3: Řešení konfliktů na sběrnici....15
Obr. 4: Bit stuffing....16
Obr. 6: Formát standardního rámce CAN 2.0A....18
Obr. 7: Formát rozšířeného rámce CAN 2.0B....19
Obr. 8: Obecné uspořádání sběrnice CAN....22
Obr. 9: Napěťové úrovně na sběrnici CAN – high-speed....23
Obr. 10: Napěťové úrovně na sběrnici CAN – low-speed....24
Obr. 11: Konektor DE-9 – male....24
Obr. 12: a) Působení EMI na sběrnici, b) Změna napětí při působení EMI....25
Obr. 13: Ukončení sběrnice s rozděleným zakončovacím terminátorem....26
Obr. 14: Zařazení tlumivky mezi sběrnicové vodiče....26
Obr. 15: STM32F4-Discovery kit [5]....28
Obr. 16: Schéma modulu budiče....30
Obr. 17: Deska plošných spojů modulu budiče....31
Obr. 18: Analyzátor USB-CAN Adapter TRIPLE drivers V4.2 s instalačním CD....31
Obr. 19: Hlavní okno záložky Messages....33
Obr. 20: Vedlejší okno Transmit Frames pro nastavení a vysílání CAN zpráv....34

Seznam tabulek

Tab. 1: Přehled přenosové rychlosti v závislosti na délce sběrnice....23
Tab. 2: Popis pinů CAN konektoru DE-9 [1]....25
Tab. 3: Vlastnosti kabelů používaných pro sběrnici CAN [2]....27
Tab. 4: Parametry mikrokontroléru STM32F407 [7]....29

Chyba! Nenalezen zdroj odkazů.

Seznam literatury a informačních zdrojů

[1] DAVIS, Leroy. CAN Bus Pin Out, and CANopen pinout, with Signal Names: 9-Pin D, CAN Bus Pin Out. *Interfacebus.com; Hardware Manufacturers, Computer Bus Descriptions, Engineering Design Information* [online]. © 1998 - 2012, Modified: 2/26/12 [cit. 2014-05-12]. Dostupné z: http://www.interfacebus.com/Can_Bus_Connector_Pinout.html

[2] ČSN EN 62026-3. *Spínací a řídicí přístroje nízkého napětí: Rozhraní řadič - zařízení (CDI) - Část 3: DeviceNet*. Praha: Úřad pro technickou normalizaci, metrologii a státní zkušebnictví, 2010.

[3] SN65HVD230 | CAN | Interface | Description & parametrics. *Analog, Embedded Processing, Semiconductor Company, Texas Instruments - TI.com* [online]. © 1995-2014 [cit. 2014-05-17]. Dostupné z: http://www.ti.com/product/SN65HVD230?keyMatch=SN65HVD230DR&tisearch=Search-EN

[4] *SN65HVD230: Datasheet* [online]. © 2001–2011, revised february 2011 [cit. 2014-05-17]. ISBN SLOS346K. Dostupné z: http://www.ti.com/lit/ds/symlink/sn65hvd230.pdf

[5] STM32F4DISCOVERY Discovery kit for STM32F407/417 lines - with STM32F407VG MCU. *STMicroelectronics* [online převzatý obrázek]. © 2014 [cit. 2014-05-18]. Dostupné z: http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF252419

[6] STM32F407VG High-performance and DSP with FPU, ARM Cortex-M4 MCU with 1 Mbyte Flash, 168 MHz CPU, Art Accelerator, Ethernet. *STMicroelectronics* [online]. © 2014 [cit. 2014-05-18]. Dostupné z: http://www.st.com/web/catalog/mmc/FM141/SC1169/SS1577/LN11/PF252140

[7] ARM 1. díl - STM32F4 Discovery kit. *TajNed - Electronics and software* [online převzatá tabulka]. 2013 - 2014 [cit. 2014-05-18]. Dostupné z: http://www.tajned.cz/2013/10/stm32f4_discovery/

[8] STM32F4DISCOVERY Discovery kit for STM32F407/417 lines - with STM32F407VG MCU. *STMicroelectronics* [online]. © 2014 [cit. 2014-05-18]. Dostupné z: http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF252419

[9] USB-CAN adapter – TRIPLE drivers V4.2 (high/ low/ one wire). *IMFsoft, s.r.o.* [online]. © 2006 – 2014 [cit. 2014-05-18]. Dostupné z: http://imfsoft.com/hardware/usb-can-adapter-triple-drivers-v4-2-high-low-one-wire

[10] USB-CAN adapter – TRIPLE drivers V4.2 (high/ low/ one wire). *IMFsoft, s.r.o.* [online převzatý obrázek]. © 2006 – 2014 [cit. 2014-05-18]. Dostupné z: http://imfsoft.com/hardware/usb-can-adapter-triple-drivers-v4-2-high-low-one-wire

[11] STSW-STM32068 STM32F4DISCOVERY board firmware package, including 22 examples (covering USB Host, audio, MEMS accelerometer and microphone) (AN3983): Sample & Buy. *STMicroelectronics* [online datový archiv ZIP]. Version 1.1.0. © 2014 [cit. 2014-05-19]. Dostupné z: http://www.st.com/web/en/catalog/tools/PF257904#

[12] CAN bus. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 30 April 2014 [cit. 2014-05-19]. Dostupné z: http://en.wikipedia.org/wiki/CAN_bus

[13] OSI Protocol Stack Description. *Interfacebus.com; Hardware Manufacturers, Computer Bus Descriptions, Engineering Design Information* [online]. © 1998 - 2012 [cit. 2014-05-19]. Dostupné z: http://www.interfacebus.com/Design_OSI_Stack.html

[14] Encoding Dictionary, terms, and definitions. *Interfacebus.com; Hardware Manufacturers, Computer Bus Descriptions, Engineering Design Information* [online]. © 1998 - 2012, Last Modified 3/04/12 [cit. 2014-05-19]. Dostupné z: http://www.interfacebus.com/Definitions.html#NRZ_Encoding

Přílohy

Příloha 1 – Zdrojový kód souboru main.c

```
#include <stm32f4xx.h>
#include "LCD16x2lbr.h"
#include "can.h"
#include "mco.h"
#include "mcohw.h"

#define NULY 0x0000000000000000

GPIO_InitTypeDef GPIO_InitStructure;
CAN_FilterInitTypeDef CAN_FilterInitStructure;
CanTxMsg TxMessageStructure;
CAN_MSG TxMessage;
CanRxMsg RxMessageStructure;
CAN_MSG RxMessage;
NVIC_InitTypeDef NVIC_InitStructure;
TIM_TimeBaseInitTypeDef TIM_InitStructure;

int main(void)
{
    uint16_t Id1, Id2, Id3, Id4;                    // 4 identifikatory pro jednu filtracni banku
    Id1 = 0x007;   Id2 = 0x072;      Id3 = 0x073;      Id4 = 0x002;

    /* konfigurace - vystupy na diodu ----------------------------------------------------------*/

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);   // Enable clock for GPIOD peripheral
    GPIO_DeInit(GPIOD);                             // Assert and immediately release GPIOD peripheral reset
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15 ; // Set output
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;          // Set Output push-pull mode
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOD, &GPIO_InitStructure);              // Initialize output GPIOD

    /* konfigurace - vstupy na tlacitko --------------------------------------------------------*/

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);       // Enable clock for GPIOA peripherals
    GPIO_DeInit(GPIOA);                             // Assert and immediately release GPIOA peripheral reset
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 ;                                      // Set output GPIOA Init Structure
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;                                    // Set Input push-pull mode
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);                                          // Initialize input GPIOA

    /* konfigurace - vystupy na displej LCD 16x2 -----------------------------------------------*/

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);   // Enable clock for GPIOE peripheral
    GPIO_DeInit(GPIOE);                             // Assert and immediately release GPIOE peripheral reset
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3; // Set output GPIOE
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;          // Set Output push-pull mode
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOE, &GPIO_InitStructure);              // Initialize output GPIOE

    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);   // Enable clock for GPIOB peripheral
    GPIO_DeInit(GPIOB);                             // Assert and immediately release GPIOB peripheral reset
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9 ; // Set output GPIOB Init Structure
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;          // Set Output push-pull mode
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);              // Initialize output GPIOB

    /* CAN1 configure and init -----------------------------------------------------------------*/
    MCO_Init(100, 1, 2000);                                     // 1 a 2000 je zatim zvoleno nahodne

    /* CAN1 filter init ------------------------------------------------------------------------*/
```

```
    MCOHW_SetCANFilter(Id1);
    MCOHW_SetCANFilter(Id2);
    MCOHW_SetCANFilter(Id3);
    MCOHW_SetCANFilter(Id4);

    /* timer init ------------------------------------------------------------------*/

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);          // TIM2 clock enable
    TIM_DeInit(TIM2);
    TIM_InitStructure.TIM_Prescaler = 84 - 1;                     // 24 MHz Clock down to 1 MHz
    TIM_InitStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_InitStructure.TIM_Period = 1000 - 1;                      // 1 MHz down to 1 KHz (1 ms)
    TIM_InitStructure.TIM_ClockDivision = TIM_CKD_DIV1;
    TIM_TimeBaseInit(TIM2, &TIM_InitStructure);
    TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);                    // TIM IT enable
    TIM_Cmd(TIM2, ENABLE);                                        // TIM2 enable counter

    /* NVIC Configuration for TIM1 -------------------------------------------------------*/

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
    NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    init_lcd();                                                   // Inicializace LCD displeje
    zobrazLCD_hexa(NULY);                                         // Po inicializaci vypise na displej nuly
    GoToXY(0,1);
    write_string("HEXA");                                         // Na druhy radek displeje vypise HEXA

    while(1)
    {
        /* zmacknuti tlacitka User ---------------------------------------------------*/
        if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0))
        {
            GPIO_WriteBit(GPIOD, GPIO_Pin_15, Bit_SET);           // GPIO_Pin_15 modraLED LD6
            GPIO_WriteBit(GPIOD, GPIO_Pin_14, Bit_RESET);         // vynulovani signalizace preteceni
        }
        else
        {
            GPIO_WriteBit(GPIOD, GPIO_Pin_15, Bit_RESET);
        }

        /* nastaveni vysilani Tx a odeslani dat ----------------------------------------*/
//          TxMessage.ID = 0x007;                   // Message Identifier
//          TxMessage.LEN = 1;                      // Data length (0-8)
//          TxMessage.BUF[0] = 0xAB;                // Data buffer
//          MCOHW_PushMessage(&TxMessage);

        /* signalizace prichodu zpravy pomoci zelene LED -------------------------------*/
        if((CAN1->RF0R & 0x03) > 0)
            GPIO_WriteBit(GPIOD, GPIO_Pin_12, Bit_SET);           // GPIO_Pin_12 zelenaLED LD4
        else
            GPIO_WriteBit(GPIOD, GPIO_Pin_12, Bit_RESET);
    }
}
```

Příloha 2 – Zdrojový kód souboru mcohw.c

```
/* Includes ------------------------------------------------------------------*/
#include "mcohw.h"
#include "can.h"
```

```
/* Private variables ---------------------------------------------------------*/
uint32_t FilterCounter = 0;
unsigned short int volatile TimCnt1ms = 0;    // Global timer/counter variable, incremented every millisecond
/*******************************************************************************/
/*                      Cortex-M4 Processor                                    */
/*******************************************************************************/

/**
  * @brief  This function implements the initialization of the CAN interface.
  * @param  None
  * @retval None
  */
uint16_t MCOHW_Init(uint16_t BaudRate)
{
    uint32_t IfaceTout = 50;                          // Waiting max 50ms
    uint16_t stav;
    stav = CAN_Initialization(BaudRate, IfaceTout);

    if(stav == 1)
        return 0;
    else
        return 1;
}

/**
  * @brief  This function implements the initialization of a CAN ID hardware filter.
  * @param  None
  * @retval None
  */
uint16_t MCOHW_SetCANFilter(uint16_t CANID)
{
    FilterCounter++;                                  // Increment a counter of CAN filter
    if (FilterCounter > 28)
    {
        return 0;
    }
    else                                               // Filter is available
    {
        CAN_SetFilter(FilterCounter - 1, CANID, CAN_FilterMode_IdList, CAN_Filter_FIFO0); // configure the filter
        return 1;
    }
}

/**
  * @brief  This function implements a CAN transmit queue.
```

```
  * @param  None
  * @retval None
  */
uint16_t MCOHW_PushMessage(CAN_MSG *pTransmitBuf)
{
    uint8_t i;
    CanTxMsg TxMessage;
    TxMessage.StdId = pTransmitBuf->ID;
    TxMessage.IDE = CAN_Id_Standard;
    TxMessage.RTR = CAN_RTR_Data;
    TxMessage.DLC = pTransmitBuf->LEN;
    for(i = 0; i < pTransmitBuf->LEN; i++)
        TxMessage.Data[i] = pTransmitBuf->BUF[i];
    if (CAN_Transmit(CAN1, &TxMessage) != CAN_TxStatus_NoMailBox)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

/**
  * @brief  This function implements a CAN receive queue.
  * @param  None
  * @retval None
  */
uint16_t MCOHW_PullMessage(CAN_MSG *pReceiveBuf)
{
    uint8_t i;
    CanRxMsg RxMessage;
    CAN_Receive(CAN1, CAN_FIFO0, &RxMessage);
    pReceiveBuf->ID = (uint32_t)(RxMessage.StdId);
    pReceiveBuf->LEN = RxMessage.DLC;
    for(i = 0; i < pReceiveBuf->LEN; i++)
        pReceiveBuf->BUF[i] = RxMessage.Data[i];
    return 1;
}

/**
  * @brief  This function gives the Timer Counter value.
  * @param  None
  * @retval None
  */
```

```
uint32_t MCOHW_GetTime(void)
{
   return TimCnt1ms;
}

/**
 * @brief  This function checks if a TimeStamp expired.
 * @param  None
 * @retval None
 */
uint8_t MCOHW_IsTimeExpired(uint32_t timestamp)
{
    uint32_t time_now;
    time_now = TimCnt1ms;
    if (time_now > timestamp)
    {
        if ((time_now - timestamp) < 0x8000)
            return 1;
        else
            return 0;
    }
    else
    {
        if ((timestamp - time_now) > 0x8000)
            return 1;
        else
            return 0;
    }
}

/**
* @}
*/

/*********************************END OF FILE**********************************/
```

Příloha 3 – Zdrojový kód souboru can.c

```
/* Includes ------------------------------------------------------------------*/
#include "can.h"
#include "mcohw.h"

/* Private structures ---------------------------------------------------------*/
extern GPIO_InitTypeDef GPIO_InitStructure;
extern NVIC_InitTypeDef  NVIC_InitStructure;
```

```
extern CAN_FilterInitTypeDef CAN_FilterInitStructure;

/**
  * @brief  This function sets a baud rate.
     PCLK1 must be set on freguency 42MHz
  * @param  None
  * @retval None
  */
uint16_t CAN_BaudRate(uint16_t BaudRate)
{
    char CAN_Prescaler, CAN_SJW, CAN_BS1, CAN_BS2, CAN_Mode = CAN_Mode_Normal;

    switch (BaudRate)
    {
        case (10):                      // 10 kbit/s
            CAN_Prescaler = 168;        // Specifies the length of a time quantum from 1 to 1024
            CAN_SJW = CAN_SJW_2tq;      // Specifies the maximum number of time quanta
            CAN_BS1 = CAN_BS1_16tq;     // Specifies the number of time quanta in Bit Segment 1
            CAN_BS2 = CAN_BS2_8tq;      // Specifies the number of time quanta in Bit Segment 2
        break;
        case (20):                      // 20 kbit/s
            CAN_Prescaler = 84;
            CAN_SJW = CAN_SJW_2tq;
            CAN_BS1 = CAN_BS1_16tq;
            CAN_BS2 = CAN_BS2_8tq;
        break;
//      case (0x02):                    // 33,3 kbit/s
//          CAN_Prescaler = 84;
//          CAN_SJW = CAN_SJW_2tq;
//          CAN_BS1 = CAN_BS1_9tq;
//          CAN_BS2 = CAN_BS2_5tq;
//      break;
        case (50):                      // 50 kbit/s
            CAN_Prescaler = 42;
            CAN_SJW = CAN_SJW_2tq;
            CAN_BS1 = CAN_BS1_12tq;
            CAN_BS2 = CAN_BS2_7tq;
        break;
//      case (0x04):                    // 83,3 kbit/s
//          CAN_Prescaler = 42;
//          CAN_SJW = CAN_SJW_2tq;
//          CAN_BS1 = CAN_BS1_7tq;
//          CAN_BS2 = CAN_BS2_4tq;
//      break;
        case (100):                     // 100 kbit/s
```

```
            CAN_Prescaler = 42;
            CAN_SJW = CAN_SJW_2tq;
            CAN_BS1 = CAN_BS1_6tq;
            CAN_BS2 = CAN_BS2_3tq;
        break;
        case (125):                 // 125 kbit/s
            CAN_Prescaler = 42;
            CAN_SJW = CAN_SJW_2tq;
            CAN_BS1 = CAN_BS1_5tq;
            CAN_BS2 = CAN_BS2_2tq;
        break;
        case (250):                 // 250 kbit/s
            CAN_Prescaler = 21;
            CAN_SJW = CAN_SJW_2tq;
            CAN_BS1 = CAN_BS1_5tq;
            CAN_BS2 = CAN_BS2_2tq;
        case (500):                 // 500 kbit/s
            CAN_Prescaler = 21;
            CAN_SJW = CAN_SJW_2tq;
            CAN_BS1 = CAN_BS1_2tq;
            CAN_BS2 = CAN_BS2_1tq;
        break;
        case (800):                 // 800 kbit/s
            CAN_Prescaler = 4;
            CAN_SJW = CAN_SJW_2tq;
            CAN_BS1 = CAN_BS1_7tq;
            CAN_BS2 = CAN_BS2_5tq;
        break;
        case (1000):                // 1 Mbit/s
            CAN_Prescaler = 2;
            CAN_SJW = CAN_SJW_2tq;
            CAN_BS1 = CAN_BS1_12tq;
            CAN_BS2 = CAN_BS2_8tq;
        break;
        default:    //state = CAN_BautRate_Failed;
            return 0;
    }
    /* Set the bit timing register */
    CAN1->BTR = (uint32_t)((uint32_t)CAN_Mode << 30) | \
            ((uint32_t)CAN_SJW << 24) | \
            ((uint32_t)CAN_BS1 << 16) | \
            ((uint32_t)CAN_BS2 << 20) | \
           ((uint32_t)CAN_Prescaler - 1);
    return 1;
}
```

```
/**
  * @brief  This function sets a baud rate.
  * @param  None
  * @retval None
  */
uint16_t CAN_Initialization(uint16_t Bitrate, uint32_t IfaceTransferTout)
{
    uint32_t cTime, timeNow;
    cTime = MCOHW_GetTime();                // Get current time

    /* CAN1 configure and init ----------------------------------------------------------*/
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);      // Enable the clock for the CAN GPIO
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 ;         // Set output GPIOD Init Structure
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;                    // Set Alternate function Mode
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOD, &GPIO_InitStructure);                           // Initialize output GPIOD
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource0, GPIO_AF_CAN1); // Connect the involved CAN pins to AF9
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource1, GPIO_AF_CAN1); // Connect the involved CAN pins to AF9

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);  // Enable the CAN interface clock
    CAN_DeInit(CAN1);               // Deinitializes the CAN peripheral registers to their default reset values
    CAN_OperatingModeRequest(CAN1, CAN_OperatingMode_Initialization); // Specifies CAN operating mode
    CAN_BaudRate(Bitrate);          // Initializes the CAN baud rate according to the defined parameter
    CAN_OperatingModeRequest(CAN1, CAN_OperatingMode_Normal); // Specifies CAN operating mode
    CAN_ITConfig(CAN1, CAN_IT_FMP0, ENABLE);            // Enable FIFO 0 message pending Interrupt

    /* CAN1 enable interrupts -----------------------------------------------------------*/
    /* NVIC Configuration for CAN1_RX0 --------------------------------------------------*/
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
    NVIC_InitStructure.NVIC_IRQChannel = CAN1_RX0_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    /* Overeni maximalni doby cekani na presun dat mezi registry rozhrani -----*/
    timeNow = MCOHW_GetTime();
    if (timeNow > cTime)
    {
        if ((timeNow - cTime) <= IfaceTransferTout)
            return 1;
        else
            return 0;
    }
```

```
    else
    {
        if (((0xFFFF - cTime) + timeNow) <= IfaceTransferTout)
            return 1;
        else
            return 0;
    }
}

/**
  * @brief  This function sets a CAN filter.
  * @param  None
  * @retval None
  */
uint16_t CAN_SetFilter(uint32_t FilterBank, uint16_t CANID, uint16_t MaskOrList, uint16_t Fifo)
{
    if((FilterBank % 4) == 0)
        CAN_FilterInitStructure.CAN_FilterIdLow = (0x0000 | ((uint16_t)CANID << 5));
 // Specifies the filter identification number (LSBs for a 32-bit configuration, second one for a 16-bit configuration)
    if((FilterBank % 4) == 1)
        CAN_FilterInitStructure.CAN_FilterMaskIdLow = (0x0000 | ((uint16_t)CANID << 5));
// Specifies the filter mask number or identification number)
    if((FilterBank % 4) == 2)
        CAN_FilterInitStructure.CAN_FilterIdHigh = (0x0000 | ((uint16_t)CANID << 5));
// Specifies the filter identification number (MSBs for a 32-bit configuration, first one for a 16-bit configuration)
    if((FilterBank % 4) == 3)
        CAN_FilterInitStructure.CAN_FilterMaskIdHigh = (0x0000 | ((uint16_t)CANID << 5));
// Specifies the filter mask number or identification number)
    CAN_FilterInitStructure.CAN_FilterFIFOAssignment = Fifo;
// Specifies the FIFO (0 or 1) which will be assigned to the filter
    CAN_FilterInitStructure.CAN_FilterNumber = (FilterBank / 4);    // Specifies the filter which will be initialized
    CAN_FilterInitStructure.CAN_FilterMode = MaskOrList;      // Specifies the filter mode to be initialized
    CAN_FilterInitStructure.CAN_FilterScale = CAN_FilterScale_16bit;     // Specifies the filter scale
    CAN_FilterInitStructure.CAN_FilterActivation = ENABLE;   // Enable or disable the filter
    CAN_FilterInit(&CAN_FilterInitStructure);
// Configures the CAN reception filter according to the specified parameters
    return 1;
}

/**
* @}
*/

/*********************************END OF FILE*********************************/
```

Příloha 4 – Zdrojový kód souboru mco.c

```
/* Includes ------------------------------------------------------------------*/
#include "mco.h"
#include "mcohw.h"

/**
  * @brief  This function initializes the CANopen protocol stack.
  * @param  bitrate - may be one of the values 1000, 800, 500, 250, 125, 50, 25 or 10
     Node_ID - may be in the range of 1 to 126
     heartbeat - heartbeat time is specified in milliseconds
  * @retval None
  */
void MCO_Init(uint16_t bitrate, uint8_t Node_ID, uint32_t heartbeat)
{
    MCOHW_Init(bitrate);
}

/**
* @}
*/

/**********************************END OF FILE**********************************/
```

Příloha 5 – Zdrojový kód souboru LCD16x2lbr.c

```
// LCD Display 16x2 LIBRARY
// Modified to work with STM32 Cortex M4

#include "stm32f4xx_rcc.h"
#include "LCD16x2lbr.h"

/*****************************************
          P I N S   D E F I N E
*****************************************/
#define PULS_ENABLE {GPIO_WriteBit(GPIOB, GPIO_Pin_9, Bit_SET); GPIO_WriteBit(GPIOB, GPIO_Pin_9,
Bit_RESET);}
#define SET_DATA(d) {GPIO_Write(GPIOE, d);}
/*****************************************/

unsigned char b, bb;
unsigned int        i = 0, cek;                    // promenna pro cekani

void delay_ms(unsigned int cas)                    // cekaci funkce
```

```
{
    unsigned int i;
    for(i=0;i<cas;i++)
    {
        for(cek=0; cek<6500; cek++)
            ;
    }
}

void delay_us(unsigned int cas)          // cekaci funkce
{
    unsigned int i;
    for(i=0;i<cas;i++)
    {
        for(cek=0; cek<100; cek++)
            ;
    }
}

void write_data(unsigned char b)
{
    GPIO_WriteBit(GPIOB, GPIO_Pin_8, Bit_SET);
    bb = b & 0xf0;
    bb >>= 4;
    SET_DATA(bb);
    PULS_ENABLE;
    delay_ms(2);                         // cekani
    bb = b & 0x0f;
    SET_DATA(bb);
    PULS_ENABLE;
    delay_ms(2);                         // cekani
}

void write_cmd(unsigned char b)
{
    GPIO_WriteBit(GPIOB, GPIO_Pin_8, Bit_RESET);
    bb = b & 0xf0;
    bb >>= 4;
    SET_DATA(bb);
    PULS_ENABLE;
    delay_ms(2);                         // cekani
    bb = b & 0x0f;
    SET_DATA(bb);
    PULS_ENABLE;
    delay_ms(2);                         // cekani
```

```
}

void write_lcd(unsigned char b)
{
    write_data(b);
}

void clear_lcd(void)                    // For Clearing the screen.
{
    write_cmd(0x01);
    delay_ms(5);                        // cekani
}

void write_string(char *msg)            // For Writing Text Message at Current cursor position.
{
    while(*msg!='\0')
    {
        write_lcd(*msg);
        msg++;
    }
}

void GoToXY(char sloupec, char radek)           // prikaz nastav kurzor na radek,sloupec
{
    if(sloupec<40)
    {
        if(radek)
            sloupec|=0x40;                      // druhy radek
        sloupec|=0x80;                          // prvni radek
        write_cmd(sloupec);
    }
}

void init_lcd(void)
{
    delay_ms(70);                               // cekani

    SET_DATA(0x03); // init 8-bit    0000 0011
    PULS_ENABLE;
    delay_ms(10);                               // cekani

    SET_DATA(0x03); // init 8-bit    0000 0011
    PULS_ENABLE;
    delay_ms(10);                               // cekani
```

```
    SET_DATA(0x03);                          // init 8-bit 0000 0011
    PULS_ENABLE;
    delay_ms(10);                            // cekani

    SET_DATA(0x02); // zapni 4-bit 0000 0010
    PULS_ENABLE;
    delay_ms(4);                             // cekani

    write_cmd(0x2c);                         // 0010 1100 8/4 bitovy rezim, 1/2 radky, samotný znak
    write_cmd(0x0c);                         // 0000 1100 display on/off, cursor on/off, brinking on/off
    clear_lcd();                             // 0000 0001 maze displej
    write_cmd(0x06);                         // 0000 0110 increment/decrement, ne/posunutí displeje
}

void hexa_cisla(char cislo)
{
    if(cislo>=0 && cislo <=9)
        write_lcd(cislo + 48);
    if(cislo==0xA)
        write_lcd('A');
    if(cislo==0xB)
        write_lcd('B');
    if(cislo==0xC)
        write_lcd('C');
    if(cislo==0xD)
        write_lcd('D');
    if(cislo==0xE)
        write_lcd('E');
    if(cislo==0xF)
        write_lcd('F');
}

void zobrazLCD_hexa(unsigned long long hex)
{
    char nibble, kroky;
    unsigned long long hexaMaska;
    hexaMaska = 0xf000000000000000;
    GoToXY(0,0);
    nibble=15;
    for(kroky=0;kroky<=15;kroky++)           // provede se 16x, protoze cislo muze mit 64 bitu, coz je 16 niblu
    {
        hexa_cisla((hex & hexaMaska) >> nibble*4);
// vybere se postupne jeden nibl maskou, posune se na zacatek a vypise se na displej
        hexaMaska = hexaMaska >> 4;          // upraveni masky - posun F o jednu pozici doprava
        nibble--;
```

```
    }
}
```

Příloha 6 – Zdrojový kód souboru stm32f4xx_it.c

```
/* Includes ------------------------------------------------------------------*/
#include "stm32f4xx_it.h"
#include "LCD16x2lbr.h"

/* Private variables ---------------------------------------------------------*/
extern unsigned short int volatile TimCnt1ms;
/* Private structures ---------------------------------------------------------*/
extern CanRxMsg RxMessageStructure;

/**
  * @brief  CAN1_RX0_IRQHandler
    This function handles CAN1_RX0 interrupt request.
  * @param  None
  * @retval None
  */
void CAN1_RX0_IRQHandler(void)
{
    /* signalizace prichodu zpravy pomoci zelene LED ----------------------------------------*/
    if((CAN1->RF0R & 0x03) > 0)
        GPIO_WriteBit(GPIOD, GPIO_Pin_12, Bit_SET);         // GPIO_Pin_12 zelenaLED LD4
    else
        GPIO_WriteBit(GPIOD, GPIO_Pin_12, Bit_RESET);

    /* signalizace preteceni FIFA pomoci cervene LED ----------------------------------------*/
    if((CAN1->RF0R & 0x10) == 0x10)
        GPIO_WriteBit(GPIOD, GPIO_Pin_14, Bit_SET);         // GPIO_Pin_14 cervenaLED LD5

    /* signalizace plneho FIFA pomoci oranzove LED ----------------------------------------*/
    if((CAN1->RF0R & 0x08) == 0x08)
        GPIO_WriteBit(GPIOD, GPIO_Pin_13, Bit_SET);         // GPIO_Pin_13 oranzovaLED LD3
    else
        GPIO_WriteBit(GPIOD, GPIO_Pin_13, Bit_RESET);

    /* Prijem Rx zpravy ----------------------------------------------------------------*/
    CAN_Receive(CAN1, CAN_FIFO0, &RxMessageStructure);

    /* vypis na diplej -----------------------------------------------------------------*/
    zobrazLCD_hexa((uint64_t)RxMessageStructure.Data[0]
        | ((uint64_t)RxMessageStructure.Data[1] << 8)
        | ((uint64_t)RxMessageStructure.Data[2] << 16)
```

```
        | ((uint64_t)RxMessageStructure.Data[3] << 24)
        | ((uint64_t)RxMessageStructure.Data[4] << 32)
        | ((uint64_t)RxMessageStructure.Data[5] << 40)
        | ((uint64_t)RxMessageStructure.Data[6] << 48)
        | ((uint64_t)RxMessageStructure.Data[7] << 56));        // na prvnim radku displeje zobrazi prijata data
}

/**
  * @brief  TIM2_IRQHandler
  *         This function handles Timer2 Handler.
  * @param  None
  * @retval None
  */
void TIM2_IRQHandler(void)
{
    if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
    {
        TIM_ClearFlag(TIM2, TIM_FLAG_Update);
        TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
    }
    TimCnt1ms++;
}

/**
* @}
*/

/****************** (C) COPYRIGHT 2011 STMicroelectronics *****END OF FILE****/
```

Příloha 7 – Zdrojový kód mcohw.h

```
/* Includes ------------------------------------------------------------------*/
#include "stm32f4xx.h"

/* Exported types ------------------------------------------------------------*/
// Data structure for a single CAN message
typedef struct
{
  uint32_t ID;              // Message Identifier
  uint8_t LEN;               // Data length (0-8)
  uint8_t BUF[8];            // Data buffer
} CAN_MSG;
/* Exported functions ------------------------------------------------------- */
uint16_t MCOHW_Init(uint16_t BaudRate);
uint16_t MCOHW_SetCANFilter(uint16_t CANID);
```

```
uint16_t MCOHW_PushMessage(CAN_MSG *pTransmitBuf);
uint16_t MCOHW_PullMessage(CAN_MSG *pReceiveBuf);
uint32_t MCOHW_GetTime(void);
uint8_t MCOHW_IsTimeExpired(uint32_t timestamp);

/************************END OF FILE*****************************************/
```