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 2015

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta elektrotechnická
Akademický rok: **2014/2015**

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: **15. října 2014**

Termín odevzdání diplomové práce: **11. května 2015**

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

Západočeská univerzita v Plzni
Fakulta elektrotechnická
4

Doc. Dr. Ing. Vjačeslav Georgiev
vedoucí katedry

V Plzni dne 15. října 2014

Abstrakt

Herink Ondřej. MicroCANopen na platformě STM32. Katedra aplikované elektroniky a telekomunikací, Západočeská univerzita v Plzni – Fakulta elektrotechnická, 2015, vedoucí: Ing. Petr Krist, Ph.D.

Předkládaná diplomová práce se zabývá specifikacemi komunikačního protokolu CAN, popisuje princip sběrnice CAN. 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, budič sběrnice, 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, 2015, head: Ing. Petr Krist, Ph.D.

This thesis deals with the specification of the communication CAN protocol, describes the principle of CAN bus. 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, bus driver, 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 11.5.2015 Jméno příjmení

……..

Poděkování

Tímto bych rád poděkoval vedoucímu diplomové práce Ing. Petru Kristovi, Ph.D. za vedení práce.

Obsah

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. 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 vybraný zdrojový kód.

Seznam symbolů a zkratek

ACK	Acknowledge – potvrzovací bit bezchybného přijetí zprávy
BTR	Bit timing registr – registr pro nastavení přenosové rychlosti
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
EOF	End Of Frame – ukončovací rámec zprávy CAN obsahující sedm bitů
ERC	End of CRC – bit, který ukončuje CRC pole v CAN zprávě
FIFO	First In, First Out – (první dovnitř, první ven), paměť fronty
GND	GrouND – zemnící potenciál, referenční bod
Heartbeat	Speciální zpráva ke zjištění dostupnosti stanice
I^2C	Inter-Integrated Circuit – multi-masterová počítačová sériová sběrnice
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
MCO	MicroCANopen
MSB	Most Significant Bit – nejvýznamnější bit
NMT	Network Manager Master – souvisí s master stanicí
NVIC	Nested Vectored Interrupt Controller – řízení vektoru přerušení
OD	Object Dictionary – adresář deskriptorů

OTG	On-The-Go – pracuje jako hostitel/periferie nebo obráceně
PDO	Process Data Object – procesní datové objekty
Pointer	Ukazatel v paměti na určitou adresu
RCC	Reset and clock control – ovládání hodin pro periférie
RTR	Remote Transmision Request – bit určující, zda CAN správa obsahuje data, nebo o ně žádá
Rx	Receive – označení signálu pro příjem
SCE	Status Change Error – chyba změny stavu
SJW	Synchronization Jump Width – programovatelná šířka synchronizačního skoku pro prodloužení či zkrácení šířky jednoho bitu
Sleep mod	Mikrokontrolér se nachází ve stavu „spánku“
SOF	Start Of Frame – první bit označující začátek rámce CAN zprávy
SRR	Substitute Remote Request – bit, který nahrazuje RTR bit v původním standardním CAN 2.0A rámci
Stack	Zásobník – datová struktura
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
Watchdog	tzv. „hlídací pes“

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 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 zachytí každý uzel na sběrnici. Jedná se o tzv. Broadcast zprávy. Zachycená zpráva pak musí projít přes přijímací filtr, který rozhodne, zda zpráva bude přijata. 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 hodnotu. Uzel dokáže detekovat chybu při odesílání zprávy (jedná se o případ, kdy na sběrnici právě vysílá jiný uzel) 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 (arbitráž), vytváření a dekódování rámců pro data, zabezpečení dat, řízení rámce, filtrování příchozích zpráv, potvrzování bezchybného příjmu zpráv, detekci a signalizaci chyb. Některé definované položky standardu CAN budou dále stručně popsány nebo vysvětleny. [13]

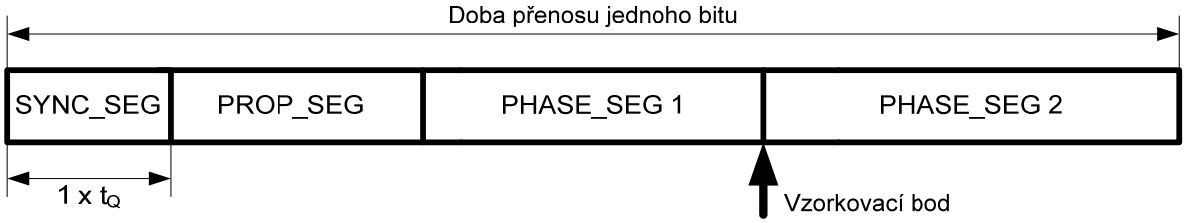
1.3 Arbitráž

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 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. [12]

1.4 Doba přenosu jednoho bitu

Klade se velký důraz na časování sběrnice, protože jak už víme z kapitoly 1.3 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 synchronizace neexistuje, protože sem zasahují vlivy zpoždění signálu, jitter, nebo reálné vlastnosti vedení. Z toho důvodu jsou vylepšeny možnosti konfigurace časování přenosu jednoho bitu.



Obr. 2: Časové segmenty jednoho bitu

Na obr. 2 je znázorněna časová šířka jednoho bitu. Tato šířka je rozdělena na čtyři 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 dobami 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.

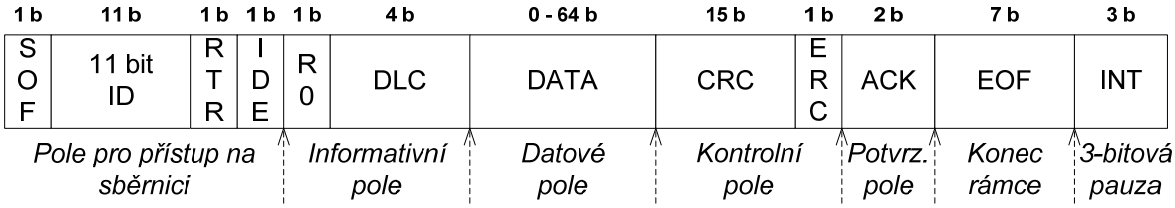
Někdy při nastavování přenosové rychlosti se PROP_SEG vůbec neuvádí a je zahrnut v PHASE_SEG 1. V podstatě se dá nastavit libovolná přenosová rychlost, ovšem maximálně do 1Mbit/s z důvodu přísných požadavků na časování signálů. Nejčastěji se používají přenosové rychlosti: 5; 10; 20; 50; 100; 125; 250;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

V mé práci je použitá přenosová rychlost sběrnice 100 kbit/s. Jedno časové kvantum v tomto případě trvá 1 μs. Doba PHASE_SEG 1 je spočtena na šest časových kvant a doba PHASE_SEG 2 na 3 časová kvanta. Synchronizační skok SJW je zvolen na dvě časová kvanta. Zpožďovací segment PROP_SEG je zahrnut v PHASE_SEG 1. [12]

1.5 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 jedenáct bitů a rozšířený formát CAN 2.0B má identifikátor dlouhý dvacet devět bitů. Podle bitové délky identifikátoru se na sběrnici může přenášet dvě stě jedenáct nebo dvě stě dvacet devět objektů. Na jedné sběrnici se smí používat oba dva formáty zpráv, avšak vyšší prioritu má formát CAN 2.0A. [12]

Standardní formát CAN 2.0A



Obr. 3: Formát standardního rámce CAN 2.0A

Na obr. 3 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 jen 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 tříbitová pauza, která zajistí čas pro přijímač právě zpracovávající zprávu. Toto tříbitové pole INT je v recesivním stavu. [12]

Rozšířený formát rámce CAN 2.0B v mé práci není použit, a proto jej nemá smysl více rozebírat. Více informací se dá získat v odkaze [12].

1.6 Typy zpráv

Jak již bylo zmíněno v kapitole 1.5, 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. 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, která 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 nejdříve 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 1.5) 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), 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 takové situace vyřeší. Jakým způsobem funguje takový nástroj, je popsáno v následující kapitole. [12]

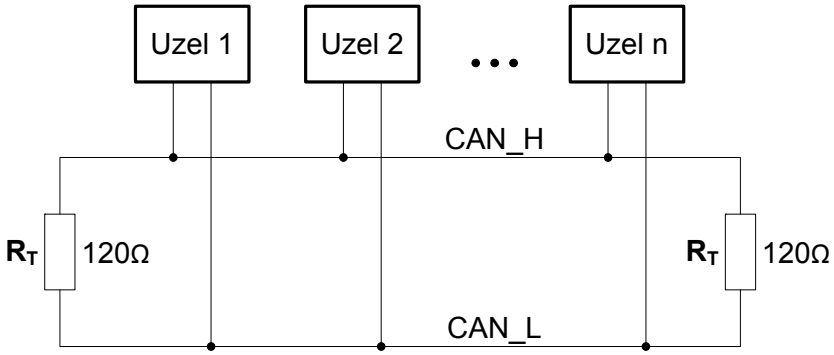
1.7 Čítače chybových stavů

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 osm a při správném příjmu nebo odvysílání se dekrementuje pouze o jeden. Jeden chybný stav se tedy 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 odpojí uzel 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 sto dvacet osm stavů v nečinnosti. Jeden časový stav odpovídá šířce jedenáct bitů, takže musí počkat celkem čtrnáct set osm bitových dob a pak se může znovu připojit na sběrnici.

1.8 Propojení sběrnice CAN

Fyzické propojení uzlů se sběrnicí CAN předvádí obr. 4. 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. 4: Obecné uspořádání sběrnice CAN

Vodiče CAN_H a CAN_L jsou na obou koncích zakončeny odpory o hodnotě 120 Ω, jimž se říká terminátory. Spolu s dvěma vodiči je také vhodné vést referenční vodič (na obr. 4 není zakreslen), od kterého se odvíjí zemnící potenciál budičů uzlů. V mé práci je tento referenční vodič uplatněn. [12]

2 MicroCANopen

MicroCANopen implementuje nezbytné vlastnosti, služby a funkce tak, aby daný uzel mohl bezproblémově komunikovat s uzly implementující komunikační protokol CANopen. MicroCANopen je koncipován tak, aby se z něho mohl stát plnohodnotný protokol CANopen pouhým přidáním potřebných služeb. V následující tabulce je srovnání vlastností protokolů CANopen a MicroCANopen.

	CANopen	**MicroCANopen**
Přenosová rychlost [kb/s]	10, 20, 50, 125, 250, 500, 800, 1000	10, 20, 50, 125, 250, 500, 800, 1000
Maximální počet uzlů v segmentu	127	127
Řízení sítě *(Network Management)*	NMT Manager nebo autostart	NMT Manager nebo autostart
Test aktivity uzlu *(Node guarding/heartbeat)*	Node guarding/heartbeat	Heartbeat
Podpora chybových zpráv *(Emergency support)*	Ano	Ne
Konfigurace uzlů	Uzly mohou být konfigurovány přes síť	Konfigurace přednastavena, nelze měnit během funkce
Uložení změny konfiguračních parametrů	Ano	Ne
Položky *Object Dictionary*	Jakákoli velikost a typ	Datové typy s velikostí do 32 bitů
Maximální velikost procesních dat	Neomezena	254 byte
Maximální počet PDO	1024	8
Přístup k PDO přes *Object Dictionary*	Ano	Ne
Konfigurace a mapování PDO v *Object Dictionary*	Ano	Ne
Způsob spouštění CAN zpráv obsahujících PDO	Jakákoli kombinace na základě časového spouštění, dotazu, změny stavu a synchronizovaného spouštění. Podpora *Inhibit Time*.	Na základě časového spouštění (time-triggered), změny stavu *(COS)* . Podpora *Inhibit Time*.

Tab. 1: Srovnání vlastností protokolu CANopen a MicroCANopen

2.1 Ovladače komunikačního profilu MicroCANopen

Komunikační profil MicroCANopen se skládá z hardwarového ovladače neboli driveru a aplikační vrstvy. Oba dva bloky musí implementovat povinné funkce.

2.1.1 Hardwarový ovladač

Hlavním úkolem hardwarového ovladače je nastavit hardwarové rozhraní CAN a implementovat časovač s hodinovým tikem jedné milisekundy. Dalším jeho úkolem je nastavit filtrační banky, přijmout a odeslat zprávu, získat statusy a měřit čas.

Povinné funkce hardwarového ovladače jsou:

BYTE MCOHW_GetStatus (void);
BYTE MCOHW_Init (WORD BaudRate);
BYTE MCOHW_SetCANFilter (WORD CANID);
*BYTE MCOHW_PushMessage (CAN_MSG *pTransmitBuf);*
*BYTE MCOHW_PullMessage (CAN_MSG *pTransmitBuf);*
WORD MCOHW_GetTime (void);
BYTE MCOHW_IsTimeExpired (WORD timestamp);

2.1.2 Aplikační vrstva

Aplikační vrstva se rozděluje na dvě sady funkcí. Jedna sada jsou přímé funkce, které lze vyvolat v průběhu chodu aplikace inicializující komunikační stack MicroCANopen a které umí předat procesní data do komunikačního stacku. Druhá sada jsou „call-back“ funkce, které vyvolává MicroCANopen, aby informovaly aplikaci o důležitých událostech či příjmu procesních dat.

Sada přímých funkcí:

void MCO_Init (WORD Baudrate, BYTE Node_ID, WORD heartbeat);
*void MCO_InitTPDO (BYTE PDO_NR, WORD CAN_ID, WORD event_tim, WORD inhibit_tim, BYTE len, BYTE *dat);*
*void MCO_InitRPDO (BYTE PDO_NR, WORD CAN_ID, BYTE len, BYTE *dat);*
BYTE MCO_ProcessStack (void);

Sada „call-back“ funkcí:

void MCOUSER_ResetApplication (void);
void MCOUSER_ResetCommunication (void);
void MCOUSER_FatalError (WORD ErrCode);

Vysvětlení všech funkcí bude popsáno později v kapitole 4.

2.2 Protokolový stack MicroCANopen

Obsluha protokolového stacku MicroCANopen je vyvolána cyklicky z hlavní funkce *main()*. O kompletní obsluhu protokolového stacku se stará funkce *MCO_ProcessStack()*. Realizuje příjem a vysílání veškerých procesních, servisních, diagnostických a řídících zpráv protokolu. Obsluha různých požadavků se provádí sekvenčně. S každým vyvoláním funkce se tedy provede jen jeden požadavek pro zpracování události. Algoritmus protokolového stacku je znázorněn v simulovaném kódu, převzatého z původního doporučení [14].

```
BEGIN MCO_ProcessStack

__IF CAN message received
____IF NMT master message received
______Handle NMT state machine
______Exit MCO_ProcessStack
____ELSEIF SDO request (OD access) received
______Handle SDO request
______Exit MCO_ProcessStack
____ELSEIF PDO message received
______Copy data received to destination location
______Exit MCO_ProcessStack
____ENDIF
__ENDIF CAN message received

__IF NMT state is "operational"
____Check next TPDO (one with each call to this function),
____is TPDO due for transmit?:
____IF EVENT_TIME is specified and EVENT_TIME is expired
______Copy process data to transmit message buffer
______Reset event timer
______Queue TPDO message for transmit
______Exit MCO_ProcessStack
____ELSEIF INHIBIT_TIME is specified
______IF message is already pending (waiting for inhibit to expire)
________Reset inhibit timer
________Queue TPDO message for transmit
________Exit MCO_ProcessStack
______ELSEIF data changed since last transmit
________Copy process data to transmit message buffer
________IF inhibit time already expired
__________Reset inhibit timer
__________Queue TPDO message for transmit
__________Exit MCO_ProcessStack
________ELSE
```

```
__________Mark message as pending (waiting for inhibit to expire)
________ENDIF
______ENDIF
____ENDIF EVENT_TIME / INHIBIT_TIME used
__ENDIF NMT state "operational"

__IF heartbeat expired
____Reset Heartbeat timer
____Queue Heartbeat message for transmit
__ENDIF

END MCO_ProcessStack
```

2.3 Charakteristika protokolového stacku MCO

Všechny uzly na sběrnici musí být nastaveny na stejnou přenosovou rychlost. Každý uzel má přiřazené své sedmibitové identifikační číslo ID_Node. Maximální počet uzlů na sběrnici je tedy sto dvacet sedm. Identifikační číslo uzlu spolu se součtem identifikátoru typu zprávy konkrétní služby tvoří jedenácti bitový identifikátor CAN.

Každý uzel, který je právě po inicializaci, odešle boot-up zprávu master stanici a přepne do předoperačního módu. V tomto módu nemůže přijímat ani vysílat PDO objekty. Uzel jen cyklicky odesílá heartbeat zprávu s informací, že je s tomto módu až do té doby, než mu master stanice dá příkaz (NMT zpráva), aby přešel do operačního módu. Uzel, který má nastaven autostart, na tento příkaz nečeká a rovnou sám přepne do operačního módu. V tomto módu může uzel vysílat i přijímat PDO objekty, zpracovávat SDO služby a ECMY zprávy a odesílat heartbeat zprávu. Ve stop módu je zastavena komunikace, uzel může zpracovávat jen NMT zprávy.

SDO služby odesílá master stanice na konkrétní uzel, který službu zpracuje, případně může master stanici odpovědět (request - response). Oproti tomu PDO objekty může vysílat i přijímat kterýkoli uzel. PDO objekty předávají procesní data a dokonce jeden PDO objekt dokáže předat více proměnných v jedné zprávě. TPDO objekty mohou být vysílány třemi způsoby. Pokud je nastavena hodnota event-time, TPDO objekt se vysílá periodicky v určitém časovém intervalu. Pokud je nastavena hodnota inhibit-time, TPDO objekt se vyšle při každé změně datové hodnoty. Aby se nezahltila sběrnice, musí být splněna podmínka minimální časové prodlevy inhibit-time mezi vysíláním TPDO zpráv. Třetí způsob vysílání TPDO objektů je kombinace dvou předchozích. Objekt se tedy vysílá podle změny datové hodnoty, ale když ke změně nedojde, zpráva se vysílá periodicky podle časového intervalu event-time.

Přijímací RPDO a vysílací TPDO zprávy mají jasně definovaný CAN identifikátor. V následující tabulce je vypsán přehled všech objektů a popsán jejich význam.

Přijímané zprávy		
CAN ID	**Označení**	**Význam**
0x000	**NMT**	Řídící zpráva Master stanice
0x200 + ID_Node	**RPDO1**	Přijímací procesní objekt 1
0x300 + ID_Node	**RPDO2**	Přijímací procesní objekt 2
0x400 + ID_Node	**RPDO3**	Přijímací procesní objekt 3
0x500 + ID_Node	**RPDO4**	Přijímací procesní objekt 4
0x600 + ID_Node	**SDO Rx**	Požadavek servisní služby
Vysílané zprávy		
CAN ID	**Označení**	**Význam**
0x080 + ID_Node	**EMCY**	Chybová zpráva uzlu
0x180 + ID_Node	**TPDO1**	Vysílací procesní objekt 1
0x280 + ID_Node	**TPDO2**	Vysílací procesní objekt 2
0x380 + ID_Node	**TPDO3**	Vysílací procesní objekt 3
0x480 + ID_Node	**TPDO4**	Vysílací procesní objekt 4
0x580 + ID_Node	**SDO Tx**	Odpověď servisní služby
0x700 + ID_Node	**Boot-Heart**	Informace o aktivitě uzlu

Tab. 2: Přehled RPDO a TPDO objektů

Adresář deskriptorů Object Dictionary (OD) obsahuje objekty komunikačního profilu, konfigurační objekty, procesní objekty a objekty definované výrobcem. Každý objekt má definovaný svůj šestnáctibitový index a osmibitový subindex. Seznam povinných položek OD je uspořádán v tab. 3.

Index	**Subindex**	**Délka [bit]**	**Popis**	**Význam**
1000H	00H	32	Device type	Typ zařízení
1001H	00H	8	Error Register	Chybový registr
1018H	00H	8	Number Subindexes	Počet podindexů záznamu = 4
1018H	01H	32	Vendor ID	Kód výrobce
1018H	02H	32	Product Code	Číslo produktu
1018H	03H	32	Revision Number	Číslo revize
1018H	04H	32	Serial Number	Sériové číslo

Tab. 3: Povinné položky Object Dictionary

Device type – prvních šestnáct bitů značí číslo komunikačního profilu (typicky DS401), dalších šestnáct bitů obsahuje informace profilu

Error Register – jednotlivé bity signalizují obecnou chybu, chybu proudu, chybu napětí, chybu teploty, chybu komunikace, chybu definovanou komunikačním profilem, chybu specifikovanou výrobcem zařízení
Popis dalších položek je jasný ze sloupce Význam a není tedy potřeba je dále popisovat.

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.

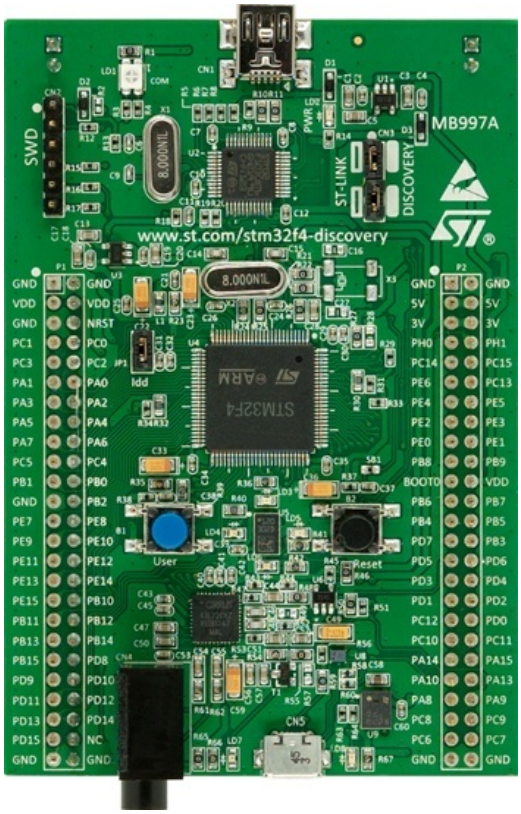
Diagnostický nástroj je dvouřádkový šestnácti znakový (16x2) LCD displej ATM1602B. Pomocí něhož se na prvním řádku zobrazuje chybový kód v hexadecimálním tvaru (viz dále popsaná funkce *MCOUSER_FatalError()*) a na druhém řádku se zobrazuje stavový registr v binárním tvaru (viz dále popsaná funkce *MCOHW_GetStatus()*).

Discovery kit postrádá budič a konektor pro CAN sběrnici, takže se vyrobil 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. 5.



Obr. 5: 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. Discovery kit je napájen 5V z USB konektoru, který rovněž slouží pro programování a debug. Dále se na kitu nachází čtyři LED uživatelské diody, dvě tlačítka (User a Reset), dvě dvouřadé lišty 2x25 pro vstupy/výstupy mikrokontroléru, na nichž je připojen displej a modul budiče. [8]

Čtyři uživatelské diody jsou použity jako další pomocný ladící nástroj při přijímání zpráv. Pokud přijatá zpráva projde přes nastavené filtry, rozsvítí se zelená LED dioda. Ta svítí do té doby, než je zpráva zpracována. Pokud se zprávy nestíhají zpracovávat a FIFO paměť se všemi třemi schránkami pro přijaté zprávy se zaplní, oranžová dioda začne signalizovat zaplnění paměti FIFO. Červená dioda signalizuje přetečení paměti a ztrátu dat. Modrá dioda vykonává příkazy na základě přijaté RPDO zprávy. RPDO zpráva s identifikátorem 0x207 a datovou hodnotou 1 zapříčiní rozsvícení diody. Zpráva s tím samým identifikátorem, ale s datovou hodnotou 0, diodu zase zhasne. Tlačítko User má také své uplatnění při zjišťování funkčnosti odesílání zpráv. Pokud není tlačítko sepnuté, vysílá se periodicky zpráva TPDO

s identifikátorem 0x187 a datovou hodnotou 0. Po dobu sepnutí tlačítka se odesílá ta samá zpráva s datovou hodnotou 1.

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.

Mikrokontrolér STM32F407

Hlavní součástkou na Discoverx kitu je samozřejmě 32-bitový mikrokontrolér STM32F407 na jádře ARM Cortex-M4 taktéž od firmy STMicroelectronics. V tab. 2 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		

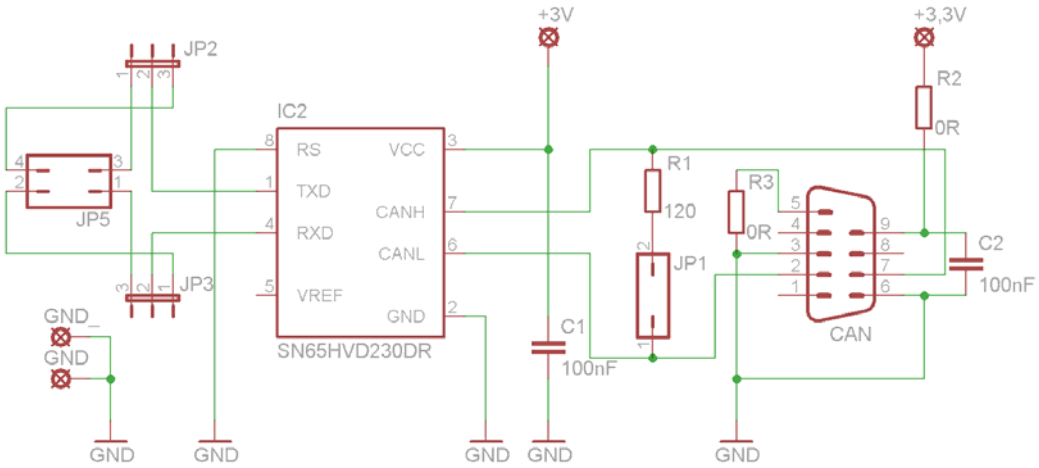
Tab. 4: Parametry mikrokontroléru STM32F407 [7]

3.2 Budič SN65HVD230DR

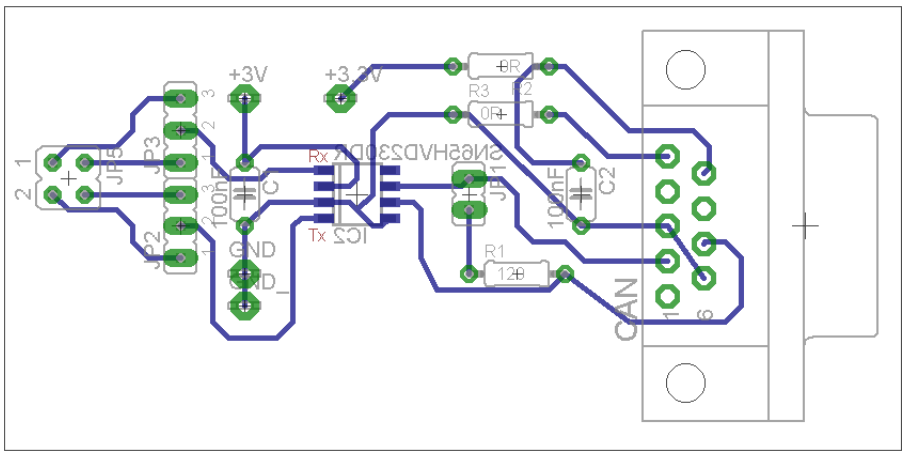
Jelikož Discovery kit nemá v sobě implementovaný budič CAN, musel být vyroben modul s budičem a konektorem. Modul určený přímo pro Discovery kit se sice dá koupit, ale výroba modulu vyšla mnohem levněji. K Discovery kitu se tento modul externě připojuje. Budič je vlastně převodník mezi TTL úrovní signálů 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 stand-by režimy. High-speed režim umožňuje vysokorychlostní přenos po sběrnici. Slope control režim se používá pro low-speed

přenosovou rychlost. Stand-by 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 stand-by režimu přepne na jiný režim. Tento režim má ještě jedno chytré využití. Když se řadič CAN dané řídící jednotky vymkne kontrole, budič přejde do stand-by režimu, tím se zablokuje komunikace a jednotka se vyřadí z provozu. Budič v mém modulu je nastavený na high-speed režim a změna na jiný režim není možná. Na obr. 6 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 1.8. 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. 7) byla vytvořena v návrhovém prostředí Eagle. [3]



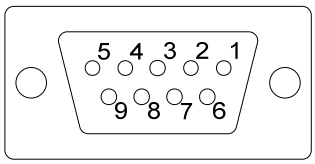
Obr. 6: Schéma modulu budiče



Obr. 7: Deska plošných spojů modulu budiče

CAN konektor

Pro úplné vysvětlení je zde ještě popsán CAN konektor DE-9, který je typický pro tuto sběrnici. Na obr. 8 je znázorněno, jak jsou v konektoru umístěné jeho piny a v tab. 3 je popsán význam jednotlivých pinů.



Obr. 8: 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. 5: Popis pinů CAN konektoru DE-9 [1]

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. 9 je ukázka analyzátoru s CD, na kterém je uložena aplikaci USB-CAN Adapter V4.11, ovladače pro analyzátor a jiné dokumenty.

Obr. 9: Analyzátor USB-CAN Adapter TRIPLE drivers V4.2 s instalačním CD [10]

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)

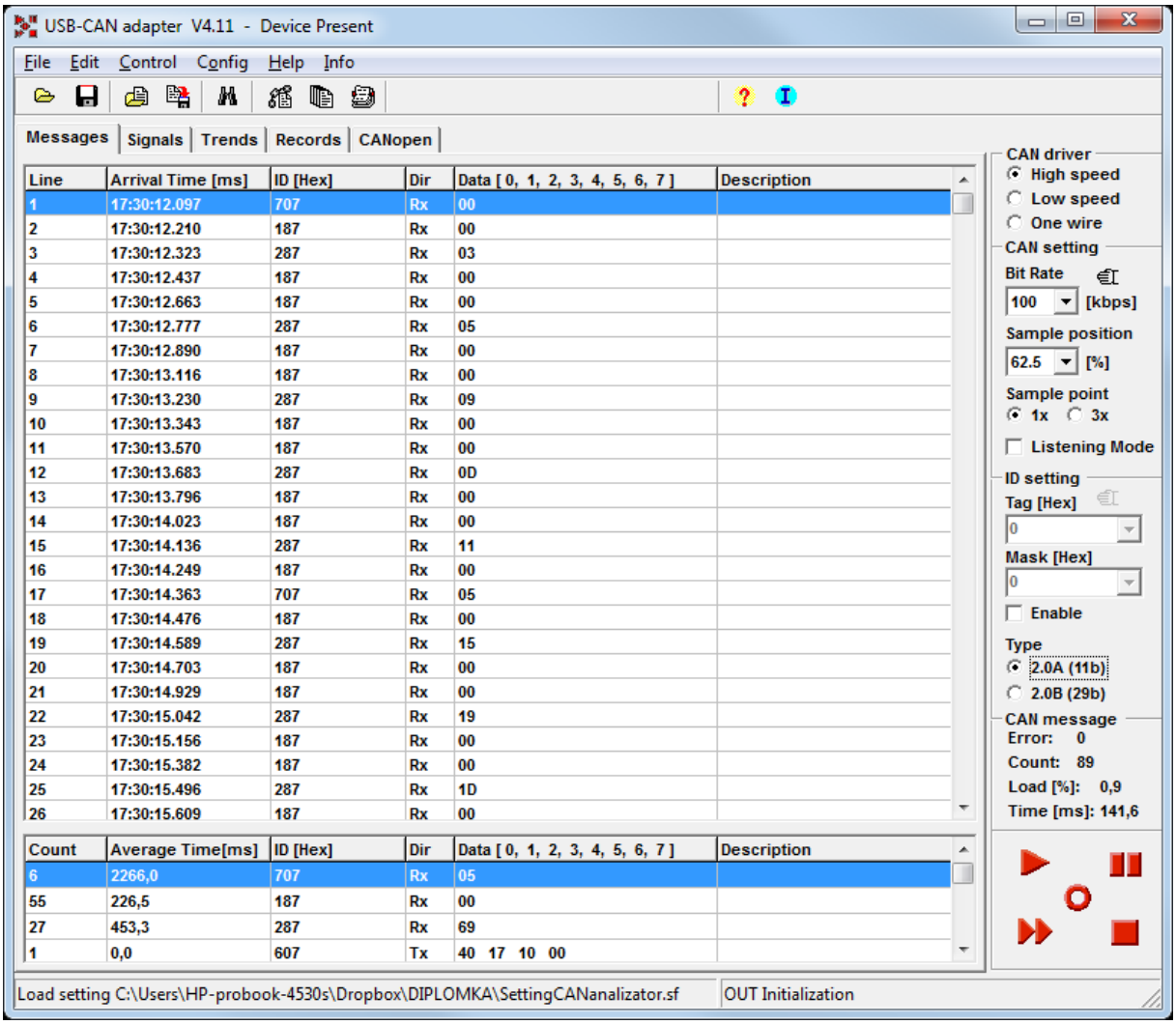
- Standardní zapojení CANNON konektoru
- Napájení ze sběrnice USB
- Ochrana proti přepětí a přepólování (Transil)“[9]

Aplikace USB-CAN Adapter

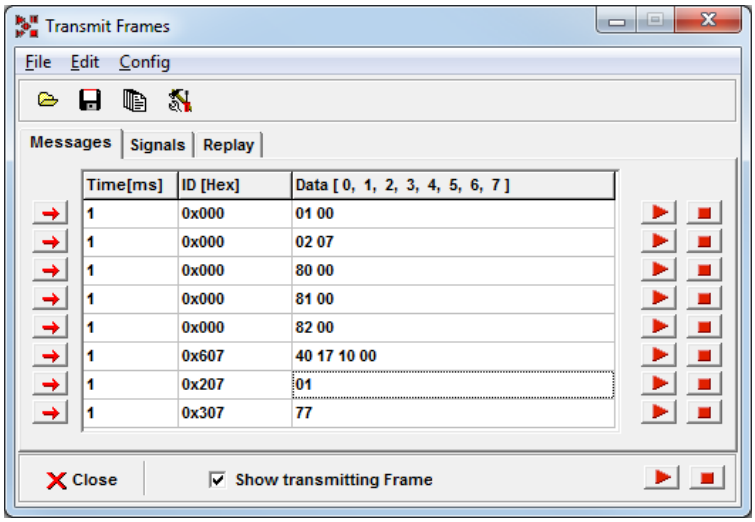
Aplikace USB-CAN Adapter je uživatelsky velmi přívětivá a snadná na ovládání. Po instalaci aplikace a připojení analyzátoru na sběrnici stačí jen nastavit přenosovou rychlost sběrnice CAN, inicializovat adapter a v tento okamžik se už monitoruje činnost na sběrnici. Hlavní okno aplikace obsahuje prvky 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 patnácti 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 vysílání CAN zpráv se stará vedlejší okno Transmit Frames. V tomto okně se dá nastavit až osm různých CAN rámců. U každého rámce se nastaví identifikátor a datové pole. Lze také nastavit, odeslání zprávy jen jednou nebo cyklicky podle nastavitelného intervalu. Rámce se mohou odesílat jednotlivě nebo postupně všechny za sebou. Odeslané rámce se samozřejmě zobrazují v záložce Messages. Na obr. 10 je ukázka uživatelského prostředí hlavního okna se záložkou Messages a na obr. 11 je vedlejší okno Transmit Frames. [9]



Obr. 10: Hlavní okno záložky Messages



Obr. 11: Vedlejší okno Transmit Frames pro 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.

Na webových stránkách [14] společnosti Embedded Systems Academy, GmbH, zabývající se produkty, poradenstvím a školením Embedded systémů, byla ukázka příkladu zdrojového kódu MicroCANopen. Tímto příkladem byla má práce inspirována.

V následujících podkapitolách budou popsány hardwarově závislé moduly *can.c, mcohw.c, stm32f4xx_it.c, LCD16x2lbr.c* a hardwarově nezávislé moduly *mco.c, main.c*. Bude také vysvětlena činnost jejich funkcí. Každý z těchto modulů kromě *main.c* má svůj hlavičkový soubor, kde jsou deklarovány názvy všech funkcí příslušných modulů, případně konstanty a struktury. Některé vybrané části hlavičkových souborů budou taktéž popsány.

4.1 Zdrojový modul can.c

Zdrojový modul *can.c* obsahuje celkem tři funkce:

- *uint16_t CAN_BaudRate (uint16_t BaudRate)* – Funkce umí nastavit přenosovou rychlost sběrnice. Na výběr je z devíti přenosových rychlostí (1000, 800, 500, 250, 125, 100, 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 1.4. Podle zvolené přenosové rychlosti se nastaví Bit timing registr (BTR) dle předem vypočítaných hodnot. Po správném nastavení přenosové rychlosti funkce vrací hodnotu 1. Pokud je špatně zvolena přenosová rychlost, funkce vrací hodnotu 0.
- *uint16_t CAN_Initialization (uint16_t Bitrate, uint32_t IfaceTransferTout)* – Funkce nastavuje a inicializuje CAN1 rozhraní a veškeré použité hardwarové periferie. Hodiny RCC musí být spuštěny pro jakoukoli periferii, která má být použita. Většina nastavení jsou zde provedena pomocí funkcí z knihovny STM32F4xx_StdPeriph_Driver. Nejprve se ve funkci konfiguruje port GPIOD pro diagnostické LED diody, dále port GPIOA pro vstup tlačítka, výstupy portů GPIOE a GPIOB pro LCD displej. Nastavení portu se dělá tak, že se povolí hodiny pro daný port pomocí funkce *RCC_AHB1PeriphClockCmd(),* poté

se port resetuje funkcí *GPIO_DeInit()*, pak se pomocí struktury zvolí konkrétní pin, nastaví se, zda jde o vstup nebo výstup a nakonec se funkcí *GPIO_Init()* port inicializuje. Následuje nastavení časovače TIM2 na 1 ms a současně se pro něj nastavuje i povoluje vektor přerušení NVIC. Pro časovač se povolí hodiny a následně se resetuje, obdobně jako to bylo u nastavení portů. Dále se nastaví předdělič, časová perioda a inkrementace. Nastavení způsobí to, že časovač přeteče každou milisekundu. To znamená, že pokud je povolen interrupt pro tento časovač, každou milisekundu se vyvolá přerušení a jeho obslužný program. Na port GPIOD, piny 0 a 1 se připojuje budič CAN a proto se pro tyto piny nastavuje alternativní funkce rozhraní CAN1. Aby se mohla nastavit přenosová rychlost sběrnice, musí se CAN1 nejprve přepnout do inicializačního módu. Toho se dosáhne za pomoci funkce *CAN_OperatingModeRequest(CAN1, CAN_OperatingMode_Initialization)*. Dále se nastaví přenosová rychlost pomocí funkce *CAN_BaudRate()* ze zdrojového modulu *can.c* a CAN1 se přepne zpět do normálního operačního módu. Následuje povolení přerušení pro příjem zpráv, Error Interrupt, Bus-off Interrupt, Error passive Interrupt a Last error code Interrupt. Nakonec se nastaví vektor přerušení NVIC pro příjem CAN zpráv a chyby změny stavu SCE (Status Change Error). Na konci funkce se ověřuje, zda doba všech nastavení v této funkci nepřesáhla čas 50 ms. Pokud čas nevypršel, návratová hodnota je rovná 1, v opačném případě je návratová hodnota rovná 0.

- *uint16_t CAN_SetFilter (uint32_t FilterBank, uint16_t CANID, uint16_t MaskOrList, uint16_t Fifo)* – Funkce nastavuje filtr pro příjem zpráv a do jedné filtrační banky dokáže uložit až čtyři standardní identifikátory. Filtr umí zadávat jen standardní identifikátory. Využívá k tomu funkci *CAN_FilterInit* použitou z knihovny STM32F4xx_StdPeriph_Driver. Identifikátor filtru ukládá do paměti FIFO0. Jako mód pro filtr je pevně zvolen List mode, čili identifikátor zachycené zprávy se porovnává se seznamem nastavených identifikátorů. Pokud se identifikátor shoduje s jedním z nastavených, zpráva se přijme. Funkce nastavuje filtr automaticky pro standardní identifikátory bez možnosti volby.

4.2 Zdrojový modul mcohw.c

Modul mcohw.c implementuje hardwarový ovladač pro mikrokontrolér STM32F407. Obsahuje funkce a proměnné, které souvisí s hardwarovou vrstvou. Mnohé funkce využívají služeb ze zdrojového modulu *can.c*.

Zdrojový modul *mcohw.c* inicializuje tři globální proměnné. Proměnná *FilterCounter*

čítá počet nastavených filtrů, nesmí však překročit hodnotu 28 filtrů. Po inicializaci má hodnotu 0. Proměnná hodinového tiku *TimCnt1ms* se inkrementuje přesně každou 1 ms a to díky timeru TIM2 v obsluze jeho přerušení. *TimCnt1ms* je přístupný pouze pro funkce *MCOHW_GetTime()* a *MCOHW_IsTimeExpired()*. Proměnná *Status* se tváří jako osmibitový registr, ale ve skutečnosti je to unsigned char. Proměnnou *Status* si vyvolává funkce *MCOHW_GetStatus()*.

Zdrojový modul mcohw.c definuje následující funkce:

- *uint8_t MCOHW_GetStatus (void)* – Návratová hodnota funkce je proměnná *Status,* která má po inicializaci hodnotu nula. Tato proměnná může být kdykoliv změněna. U většiny případů se tak děje při příslušném přerušení. Hlavičkový soubor *mcohw.h* definuje význam jednotlivých stavových bitů proměnné *Status*.

  INIT 0x01 – 0 bit - Nastaví log. 1 po kompletní inicializaci hardwarového driveru, nechá log. 0 dokud nebyla inicializace dokončena nebo při ní vznikla chyba.

  CERR 0x02 – 1 bit - Nastaví log. 1 když nastane chyba na CAN sběrnici.

  ERPA 0x04 – 2 bit - Nastaví log. 1 když počet chyb dosáhne limitu "error passive".

  RXOR 0x08 – 3 bit - Nastaví log. 1 když přeteče paměť FIFO pro příjem zpráv.

  TXOR 0x10 – 4 bit - Nastaví log. 1 když přeteče paměť FIFO pro vysílání zpráv.

  BOFF 0x80 – 7 bit - Nastaví log. 1 když se fyzicky přeruší sběrnice.

- *uint16_t MCOHW_Init (uint16_t BaudRate)* – Tato funkce má inicializovat rozhraní a připouštět přenosové rychlosti sběrnice 1000, 800, 500, 250, 125, 50, 25 nebo 10 kbit/s. Používá k tomu funkci *CAN_Initialization* definovanou v *can.c* zdrojovém modulu. Přípustná doba inicializace hardwaru je stanovena na maximálně 50 ms. Návratová hodnota je rovná jedné, když inicializace proběhne v pořádku a doba čekání na provedení inicializace nepřesáhne zmíněných 50 ms. V opačném případě by byla návratová hodnota rovná 0.
- *uint16_t MCOHW_SetCANFilter (uint16_t CANID)* – Funkce se stará o nastavení CAN filtru. Využívá k tomu funkci *CAN_SetFilter* definovanou v modulu *can.c*. Defaultně nemá mikrokontrolér po resetu nastavený žádný filtr. Při každém vyvolání funkce nastaví jeden identifikátor a inkrementuje proměnnou *FilterCounter*. Po úspěšném nastavení filtru vrací hodnotu 1. Dokáže nastavit až 28 filtračních bank. Po pokusu o nastavení více jak 28 filtračních bank vrací hodnotu 0.
- *uint16_t MCOHW_PushMessage (CAN_MSG *pTransmitBuf)* – Hlavním úkolem funkce je vysílání zpráv. Tato funkce implementuje CAN vysílací frontu. S každým vyvoláním funkce je CAN zpráva přidána do vysílací fronty. Vysílací fronta dokáže pojmout až 3

zprávy. Při dalším pokusu o přidání zprávy do plné vysílací fronty se změní hodnota proměnné *Status* podle TXOR stavového bitu. Po úspěšném odvysílání je návratová hodnota rovná 1, po neúspěchu 0. Využívá funkci *CAN_Transmit* z knihovny STM32F4xx_StdPeriph_Driver. Vysílá jen zprávy se standardním identifikátorem. MCO stack se nesnaží hned opakovaně přidávat další zprávy do fronty, protože s každým vyvoláním funkce *MCO_ProcessStack* je maximálně jedna zpráva přidána do fronty.

- *uint16_t MCOHW_PullMessage (CAN_MSG *pReceiveBuf)* – Funkce se stará o příjem zpráv. Obdržené zprávy, které prošly přijímacím filtrem CAN, se předají do přijímací paměti fronty. S každým vyvoláním funkce se z paměti fronty vyčte zpráva. K přijetí zprávy je využita funkce *CAN_Receive()* z knihovny STM32F4xx_StdPeriph_Driver. Po vyvolání této funkce se nejprve zkontroluje pomocí CAN_RF0R registru a jeho FMP0[1:0] bitů, jestli existuje v přijímací frontě nějaká zpráva, která čeká na obsloužení. Pokud existuje, její identifikátor, délka dat a samotná data se uloží do struktury pointru **pReceiveBuf* pro následné zpracování. Návratová hodnota je v tomto případě rovna 1. Návratová hodnota 0 nastane, když registr CAN_RF0R a jeho bity FMP0[1:0] jsou nulové. Znamená to tedy, že žádná zpráva se nenachází v přijímací paměti fronty.
- *uint32_t MCOHW_GetTime (void)* – Funkce vrací aktuální hodnotu 32-bitové proměnné *TimCnt1ms*, která je inkrementována každou milisekundu při vyvolání přerušení časovače TIM2. Tato funkce plní úlohu systémového časovače.
- *uint8_t MCOHW_IsTimeExpired (uint32_t timestamp)* – Funkce porovnává čas daný parametrem funkce *timestamp* (v milisekundách) s hodnotou proměnné *TimCnt1ms* vnitřního časovače a říká, zda čas daný parametrem už vypršel (návratová hodnota je 0), či nikoliv (návratová hodnota je 1). Maximální porovnávaná hodnota je 0x8000, což je kolem 32 sekund.

Hlavičkový soubor *mcohw.h* definuje datovou strukturu pro samostatné CAN zprávy.

```
typedef struct
{
  uint32_t ID;              // Identifikátor zprávy
  uint8_t LEN;              // Délka datové zprávy (může nabývat hodnoty 0-8)
  uint8_t BUF[8];           // Data
} CAN_MSG;
```

4.3 Zdrojový modul stm32f4xx_it.c

Tento zdrojový modul je speciálně vytvořený soubor, který obsahuje jen funkce, na které dojde při přerušení neboli interruptu. Obecně by funkce vyvolané přerušením neměly být nějak obzvlášť dlouho trvající, aby při častém přerušení neohrozily správný chod programu.

Modul obsahuje tři funkce. Funkce *void CAN1_RX0_IRQHandler (void)* se vyvolá při přerušení, když přijde CAN zpráva. Tato funkce řeší rozsvěcování LED diod pro diagnostické účely. Zelená LED se rozsvítí při příjmu zprávy. Oranžová LED se rozsvítí, když je paměť fronty pro příjem zpráv zcela zaplněna. Červená LED se rozsvítí, když paměť fronty přeteče a dojde tak ke ztrátě dat. Dalším úkolem funkce je zaznamenat tento chybový stav zapsáním log. 1 do bitu RXOR v globální proměnné *Status*. Pro zjištění stavu paměti fronty je využíván CAN_RF0R registr.

Druhá přerušovací funkce *void CAN1_SCE_IRQHandler (void)* se vyvolá, když přijde hlášení o chybě pomocí chybových flagů v CAN_ESR registru. Nastane přerušení a podle toho se nastaví příslušný stavový bit v globální proměnné *Status* na log. 1.
Přerušení může nastat:

- Když se fyzicky přeruší sběrnice CAN (stavový bit BOFF).
- Když počet chyb dosáhne limitu Error pasiv (stavový bit ERPA).
- Když se vyskytne nějaká chyba na sběrnici CAN (stavový bit CERR).

Význam všech stavových bitů proměnné *Status* je popsán v kapitole 4.2 u funkce *MCOHW_GetStatus()*.
Třetí funkce *void TIM2_IRQHandler (void)* řeší obsluhu přerušení časovače TIM2. Přerušení se vyvolá každou 1 ms. Při této události se inkrementuje proměnná hodinového tiku *TimCnt1ms* a vynuluje se příslušný flag. Časové přerušení musí být velmi přesné (přesně 1ms), protože plní funkci systémového časovače a zároveň se využívá ve funkcích, které pracují s časem.

4.4 Zdrojový modul LCD16x2lbr.c

Modul *LCD16x2lbr.c* je vytvořená knihovna pro dvouřádkový 16-ti znakový LCD displej. Definice příkazu *PULS_ENABLE* pro pin 9, který je propojen s LCD displejem, představuje příkaz pro vznik jednoho pulsu, který umožní zapsat připravená data na displej. Další definice příkazu *SET_DATA* zapíše na celý port GPIOE (který je propojen s datovými piny displeje) datový příkaz pro displej.

Funkce *void delay_ms (unsigned int cas)* předstírá čekací funkci. Funkce *void write_data (unsigned char b)* zapisuje na displej data, naopak funkce *void write_cmd (unsigned char b)* ovládá displej příkazy. Funkce *void write_lcd (unsigned char b)* jen vyvolá funkci *write_data()*. Pro mazání displeje je tu funkce *void clear_lcd (void),* která odešle speciální příkaz pro mazání. Na displej se můžou psát textové řetězce pomocí funkce *void write_string (char *msg)*. Funkce *void GoToXY (char sloupec, char radek)* dokáže umístit kurzor na kterékoli pole displeje a odtud pak začít vypisovat znaky. Pro inicializaci a nastavení displeje poslouží funkce *void init_lcd (void)*, která je vyvolána v hlavní funkci *main()*. Funkce *void hexa_cisla (char cislo)* umí decimální číslo převést na hexadecimální a pak ho vypsat na displej. Funkce *void zobrazLCD_RxMessage (unsigned long long hex)* byla speciálně vytvořena pro ladící účely příjmu zpráv, protože dokáže na displeji zobrazit celé datové pole CAN zprávy v hexadecimálním tvaru. Momentálně ale není v práci využita. Funkce *void zobrazLCD_ErrCode (unsigned int code)* je vyvolána funkcí *MCOUSER_FatalError()* ze zdrojového modulu *mco.c* a zobrazuje na prvním řádku displeje kód fatální chyby v hexadecimálním tvaru. Na druhém řádku displeje se vypisuje aktuální stav uzlu. Výpis provádí funkce *void zobrazLCD_Status (unsigned char status)*, která je vyvolána v hlavní funkci *main()*. Funkce se stará o výpis proměnné *Status* v binární hodnotě. Více o této proměnné je zmíněno v popisu funkce *MCOHW_GetStatus()* v kapitole 4.2.

4.5 Zdrojový modul mco.c

Tento zdrojový modul inicializuje funkce pro aplikační vrstvu. Jak již bylo popsáno v kapitole 2.1.2, aplikační vrstva se rozděluje na přímé a „call-back“ funkce.

Aplikační vrstva s přímými funkcemi definuje následující povinné funkce:

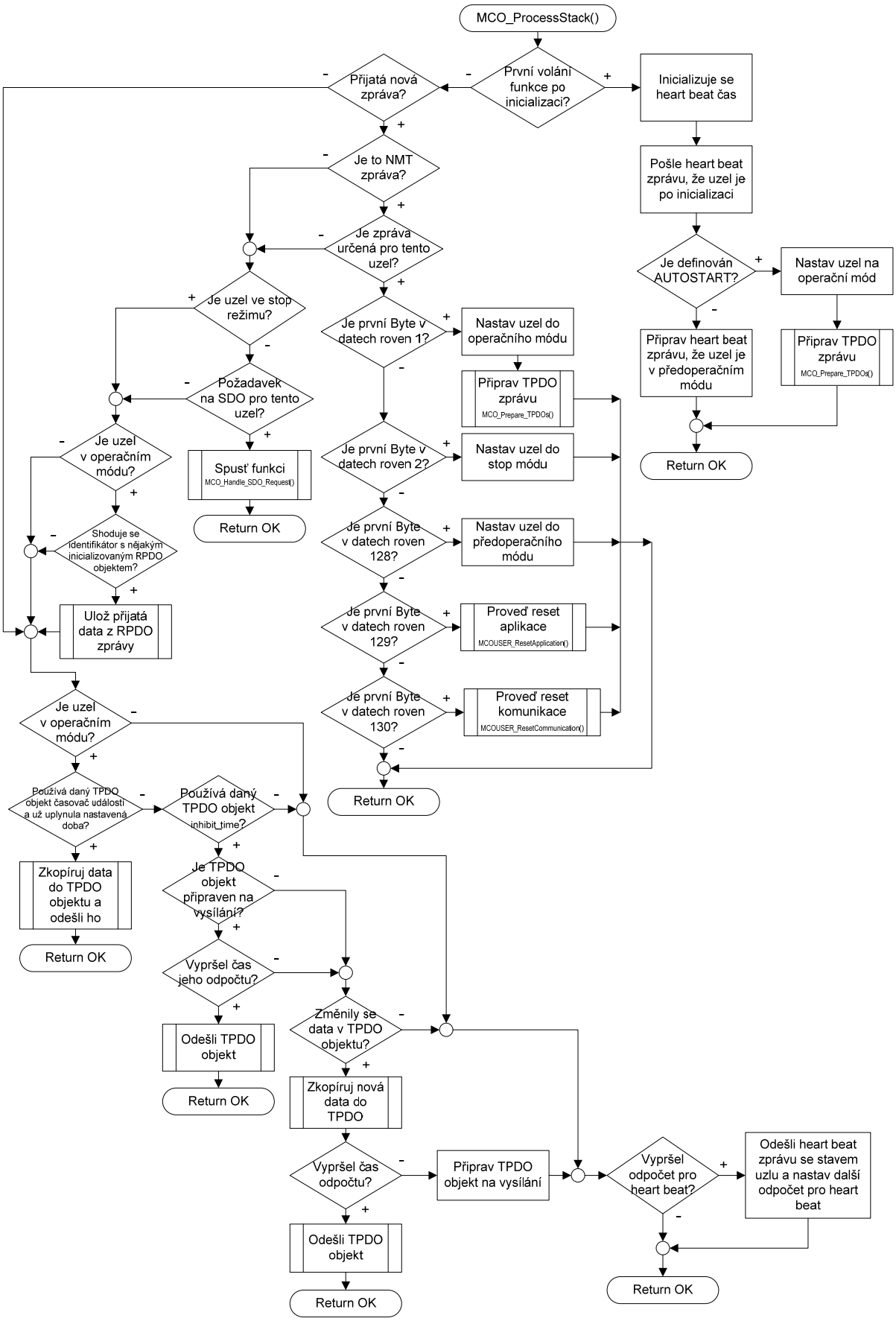
- *void MCO_Init (uint16_t bitrate, uint8_t Node_ID, uint32_t heartbeat)* – Funkce inicializuje protokolový stack MCO. Je vyvolána z uživatelské „call-back“ funkce *MCOUSER_ResetApplication* (viz níže). Pomocí funkce *MCOHW_Init* se nastavuje přenosová rychlost sběrnice (parametr *bitrate*), identifikátor uzlu (parametr *Node_ID*) a interval pro opakované vysílání heartbeat (parametr *heartbeat*). Identifikátor uzlu může být v rozsahu 0x700 + 1 až 126. Heartbeat interval se udává v milisekundách. Ve funkci se nejprve předají parametry *Node_ID* a *heartbeat* do proměnných a inicializují se TPDO a RPDO objekty. V této práci jsou použity jen dva vysílací a dva přijímací procesní objekty. Dále se pomocí funkce *MCOHW_Init(bitrate)* nastaví přenosová rychlost. V případě neúspěchu nastavení se na displeji zobrazí chybový kód 0x8802. Dále se nastaví filtr na příjem identifikátoru 0, což jsou hlavní NMT zprávy. Pro SDO zprávy se nastaví filtr

s identifikátorem *0x600+Node_ID*, kde *Node_ID* je v tomto případě identifikátor uzlu 0x007. V případě neúspěchu nastavení kteréhokoliv filtru se na displeji zobrazí chybový kód 0x8803. Pokud se během inicializace nevyskytla žádná chyba, na konci této funkce se do proměnné *gTPDONr* uloží hodnota 0xFF a do bitu INIT v proměnné *Status* se nastaví 1. Právě hodnota 0xFF je znamení pro funkci *MCO_ProcessStack()*, že protokolový stack MCO byl právě inicializován. Pokud nějaká chyba nastala, do bitu INIT se nastaví 0.

- *void MCO_InitTPDO (uint8_t PDO_NR, uint16_t CAN_ID, uint16_t event_tim, uint16_t inhibit_tim, uint8_t len, uint8_t *dat)* – Funkce inicializuje vysílací PDO objekty. Jeden datový objekt CAN zprávy může obsahovat až 8 Bytů dat. Není tedy vyloučeno, že více proměnných nemohou být spojeny do jedné TPDO zprávy. Protokolový stack MCO po své inicializaci automaticky zpracovává přenos PDO. Aplikace mění data před spuštěním funkce *MCO_ProcessStack()* kvůli své konzistenci, protože během provádění funkce není vhodné data měnit. Parametr *PDO_NR* musí být v rozmezí 1 až 4. Parametr *CAN_ID* může být nastavený na hodnotu 0, pokud by měl být použit CANopen výchozí ID. V případě, že parametr *event_tim* nenabývá nulové hodnoty, zpráva bude vysílána každých *event_tim* milisekund. Pokud parametr *inhibit_tim* nenese hodnotu nula, zpráva je přenášena na základě změny stavu s minimálním časovým limitem *inhibit_tim* udávaným v milisekundách. Parametr *len* určuje délku dat a parametr *dat* udává přímo data.
- *void MCO_InitRPDO (uint8_t PDO_NR, uint16_t CAN_ID, uint8_t len, uint8_t *dat)* – Funkce inicializuje příjem PDO objektů. Parametr *PDO_NR* musí být v rozmezí 1 až 4 a značí pořadí RPDO objektů. Parametr *CAN_ID* může být nastavený na hodnotu 0, pokud by měl být použit výchozí identifikátor uzlu. V tomto případě se identifikátor pro filtr nastaví podle následujícího vzorce:

  *0x200 + (0x100 * ((uint16_t)(PDO_NR))) + ID_Node*

  Pokud parametr nebude nastaven na hodnotu 0, identifikátor filtru se nastaví podle parametru *CAN_ID*. Parametr *len* určuje délku dat a parametr **dat* udává přímo přijatá data. Po inicializaci protokolového stacku MCO se přijatá data zapíší automaticky do paměti, kam ukazuje pointer **dat*.
- *uint8_t MCO_ProcessStack (void)* – Funkce implementuje hlavní protokolový stack MicroCANopen. Tato funkce musí být často vyvolána za účelem zajištění správné činnosti komunikačního stacku. Funkce je vyvolána z opakující se smyčky while v hlavní funkci *main()*. Vysvětlení této funkce nejlépe popisuje vývojový diagram na obr. 12.



Obr. 12: Vývojový diagram funkce MCO_ProcessStack()

- *uint8_t MCO_Handle_SDO_Request (uint8_t *pData)* – Funkce se zabývá kompletním hledáním a přístupem k OD (Object Dictionary) objektům. Je vyvolána v protokolovém stacku MCO s požadavkem SDO Request na zpracování této služby. Identifikátor s požadavkem musí být roven *0x600 + ID_Node.* Pro splnění některých operací používá funkce *MCO_Search_OD(), MCO_Send_SDO_Abort()*, *MCOHW_PushMessage(),* které již byly nebo ještě budou popsány. Parametr funkce **pData* je ukazatel na data SDO Request zprávy.
- *void MCO_Send_SDO_Abort (uint32_t ErrorCode)* – Funkce je vyvolána ve funkci *MCO_Handle_SDO_Request.* Generuje odpověď na přenos služeb SDO a vyšle zprávu SDO Abort. V praxi to znamená, že odesílá zprávy s identifikátorem *0x580+ID_Node* a daty, kde první 4 Byty dat jsou předem inicializovány, zbylé 4 Byty náleží parametru *ErrorCode.* Byte 0 – specifický příkaz 0x80, Byte 1 a 2 - Index, Byte 3 – Subindex. Chybové kódy (SDO_ABORT_UNSUPPORTED, SDO_ABORT_NOT_EXISTS, SDO_ABORT_READONLY, SDO_ABORT_TYPEMISMATCH, SDO_ABORT_UNKNOWNSUB, SDO_ABORT_UNKNOWN_COMMAND) pro tuto funkci jsou definované v hlavičkovém souboru *mco.h.*
- *uint8_t MCO_Search_OD (uint16_t index, uint8_t subindex)* – Funkce je vyvolána ve funkci *MCO_Handle_SDO_Request.* Hledá v SDO Response tabulce (*SDOResponseTable[]* definovaná v *mco.c* modulu) podle specifikovaného indexu a subindexu. Tato funkce slouží ke zjištění, zda je požadovaná OD položka implementována v tabulce. Návratová hodnota funkce je číslo pořadí záznamu v tabulce, s tím, že se začíná počítat od nuly. V případě neúspěchu vyhledávání je návratová hodnota rovna 255.
- *void MCO_Prepare_TPDOs (void)* – Funkce je vyvolána v rámci protokolového stacku MCO. MicroCANopen implementuje CANopen NMT stavový automat. PDO zprávy jsou zpracovány pouze v operačním módu, v jakémkoliv jiném módu jsou neaktivní. Takže s každým přechodem do operačního módu musí být odeslány všechny TPDO zprávy. Tato funkce připraví všechny TPDO zprávy na odeslání. Nejprve aktualizuje data pro daný TPDO objekt. V konfiguračních strukturách TPDO nastavuje časovač událostí *event_timestamp* a *inhibit_timestamp* na aktuální hodnotu systémového časovače. Do proměnné *inhibit_status* v konfigurační struktuře TPDO uloží signalizaci, že daná TPDO zpráva je připravena na vysílání.
- *void MCO_TransmitPDO (uint8_t PDONr)* – Funkce je vyvolána z protokolového stacku MCO, když potřebuje být odeslána TPDO zpráva. Parametr *PDONr* ukazuje na příslušný vysílací objekt TPDO. Do proměnné *inhibit_status* v konfigurační struktuře TPDO uloží

signalizaci, že byl spuštěn Inhibit timer. Do proměnné *inhibit_timestamp* v konfiguračních strukturách TPDO ukládá aktuální hodnotu systémového časovače + hodnotu *inhibit_time* z konfigurační struktury TPDO a do proměnné *event_timestamp* ukládá aktuální hodnotu systémového časovače + hodnotu *event_time*. Pro odeslání TPDO zprávy je použita funkce *MCOHW_PushMessage()*. V případě neúspěchu odeslání zprávy se na displeji zobrazí chybový kód 0x8801.

Aplikační vrstva s „call-back" funkcemi definuje následující povinné funkce:

- *void MCOUSER_ResetApplication (void)* – Funkce resetuje aplikaci. Je vyvolána v rámci protokolového stacku MCO. Pokud se přijme hlavní NMT zpráva s požadavkem na reset aplikace, dojde k vyvolání této funkce. Reset aplikace je řešen pomocí watchdog přerušení. Ve funkci se inicializuje, nastaví a povolí watchdog pomocí funkcí z knihovny STM32F4xx_StdPeriph_Driver. Hned za tímto procesem následuje prázdný cyklus *while(1).* Automaticky se spustí watchdog přerušení a mikrokontrolér se sám automaticky resetuje.
- *void MCOUSER_ResetCommunication (void)* – Funkce resetuje a zároveň inicializuje CAN rozhraní a procesní objekty. Funkce je vyvolána na začátku hlavní funkce *main()* a dále v rámci protokolového stacku MCO, když se přijme hlavní NMT zpráva s požadavkem na reset komunikace. Tato funkce vyvolá příslušné funkce: *MCO_Init, MCO_InitTPDO* a *MCO_InitRPDO.* Vyvoláním této funkce se nastaví bit RESET v CAN_MCR registru na log. 1. Následně na to se tento bit automaticky vymaže a CAN_MCR registr se nastaví do své hodnoty po resetu. To znamená, že se CAN rozhraní přepne do sleep módu. Z tohoto módu se „probudí" právě, když se CAN rozhraní přepne do inicializačního módu. Do inicializačního módu se přepne z toho důvodu, aby se mohla nastavit přenosová rychlost sběrnice (viz zdrojový modul *mcohw.c* a jeho funkce *CAN_Initialization()*).
- *void MCOUSER_FatalError (uint16_t ErrCode)* – Tato funkce je vyvolána, pokud nastane fatální chyba. Kódy chyb pro modul hardwarové vrstvy jsou udávány v rozsahu od 0x8000 do 0x87FF. Kódy chyb pro modul aplikační vrstvy jsou udávány v rozsahu od 0x8800 do 0x8FFF. Všechny ostatní kódy mohou být použity libovolně podle aplikace. Tato funkce slouží pro diagnostické účely a tak když nastane chyba, na prvním řádku LCD displeje se zobrazí její kód. Vypsání kódu na displej provádí funkce *zobrazLCD_ErrCode(ErrCode)* ze zdrojového modulu *LCD16x2lbr.c.* Z toho důvodu musí být na začátku zdrojového modulu *mco.c* inkludován hlavičkový soubor *LCD16x2lbr.h*.

4.6 Zdrojový modul main.c

Zdrojový kód modulu *main.c* obsahuje hlavní *int main (void)* funkci a *while* smyčku, ve které neustále běží hlavní program. Hned na začátku se inkludují nezbytné hlavičkové soubory *stm32f4xx.h, LCD16x2lbr.h, mco.h* a inicializují se datové proměnné *OUT_DATA_2, IN_DATA_1, IN_DATA_2,* přes které se předávají hodnoty dat pro TPDO a RPDO zprávy.

V hlavní funkci *int main (void)* se vyvolá funkce *MCOUSER_ResetCommunication()*, která způsobí reset komunikace, takže proběhne inicializace CAN rozhraní. Dále proběhne inicializace LCD displeje vyvoláním funkce *void init_lcd()*.

Ve *while* smyčce se operace provádí neustále dokola. Inkrementuje se datová proměnná *OUT_DATA_2* a předá data pro TPDO2 zprávu. Předají se datové hodnoty z RPDO1 a RPDO2 objektů do proměnných *IN_DATA_1* a *IN_DATA_2*. Testuje se zmáčknutí tlačítka a tím se nastavují data pro TPDO1 zprávu. Když není zmáčknuté tlačítko, data jsou rovná hodnotě 0. Když se stiskne tlačítko, tak se nastaví data na hodnotu 1. Kontroluje se datová proměnná *IN_DATA_1*. Když je její hodnota rovna 1, rozsvítí se modrá LED dioda. V opačném případě je dioda zhasnutá. Přijetí zprávy je signalizováno rozsvícením nebo zhasnutím zelené LED diody. Pomocí funkce *zobrazLCD_Status(MCOHW_GetStatus())* je na displej vypisován aktuální stav uzlu. Na konci *while* smyčky je vyvolána funkce *MCO_ProcessStack()* jejíž činnost již byla popsána. Pak se program vrací znovu na začátek smyčky *while*.

Závěr

Prostudoval jsem architekturu mikrokontroléru STM32F40x a seznámil jsem se s vývojovými nástroji. Popsal jsem specifikaci komunikačního profilu CAN. Program byl realizován v programovacím jazyce C.

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. Bylo naprogramováno komunikační rozhraní a konfigurační a stavové struktury komunikačního driveru MicroCANopen. Dále byla naprogramována vlastní vrstva implementující MicroCANopen protokolové služby a funkce hardwarově závislého ovladače CAN řadiče mikrokontroléru. Protokolový stack MicroCANopen je řešen formou periodicky volaného obecného stavového automatu s minimálními časovými prodlevami.

Na konečnou fázi implementace protokolového stacku MicroCANopen do systémové exekutivy ZAT nedošlo.

Seznam obrázků

Seznam tabulek

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] *Embedded Systems Academy: Products, Consulting & Training for Embedded Systems* [online]. © 1999 - 2015. [cit. 2015-05-04]. Dostupné z: http://www.esacademy.com/

Přílohy

Příloha 1 – Zdrojový kód souboru main.c

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

//extern CAN_MSG gRxCAN;
uint8_t OUT_DATA_2;                                     // Date variable for TPDO2
uint8_t IN_DATA_1;                                      // Date variable for RPDO1
uint8_t IN_DATA_2;                                      // Date variable for RPDO2
extern uint8_t gProcImg[];                  // external declaration for the process image array

int main(void)
{

    /* Resets and initializes the CAN interface -----------------------------------------*/
    MCOUSER_ResetCommunication();

    /* Initializes the user application -------------------------------------------------*/
    init_lcd();

    while(1)
    {
        /* Image datas ----------------------------------------------------------------------*/
        OUT_DATA_2++;                                   // changing data
        gProcImg[IN_digi_2] = OUT_DATA_2;

        IN_DATA_1 = gProcImg[OUT_digi_1];
        IN_DATA_2 = gProcImg[OUT_digi_2];

        /* User button ----------------------------------------------------------------------*/
        if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0))
        {
            gProcImg[IN_digi_1] = 0x01;                 // send data that button is on
        }
        else
        {
            gProcImg[IN_digi_1] = 0x00;                 // send data that button is off
        }

        /* Set blue LED ---------------------------------------------------------------------*/
        if(IN_DATA_1 == 0x01)
        {
            GPIO_WriteBit(GPIOD, GPIO_Pin_15, Bit_SET);             // GPIO_Pin_15 modraLED
        }
        else
        {
            GPIO_WriteBit(GPIOD, GPIO_Pin_15, Bit_RESET);
        }

        /* 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);

        zobrazLCD_Status(MCOHW_GetStatus());

        MCO_ProcessStack ();
    }
}
```

Příloha 2 – Zdrojový kód souboru mcohw.c

```
/* Includes ------------------------------------------------------------------*/
#include "mcohw.h"
#include "can.h"
/* Private variables ---------------------------------------------------------*/
uint8_t FilterCounter = 0;
uint16_t volatile TimCnt1ms = 0;     // Global timer/counter variable, incremented every millisecond
uint8_t volatile Status = 0x00;      // Global status variable, default set to 0x00
/* Private functions ---------------------------------------------------------*/

/**
  * @brief  This function returns the global status variable.
  * @param  None
  * @retval None
  */
uint8_t MCOHW_GetStatus(void)
{
    return Status;
}

/**
  * @brief  This function implements the initialization of the CAN interface.
  * @param  None
  * @retval None
  */
uint8_t MCOHW_Init(uint16_t BaudRate)
{
    uint16_t IfaceTout = 50;
    // Waiting max 50ms
    uint8_t stav;
    stav = CAN_Initialization(BaudRate, IfaceTout);
    if(stav == 1)
        return 1;               // Correct
    else
        return 0;               // Mistake
}

/**
  * @brief  This function implements the initialization of a CAN ID hardware filter.
  * @param  None
  * @retval None
  */
uint8_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
  */
uint8_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;           // Correct
    }
    else
    {
        Status = Status | TXOR;             // Queue overrun, TXOR bit set to 1
        return 0;           // Mistake
    }
}

/**
  * @brief  This function implements a CAN receive queue.
  * @param  None
  * @retval 0 if no message received, 1 if message received and copied to the buffer
  */
uint8_t MCOHW_PullMessage(CAN_MSG *pReceiveBuf)
```

```
{
    uint8_t i;
    CanRxMsg RxMessage;
    if((CAN1->RF0R & 0x03) > 0)    // chack message received
    {
        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;                // message received
    }
    else
        return 0;                        // no message received
}

/**
  * @brief  This function gives the Timer Counter value.
  * @param  None
  * @retval None
  */
uint16_t MCOHW_GetTime(void)
{
   return TimCnt1ms;
}

/**
  * @brief  This function checks if a TimeStamp expired.
  * @param  None
  * @retval None
  */
uint8_t MCOHW_IsTimeExpired(uint16_t timestamp)
{
    uint16_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 ---------------------------------------------------------*/
GPIO_InitTypeDef GPIO_InitStructure;
NVIC_InitTypeDef            NVIC_InitStructure;
CAN_FilterInitTypeDef CAN_FilterInitStructure;
TIM_TimeBaseInitTypeDef TIM_InitStructure;
/* Private functions ---------------------------------------------------------*/

/**
  * @brief  This function sets a baud rate.
     PCLK1 must be set on the freguency 42MHz
  * @param  None
  * @retval None
  */
uint16_t CAN_BaudRate(uint16_t BaudRate)
{
    uint8_t 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 configure a CAN1 hardware, sets an interupts and sets a NVIC.
  * @param  None
  * @retval None
  */
uint8_t CAN_Initialization(uint16_t Bitrate, uint16_t IfaceTransferTout)
{
    uint16_t cTime, timeNow;
    uint8_t stav;

    cTime = MCOHW_GetTime();                        // Get current time

    /* Hardware configure and init ----------------------------------------------------------*/
    /* 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
GPIOD Init Structure
    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
```

```
    /* 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

    /* vystupy na display 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
Init Structure
    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

    /* timer ----------------------------------------------------------------------------*/
    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);

    /* CAN1 configure and init ----------------------------------------------------------*/
```

```
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);          // CAN pins configuration: 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); // CAN pins configuration: Connect the involved CAN pins to AF9
    GPIO_PinAFConfig(GPIOD, GPIO_PinSource1, GPIO_AF_CAN1); // CAN pins configuration: Connect the involved CAN pins to AF9

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);    // Enable the CAN controller interface clock
    CAN_DeInit(CAN1);               // Deinitializes the CAN peripheral registers to their default reset values
    CAN_OperatingModeRequest(CAN1, CAN_OperatingMode_Initialization);    // Specifies the CAN operating mode
    stav = CAN_BaudRate(Bitrate);           // Initializes the CAN baud rate according to the defined parameter
    if(stav != 1)
        return 0;                           // Mistake
    CAN_OperatingModeRequest(CAN1, CAN_OperatingMode_Normal);   // Specifies the CAN operating mode

    /* CAN1 interrupts enable ------------------------------------------------------------*/
    CAN_ITConfig(CAN1, CAN_IT_FMP0, ENABLE);          // Enable FIFO 0 message pending Interrupt
    CAN_ITConfig(CAN1, CAN_IT_ERR, ENABLE);           // Error Interrupt enable
    CAN_ITConfig(CAN1, CAN_IT_BOF, ENABLE);           // Bus-off Interrupt enable
    CAN_ITConfig(CAN1, CAN_IT_EPV, ENABLE);           // Error passive Interrupt enable
    CAN_ITConfig(CAN1, CAN_IT_LEC, ENABLE);           // Last error code Interrupt enable

    /* 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 = 1;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    /* NVIC Configuration for CAN1_SCE ---------------------------------------------------*/
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
    NVIC_InitStructure.NVIC_IRQChannel = CAN1_SCE_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;                                   // Correct
        else
            return 0;                                   // Mistake
    }
    else
    {
        if (((0xFFFF - cTime) + timeNow) <= IfaceTransferTout)
            return 1;                                   // Correct
        else
            return 0;                                   // Mistake
    }
}

/**
  * @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, according to the mode (LSBs for a 32-bit configuration, second one for a 16-bit configuration)
    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, according to the mode (MSBs for a 32-bit configuration, first one for a 16-bit configuration)

    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 stm32f4xx_it.c

```
/* Includes ------------------------------------------------------------------*/
#include "stm32f4xx_it.h"
#include "LCD16x2lbr.h"
#include "mcohw.h"
#include "can.h"
/* Private variables ---------------------------------------------------------*/
extern uint16_t volatile TimCnt1ms;
extern uint8_t volatile Status;                                 // Global status variable
/* Private structures --------------------------------------------------------*/
extern CAN_MSG gRxCAN;
/* Private functions ---------------------------------------------------------*/

/******************************************************************************/
/*               STM32F4xx Peripherals Interrupt Handlers                     */
/*  Add here the Interrupt Handler for the used peripheral(s) (PPP), for the  */
/*  available peripheral interrupt handler's name please refer to the startup */
/*  file (startup_stm32f4xx.s).                                               */
/******************************************************************************/

/**
  * @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 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);

    /* signalizace preteceni FIFA ---------------------------------------------------------*/
    if((CAN1->RF0R & 0x10) == 0x10)
    {
        GPIO_WriteBit(GPIOD, GPIO_Pin_14, Bit_SET);         // GPIO_Pin_14 cervenaLED LD5
        Status = Status | RXOR;                             // A receive queue overrun occured, RXOR bit set to 1
    }
}

/**
  * @brief  CAN1_SCE_IRQHandler
  *         This function handles CAN1_SCE interrupt request.
  * @param  None
  * @retval None
  */
void CAN1_SCE_IRQHandler(void)
{
    /* signalizace preruseni sbernice -----------------------------------------------------*/
    if((CAN1->ESR & 0x04) == 0x04)
    {
        Status = Status | BOFF;                 // A CAN "bus off" error occured, BOFF bit set to 1
    }

    /* signalizace dosazeni limitu Error pasiv --------------------------------------------*/
    if((CAN1->ESR & 0x02) == 0x02)
    {
        Status = Status | ERPA;                 // A CAN "error passive" occured, ERPA bit set to 1
    }

    /* indikace stavu chyby na CAN sbernici -----------------------------------------------*/
    if((((CAN1->ESR & 0x70) >> 4) >= 1) || (((CAN1->ESR & 0x70) >> 4) <= 6))
    {
        Status = Status | CERR;                 // A CAN bit or frame error occured, CERR bit set to 1
    }

    CAN_ClearITPendingBit(CAN1, CAN_IT_ERR);   // Clears the CAN1's error interrupt pending bit and clears
LEC bits
}
```

```
/**
  * @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++;
}

/**
* @}
*/
```