ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Schwank** Jméno: **Filip** Osobní číslo: **420183**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra řídicí techniky**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Programové vybavení pro CAN Gateway

Název diplomové práce anglicky:

CAN Gateway Software

Pokyny pro vypracování:

Navrhněte a implementujte programové vybavení modulu CAN Gateway. Funkcí modulu CAN GW je přenášet/filtrovat/modifikovat zprávy standardu CAN FD mezi dvojicí rozhraní v reálném čase. Jednotlivé funkce mohou být aktivní neomezeně, po daný časový interval nebo pro daný počet zpráv. Pro parametrizaci modulu použijte také rozhraní CAN FD.

a) Navrhněte modulární strukturu programového vybavení tak, aby funkční jádro bylo přenositelné na jiné HW platformy.
b) Implementujte podporu pro nezávislou manipulaci s až 8 zprávami CAN FD současně.
c) Implementujte podporu pro aktualizaci kontrolních součtů na aplikační vrstvě při manipulaci s daty, a to správně i záměrně chybně.
d) Implementujte funkci průběžného měření počtu přenášených zpráv.
e) Programové vybavení důkladně otestujte.

Seznam doporučené literatury:

[1] Kocourek P., Novák J.: Přenos informace, ČVUT 2003
[2] Etschberger K.: Controller Area Network, IXXAT Automation 2001, ISBN: 978-3000073762
[3] CAN with Flexible Data-Rate, Bosch 2012

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Jiří Novák, Ph.D., K 13138 - katedra měření

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **27.03.2018** Termín odevzdání diplomové práce: **25.05.2018**

Platnost zadání diplomové práce:
do konce letního semestru 2018/2019

doc. Ing. Jiří Novák, Ph.D.
podpis vedoucí(ho) práce

prof. Ing. Michael Šebek, DrSc.
podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

15.5.2018
Datum převzetí zadání

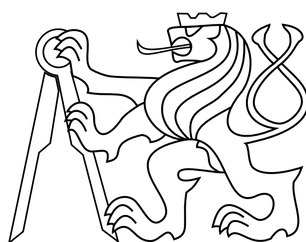
Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

diplomová práce

Programové vybavení pro CAN Gateway

Filip Schwank



Květen 2018

doc. Ing. Jiří Novák, Ph.D.

České vysoké učení technické v Praze
Fakulta elektrotechnická, Katedra měření

Poděkování

Rád bych poděkoval doc. Novákovi za jeho rady a vedení v průběhu tvorby této bakalářské práce. Vždy mi ochotně odpověděl na všechny dotazy.

Také bych chtěl poděkovat své rodině a přátelům za jejich podporu a motivaci.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Abstrakt

Tato diplomová práce se zabývá návrhem a realizací programového vybavení pro *CAN FD Gateway.* Jednotka je určena zejména pro použití jako testovací zařízení v automobilovém průmyslu, kdy jejím úkolem je blokovat, přeposílat, nebo modifikovat zprávy mezi dvěma *CAN FD* sítěmi. Pravidla pro práci se zprávami určuje třetí – řídicí *CAN FD* síť. Tato práce se zabývá vývojem aplikace nejdříve v simulaci, následně i implementací verze pro skutečný hardware s mikrokontrolérem firmy *Cypress.* Pro simulaci i reálnou aplikaci je navržena série testů pro ověření funkčnosti celé aplikace.

Klíčová slova

CAN; FDCAN; Gateway; mikrokontrolér, automotive

Abstrakt

This diploma thesis discusses realization of *CAN FD Gateway Software*. The unit is intended for testing purposes in automotive. Its task is to block, pass or modify messages bewtween two *CAN FD* networks. Rules applied on messages are determined by third – control *CAN FD* network. This thesis discusses realisation of the application first as a simulation, later with implementation on real hardware with *Cypress* microcontroller. For simulation and real application a series of tests is designed in order to verify functionality of the application.

Keywords

CAN; FDCAN; microcontroller, automotive

Obsah

1 Úvod **1**

2 Rozbor problematiky **2**
2.1 Motivace . . . 2
2.2 Analýza funkcionalit aplikace . . . 2
2.3 Komunikační protokol CAN . . . 4
2.3.1 Princip protokolu . . . 4
2.3.2 Verze CAN FD . . . 7
2.4 Aplikační CRC . . . 8

3 Popis komunikace **9**
3.1 Funkce modulu CAN Gateway . . . 9
3.2 Příkazy . . . 9
3.2.1 Formát a definice . . . 9
3.2.2 Reset . . . 10
3.2.3 Block/Pass . . . 10
3.2.4 Modify . . . 11
3.2.5 Trig . . . 13
3.2.6 Stat . . . 13
3.2.7 ACK/NACK . . . 13
3.3 Identifikátor Gateway . . . 14
3.4 Trig Broadcast . . . 14

4 Implementace emulátoru **15**
4.1 Vývojové prostředí . . . 15
4.2 Implementace . . . 16
4.2.1 Funkce *FDCAN_GW_xfer_message* . . . 17
4.2.2 Funkce *FDCAN_GW_parse_rule* . . . 19
4.2.3 Implementace statistiky . . . 20

5 Hardware **22**
5.1 Procesor . . . 23
5.2 Napěťový regulátor . . . 24
5.3 Budič CAN . . . 24

6 Implementace na hardware **25**
6.1 Vývojové prostředí . . . 25
6.2 Nástroje pro ladění . . . 26
6.2.1 J-Link . . . 26
6.2.2 I-Jet . . . 27
6.3 Oživení desky . . . 27
6.3.1 Nastavení hodin . . . 28
6.3.2 Nastavení pamětí . . . 29
6.4 Nastavení periferií aplikace . . . 29
6.4.1 Vstupně výstupní port – GPIO . . . 29
6.4.2 Knihovna PDL . . . 30
6.4.3 Časovač s přerušením . . . 30
6.4.4 Rozhraní CAN . . . 32

7 Testování **35**
7.1 Finální struktura projektu . . . 35
7.2 Testování emulátoru . . . 36
7.2.1 Metody testování . . . 38
7.2.2 Výsledky testování emulátoru . . . 40
7.3 Testování finální aplikace . . . 41
7.3.1 Kvaser Memorator Pro . . . 41
7.4 Závěr testování . . . 44

8 Závěr **45**

Literatura **46**

Přílohy

A Ukázky kódu **48**

Seznam obrázků

1 Blokové schéma aplikace 3
2 Model drátového součinu na sběrnici 4
3 Arbitráž protokolu *CAN* 5
4 CAN datový rámec 6
5 CAN FD 7
6 Porovnání klasického CAN a CAN FD rámce 8

7 Založení projektu ve Visual Studio 15
8 Hlavní programová smyčka CAN FD GW 16
9 Diagram funkce *FDCAN_GW_xfer_message* 18
10 Diagram funkce *FDCAN_GW_parse_rule* 21

11 Poskytnutý hardware – deska s procesorem *Cypress S6J3340*. 22

12 Vývojové prostředí IAR Embedded Workbench 25
13 Ladící nástroj J-Link 26
14 Ladící nástroj I-Jet 27
15 Fázový závěs *PLL*. 28
16 Princip časovače. 31

17 Ukázka konzolové aplikace 36
18 Kvaser Memorator Pro 42
19 Aplikace CAN King 43

Seznam tabulek

1 Formát konfiguračního příkazu s poly CMD a parametry. 9
2 Typy příkazů aplikace CAN FD Gateway. 10
3 Formát potvrzovací odpovědi. 10
4 Příkaz *Reset* a jeho parametry. 10
5 Příkaz *Block/Pass* a jeho parametry. 11
6 Příkaz *Modify* a jeho parametry. 12
7 Potvrzení *ACK/NACK* a jeho parametry. 13
8 Struktura identifikátoru. 14

9 Mapa rozložení pinů mezi budiči a *CAN* rozhraními. 32
10 Tabulka nastavení *CAN* periferie pro standardní a *FD BRS* rychlost komunikace. 33

11 Finální struktura projektu, jednotlivé zdrojové soubory s krátkým popisem. 36
12 Ovládání konzolové aplikace. 37
13 Příkaz *Reset* a jeho třídy ekvivalence. 38
14 Příkaz *Block/Pass* a jeho třídy ekvivalence. 38
15 Příkaz *Modify* a jeho třídy ekvivalence. 39
16 Příkaz *Block/Pass* – vygenerované testy programem *Allpairs*. 39
17 Příkaz *Modify* – vygenerované testy programem *Allpairs*. 40
18 Testovací sekvence příkazů *Reset, Block/Pass, Modify*. 40

1 Úvod

Automobilový průmysl jde vývojem neustále kupředu, což zvyšuje nároky na přenos dat uvnitř automobilu. Proto je nejvíce používaným komunikačním protokolem *CAN – Controller area network*, který v rámci modernizace má novou verzi *CAN FD – CAN with flexible data rate*. Ten umožňuje vyšší přenosovou rychlost pouze s malou změnou již použitého hardwaru. Nyní tedy mnoho automobilek začíná implementovat *CAN FD*.

Automobily ale musí projít mnoha náročnými testy, aby byly splněny všechny požadavky. Testování hotových aut je ale drahé, takže jsou také vyvíjeny simulace celého auta a jeho komponent. Proto je v této práci navrhnuto řešení programového vybavení *CAN FD Gateway* – most mezi dvěma sítěmi, které jsou řízeny ze třetí sítě. Ta pak určuje, které zprávy se budou blokovat, přenášet či modifikovat. Toto zařízení pak může sloužit jako nástroj pro záměrné uvedení jiné komponenty auta do chybového stavu a ověření její funkčnosti.

2 Rozbor problematiky

2.1 Motivace

V dnešních automobilech je mnoho komponent, které společně komunikují a zajišťují tak správný a bezpečný chod automobilu po mnoho let. Proto auta musí projít mnoha náročnými testy od mechanických testů odolnosti proti nárazu až po důkladné otestování každé elektronické součásti auta tak, aby bylo vyhověno všem normám. Protože je testů mnoho, existují v automobilkách testovací systémy, které simulují veškeré komponenty auta, snaží se navodit kritické situace a zjišťují reakce auta. Ne u každé komponenty však lze navodit snadno chybový stav, proto tématem této práce je jednotka, která na komunikačním protokolu *CAN* (resp. jeho nové verzi *CAN FD*) je schopná zprávy mezi dvěma sítěmi blokovat, modifikovat, nebo propouštět – a tím je pak jednoduše možné navodit chybový stav jiné jednotky. Je tedy možné simulovat odpojení některých jednotek ze sítě, nebo jejich komunikaci záměrně zarušit, posílat nesmyslná data, nebo i data platná, ale chybná.

2.2 Analýza funkcionalit aplikace

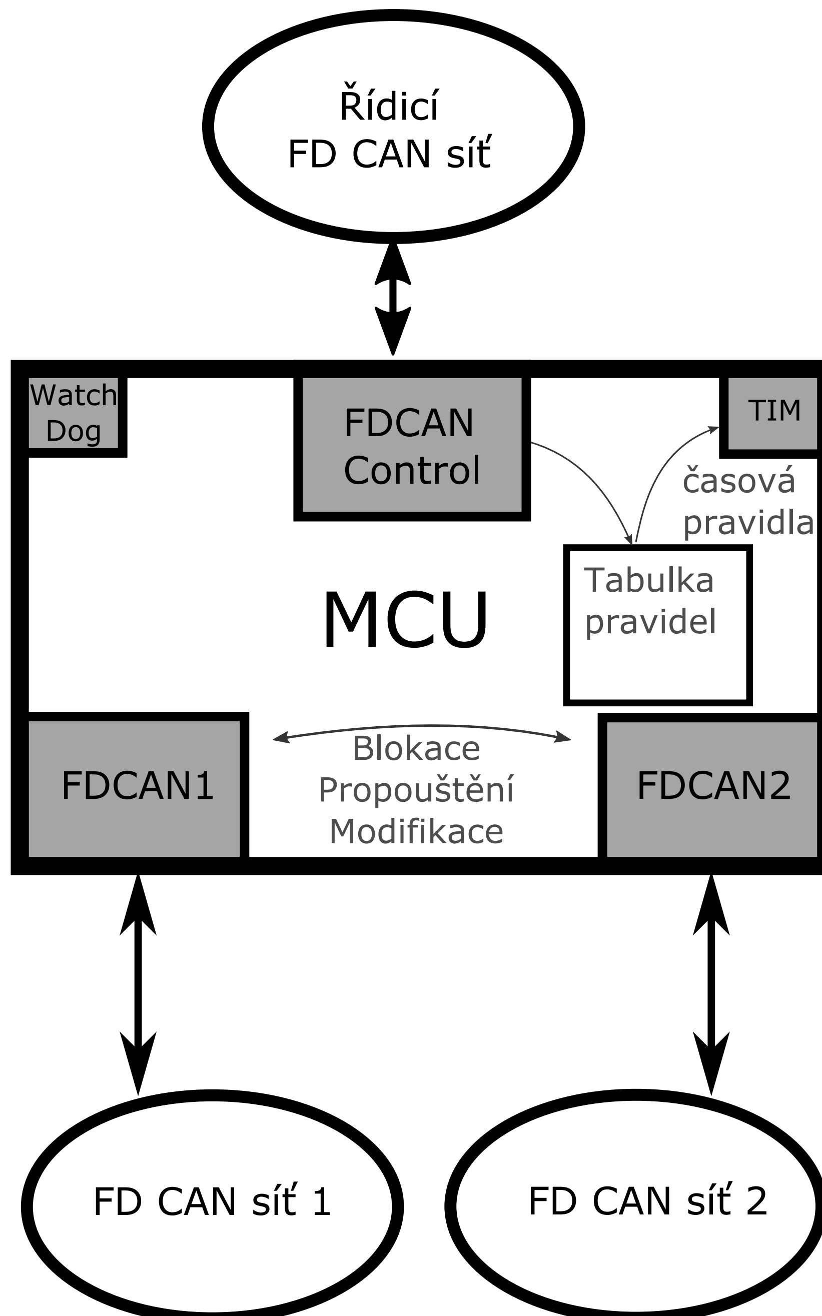
Dle zadání bylo třeba vytvořit aplikaci, která funguje jako Gateway na komunikačním protokolu *CAN*. Protože vývoj jde neustále dopředu, vyvíjí se i *CAN* - s novou verzí značenou *CAN FD* (*Flexible Data-rate* - *ISO 11898-1*).

Gateway, neboli brána je zařízení, které odděluje dvě sítě pracující s ne nutně odlišnými komunikačními protokoly [1]. V rámci aplikace pak existují 2 FD CAN sítě, kde v rámci jedné sítě si jednotky vyměňují CAN rámce. Úkolem gateway je pak tyto 2 sítě propojit - přeposlat zprávy a dle požadavků rámce zablokovat, propustit, nebo modifikovat určitým způsobem.

Pravidla určuje třetí FD CAN rozhraní, které dostává od systému příkazy, které rámce mají být blokovány/propouštěny/modifikovány. Je zde také požadavek, aby některá pravidla byla aktivní jen po určitou dobu (dobu určenou časem, nebo počtem rámců).

Další požadavky jsou:

- Přenositelnost na jiné HW platformy
- Podpora manipulace až s 8 *CAN FD* zprávami současně
- Implementace *CRC* aplikační vrstvy (správné i záměrně chybné)
- Statistiky – počet přenášených zpráv
- Důkladné otestování



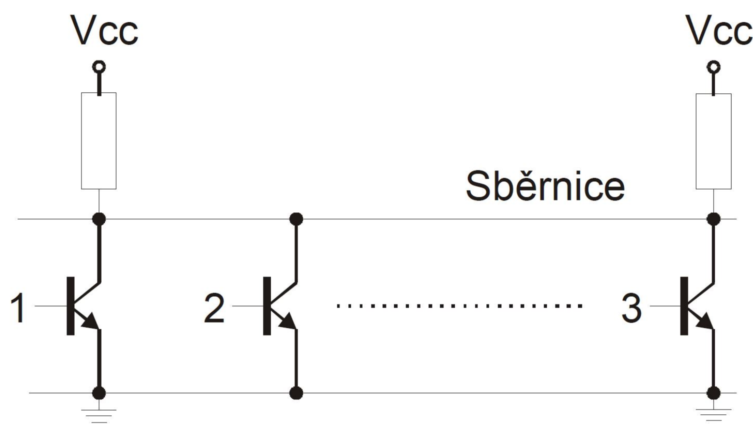
Obr. 1 Blokové schéma aplikace. Vyznačené zapojení do tří FDCAN sítí - 1, 2 a řídicí.

2.3 Komunikační protokol CAN

2.3.1 Princip protokolu

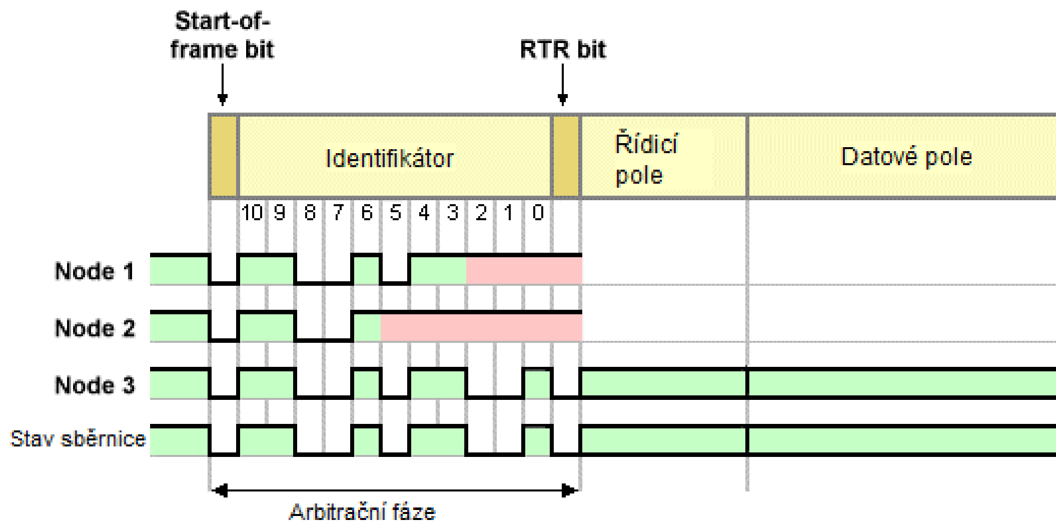
CAN je protokol vyvinutý firmou Bosch v roce 1983 a oficiálně vydán v roce 1986. V pozdější verzi 2.0 byl vydán v roce 1991 (verze 2.0A se standardním 11 bit identifikátorem a 2.0B s rozšířeným 29 bit identifikátorem). Hlavní oblastí aplikace *CAN* protokolu je průmysl a to zejména automobilový, kde se používá pro komunikaci mezi elektronickými prvky automobilu (brzdy, motor, volant atd.)[2].

Samotný protokol funguje na bázi řešení kolizí (metoda CSMA/CR - *Carrier Sense Multiple Access with Collision Resolution*), kdy se při vysílání využívá tzv. drátového součinu. V klidovém stavu je na CAN sběrnici logická 1, neboli recesivní stav. Jakmile nějaké zařízení na sběrnici chce začít vysílat stáhne sběrnici do logické 0, neboli do dominantního stavu. Pokud začne vysílat několik jednotek najednou, ničemu to nevadí, protože kolizi vyřeší arbitráž[3].



Obr. 2 Model drátového součinu na sběrnici, recesivní log. 1 (klidový stav), dominantní log. 0 při sepnutí zařízení[3].

Arbitráž toho, kdo bude v danou chvíli vysílat a celkově adresování jednotek, se určí za pomoci identifikátoru zprávy (11 nebo 29 bit). Logická 0 jakožto dominantní bit má vyšší prioritu, proto čím nižší identifikátor, tím vyšší priorita. Každé zařízení na sběrnici při vysílání také zpětně čte data ze sběrnice a pokud například jednotka v danou chvíli vyslala recesivní bit svého identifikátoru, ale na sběrnici přečetla dominantní bit, tak tím pro tuto jednotku arbitráž končí, protože nejspíše je na sběrnici zařízení, které chce vysílat s identifikátorem zprávy s vyšší prioritou. K zajištění toho, aby i po arbitráži nenastala kolize, musí být daný identifikátor v dané CAN síti jedinečný [3].



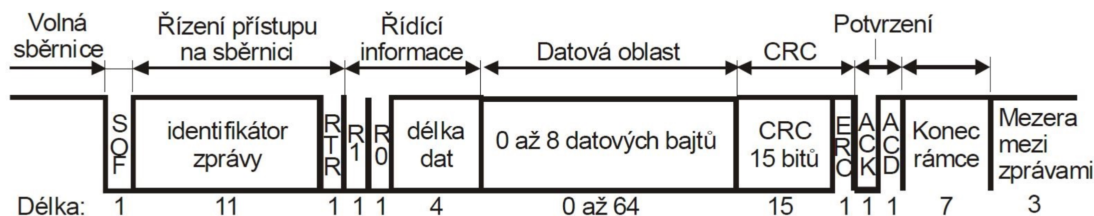
Obr. 3 Arbitráž protokolu *CAN*, sběrnici zde získá Node 3, který pak jako jediný vysílá [4].

Aby se uzel vůbec mohl pokusit o začátek komunikace, musí na sběrnici detekovat klidový stav. Při detekci komunikace vysílat nesmí a naopak potvrzuje přijetí zprávy ze sběrnice, a to už v rámci dané zprávy (*ACK* bit a teprve po něm přichází *Konec rámce*). Pokud jednotka detekuje nějakou chybu v komunikaci, oznámí to i na sběrnici (vysláním 6 po sobě jdoucích recesivních nebo dominantních bitů – dominantní pouze pokud jednotka není v tzv. *error passive* stavu). Tím také zaruší komunikaci pro ostatní uzly tak, aby už zprávu nedostaly, protože nemusejí chybu také detekovat - tj. funguje zde systém detekuji chybu, zprávu nebude mít nikdo. Odeslání dané zprávy se pak musí opakovat. Toto chování má také negativní důsledek, že vadná jednotka by neustále zarušovala komunikaci na sběrnici. Proto jsou implementovány také chybové stavy - jedná se jednoduše o počítadla, kolikrát jednotka detekovala chybu na sběrnici a dle stanovených konstant přechází mezi chybovými stavy (stav *error active* - výchozí bezchybový, *error passive* - jednotka už oznamuje chybu recesivně a zaznamenává do počítadla, *bus-off* - jednotka je prakticky oddělená od sběrnice, nemůže ani vysílat, ani přijímat, nutný *reset*).

Z hlediska ISO OSI modelu se dá *CAN* protokol zařadit následovně [3]:

- Aplikační vrstva
    - Obsah rámců
    - Kdy a za jakých podmínek jsou rámce vyslány
- Spojová vrstva
    - Samotný *CAN* protokol
    - Řízení přístupu k médiu - kolize
    - Adresace - arbitráž
    - Zabezpečení
    - Reakce na chybové stavy
- Fyzická vrstva
    - Reprezentace bitů a signálové úrovně
    - Přenosové médium
    - Parametry vedení, konektory, rychlost

CAN pracuje se čtyřmi typy rámců - datový rámec (pro přenos dat o délce 0 - 8 bajtů), rámec žádosti o data (*remote request frame* - jednotka jím žádá o data s daným identifikátorem), chybový rámec (6 dominantních nebo recesivních bitů je vysláno v případě detekce chyby), rámec přetížení (dnes nepoužívaný - stejný formát jako chybový, ale jednotka tím žádá o odložení dalšího rámce)[3].



Obr. 4 CAN datový rámec, se standardním identifikátorem[3].

V rámci zabezpečení dat a detekce chyb *CAN* používá kontrolní součet *CRC* (pokud se přijatý a vypočtený *CRC* kód liší, je vyslána chyba) a také *bit stuffing*. Je použito kódování *NRZ* (*not return to zero*) a při vysílání 5 a více bitů stejné log. hodnoty je vložen bit opačný – protože *CAN* není synchronní (tj. není rozváděn hodinový signál na který by se sběrnice synchronizovala), je nutné měnit stav sběrnice, aby byla data správně přečtena. Dalším efektem *bit stuffingu* je, že 6 bitů stejné hodnoty po sobě je již detekováno jako chyba[3].

Jak bylo řečeno výše, *CAN* není synchronní, ale kvůli různé vzdálenosti jednotek na sběrnici je nutné kompenzovat zpoždění signálů dané jejich šířením po fyzickém kanálu. K tomu *CAN* využívá tzv. *Time quant* – krátkých časových úseků ke kompenzaci šíření signálů. V podstatě jde o před-děličku frekvence na kterém jednotka běží a v rámci nastavení se určuje počet těchto kvant pro správné určení *Sample pointu* – bodu, kde aktuální stav sběrnice se bere jako platná hodnota bitu. Tímto procesem se také jednotky navzájem synchronizují. Pokud ale jsou zařízení od sebe příliš daleko při použití dané nominální přenosové rychlosti, tak maximální počet časových kvant, na kompenzaci stačit nemusí a je nutné snížit celkovou rychlost komunikace[3].

Pro správnou funkci *CAN* sběrnice je také nutné přidat zakončovací odpory – terminátory. Ty zamezují odrazům signálu na sběrnici tak, že se energie přemění jednoduše na teplo a odraz se utlumí. Obvykle se používá 120 Ohm odpor vložený na obou stranách sběrnice mezi vodiče *CAN_H* a *CAN_L*. Kanál pak využívá diferenciální (log. úroveň je dána rozdílem napětí) komunikaci mezi těmito vodiči[3].

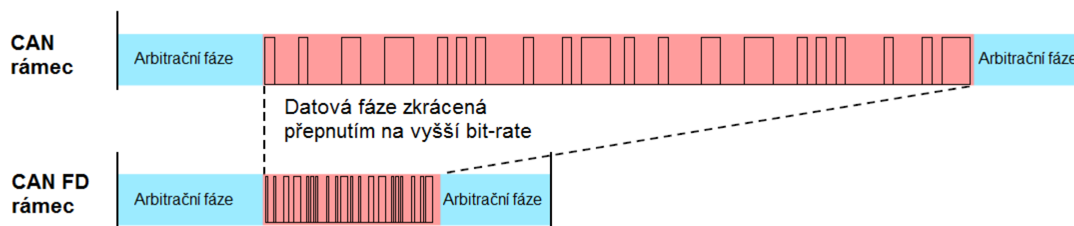
Z popisu výše také plyne, že pro správnou funkci sběrnice *CAN* jsou nutné alespoň dvě jednotky. Jedna, která vysílá, a druhá, která potvrzuje rámce.

2.3.2 Verze CAN FD

Protože v dnešních autech je požadavek na stále vyšší a vyšší propustnost dat, původní specifikace 1 *Mbit/s CANu* již nemusí stačit. Proto firma Bosch v roce 2012 přišla s dalším rozšířením, které vlastnostmi pokrývá oblast mezi standardním *CAN* a dražšími jednotkami jako *FlexRay*, *Ethernet* apod. Zároveň přechod z *CAN* na *FlexRay* nebo jinou technologii by mohlo být nákladný na hardware a implementaci. Proto ve verzi FD má vyšší propustnost, ale stále i zpětnou kompatibilitu s klasickými *CAN* jednotkami[5]. Kompatibilita funguje pouze pokud jsou vysílány standardní *CAN* rámce, *FD* rámce starší jednotka nerozezná a hlásí chyby.

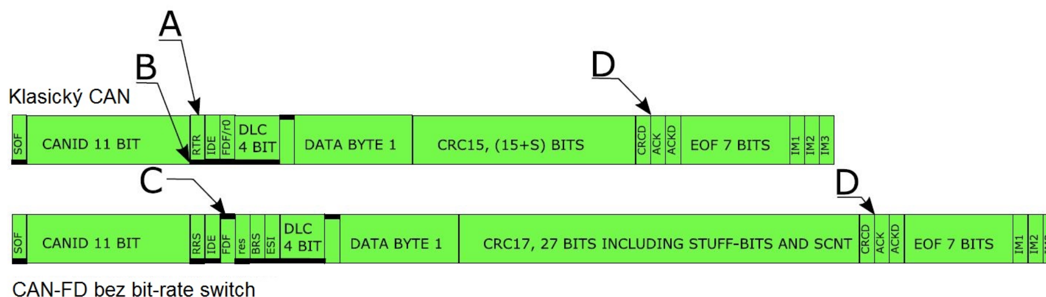
Co se týče *arbitrace* a potvrzování zpráv, *CAN FD* tuto funkci nijak nemění. Co se změnilo je přepnutí na vyšší rychlost (*bit rate switch*), datové rámce v této verzi přenáší z původních 0 - 8 bajtů nově 0 - 8, 12, 16, 20, 24, 32, 48, 64 bajtů zakódovaných v poli délky(*DLC field*). Tudíž nyní lze přenést mnohem více dat za časový úsek odpovídající přenosu až 8 bajtů původního *CAN* protokolu[5].

Z hlediska software není třeba nic měnit, pokud není zapotřebí využít vyšší rychlost. Z hlediska hardwaru je nutná změna fyzického transceiveru (budiče) – pokud je využívána rychlost vyšší jak *1Mbit/s*[5].



Obr. 5 *CAN FD* s bit rate switch – porovnání se standardním *CAN* rámcem [5].

Následující obrázek (6) porovnává klasický a *FD CAN* rámec ve významu jednotlivých bitů. S klasickým 11 bit i rozšířeným 29 bit identifikátorem je začátek rámce stejný pro obě verze *CAN*. Rozdíl mezi *RTR* a *RRS* (**A**) (*Remote request*) je pouze v přejmenování a je obvykle dominantní pro datové rámce (označení **B** značí dominantní nebo recesivní hodnotu v daném *CAN* nebo *CAN FD* rámci). Opravdová změna je u bitu *r0/FDF* (**C**), v původním *CANu* byl tento bit rezervovaný(*reserved*) a v log. 0, v *FD CAN* má význam *Flexible Data Format* – tj. určuje, zda se jedná o standardní nebo *FD CAN* rámec. Po tomto bitu následuje *DLC* u standardního a *res* bit u *FD*, takže jednotky klasického *CAN* v tuto chvíli misinterpretují rámec a ohlásí chybu. Naopak *FD CAN* jednotky jsou schopny pracovat s oběma formáty. Poslední vyznačená část **D** poukazuje na identické zakončení obou *CAN* rámců.



Obr. 6 Porovnání klasického CAN a CAN FD rámce.[5].

2.4 Aplikační CRC

CRC, neboli *Cyclic Redundancy Check* je funkce používaná nejčastěji v komunikačních protokolech k zajištění bezchybného přenosu dat – resp. k detekci chyby v přenosu (v důsledku šumu a jiných negativních vlivů na přenosovém kanálu). Cílem je maximalizovat *Hammingovu vzdálenost* (počet bitů, o který se změní jedno kódové slovo na jiné kódové slovo – čím větší vzdálenost, tím větší pravděpodobnost odhalení chyby) a minimalizovat velikost redundantních (nadbytečných) dat.

Obecně jsou data reprezentována jako binární polynom $M(x)$ stupně $n-1$ kde hodnoty koeficientů polynomu odpovídají jednotlivým datovým bitům. Tento datový polynom je pak dělen generujícím polynomem $G(x)$ stupně k. Zbytek po dělení polynomu $M(x)$ polynomem $G(x) = R(x)$ o délce k je přidán jako redundantní informace k datům a celé kódové slovo délky $n+k$ je následně odesláno. Druhá strana jakmile dostane tato data, spočítá rovněž *CRC* a pokud se zbytky po dělení liší, nastala chyba. Pokud se rovnají, stále mohlo dojít k chybě, která ale nebyla detekována a závisí na použitém *CRC* a dané pravděpodobnosti chyby, zda je dostačující pro dané požadavky aplikace.

Protože z hlediska požadavků aplikace někdy nestačí kontrolní součet na úrovni *CAN protokolu* (pravděpodobnost chyby je stále relativně vysoká), je zavedeno také aplikační *CRC*, tudíž *CAN* periferie automaticky při příjmu rámce sama spočítá a ověří *CRC* a případně oznámí chybu, ale je tedy ještě nutné dopočítat manuálně – na procesoru aplikační *CRC*. Využívá se *CRC8* – tj. generující polynom vypadá následovně:

$$x^8 + x^5 + x^3 + x^2 + x + 1 = \mathit{0x2F} \tag{1}$$

Protokol *CAN* používá generující polynom stupně 15 – *CRC15*:

$$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1 = \mathit{0x4599} \tag{2}$$

3 Popis komunikace

3.1 Funkce modulu CAN Gateway

Jak bylo řečeno výše, aplikace funguje jako *Gateway* na *CAN FD* protokolu. Pro tuto práci byly poskytnuty materiály původního pouze *CAN Gateway* a cílem této práce je vytvořit modul schopný pracovat s *CAN FD*.

V původní aplikaci byly definovány konfigurační zprávy, které určovaly platná pravidla pro zpracování rámců (blokace, přeposílání, modifikace). Aplikace ve výchozím stavu propouští všechny rámce oběma směry, pokud přijme nějaký příkaz z třetího *CAN* (*FD*) rozhraní, zachová se následovně – příkazy definující nové pravidlo si prozatím pouze uloží, pouze po přijmutí příkazu *Trig* jsou nová pravidla aplikována. Existuje také příkaz *Reset*, který uložená pravidla smaže a nastaví pouze výchozí pravidlo pro práci s rámci (blokace nebo přeposílání). Všechny příkazy jsou rozepsány podrobněji na následujících stránkách.

3.2 Příkazy

3.2.1 Formát a definice

Každý konfigurační příkaz obsahuje pole CMD a poté až 7 parametrů (dle typu příkazu).

CMD	param1	param2	param3	...

Tab. 1 Formát konfiguračního příkazu s poly CMD a parametry.

Pole *CMD* je jeden bajt dat, ještě dále rozdělený na samotný příkaz (jeho typ kódován do horních 4 bitů) a klesající pořadové číslo rámce – pokud je nutné konfigurační zprávu dělit do více rámců (data přesahující 8 resp. 64 bajtů – pro FD) má první paket pořadové číslo *N-1*, kde *N* značí počet rámců. Hodnota tedy postupně klesá až k *0*. Pole *CMD* je tedy obsaženo v každém rámci, aby se zajistilo správné doručení všech dat. Zároveň protože v poli *CMD* je také obsažen typ příkazu, usnadňuje to implementačně členění dat.

Následující tabulka shrnuje definované příkazy a jejich kódy:

Kód příkazu	Název	Popis funkce
0x00	Reset	Uvede modul do definovaného stavu.
0x01	Block/Pass	Požadavek na blokování/přeposílání zprávy.
0x02	Modify	Požadavek na modifikaci přeposílané zprávy.
0x03	Trig	Požadavek na aktivaci nastavených funkcí.
0x04	Stat	Požadavek na odeslání statistiky.

Tab. 2 Typy příkazů aplikace CAN FD Gateway.

Parametry mají proměnnou délku i počet, záleží na typu příkazu. Minimální velikost parametru je 1 bajt.

Pro potvrzení příkazu se používá následující formát – opět pole *CMD* (horní 4 bity typ příkazu na který se odpovídá, dolní 4 bity vždy 0 – není zde třeba počítat rámce). Potvrzení má také 2 parametry, kam se vkládá stavový registr *CAN* periferií, nebo statistiky.

CMD	param1	param2

Tab. 3 Formát potvrzovací odpovědi.

3.2.2 Reset

Příkaz *Reset* s jedním parametrem nastavuje výchozí pravidlo pro blokaci nebo přeposílání. Specifická pravidla (v danou chvíli rozpracovaná) smaže. Následující tabulka popisuje příkaz *Reset* a jeho parametry:

Parametr	CAN/FDCAN velikost	Hodnota	Popis
State	1B/1B	`0x00`	Po příkazu *Trig* jsou všechny rámce oběma směry transparentně přeposílány.
		`0x01`	Po příkazu *Trig* jsou všechny rámce oběma směry blokovány.

Tab. 4 Příkaz *Reset* a jeho parametry.

3.2.3 Block/Pass

Příkaz *Block/Pass* určuje pravidlo pro výjimku z výchozího chování (určeném po zapnutí, nebo příkazem *Reset*) pro určité zprávy s daným identifikátorem. Pravidla tohoto typu jsou ukládána a jsou aktivní po příkazu *Trig*. Příkaz *Reset* smaže rozpracovaná pravidla.

Následující tabulka přehledně popisuje funkce příkazu *Block/Pass*:

Parametr	CAN/FDCAN velikost	Hodnota	Popis
ID	4B/4B	11 nebo 29 bit ID	Identifikátor, jehož rámce budou použitím tohoto pravidla blokovány nebo propouštěny – opak výchozího pravidla.
Type	1B/1B	`0x00`	Stálé pravidlo, časově neomezené.
		`0x01`	Pravidlo omezené počtem rámců v parametru *Length*.
		`0x02`	Pravidlo omezené časem daným v parametru *Length*.
Length	2B/2B	Počet použití	Když *Type* = `0x01` parametr *Length* určuje počet rámců, na které se pravidlo aplikuje a pak skončí.
		Doba použití (*ms*)	Když *Type* = `0x02` parametr *Length* určuje dobu v milisekundách, po kterou je pravidlo aktivní a pak skončí.

Tab. 5 Příkaz *Block/Pass* a jeho parametry.

3.2.4 Modify

Příkaz *Modify* je v rámci aplikace asi nejsložitější, ale zároveň "nejmocnější"pravidlo. Má řadu funkcí, jako modifikace dat na základě masky a dat pravidla, je možné modifikovat i aplikační *CRC* a to buďto správně spočítat nové *CRC*, nebo záměrně špatné.

Stejně jako příkaz *Block/Pass* má i *Modify* parametry pro časově nebo počtem omezené trvání. Na rozdíl od *Block/Pass* ale *Modify* není nijak závislé na výchozím pravidle daném aktivací nebo příkazem *Reset*.

U tohoto příkazu ale existuje jisté úskalí, pokud by z nějakého důvodu bylo třeba přenést pravidlo pouze pomocí paketů o velikosti menší rovno než 8 bajtů, pak by nebylo možné použít počítadlo rámců tak jak je navržené (při 8 bajtech, kde první bajt je vždy pole *CMD*, takže efektivně 7 bajtů je nutné přenést 138 bajtů dat pravidla *Modify*, což je 20 rámců, ale počítadlo je jen 4 bitové, tj. maximálně 16 rámců). Počítadlo by tedy muselo být 5 bitů pro pořadí rámců a 3 bity pro typ příkazu. Pokud je zajištěno, že dat se přenese alespoň 12 bajtů (efektivně přeneseno 11 bajtů, tj. 13 rámců pravidla *Modify*) v každém rámci, pole *CMD* není nutné měnit.

Následující tabulka opět přehledně popisuje parametry příkazu *Modify*:

Parametr	CAN/FDCAN velikost	Hodnota	Popis
ID	4B/4B	11 nebo 29 bit ID	Identifikátor, jehož rámce budou použitím tohoto pravidla modifikovány.
Mask	8B/64B	Maska	Maska dat, pro nahrazení daty pravidla.
Data	8B/64B	Data	Data pravidla, kterými se nahradí data rámce (dle masky).
DLC/CRC	1B/1B	Délka dat/CRC	Horní 4 bity určují modifikaci délky dat, zde v původní aplikaci platil rozsah 0-8, hodnota `0xF` ponechala původní délku. Nově pro *CAN FD* je nutné toto pole rozšířit a použít standardní kódování délky (viz 2.3.2). V nové aplikaci tedy `0x10` znamená velikost beze změny a mimo jiné je nutné změnit zakódování na: horních 5 bitů je délka a spodní 3 bity CRC. Pokud CRC = `0x00` provede se pouze modifikace daty a maskou. Pokud CRC = `0x01` je CRC správně dopočítáno z modifikovaných dat a nakonec, pokud CRC = `0x02`, pak výpočet je záměrně chybný (správně spočítán a inkrementován o 1).
Seed	16B/16B	CRC Seed	Inicializační vektor pro výpočet aplikačního CRC.
Type	1B/1B	`0x00`	Stálé pravidlo, časově neomezené.
		`0x01`	Pravidlo omezené počtem rámců v parametru *Length*.
		`0x02`	Pravidlo omezené časem daným v parametru *Length*.
Length	2B/2B	Počet použití	Když *Type* = `0x01` parametr *Length* určuje počet rámců, na které se pravidlo aplikuje a pak skončí.
		Doba použití (*ms*)	Když *Type* = `0x02` parametr *Length* určuje dobu v milisekundách, po kterou je pravidlo aktivní a pak skončí.

Tab. 6 Příkaz *Modify* a jeho parametry.

3.2.5 Trig

Příkaz *Trig* nemá žádné další parametry, aktivuje doposud pouze ukládaná nastavení získaná po předchozím příkazu *Trig* nebo po zapnutí zařízení.

3.2.6 Stat

Příkaz *Stat* je jako *Trig* také bez parametrů, po přijetí tohoto příkazu aplikace potvrzuje standardně pomocí *ACK/NACK*, ale v parametrech nejsou stavové registry CAN (FD) řadičů, ale statistiky pro obě rozhraní. Počítá se průměrný počet rámců, který za sekundu projde daným rozhraním.

3.2.7 ACK/NACK

Po každém příkazu modul odesílá potvrzení na aplikační úrovni a to společně s *ID* erroru, který případně nastal a s stavovými registry obou CAN (FD) rozhraní. Následující tabulka shrnuje formát potvrzovací zprávy:

Parametr	CAN/FDCAN velikost	Hodnota	Popis
ErrID	1B/1B	ID erroru	Udává ID erroru, který nastal od posledního potvrzení *ACK/NACK*. Pokud žádný problém nenastal, ID je 0.
CAN1 FD ERR	2B/2B	Error registr	Stavový error registr CAN1 FD rozhraní
CAN2 FD ERR	2B/2B	Error registr	Stavový error registr CAN2 FD rozhraní

Tab. 7 Potvrzení *ACK/NACK* a jeho parametry.

3.3 Identifikátor Gateway

Identifikátor *Gateway* modulu je standardní 11 bitový. 2 nejvyšší bity jsou nulové, následované 5 bity – adresa modulu určená hardwarovým *DIP switchem*. Další 3 bity jsou opět nulové, poslední bit má hodnotu 0, pokud se jedná o příkaz a hodnotu 1, pokud se jedná o potvrzení *ACK/NACK*. Následující tabulka shrnuje formát identifikátoru:

2 bity	5 bitů	3 bity	1bit
00	DIP adresa	000	*ACK/NACK*

Tab. 8 Struktura identifikátoru.

3.4 Trig Broadcast

Příkaz *Trig* také podporuje *broadcastování*, kde adresa 0 je rezervována právě pro *broadcast*. Na *Trig* ať už *unicastový* nebo *broadcastový* modul odpovídá standardním způsobem popsaným výše.

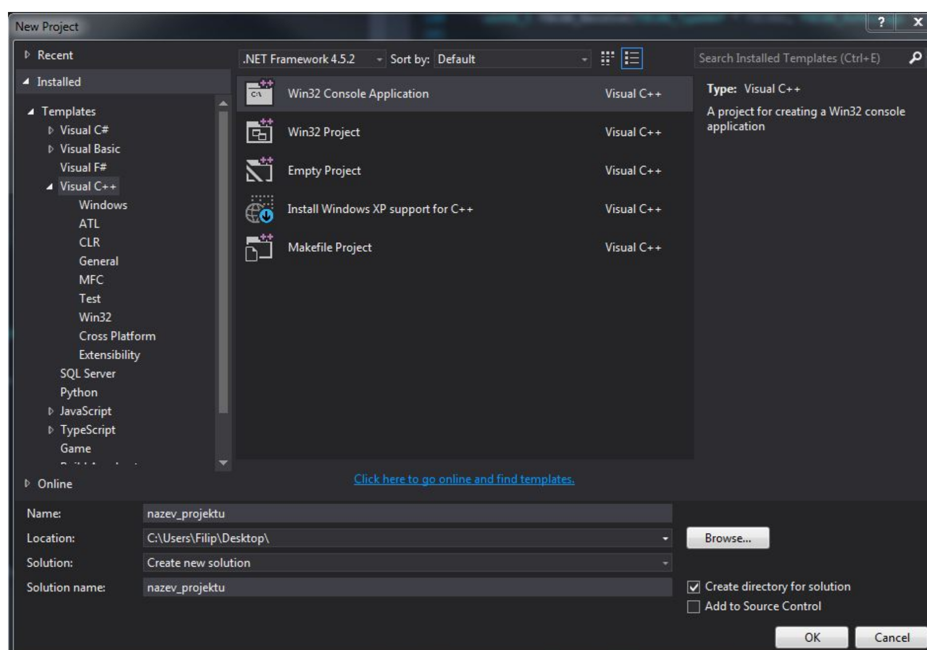
4 Implementace emulátoru

Protože hardware nebyl dostupný v době začátku této práce, bylo rozhodnuto, že se naimplementuje emulátor aplikace. Zároveň při správné abstrakci od hardwaru se aplikační programové vybavení modulu lépe odladí – není třeba dbát na komunikaci s *CAN FD* rozhraními, ale pouze na samotný simulovaný obsah zpráv.

4.1 Vývojové prostředí

Pro vývoj emulátoru bylo použito vývojové prostředí *Microsoft Visual Studio 2015* se studentskou licencí na verzi *Community*. Bylo zvoleno pro snadnou přenositelnost projektu a celkové zkušenosti s prostředím.

Samotné založení projektu je velmi snadné, jednoduše stačí spustit Visual Studio a založit nový projekt pro jazyk C/C++ (v případě nové instalace je nutné doinstalovat balíčky, aby jazyk C byl kompilovatelný). Výchozí konzolový projekt obsahuje soubor *nazev_projektu.cpp*, tzn. že je v jazyce *C++*, ale nic nebrání tomu, změnit příponu na *.c* a pokračovat ve vývoji v čistém jazyce *C*. Pokud hlavní soubor, odkud se aplikace spouští, obsahuje funkci *main*, konzole se bez problémů spustí (na začátku pro ověření funkce je dobré nechat vypsat například klasický *Hello World*).

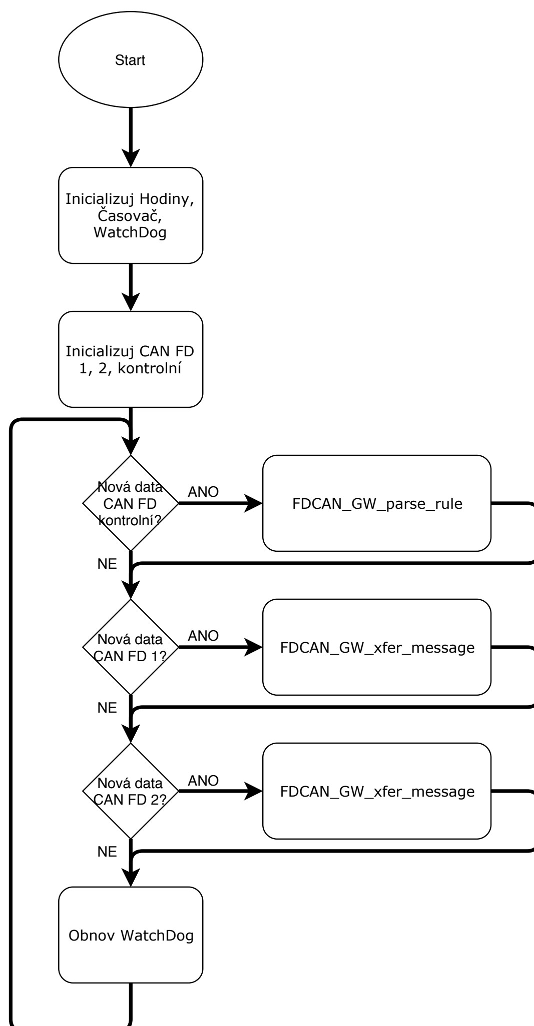


Obr. 7 Založení projektu ve *Visual Studio 2015 Community*.

4.2 Implementace

Po založení projektu stačilo začít programovat. Z hlediska aplikace bylo nejprve třeba zajistit nějaké rozhraní pro příjem(*Receive – Rx*) a odesílání(*Transmit – Tx*). Proto byl navržen prototyp v souboru *fdcan.h*, kde jsou definice *Rx* a *Tx* funkcí a samozřejmě také inicializační funkce pro *CAN FD* periferie. Dále jsou zde definované datové typy pro naplnění daty pro příjem nebo odeslání (*FDCAN_RxMessage* a *FDCAN_TxMessage*). Všechny tyto definice by měly být společné nezávisle na hardwaru, ať už se jedná o simulaci, nebo konkrétní implementaci na *HW*, tento hlavičkový soubor bude stejný.

Z hlediska emulátoru bylo pochopitelně nutné hlavičkový soubor konkrétně implementovat – k tomu slouží soubor *fdcan_emulated.c*. Pro simulaci příjmu zde postačí jednoduché překopírování staticky předdefinovaných zpráv do datových struktur *FDCAN_RxMessage*. Opačný proces tj. odesílání není třeba kopírovat, stačí v *debuggeru* vývojového prostředí umístit *breakpoint* právě do funkce *FDCAN_Trasmit* a ověřit, zda struktura *FDCAN_TxMessage* byla naplněna aplikací dle očekávání.



Obr. 8 Hlavní programová smyčka CAN FD GW – inicializace, obsluha CAN rozhraní.

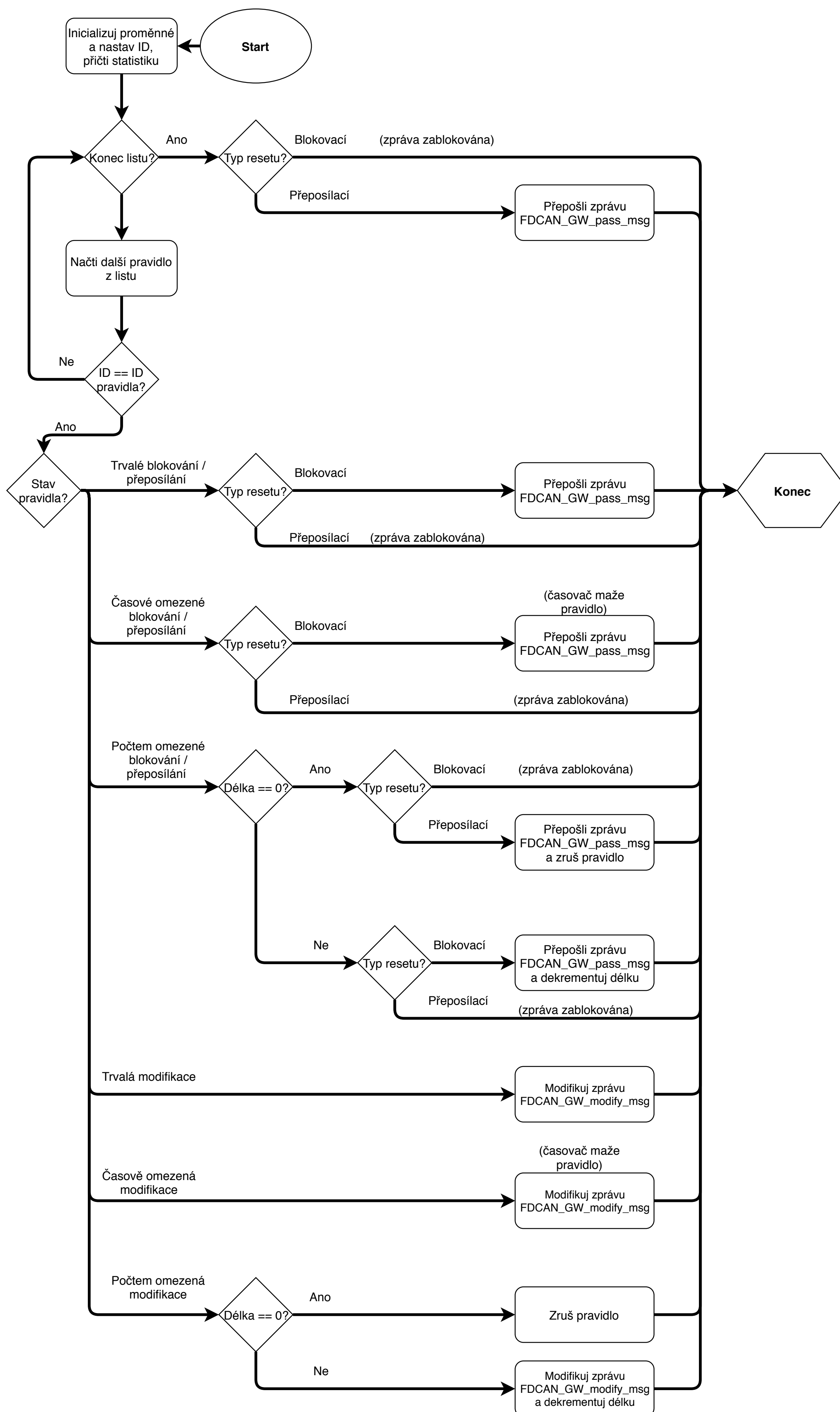
Po implementaci tohoto jednoduchého simulačního rozhraní následovala implementace samotného firmwaru tj. *Gateway*. Z hlediska hlavního procesu (viz obrázek 8) je nutné obsloužit několik funkcí, po zapnutí aplikace je nutné aktivovat veškerá rozhraní (*CAN FD* periferie, časovače, *WatchDog*). Dále následuje nekonečná smyčka, která se postupně ptá všech *CAN FD* rozhraní na nová data, pokud nějaká jsou, je nutné je patřičně zpracovat. Nejprve se aplikace dotazuje kontrolního *CAN FD* rozhraní, odkud chodí příkazy a pravidla systému. Zpracování dat kontrolního *CAN FD* rozhraní probíhá ve funkci *FDCAN_GW_parse_rule*. Dále se aplikace vyptává postupně obou *CAN FD* periferií, mezi kterými se zprávy přeposílají. Pro obě rozhraní se používá funkce *FDCAN_GW_xfer_message* na zpracování nových dat. Tím hlavní smyčka končí (ještě je třeba *refreshovat WatchDog*, ale to se v době implementace emulátoru neuvažovalo).

Hlavní část aplikace se skrývá uvnitř funkcí *FDCAN_GW_parse_rule* a *FDCAN_GW_xfer_message*, proto jsou dále rozepsány podrobněji v textu a také v procesních diagramech.

Aby celá aplikace mohla fungovat jak má, existuje ještě několik funkcí mezi *Transmit/Receive* a přeposílajícími/parsovacími funkcemi. Zde je myšlenka velmi snadná, jedná se o vlastní provedení příkazu, nebo jeho dekódování. V případě *Block/Pass* se data dané zprávy jednoduše překopírují do dat k odeslání na druhém rozhraní, než ze které zpráva přišla (funkce *FDCAN_GW_pass_msg*). Pro příkaz *Modify* je zde funkce *FDCAN_GW_modify_msg*, která kromě překopírování, data také modifikuje na základě masky a dat pravidla, případně zkracuje nebo prodlužuje délku (parametr *DLC*) a nakonec také počítá *CRC* pokud je požadováno – a to buďto správně, nebo záměrně chybně (dle parametru *CRC*) s pomocí inicializačního vektoru (*CRC Seed*). Poslední důležitou funkcí je *FDCAN_GW_progress_mod_rule*, která s použitím globálních indexů rozpracovaného pravidla (index pro pozici v listu a index pro pozici uvnitř struktury tvořeného pravidla) zpracovává parametry příkazu *Modify*, který je nutné dělit do několika rámců (max. délka dat je 64 bajtů, tj. minimálně 3 rámce pro *Modify*). Funkce postupně plní strukturu nového pravidla, jakmile je dokončeno, jsou indexy resetovány. V jednu chvíli může být rozpracováno pouze jedno pravidlo typu *Modify*.

4.2.1 Funkce *FDCAN_GW_xfer_message*

Funkce *FDCAN_GW_xfer_message* (viz obrázek 9) se stará o samotné přeposílání zpráv mezi *CAN FD* 1 a 2 rozhraními. Rozhoduje se na základě aktuálně platných pravidel daných uvnitř statického listu až o 10 položkách (teoreticky lze určit jakýkoli počet pravidel, protože je dáno konstantou, kolik pravidel je možné v jednu chvíli aplikovat – samozřejmě při velkém počtu pravidel to může mít vliv na celkový výkon aplikace a je nutné také dbát na dostatek paměti).



Obr. 9 Diagram funkce *FDCAN_GW_xfer_message*.

Implementačně funkce *FDCAN_GW_xfer_message* postupuje následovně. Nejdříve jsou inicializovány potřebné proměnné a hlavně je vzato *ID* zprávy, o které se rozhoduje. Dále je nutné projít celý aktivní list pravidel, zda pro dané *ID* zprávy neexistuje speciální pravidlo. Pokud se žádné takové nenajde, postupuje se dle výchozího pravidla (blokovací nebo propouštěcí pravidlo dáno po aktivaci nebo posledním *Reset* příkazem). Pokud se *ID* shoduje s některým z *ID* pravidla v listu, je dále rozhodováno, o jaké pravidlo jde a jak se zprávou naložit. Rozhodne se pomocí výčtového typu, který definuje *stav* pravidla. *Stav* je spojení několika parametrů pravidla pro snažnější rozčlenění. Pravidla mohou být typu *block/pass* nebo *modify* a mohou platit neomezeně, po určitý čas, nebo počet zpráv – tj. celkem 6 typů pravidel.

U *Block/pass* pravidel je nutné brát v potaz také stav výchozího pravidla, neboť aplikace musí být opačná – propouštěcí *Reset* zablokuje zprávu s daným *ID*. U pravidel typu *Modify* je pravidlo aplikováno nezávisle na výchozím *Reset* pravidlu.

Z hlediska časových typů pravidel, ty neomezené platí vždy pro dané ID, ale u pravidel omezených počtem se musí snižovat parametr *Length*, jakmile dojde k *0*, je pravidlo smazáno a daná zpráva je již zpracována jako bez speciálního pravidla. Časově omezená pravidla maže časovací funkce (v případě emulátoru samostatný *Thread*, v případě aplikace na *HW* se o vymazání pravidla stará časovač – resp. jeho *callback*). Opět je zde snižován parametr *Length* a časovací funkce vymaže pravidlo, když dojde k *0*. Takže v rozhodování je časové pravidlo aplikováno stejně jako to neomezené.

4.2.2 Funkce *FDCAN_GW_parse_rule*

Funkce *FDCAN_GW_parse_rule* (viz obrázek 10) je druhá důležitá součást aplikace. Jak název nartpovídá, slouží k dekódování nového pravidla, které přišlo z kontrolního *CAN FD* rozhraní. Všechna pravidla je možné přenést v jediném rámci (64 bajtů), kromě pravidla *Modify*, které musí být rozdělené. Implementačně se nejdříve inicializují proměnné a vybere se z datové oblasti zprávy samotný příkaz, který je vždy obsažen v prvním bajtu dat – to zároveň usnadňuje rozčlenění. Dle kódu příkazu je pak rozhodnuto, o který příkaz se jedná, a ten je pak příslušně zpracován. Nejsnadnější pravidla z hlediska zpracování jsou pravidla *Reset, Trig, Stat*. *Reset* jednoduše nastaví *flag*, že byl proveden a také se nastaví nové výchozí pravidlo (pouze uloží, aplikováno je až po příkazu *Trig*) dané druhým bajtem dat zprávy. Příkaz *Trig* funguje jako spouštěč nového uloženého nastavení, takže jeho zpracování je také velmi jednoduché. Pokud byl nastaven *Reset flag*, je příkazem *Trig* aktivováno nové výchozí pravidlo a ukazatele na listy aktivního a nového nastavení jsou prohozeny. Nový list (v tuto chvíli se starým nastavením) je smazán.

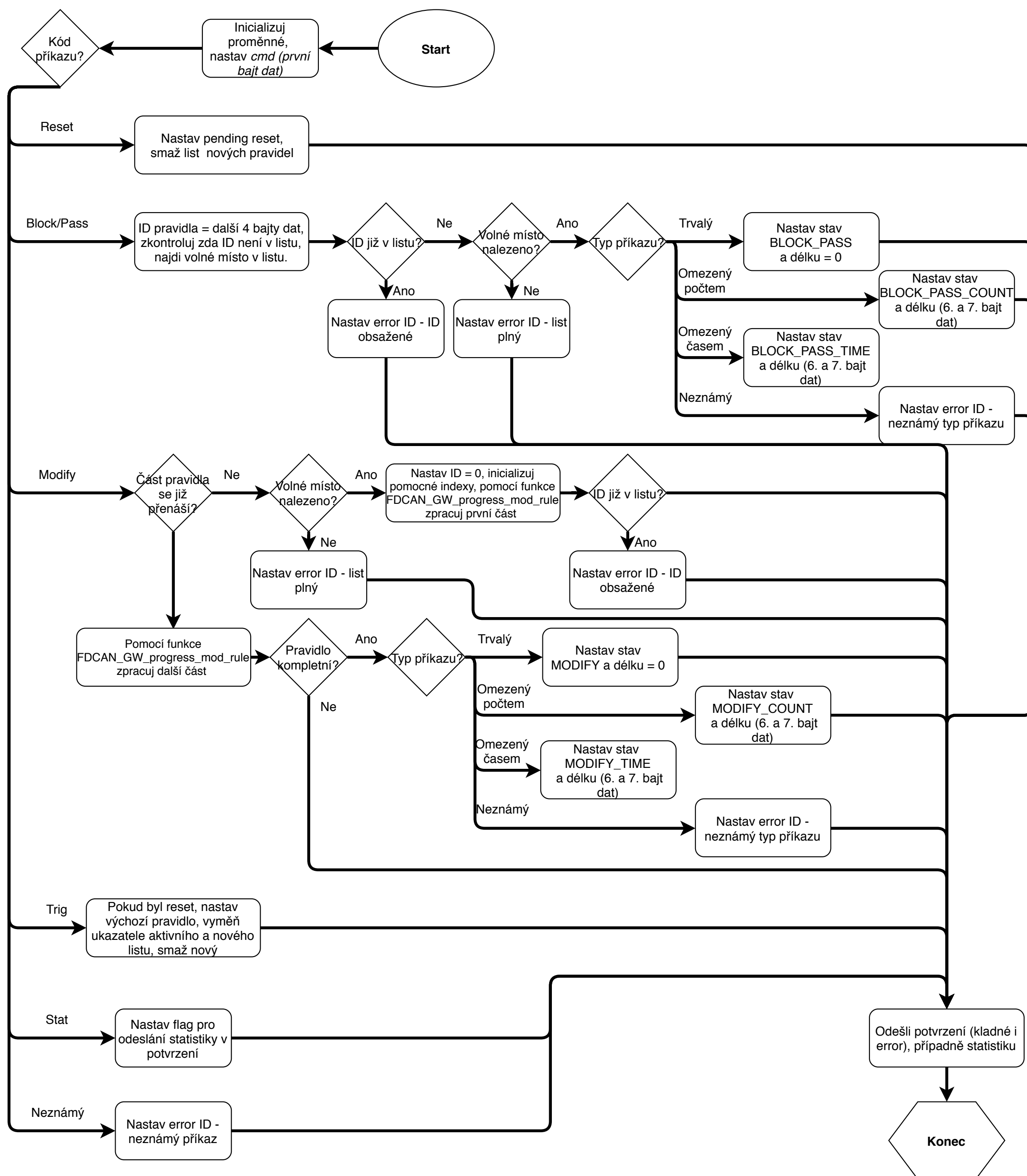
Pravidlo s kódem *Block/Pass* již má více parametrů než jednoduché příkazy a také je to jedno z typu pravidel, které jsou ukládány do listu. Nejdříve je nutné určit *ID*, pro které bude pravidlo platit. Musí se prohledat celý list, jestli už náhodou pravidlo se stejným *ID* neexistuje, pokud ano, je nastaven *error ID* a v dalším potvrzení je odeslán *error*. Chyba je generována také pokud v listu není nalezeno místo. Pokud místo existuje a není žádné konfliktní *ID*, je nutné už jen určit *stav* pravidla (tj. výčtový typ – spojení

kódu příkazu a typu platnosti – viz strana 17). Jakmile je i *stav* nastaven, je odesláno potvrzení a tím je nové pravidlo *Block/Pass* zpracováno.

Implementačně nejsložitější je zpracování pravidla *Modify* (viz obrázek 10), protože je dělené do několika rámců. Proto byly zavedeny pomocné globální proměnné, které určují index rozpracovaného pravidla a index do struktury samotného pravidla, na který se zapisují další data do příslušných parametrů. Nejdříve je nutné zjistit na základě zmíněných globálních proměnných, zda již nějaké pravidlo *Modify* je rozpracované – tudíž, jestli se jedná o další rámec jednoho pravidla, nebo je přijímáno nové. Pokud se jedná o nové pravidlo, tak je nutné stejně jako u pravidla *Block/Pass* vyhledat prázdné místo v listu (jinak *error*), dále nastavit *ID* prázdného místa na *0*, inicializovat globální proměnné (index rozpracovaného na nalezené prázdné místo, index struktury pravidla na *0*). Následným zavoláním funkce *FDCAN_GW_progress_mod_rule* jsou data ze zprávy přesunuta do struktury pravidla. Až v tuto chvíli je možné ověřit, zda pravidlo s tímto *ID* není již v listu. Pokud již takové pravidlo existuje, je nastaven *error* a nedokončené pravidlo je zahozeno a vymazáno z listu. Pokud žádný problém není, čeká se na další rámec, tj. další část pravidla *Modify*. Jakmile dorazí další část, funkce *FDCAN_GW_progress_mod_rule* se opět postará o vložení dat do struktury. Pokud je pravidlo kompletní, je nutné opět rozhodnout o *stavu* pravidla, a protože je již známý i *časový typ*, je možné určit v tuto chvíli i *stav*. Globální proměnné jsou nastaveny na neplatné hodnoty a tím je pravidlo *Modify* zpracováno.

4.2.3 Implementace statistiky

Pro sledování zátěže jednotlivých *CAN FD* rozhraní, mezi kterými se přeposílají zprávy, byla implementována jednoduchá statistika. O tuto funkci se opět stará časovací funkce (*Timer* nebo *Thread*), která, kromě sledování časových pravidel, jednoduše přidává do kruhového bufferu každou milisekundu počet přijatých zpráv na každém rozhraní zvlášť. Pokud je následně požádáno o odeslání statistiky, je místo *error* registru odeslán průměrný počet přijatých zpráv každého rozhraní za sekundu. Průměrování se bere přes celý kruhový buffer.

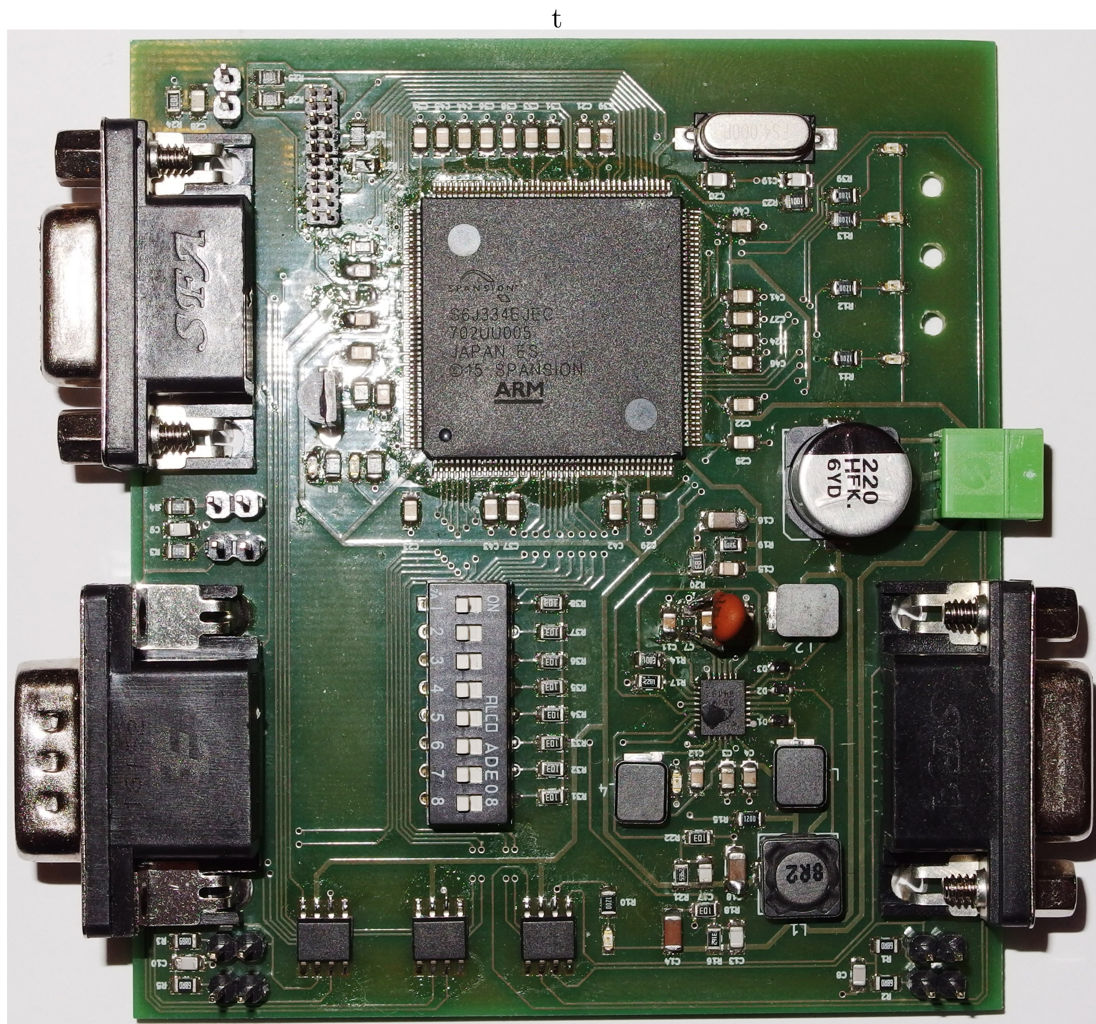


Obr. 10 Diagram funkce *FDCAN_GW_parse_rule*.

5 Hardware

Pro tuto diplomovou práci byl poskytnut hardware vyvinutý přímo pro tento projekt. Protože do posledních chvílí nebyl úplně k dispozici, hlavní část vývoje aplikace byla v simulaci a na hardware byla celá aplikace naportována. Díky velké míře abstrakce vůči *CAN* rozhraním by neměl být problém naportovat aplikaci na jakýkoli hardware.

t



Obr. 11 Poskytnutý hardware – deska s procesorem *Cypress S6J3340* a třemi *CAN* budiči a konektory.

Samotná deska obsahuje jen několik základních komponent. Hlavním srdcem systému je *mikrokontrolér*, který řídí celou aplikaci a komunikaci. Ten pro svůj život potřebuje napájení, přičemž samotný testovací systém disponuje rozvodem 12 Voltů, takže je třeba pro *mikrokontrolér* použít ještě regulátor napětí. Pro komunikaci se světem je použité

rozhraní *CAN*, to ale jako periferie *mikrokontroléru* potřebuje ještě tzv. *transceiver* – neboli převod *CAN* signálů na diferenciální signály *CAN_H* a *CAN_L*. Pro programování je ještě nutné nějaké ladící rozhraní – v tomto případě *JTAG*. Jako periferie je obsažen přímo v procesoru a stačí jednoduše vyvést signály do patřičného *debuggeru* (většinou externí přístroj komunikující s procesorem a na druhé straně s PC – většinou pomocí *USB*).

5.1 Procesor

Na desce je osazený *mikrokontrolér* firmy *Cypress semiconductors*. Jedná se o procesor určený zejména do *automotive* – automobilového průmyslu. Základem je procesorové jádro *ARM Cortex R5F*. Přesné označení je S6J334EJE, další klíčové vlastnosti jsou[6]:

- 32-bitový procesor *ARM Cortex R5F*, taktovatelný až na 240MHz.
- Paměť *Flash* o velikosti 4160kB Program + 112kB *Work Flash*.
- Paměť *RAM* o velikosti 512kB Hlavní, 16+16kB *Backup RAM*.
- Až 148 vstupně výstupních portů.
- 12-bitový *AD* převodník s až 48 kanály.
- Externí přerušení – až 24 kanálů.
- *Timer* s časovou základnou – až 32 kanálů.
- *Reload Timer a Freerun Timer* – až 6 a 8 kanálů.
- *Input capture a Output compare* jednotka – až 12 a 12 kanálů.
- Řízení krokového motoru.
- Hodiny reálného času.
- *DMA* – přímý přístup k paměti (koprocesor pro přenos dat).
- *JTAG* debugovací rozhraní.
- Podpora 2D grafiky a LCD.
- *CAN-FD* rozhraní – až 6 kanálů.
- Sériová komunikace – *UART, LIN, CSIO, I2C*.
- Systémy pro zabezpečení – *SHE, MPU, TPU, ECC, CRC, WatchDog, LVD*.

Z popisu vyplývá, že tento procesor je velmi bohatě vybavený, v podstatě by pro tuto aplikaci byl vhodnější nějaký slabší procesor, který splňuje podmínky aplikace (3 *FD-CAN* rozhraní a procesorové jádro *ARM*), jenže takových v tuto chvíli moc neexistuje (výrobci teprve začínají s implementací *FDCAN* periferie, tudíž jsou *mikrokontroléry* osazeny obvykle maximálně dvěma *FDCAN* rozhraními).

5.2 Napěťový regulátor

Pro funkci procesoru je nutný napěťový regulátor, který převádí vstupní 12voltové napětí na 5 V, ze kterých je procesor napájen. Deska je osazena *Triple Step-Down* přepínacím regulátorem firmy *Linear Technology*, přesné označení *LT3514*. Klíčové vlastnosti jsou[7]:

- Široký rozsah vstupního napští – od 3,2Voltů až do 36 Voltů.
- Tři výstupy – jeden s proudem 2 A, dva s proudem 1 A.
- Operace při až 100 % střídy.
- Odolnost vůči zkratu.
- Vypnutí při vysoké teplotě.
- Aplikace – regulace baterií v *automotive*, průmyslové řídicí jednotky, distribuovaná regulace.

5.3 Budič CAN

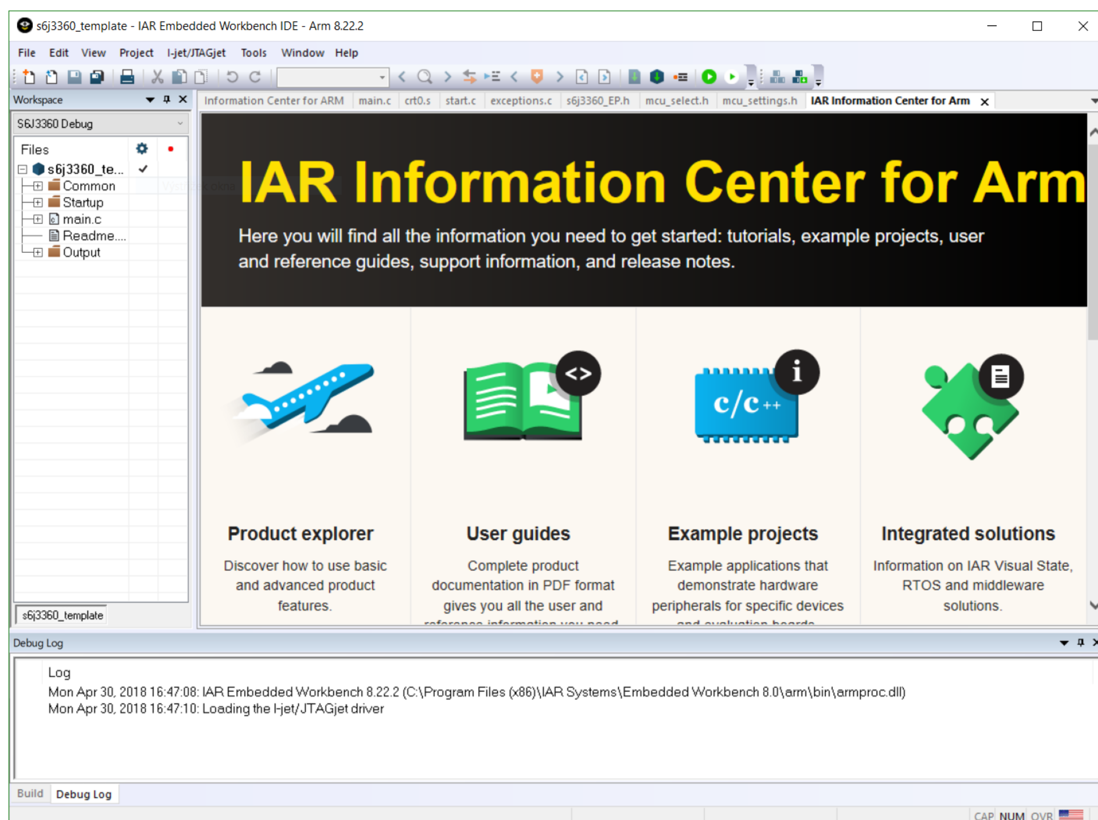
Aby mohl procesor komunikovat se zbytkem světa, a hlavně, aby mohl komunikovat s připojenými *CAN* sítěmi je nutný *transceiver* – převaděč *CAN* signálů na diferenciální signály *CAN_H* a *CAN_L*. pro tento účel je na desce osazen budič firmy *NXP – TJA1051*. Následující list opět shrnuje několik klíčových vlastností[8]:

- Plně vyhovující ISO 11898-2:2003.
- Rychlost až 5 Mbit/s v *CAN FD fast phase*.
- Vhodný pro 12 i 24 V systémy.
- Nízká elektromagnetická emise (*EME*) a vysoká elektromagnetická imunita (*EMI*).
- V*IO* pro přímé propojení s *mikrokontroléry* (3 až 5 Voltů).
- Vstup *Enable* (*EN*) pro přepnutí to režimu velmi nízké spotřeby.
- Odolnost proti *ESD* – Elektrostatickém výboji.
- Detekce nízkého napětí.

6 Implementace na hardware

Z hlediska finální implementace *CAN FD Gateway* na skutečný hardware – desku popsanou v předchozí kapitole, je třeba ještě vyřešit už jen několik věcí. Nejprve výběr prostředí pro programování, oživení samotné desky a následně aktivace jednotlivých periferií. K funkci aplikace jsou zapotřebí samozřejmě hodiny, které umožňují procesoru vykonávat svou výpočetní funkci, dále vstupně výstupní piny, aby aplikace mohla zkoumat své okolí – od otestování / signalizace pomocí *LED*, přes čtení *DIP* přepínače pro zjištění adresy až po alternativní funkci pro *CAN*. Kromě *GPIO* je nutná samozřejmě také samotná *CAN* periferie – příjem a odesílání zpráv, *Timer* pro přesné odpočítání platnosti časového pravidla a nakonec také *WatchDog* kvůli bezpečnosti – pokud by se procesor někde zasekl, nebude moci *refreshovat WatchDog*, který pak při nesplnění časového okna resetuje procesor.

6.1 Vývojové prostředí



Obr. 12 Vývojové prostředí IAR Embedded Workbench.

Protože celá aplikace je psaná v jazyce C a zároveň protože jazyk C je nejrozšířenější v oblasti *mikrokontrolérů*, není zde žádný důvod, proč s finální verzí aplikace přecházet na jiný jazyk. Procesory rodiny *Cypress Traveo* v tuto chvíli podporuje jen vývojové prostředí *IAR Embedded Workbench. IAR* je prostředí pro programování *mikrokontrolérů* zejména na jádře *ARM*, ale podporuje i jiné architektury jako *8051, AVR* a další. Pro programování si lze vybrat mezi jazykem C nebo v tomto případě *ARM* asemblerem, je ale jasné, že C se hodí mnohem více k programování a asembler spíše k ladění programu, kdy přeložený kód lze číst v tzv. zobrazení *disassembly.* Kromě firmy *Cypress IAR* také podporuje *mikrokontroléry* dalších firem jako *ST Microelectronics, NXP, Texas Instruments a další.*

6.2 Nástroje pro ladění

Aby bylo možné vůbec s deskou komunikovat a nahrávat do ní kód, je zapotřebí mít tzv. *debugger* – na straně procesoru komunikuje pomocí *JTAG*, s PC pak pomocí *USB.* K tomuto účelu byly poskytnuty *debuggery J-Link* od firmy *Segger* a *I-Jet* od firmy *IAR.* Oba ladící nástroje jsou podporovány *mikrokontroléry* z rodiny *Traveo.*

6.2.1 J-Link

Ladící nástroj *J-Link* firmy *Segger* je podporován všemi populárními *IDE* jako *IAR, Keil, Eclipse,* a další. Funguje pod všemi používanými OS – Windows, Linux, Mac. Obsahuje *JTAG* a *SWD* rozhraní pro připojení k *mikrokontroléru.* Podporuje architektury *ARM, Microchip, Renesas*[9].

Obr. 13 Ladící nástroj J-Link [10].

6.2.2 I-Jet

Stejně jako *J-Link*, je také *I-Jet* vybaven konektorem pro komunikaci a ladění pomocí *JTAG a SWD*. V podstatě *J-Link* není již tak moc rozšířený (zejména jeho starší verze s deskou vůbec nefungují) a je nahrazen spíše nástrojem *I-Jet*, přímo od firmy *IAR*. Proti *J-Linku* je *I-Jet* rychlejší v komunikaci.

Obr. 14 Ladící nástroj I-Jet [11].

Bohužel ale v rámci zmenšování používá také menší *JTAG* konektor, který ale nemá stejný pinout jako velký klasický 20-pin konektor, proto je třeba redukce. V případě desky je to o to složitější, protože na desce je konektor s malou roztečí, ale standardním *JTAG* pinoutem, takže jsou třeba dvě redukce (z malého na velký a z velkého zpátky na malý).

6.3 Oživení desky

Protože deska byla vyvíjena souběžně s touto prací, z výroby přišla až ke konci a bylo nutné ji oživit. Po jejím osazení a důkladné kontrole všech zapojení proběhly první pokusy o oživení. K dispozici byl *12 V* zdroj napětí a ladící nástroje *J-Link* a *I-Jet*, které jsou, dle manuálu [12], podporovány procesorem.

Po prvním zapojení desky na napájení a rozsvícení signalizačních *LED*, bylo načase začít s nějakým programem. Bohužel pro procesor osazený na desce (S6J334) neexistuje žádný příklad. Existuje ale pro velice podobný procesor (S6J3360), který je naopak hůře vybavený a je schopný běžet jen na nižších taktech, tudíž při startu a přechodu na vyšší takty by mělo být nastavení stejné.

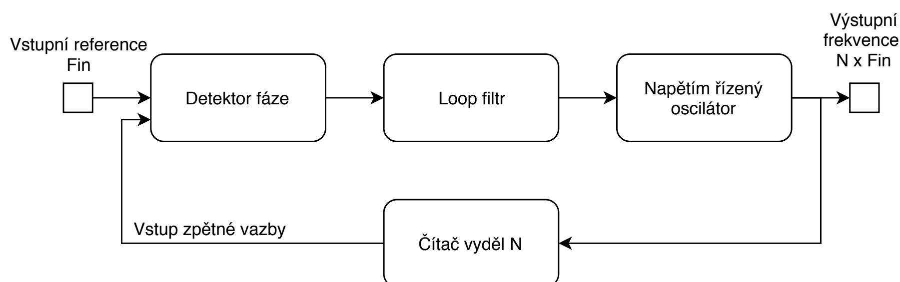
Použitý příklad lze nalézt přímo na úvodní stránce *IAR IDE* mezi vzorovými projekty – vybere se příslušný výrobce *mikrokontroléru* a nechá se stáhnout celá knihovna příkladů. Zde pak pro procesor S6J3360 existuje šablona, která inicializuje hodiny, paměti a další základní funkce.

Prvotní nahrání projektu mělo také své problémy. Nástroj *I-Jet* se přes *IAR* vůbec nepřipojí, tvrdí, že pin *TDO JTAGu* je stále v logické 0, ale přes konfigurační utilitu instalovanou společně s ovladači procesor na *JTAGu* vidí a s připojením není problém. *J-Link* byl schopný se připojit a nahrát program, ale běh programu byl nahodilý a *debugger* se odpojoval. Trvalo několik dní zjistit, že *Flash loader* (program nahrávající binární data do paměti *Flash*) určený přímo pro procesor neumí smazat paměť, resp. odpojuje se, ale nativní *Flash loader* mazat umí, ale neumí nahrávat program. Tudíž ve výsledku při použití *Flash loaderu* pro procesor se data nahrávají přes nesmazanou paměť *Flash* a vznikají nesmyslné instrukce.

Správné nastavení pro ladění programu tedy je použít nativní *Flash loader* pro smazání a dedikovaný pro nahrání programu, pak je možné program nahrát i krokovat.

6.3.1 Nastavení hodin

Projekt z knihovny příkladů pro *IAR* už sám umí nastavit hodiny. Základem je přepnutí z *4 MHz* vnitřních hodin na externí krystal, taktéž na *4 MHz*. Samozřejmě by bylo možné celé nastavení vynechat (jak bylo i při testování uděláno, aby se program dostal do hlavní programové smyčky funkce *main*), ale protože vnitřní oscilátor není tak přesný, tak je lepší přepnout na externí a případně pomocí fázového závěsu (*PLL*) zvýšit frekvenci. Zároveň periferie *CAN* má zdroj hodin právě *PLL*.



Obr. 15 Fázový závěs *PLL*.

Zde ale opět nastával problém, už při krokování programu se v jedné části *debugger* náhle odpojil. Důvod pro to byl velice jednoduchý, ale najít ho chvíli trvalo. V šabloně je vykonávána také část pro nastavení tzv. *Expand PLL*, která ale jak bylo zjištěno z poznámky pod čarou v manuálu [13], není podporována procesorem *S6J334*. Po odstranění této části již procesor bez problémů prošel nastavením hodin a nastavil *80 MHz* pomocí *PLL*.

Skrze makra lze samozřejmě také změnit nastavení hodin na vyšší či nižší frekvenci a to při použití *PLL* změnou jen 3 parametrů (hodinový takt je násoben nebo dělen parametry *PLLN/M/L*). V tuto chvíli i v budoucnu by ale takt *80 MHz* měl stačit na veškeré úkony, které musí procesor zvládnout.

6.3.2 Nastavení pamětí

Vzorový program také nastavuje tzv. *Wait state* paměti *Flash* – kolik cyklů má procesor počkat, aby dostal data z *Flash*. Toto nastavení je použitelné tak jak je v šabloně, ale protože do těchto registrů se smí zapsat pouze jednou a *debugger* toto nastavení také provádí, je nutné při ladění tuto část kódu zakomentovat, nebo ošetřit preprocesorovým makrem. Ve finální aplikaci pak musí být nastavení pamětí aktivní.

6.4 Nastavení periferií aplikace

Po tom, kdy se program konečně dostal až do funkce *main*, bylo načase implementovat použití periferií nutných pro aplikaci *FD CAN Gateway*. Od blikání signalizační *LED*, nebo čtení *DIP* přepínače, přes časovač až k periferii *CAN*.

6.4.1 Vstupně výstupní port – GPIO

K nastavení vstupně výstupního portu, je třeba určit jen několik věcí. Jaký typ *GPIO* to bude – analogový či digitální vstup nebo výstup, použití *pull-up* či *pull-down* rezistorů a také zdroj či příjemce dat – datový registr *GPIO* nebo nějaká alternativní funkce (periferie). Všechna tato nastavení lze najít v manuálu[14], společně s procedurou jak správně pin nastavit.

Pro jednoduchou signalizační *LED* je třeba nejprve najít její umístění. Dle schématu desky jsou uživatelské *LED* umístěné na pinech 55-58 – to jsou ale čísla pouzdra, proto je nutné zjistit číslo přímo pinu procesoru. Piny 55-58 odpovídají dle manuálu[15] pinům *P0_27-30*.

Jakmile je známo kde je *LED* zapojena, je třeba nastavit pin jako výstup, bez jakýchkoli *pull* rezistorů a jako zdroj dat použít výchozí hodnotu – registr *GPIO*. Protože procesor *S6J334* je určený zejména pro *automotive*, jsou zde implementovány ochrany proti náhodnému přepisu a změně funkce. Proto je nutné nejprve registr pro nastavení pinu odemknout klíčovým registrem, aby byl umožněn zápis.

Pro odemčení registru je nutné 4x po sobě zapsat do klíčového registru(*GPIO_KEYCDR*) následující hodnoty, jinak dojde k *Bus erroru*:

- Bity 31-30 – Posloupnost 0,1,2,3.
- Bity 29-28 – Způsob přístupu (zápis *byte, half word, word*).
- Bity 14-0 – Offsetová adresa pinu – lze dohledat z tabulky v manuálu[15].

Jakmile je registr odemčen, lze nastavit příslušný pin jako výstup (registr *PPC_PCFGR*):

- Bit 15 – Povolení výstupu.
- Bit 14 – Počáteční výstupní hodnota.
- Bity 2-0 – Zdroj dat – 0 pro *GPIO* registr.

Nakonec je ještě třeba určit směr výstupu, což se určí registrem *GPIO_DDSR*, který je také nutné odemknout pomocí registru *GPIO_KEYCDR*, ve stejném formátu, ale s jiným offsetem. Následně je pak možné zapsat na příslušný bit registru pro nastavení směru na výstup (bit se určí jako pozice na bráně, v případě *LED* na *P0_27* je nastaven bit 27)[14].

6.4.2 Knihovna PDL

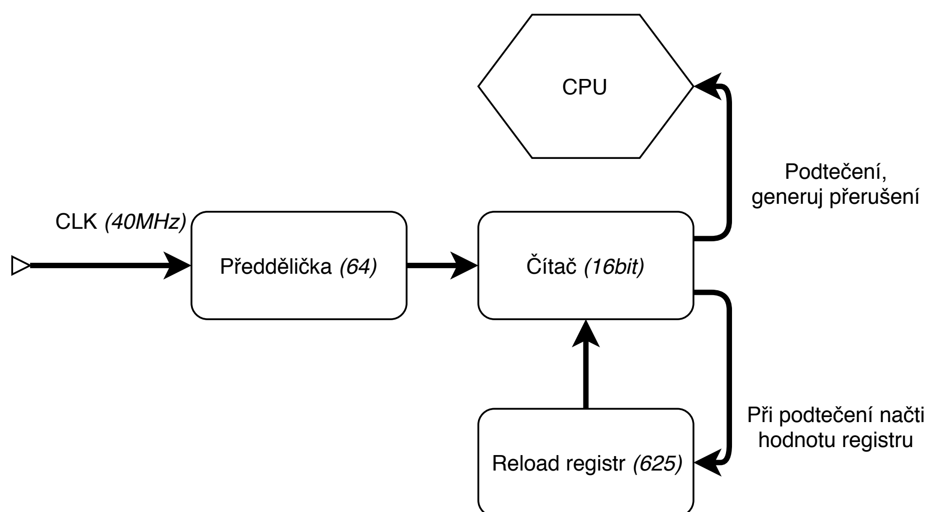
Pro menší ulehčení práce s procesorem, byla použita knihovna *PDL* (*Peripheral Driver Library*). Původně ani nebylo v plánu ji použít, protože sama o sobě dle dokumentace není ani určena pro jádro R5, které na procesoru je. Protože ale v repositáři příkladů *IAR* existuje několik projektů pro procesory řady *S6J3200* s použitím právě knihovny *PDL*, byla vyzkoušena funkčnost knihovny nejprve na obyčejné *LED*, přičemž knihovní funkce bez problémů fungovaly.

Jak je ale popsáno dále, knihovna (zejména pro rozhraní *CAN*) není úplně abstrahovaná od hardwaru, je to spíše několik ukázkových funkcí, jak s periferiemi pracovat, ale nikoliv universální nástroj. Proto bylo třeba mnoho funkcí upravit a také zobecnit, aby bylo možné snadno nastavit všechna rozhraní *CAN* najednou.

6.4.3 Časovač s přerušením

Pro funkci časových pravidel aplikace *CAN Gateway* je samozřejmě nutná implementace funkce reálného času. Minimální jednotkou je milisekunda, proto není nic snadnějšího, než vybrat jednu z *Timer* periferií a generovat přerušení tak, aby je procesor dostával každou *ms*.

Pro správné nastavení časovače je nutné modifikovat jen několik registrů. Pro přerušovací funkci byl vybrán časovač označený jako *Base Timer*(v procesoru značený jako *Base Timer 2_1*) v režimu *Reload Timer*. Pokud je do periferie časovače přiveden hodinový takt *40MHz* (s předděličkou), rozsah čítače je 16 bitů, je nutné nastavit předděličku časovače tak, aby čítací registr nepřetekl. Tj. při *40MHz*, vyděleno 64 dá frekvenci *625kHz*, tudíž pro milisekundu (*1kHz*) je nutné nastavit resetovací hodnotu na 625. Samotný čítač pak pracuje tak, že s frekvencí danou předděličkou (625kHz) odečítá postupně čítací registr a pak při podtečení se nastaví hodnota z resetovacího registru (625). Tato událost se dá z periferie zachytit tak, že se povolí přerušení do procesoru.



Obr. 16 Princip časovače. Hodnoty v závorce znamenají použité nastavení.

Samotné nastavení čítače se dá provést samozřejmě ručně (což bylo také použito, ale v další fázi zakomentováno), nebo pomocí knihovny. Pomocí struktury *stc_rlt_config_t* definované v knihovně pak lze jednoduše nastavit všechny parametry časovače a zavoláním funkcí *Bt_Rlt_Init* a *Bt_Rlt_Start* vše nastavit a časovač zapnout. Nastavení přerušení se pak děje uvnitř těchto funkcí, kde se kromě samotných přerušovacích linek a masek nastavuje také obsluha přerušení, která si také z struktury vezme ukazatel na funkci, která se má zavolat, při dané události. V tomto případě se tedy při podtečení zavolá funkce *RLTUnderflowCallback*.

Zde se ale objevil jeden z několika rozdílů mezi procesory, resp. jeden z rozdílů v definicích daných soubory *interrupts.c* pro procesor *S6J3200* a použitý procesor *S6J3340* resp. z šablony *S6J3360*. Šlo o to, že řada *3360* neměla definováno tolik přerušovacích registrů (ale řada *3200* i *3340* měla definice přibližně stejné), takže při prvních pokusech nebylo možné z časovače dostat přerušení, i když běžel a daná událost nastávala. Řešení bylo nakonec velmi snadné, vyměnit soubory *interrupts.h a interrupts.c* za soubory z příkladu pro *S6J3200*. Je nutné ale při dalším vývoji dávat velký pozor, zda se některé definice neliší, pro účely časovače jsou stejné.

6.4.4 Rozhraní CAN

Z hlediska portování simulované aplikace na reálný hardware byla nejtěžší část právě při implementaci rozhraní *CAN*. Nejen, že byly opět problémy s kompatibilitou příkladů a procesorů, ale také byla objevena chyba přímo na desce. Deska byla vytvořena v rámci jiné – bakalářské práce a k dispozici byla jen její první verze, která měla prohozené *D-Sub*(9 pinů) konektory pro *CAN* – tam kde měl být zásuvka, byla zástrčka (*female, male* konektor) a naopak.

Pro správnou funkci rozhraní *CAN* je třeba zapnout budiče převádějící logické hodnoty z procesorových pinů na diferenciální pár *CAN_H* a *CAN_L* (viz. obrázek mapování pinů 9). Díky knihovně se toto nastavení provede snadno, stačí aktivovat piny *P0_00, P0_02, P0_04* – *Enable* (povolení) všech tří budičů, když je nastavena hodnota log. 1. A také piny *P0_01, P0_03, P0_05*, které jsou připojeny na pin *Silent* budičů – log. 1 znamená pouhé odposlouchávání sběrnice, ale nikoliv aktivní přístup, proto se tyto piny musí nastavit do log. 0, aby *CAN* jednotky komunikovaly aktivně na sběrnici – účastnily se komunikace. Z hlediska implementace tedy opět stačí naplnit strukturu pro aktivaci *GPIO* (*stc_port_pin_config_t*) a zavolat funkci *Port_SetPinConfig*.

CAN	Tx(package, procesor)	Rx(package, procesor)	EN(package, procesor)	S(package, procesor)
01/0	71, P3_10	70, P3_09	2, P0_00	3, P0_01
31/3	99, P3_17	98, P3_16	4, P0_02	5, P0_03
51/5	107, P3_20	106, P3_19	6, P0_04	7, P0_05

Tab. 9 Mapa rozložení pinů mezi budiči a *CAN* rozhraními.

Když jsou budiče aktivní, stačí už jen aktivovat samotnou periferii *CAN* uvnitř *mikrokontroléru*. Samozřejmě je také nutné kromě výstupních pinů k budičům aktivovat také vstup a výstup periferie *CAN*. Na desce jsou použity *CAN* rozhraní s kanály 0, 3, 5. To z hlediska rozložení desky odpovídá pinům Tx/Rx – *P3_10/P3_09, P3_17/P3_16, P3_20/P3_19* – viz. tabulka 9 rozložení pinů.

Nyní jsou sice *GPIO* nastavena, ale protože všechna *CAN* rozhraní jsou dle značení ve skupině 1 a ne nula, musí se ještě nastavit tzv. *resource* registry, které propojí *CAN* s *GPIO*. Z hlediska výstupu se jen jednoduše nastaví při konfiguraci *GPIO* bity *POF* na hodnotu 7 (nebo uvnitř konfigurační struktury prvek *enOutputFunction* na *PortOutputResourceH*) – což odpovídá dle manuálu[14] nastavení pro směrování výstupu *CAN* periferie na *GPIO*. Z hlediska vstupu je to o něco málo složitější, protože to nelze nastavit přímo ve struktuře, ale skrze *resource* registry. S nahlédnutím do manuálu[15], který registr se má nastavit a pomocí funkcí *Port_SelectInputPort* a *Port_SelectInputSource* je to stále velmi jednoduchá operace. V tomto případě je nutné nastavit registry *RESIN133,136,139* na *PortInputPortB* a *PortInputSourceB*, což propojí vstup periferie *CAN* s *GPIO*.

Po nastavení všech *GPIO* se nastaví konečně samotný *CAN*. Přímo pro *FD CAN* vydal *Cypress* také aplikační manuál[16], který obsahuje i ukázky kódu pracující přímo s registry (příklad z knihovny je také velice podobný, ale přes definované struktury). Manuál také obsahuje trochu specifičtější popis než velký referenční manuál[14] pro všechny periferie.

Pro zprovoznění periferie *CAN*, stejně jako u časovače je nutné znát vstupní frekvenci do periferie – což je *40 MHz*. Pomocí funkce *Can_PrescalerInit* se nastavuje předdělička před periferií *CAN*, ale dle manuálu[14] pro vyšší rychlosti komunikace je doporučeno nechat *40 MHz* – také všechny dostupné příklady (aplikační dokument i příklad pro *S6J3200*) pracují s touto frekvencí, tudíž předdělička byla nastavena na hodnotu 1 (ve volání funkce 0). Pro dané požadavky na rychlost – standardní rychlost *500 kBit/s*, rychlost při *FD Bit rate switch 2 Mbit/s* – je třeba správně nastavit vnitřní předděličku *CAN* rozhraní, oba *Time segmenty* a synchronizační skok. Také je dobré dát si pozor na to, že všechna nastavení uvnitř struktury/registrů jsou dále použita jako hodnota *+1*. Takže pokud se nastaví předdělička na 4 pro standardní rychlost, je výstupem *10 MHz*, neboli *100 ns*. Pokud se touto hodnotou vydělí rychlost *500 kBit/s* neboli $2\mu s$, je třeba 20 časových kvant pro dosažení žádané rychlosti. Protože by *Sample point* měl být kolem *80*%, je nastaven *TSeg1* na 15 a *TSeg2* na 4. To dá se synchronizačním segmentem dohromady 20 časových kvant. Pro *FD BRS* rychlost se použije dělení 1, tj. vstupem je nezměněných *40 MHz*, neboli *25 ns*, takže pro rychlost *2 Mbit/s*, neboli *500ns* je třeba použít opět 20 časových kvant. Takže nastavení *TSeg1* a *TSeg2* je stejné jako u standardní rychlosti. Synchronizační skok je u obou rychlostí nastaven na 4. Následující tabulka shrnuje nastavení periferie *CAN*.

	Nominální rychlost *500kBit/s*	Datová FD BRS rychlost *2Mbit/s*
Prescaler	4 (3)	1(0)
TSeg1	15(14)	15(14)
TSeg2	4(3)	4(3)
SJW	4(3)	4(3)

Tab. 10 Tabulka nastavení *CAN* periferie pro standardní a *FD BRS* rychlost komunikace.

Výše uvedená nastavení shrnutá v tabulce 10 byla vložena do struktury a následně voláním funkce *CanFD_Init* byla aktivována periferie. Aby to nebylo tak jednoduché, tak v tuto chvíli bylo možné přijímat (potvrzením) i vysílat standardní *CAN* rámce ale opět kvůli rozdílu mezi procesory, nebyl aktivní *FDF* a *BRS* (*FD frame* a *Bit rate switch*), protože procesor z řady *3200* měl trochu odlišné definice registrů (dvěma bity registru *CCCR* nastavoval obě funkce *BRS,FDF*, ale procesor desky je má oddělené), tudíž byla provedena malá modifikace a kontrola všech potřebných registrů, zda sedí s daným procesorem. Po nastavení *BRS, FDF* bitů bylo již možné *FD* rámce přijmout, ale nebylo je možné odeslat. Také *BRS* hlásil mnoho chybových rámců a nebyl funkční.

Komunikace ve zvýšené *BRS* rychlosti dělala problémy, protože příklad nastavoval také tzv. *Transmitter delay compensation* – kompenzace zpoždění vysílacího uzlu, zavádí druhotný bod vzorkování (*SSP*) posunutý o offset od standardního *SP*. Čítačem se měří doba od spádové hrany *FDF* bitu do *res* bitu, přičemž je měřen čas mezi *FDF* spádovou hranou na *Tx* pinu a bodem, kdy se spádová hrana objeví i na *Rx* pinu. K této hodnotě je také přičítán registr *TDCO*, který přidává kompenzační čas daný vlastnostmi budiče (tuto hodnotu lze najít v datasheetu budiče). Nejjednodušší pro zprovoznění *BRS* rychlosti bylo tuto funkci vypnout, což také funguje – je ale nutné si v budoucnu dát velký pozor, pokud budou vznikat chyby ve vyšší rychlosti komunikace, je možné, že problém bude právě v kompenzaci vysílače a bude nutné tuto funkci opět zapnout. V tuto chvíli nebude uvažována. Díky tomu, bylo nyní možné přijímat také rámce s *FD BRS*.

Pro odeslání *FD* rámců bylo zapotřebí další modifikace, periferie sice měla povolena tato nastavení, ale to ještě neznamená, že každá zpráva tak musí být vyslána. Opět byl problém v drobném rozdílu mezi procesory, kdy v knihovním příkladu *FDF* a *BRS* bity pro odeslání zprávy nebyly vůbec uvažovány. Tudíž byla modifikována i samotná struktura pro odeslání zprávy tak, aby bylo možné měnit typy rámců (standardní, *FD*, *FD* s *BRS*). Reálně se pak zapsalo do bitů *FDF,BRS* registru *T1* každé zprávy.

Nyní tedy bylo možné odesílat, ale přijímat pouze potvrzením (rozhraní *CAN* potvrdilo rámec, tudíž nenastala žádná chyba). Příklad byl opět až příliš specificky navržen a zaváděl hardwarový filtr, který pouze rámce s určitým identifikátorem zprávy uložil do dedikovaného bufferu, ale ostatní zahodil. Pro aplikaci *FD CAN Gateway* je ale nutné přijímat všechny zprávy, tudíž tento filtr byl odstraněn a byl zaveden filtr přijmi vše a ulož do *Rx FIFO*. Proti dedikovanému bufferu má *FIFO* tu výhodu, že je schopné uložit několik zpráv (až 64 *FD CAN* zpráv) najednou, proti tomu buffer pouze jednu a je nutné ho vyprázdnit před dalším přijetím (používá se zejména pro ukládání speciálních zpráv, které nechodí tak často). Pro aplikaci je také *FIFO* mnohem vhodnější, neboť při zpracování zpráv v hlavní smyčce (bez přerušení) trvá různě zprávy zpracovat (například zablokovaná zpráva vs zpráva s pravidlem pro modifikaci a výpočet *CRC*), tudíž je nutné zprávy ukládat, aby se nějaká neztratila. Zde tedy tento problém vyřešil hardware.

7 Testování

7.1 Finální struktura projektu

Pro snadnější orientaci v projektu shrnuje tabulka 11 celou strukturu zdrojových kódů. Jsou vypsány jen ty nejdůležitější, nejsou zde vypsány například soubory z šablony pro portování aplikace na procesor, protože důležité jsou jen ty, které mají velkou souvislost s aplikací *FD CAN Gateway*.

Složka	Soubor(/složka)	Popis
FDCAN_GW_sources		Složka zdrojových kódů – hlavní část aplikace.
	fdcan_gateway.h	Hlavičkový soubor deklarací a maker pro aplikaci *FD CAN Gateway*, nezávislé na HW ani na emulátoru
	fdcan_gateway.c	Hlavní část aplikace, zpracování zpráv a pravidel.
	fdcan.h	Definice struktur a deklarace funkcí pro příjem, odesílání zpráv, inicializace rozhraní FD CAN.
FDCAN_GW_Emulator		Složka emulátoru pro PC.
	fdcan_emulated.c	Aplikace definic z *fdcan.h*, definice funkcí pro příjem odesílání zpráv, inicializace rozhraní *FD CAN* z hlediska emulátoru. Fyzicky načítání předdefinovaných zpráv.
	FDCAN_GW_Emulator.h	Globální definice proměnných pro práci s emulátorem – deklarace předdefinovaných zpráv.
	FDCAN_GW_Emulator.c	Hlavní část emulátoru – od statického naplnění deklarovaných zpráv, po zpracování vstupu uživatele a řízení aplikace *FD CAN Gateway*. Implementuje *fdcan.h*.
	kvaser.h	Deklarace funkcí pro práci s přípravkem Kvaser.
	kvaser.c	Definice funkcí pro práci s přípravkem *Kvaser*, příjem a odesílání dat, inicializace.
	/tests	Složka s testy, které je možné vložit na vstup emulátoru.
	⋮	⋮

Složka	Soubor(/složka)	Popis
FDCAN_GW_Cypress		Složka souborů pro práci s procesorem S6J3340 – od inicializace po komunikaci pomocí rozhraní FD CAN
	main.c	Hlavní programová smyčka na HW, volá hlavní proces aplikace.
	s6j3340_lib.h	Deklarace funkcí pro práci s procesorem *S6J3340*, inicializace *LED, Timeru*, rozhraní *FD CAN* atd.
	s6j3340_lib.c	Definice funkcí pro práci s HW, implementuje s6j3340_lib.h.
	fdcan_s6j3340.c	Definice funkcí pro příjem, odesílání zpráv, inicializace rozhraní *FD CAN*. Implementuje *fdcan.h*.
	/pdl	Složka knihovny *PDL – Peripheral driver library*
	/common	Šablonové soubory pro práci s procesorem – inicializace samotného procesoru.

Tab. 11 Finální struktura projektu, jednotlivé zdrojové soubory s krátkým popisem.

7.2 Testování emulátoru

Aby mohla být simulovaná aplikace *CAN Gateway* dobře otestována, je nutné nejdříve mít nějaké rozhraní, kam lze zadávat data a získávat výsledky. V ideálním případě je dobré mít testy automatizované, protože to usnadňuje kontrolu správnosti, než manuální kontrola každého parametru. Bohužel, příkazy a vzniklá pravidla uvnitř aplikace mají velmi mnoho parametrů, proto byl automatizován pouze vstup, ověření správnosti je již manuální. Existují ale metody, jak snížit počet vstupních dat tak, aby testovaná aplikace byla stále dobře otestovaná.

```
Hello
FD CAN GW Emulator ready, enter one of the following cmds:
******************************************************************************
Trigger (t), Reset (r), Modify (m), Block/Pass (b),
Send msg from CAN 1 to 2 (s), Msg from 2 to 1 (d), Exit (e)
******************************************************************************
b
Enter Block pass parameters in format:
ID TYPE LENGTH11 0 0
CMD Blockpass sent


----------------------------------------------------------------------------------
FDCAN receiving message with ID 33
10 00 00 00 0B 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
----------------------------------------------------------------------------------


.....................................................................................
FDCAN sending message with ID 1
Data: 00 00 00 00 00 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
.....................................................................................
FD CAN GW Emulator ready, enter one of the following cmds:
******************************************************************************
Trigger (t), Reset (r), Modify (m), Block/Pass (b),
Send msg from CAN 1 to 2 (s), Msg from 2 to 1 (d), Exit (e)
******************************************************************************
```

Obr. 17 Ukázka konzolové aplikace.

Pro zadání příkazů byla vytvořena jednoduchá konzolová aplikace. Její ovládání je velmi snadné, reaguje na vstup z klávesnice, každý příkaz je aktivován jedinou klávesou, ale uživatel může být dále požádán o další parametry (jako je identifikátor, typ platnosti příkazu atd.). Rozsáhlé datové parametry (jako např. u příkazu *Modify*) jsou předdefinovány přímo uvnitř kódu, protože zadávání více jak 100 parametrů by bylo pro uživatele časově náročné. Následující tabulka (12) popisuje jednotlivé klávesové příkazy a jejich parametry.

Klávesa	Příkaz	Další žádané parametry	Popis
T	*Trig*	–	Aktivací příkazu *Trig* se nastaví nová konfigurace.
R	*Reset*	Typ *resetu* (0 – propouštěcí, 1 – blokující)	Příkaz *Reset* maže novou konfiguraci a nastavuje nové výchozí pravidlo.
M	*Modify*	Identifikátor, typ a délka platnosti	Příkaz *Modify* určuje pravidlo pro modifikaci rámců s daným *ID*, trvající *Length* rámců nebo časových jednotek dané parametrem *Type*.
B	*Block/Pass*	Identifikátor, typ a délka platnosti	Příkaz *Block/Pass* určuje pravidlo pro zablokování nebo naopak přeposlání (opak výchozího pravidla) rámců s daným *ID*, trvající *Length* rámců nebo časových jednotek dané parametrem *Type*.
S	*Send1to2*	Identifikátor zprávy	Použitím tohoto příkazu se vnitřně pošle rámec z *CAN* rozhraní 1 na rozhraní 2, aktivní pravidla rozhodují, jak se zpráva zpracuje.
D	*Send2to1*	Identifikátor zprávy	Použitím tohoto příkazu se vnitřně pošle rámec z *CAN* rozhraní 2 na rozhraní 1, aktivní pravidla rozhodují, jak se zpráva zpracuje.
E	*Exit*	–	Ukončení aplikace.
F	*File*	Název souboru	Načítá soubor, který je použit místo vstupu klávesnice, umožňuje automatizaci.

Tab. 12 Ovládání konzolové aplikace.

Jak ukazuje tabulka výše (12), testovací aplikace také umí načítat vstup ze souboru. Vstupní soubor musí být umístěný ve složce projektu (nebo v podsložce – potom se soubor načte následovně: *podslozka/nazev.txt*) a jednotlivé příkazy musí být odděleny novým řádkem nebo mezerou a jejich parametry striktně mezerami.

7.2.1 Metody testování

Pro úplné otestování je samozřejmě možné vygenerovat všechny kombinace vstupních parametrů, v tomto případě všech parametrů pravidel. Takové testování bývá ale celkově velmi náročné – kombinací je mnoho, zabírá paměť, testování může být zdlouhavé, než se provedou všechny kombinace a nakonec manuální kontrola každého *test case* je pro člověka velmi náročná, aby nepřehlédl žádnou chybu.

Existuje mnoho technik testování, zmíněné základní a nejvíce náročné, ale nejvíce pokrývající testování, při použití všech kombinací se nazývá *MCC – Multiple Condition Coverage*. Další technika se nazývá *MC/DC – Modified Condition/Decision Coverage*, zde se testují všechny kombinace, které mají vliv na výsledek rozhodovacího výrazu, tj. každý z výsledků všech podmínek v logickém výrazu je otestován alespoň jednou a zároveň každá z podmínek nezávisle ovlivnila výsledek logického výrazu. Další technikou je *Pairwise testing*, v každém páru vstupů jsou pokryty všechny kombinace. Další techniky jako *C/DC, CC, DC* už mají jen malé pokrytí testů, proto se používají spíše pro nekritické části aplikace – fungují na principu splnění/nesplnění celé podmínky (*DC – Decision Coverage*), nebo části logického výrazu (*CC – Condition Coverage*)[17].

Pro otestování aplikace byla vybrána metoda *Pairwise testing*, protože proti *MCC* není tak náročná, ale zároveň pokryje ještě dostatek případů k otestování celé aplikace. K vygenerování párů byl použit program *Allpairs* [18], který je na Internetu volně ke stažení. Existují také komerční a také lepší aplikace, ale k tomuto testování výborně poslouží *Allpairs*.

Pro *Pairwise testing* je nejdříve třeba stanovit tzv. třídy ekvivalence – systém se má podle specifikace (na základě které se testuje) pro všechny hodnoty z konkrétní třídy ekvivalence chovat stejným způsobem. Dá se, podle typu vstupu, definovat intervalem (např. částka, věk apod.) nebo diskrétními hodnotami (typ prohlížeče, metoda platby, položka v menu)[17].

Reset
Typ
Propouštěcí (0)
Blokující(1)

Tab. 13 Příkaz *Reset* a jeho třídy ekvivalence.

Block/Pass		
ID	Typ	Trvání
Standardní	Stálé	0
Rozšířený	Časové	uint16
Existující	Počet	

Tab. 14 Příkaz *Block/Pass* a jeho třídy ekvivalence.

Pro aplikaci *CAN FD Gateway* jsou vstupem zprávy a příkazy, zde je tedy možné určit třídy ekvivalence. U obecné zprávy záleží jen na identifikátoru, který může být standardní, rozšířený a z hlediska aplikace může spustit aplikaci nějakého pravidla. Parametry jako data a délka jsou již jednou třídou ekvivalence, není zde žádný rozdíl, jestli bude posláno 8 bajtů, nebo 64, stejně tak není rozdílný dopad v tom, jaká data jsou odesílána. Z hlediska příkazů – pravidel je již dopad jiný. Příkaz *Trig* nemá žádné parametry, ale je třeba ho otestovat, příkaz *Reset* má jediný parametr a to je typ resetu, tj. 2 možnosti k otestování. Zajímavější příkazy k otestování jsou pak *Block/Pass* a *Modify*. Oba mají parametry identifikátor, typ (neomezený, časově omezený, omezený počtem rámců) a délku trvání (nulová délka, jakékoli 16 bitové číslo kromě 0). U identifikátoru lze stanovit třídy ekvivalence jako identifikátory typu standardní, rozšířený nebo již zahrnutý v nějakém pravidle, protože každá třída otestuje jinou část aplikace. Stejně tak typ se třemi třídami má pokaždé jiný dopad na aplikaci. Nakonec příkaz *Modify* má ještě více parametrů, které by měly být otestovány. Data a maska jsou opět parametry, u kterých nezáleží, jaké budou, resp. jejich dopad je vždy stejný, tj. dají se shrnout do jedné třídy ekvivalence. Proti tomu parametry *DLC* a *CRC* mají opět různý dopad na aplikaci v závislosti na hodnotě parametru. *DLC* by se dalo zařadit následovně – zkracující, nezkracující, prodlužující délku zprávy. *CRC* – nepočítat, správně či chybně spočítat. Celkem tedy pro příkaz *Block/Pass* existuje jen 18 možností (což je ještě dobře manuálně proveditelné), ale příkaz *Modify* má již možností 162. Při zadání všech těchto parametrů a jejich tříd ekvivalence do programu *Allpairs*, je třeba otestovat pouze 9 možností pro *Block/Pass* a 11 pro *Modify*. Jednotlivé třídy ekvivalence a jednotlivé *test case* popisují následující tabulky (znak ~znamená, že na daném parametru nezáleží, protože byl již dříve otestován).

Modify				
ID	Typ	Trvání	DLC	CRC
Standardní	Stálé	0	Zkracující	Nepočítat
Rozšířený	Časové	uint16	Prodlužující	Správně spočítat
Existující	Počet		Stejná délka	Chybně spočítat

Tab. 15 Příkaz *Modify* a jeho třídy ekvivalence.

TEST CASES – Block/Pass			
case	ID	Typ	Trvání
1	Standardní	Stálé	0
2	Standardní	Časové	uint16
3	Rozšířený	Stálé	uint16
4	Rozšířený	Časové	0
5	Existující	Počet	0
6	Existující	Stálé	uint16
7	Standardní	Počet	uint16
8	Rozšířený	Počet	~0
9	Existující	Časové	~0

Tab. 16 Příkaz *Block/Pass* – vygenerované testy programem *Allpairs*.

TEST CASES – Modify					
case	ID	Typ	Trvání	DLC	CRC
1	Standardní	Stálé	0	Zkracující	Nepočítat
2	Standardní	Časové	uint16	Prodlužující	Správně spočítat
3	Rozšířený	Stálé	0	Prodlužující	Chybně spočítat
4	Rozšířený	Časové	uint16	Zkracující	Nepočítat
5	Existující	Stálé	uint16	Stejná délka	Správně spočítat
6	Existující	Časové	0	Stejná délka	Chybně spočítat
7	Standardní	Počet	uint16	Stejná délka	Chybně spočítat
8	Rozšířený	Počet	0	Zkracující	Správně spočítat
9	Existující	Počet	∼0	Prodlužující	Nepočítat
10	Rozšířený	∼Stálé	∼uint16	Stejná délka	Nepočítat
11	Existující	∼Časové	∼uint16	Zkracující	Chybně spočítat

Tab. 17 Příkaz *Modify* – vygenerované testy programem *Allpairs.*

7.2.2 Výsledky testování emulátoru

Aby byly výsledky testování snadno ověřitelné, bylo vytvořeno několik testovacích souborů, které se, jak bylo popsáno výše, jednoduše vloží jako vstup emulátoru.

Sekvence testů		
Reset	Block/Pass	Modify
s 1	b 1 0 0	m 1 0 0 5 0
r 0	b 2 2 2	m 2 2 2 15 1
t	b 4096 0 2	m 4096 0 0 15 2
s 1	b 4097 2 0	m 4097 2 2 5 0
r 1	b 1 1 0	m 1 0 2 16 1
t	b 2 0 2	m 4096 2 0 16 2
s 1	b 3 1 2	m 3 1 2 5 1
	b 4098 1 2	m 4098 1 0 5 1
	b 4098 2 0	m 4098 1 0 15 0
	t	m 4099 0 2 16 0
	s 1	m 4099 2 2 5 2
	s 1	t
	s 3	s 1
	s 3	s 2
	s 3	s 4096
	s 2	s 4097
	s 4096	d 3
	s 4097	d 3
	s 4098	d 3
	s 4098	d 4098
	s 4098	d 4099

Tab. 18 Testovací sekvence příkazů *Reset, Block/Pass, Modify.*

Tabulka 18 ukazuje sekvenci testování jednotlivých příkazů. V případě *Reset* je nejprve vyslán rámec bez jakýchkoli pravidel, následuje propouštěcí *Reset* a jeho aktivace příkazem *Trig*. Opět je odeslán rámec, který je bez problémů přeposlán na druhé *CAN* rozhraní. Testuje se také blokující *Reset*, opět potvrzený a vyzkoušený vysláním rámce. Tentokrát se data na druhé rozhraní nedostanou, protože zpráva byla zablokována. Pro ověření si musí čtenář spustit aplikaci, neboť je výstup moc dlouhý.

Pro příkaz *Block/Pass* je opět vytvořena testovací sekvence dle tabulky(14) vygenerované programem *Allpairs*. Jsou tedy postupně zadána jednotlivá pravidla s jejich parametry a následně aktivována příkazem *Trig*. Pokud je parametr *ID* v *test case* jako existující, aplikace na to zareaguje a dané pravidlo není zařazeno do seznamu, protože takové pravidlo s daným *ID* již existuje. Tudíž se v tomto případě vytvoří 6 pravidel, která se stanou aktivní. Následuje blok odesílání rámců, kde každé *ID* je testováno alespoň jednou, a pravidla s platností na určitý počet rámců jsou testována počtem o jedna vyšší, aby se dokázalo, že pravidlo úspěšně skončilo svou platnost. Protože konzole také vypisuje, o jaký typ a délku platnosti pravidla jde, lze také ověřit, že žádné pravidlo s *ID Existující* nebylo zahrnuto do systému. Spuštěním aplikace lze opět ověřit celý test.

Posledním testovaným příkazem je *Modify*. Stejně jako u *Block/Pass* a *Reset*, sekvence je vytvořena dle *test case* tabulky 15. Zde kromě parametrů navíc je třeba také dodržet určitá omezení pro zadání parametru *DLC*. Čísla 0 - 15 kódují délku zprávy v bajtech (0, 1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 32, 48, 64) a číslo 16 odpovídá parametru *DLC* s hodnotou *Stejná délka*. Sekvence příkazů opět nejdříve načte všechna pravidla, kterých je sice dle tabulky (15) 11, ale protože některá jsou s parametrem *ID Existující*, nebudou zahrnuta a pravidel bude pouze 9 z maxima daným aplikací – 10. Samozřejmě, pokud by došlo k chybě, nebo by byl test modifikován na 11 pravidel s unikátním *ID*, aplikace by mohla načíst své maximum, ale potom by ohlásila, že list pravidel je plný. Po zadání všech pravidel opět následuje příkaz *Trig* a opět sekvence několika odeslání zpráv, pro otestování každého pravidla.

7.3 Testování finální aplikace

7.3.1 Kvaser Memorator Pro

Pro testování finální aplikace byl poskytnut Kvaser Memorator Pro – nástroj pro logování, ale také jako aktivní jednotka na sběrnici *CAN* s podporou filtrace, detekce a generace chyb. Disponuje dvěma *CAN FD* rozhraními (komunikační rychlost v rozmezí 50-1000 *kbps*) galvanicky oddělenými od samotné desky a pro připojení k PC využívá USB.

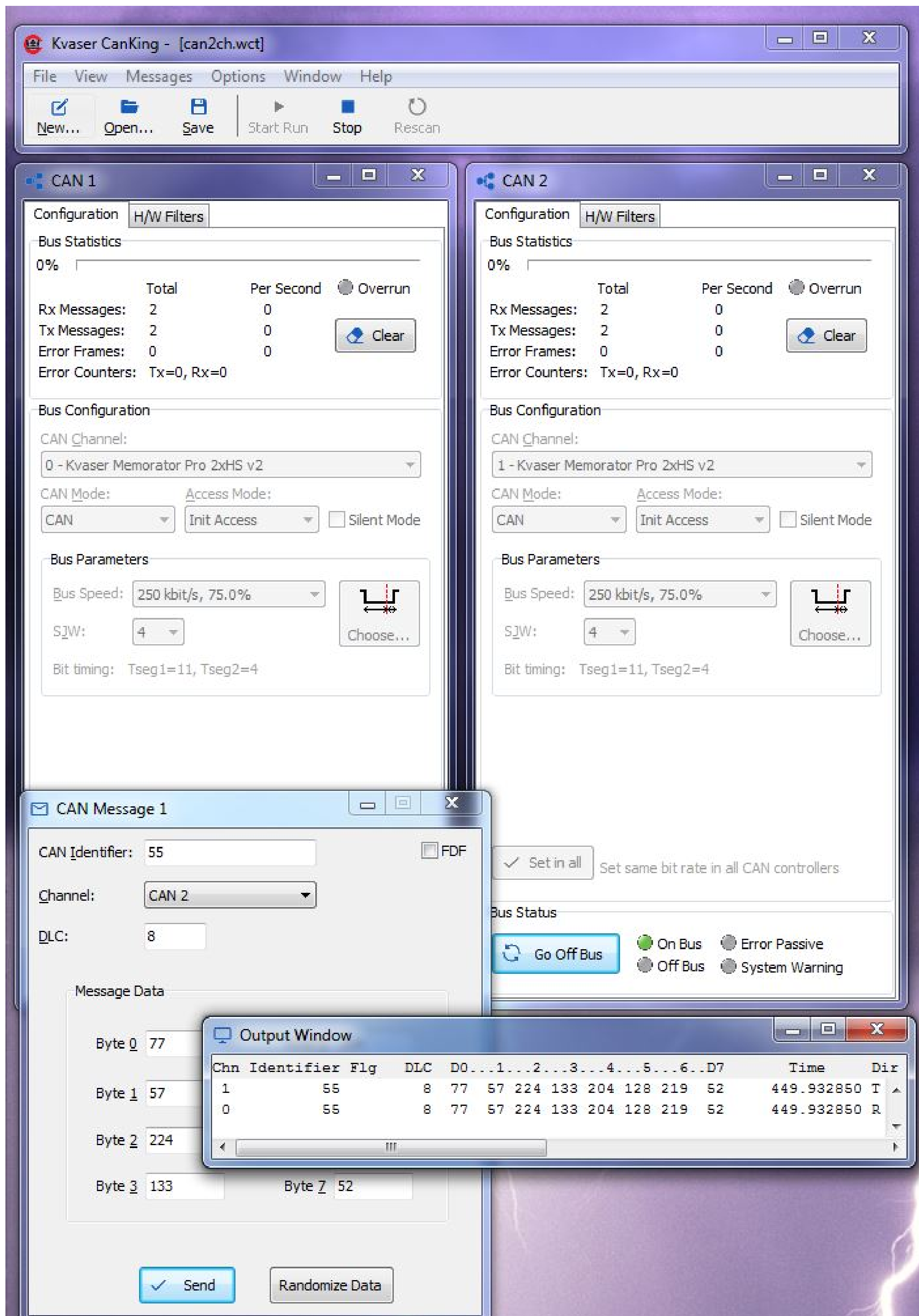
Z hlediska softwaru je na stránkách výrobce volně ke stažení program *CAN King* a balík *SDK*(Software Development Kit) pro snadnou implementaci tohoto nástroje v rámci aplikace.

Obr. 18 Kvaser Memorator Pro.

Program *CAN King* slouží ke snadnému monitorování připojené *CAN* sítě, ale také může jednotka být na síti aktivní, tj. aktivně se podílet na komunikaci a potvrzovat přijaté rámce a samozřejmě také odesílat rámce definované uživatelem.

Pro prvotní otestování byl použit právě program *CAN King*, kdy obě *CAN* rozhraní byla propojena za pomoci terminované propojky (pro fyzickou komunikaci v síti CAN se používá 9 pinový konektor a ještě k tomu musí být mezi vodiči *CAN_H, CAN_L* 120 Ohmový odpor pro zakončení, jinak by docházelo k odrazům na vedení). Propojením tedy vznikl tzv. *Loopback* – když jedno rozhraní pošle rámec, druhé ho přijme a naopak. V *Loopback* módu musí být také obě rozhraní v rámci sítě aktivní, protože *CAN* vyžaduje, aby byl každý rámec někým potvrzen. Po propojení a stisknutí tlačítka *Start Run* pro aktivaci obou rozhraní s daným nastavením je už pak generování zprávy pro otestování velice jednoduché. Obrázek ukazuje krátký záznam, že propojení obou jednotek v aplikaci *CAN King* bylo úspěšné.

Po ověření, že nástroj funguje, následoval další krok a to odesílání a příjem zpráv pomocí *SDK*, aby bylo ve finále možné testovat reálnou aplikaci, stejně jako výše popsanou simulaci. *SDK* obsahuje kromě samotného *API* i řadu příkladů a to v různých programovacích jazycích. Podporovány jsou jazyky *C, C++, C#, Delphi a Python.* Protože simulace je psaná v jazyce *C* a značná část je také určena pro testování, byl jazyk *C* jasnou volbou. Tudíž projekt simulace byl rozšířen o několik funkcí podporujících práci s nástrojem *Kvaser*. V rámci hlavní simulační smyčky se pak místo samotné aplikace volá odesílání, nebo příjem z *CAN* rozhraní *Kvaser*, takže ve výsledku ten stejný vstup uživatele, nebo testovací soubor může být použit i pro otestování reálné aplikace.



Obr. 19 Aplikace CAN King.

7.4 Závěr testování

Simulace i reálná aplikace na hardwaru prošly sekvencemi testů popsanými výše. Není zde žádná automatická kontrola testů, ale pouze manuální. Bylo objeveno a odstraněno několik chyb na straně testovacího programu. Samotná aplikace se, dle testů, zdá být funkční dle specifikace. Samozřejmě při manuální kontrole může dojít k chybám a také testy nemusí odhalit všechny chyby. Také budoucí nasazení aplikace do reálné sítě může odhalit další chyby, které testy neprověřily, nebo ani prověřit nemohly, ale to již není náplní této práce.

8 Závěr

Tato práce realizuje programové vybavení pro *CAN FD Gateway*. Protože hardware nebyl na začátku dostupný, byla nejdříve implementována simulace i s konzolovým uživatelským rozhraním pro zadávání parametrů. Pro simulaci a ve finále i reálnou aplikaci byly navrženy testy pro ověření správné funkce aplikace.

Navržená aplikace používá definovaný protokol pro správu *Gateway*. Jsou definovány 3 sítě – 2 mezi kterými jsou přeposílány rámce a třetí, která řídí pravidla pro zpracování (pravidla jsou typu blokace, přenos, modifikace zprávy).

Jakmile byl i hardware dostupný, jehož oživení provázely jisté problémy, které byly ale překonány, byla implementována vrstva mezi samotnou aplikací a hardwarem. Výpočetní funkci zajišťuje mikrokontrolér firmy *Cypress*. Pro 3 definovaná *CAN* rozhraní jsou připraveny 3 *D-Sub* konektory, budiče a použitý mikrokontrolér s implementací *FD CAN* periferie. Díky abstrakci aplikace od hardwaru je možné vytvořené řešení teoreticky naportovat na jakýkoli jiný hardware nebo i simulaci.

Také výsledná aplikace na reálném hardwaru prošla všemi testy a do budoucna se počítá s reálným nasazením v *CAN FD* síti, pro kterou je řešení navrženo. Tímto jsou tedy všechny body zadání splněny a aplikace se, v laboratorních podmínkách, zdá být plně funkční.

Literatura

[1] Wikipedia. *Gateway (telecommunications)*. 2018. URL: https://en.wikipedia.org/wiki/Gateway_(telecommunications) (cit. 22. 03. 2018).

[2] Wikipedia. *CAN bus*. 2018. URL: https://en.wikipedia.org/wiki/CAN_bus (cit. 22. 03. 2018).

[3] J. Novák. *Přednáška KRP - CAN bus a jeho aplikace ve vozidlech*. 2018. URL: https://moodle.fel.cvut.cz/pluginfile.php/104928/mod_resource/content/1/CAN_Automotive.pdf (cit. 22. 03. 2018).

[4] Automotive Hub. *CAN basics*. 2016. URL: http://techiscafe.blogspot.cz/p/the-controller-area-network-can.htmll (cit. 01. 05. 2018).

[5] Reutlingen Robert Bosch GmbH. *CAN FD - CAN with Flexible Data-rate*. 2012. URL: https://www.can-cia.org/fileadmin/resources/documents/proceedings/2012_hartwich.pdf.

[6] Cypress Semiconductor. *32-bit Microcontroller Traveo Family*. 2018. URL: http://www.cypress.com/file/233626/download (cit. 22. 03. 2018).

[7] Linear Technology. *LT3514 - Triple Step-Down Switching Regulator with 100% Duty Cycle Operation*. 2013. URL: http://www.analog.com/media/en/technical-documentation/data-sheets/3514fa.pdf (cit. 22. 03. 2018).

[8] NXP. *TJA1051 High-speed CAN transceiver*. 2017. URL: https://cz.mouser.com/datasheet/2/302/TJA1051-1127850.pdf (cit. 22. 03. 2018).

[9] Segger. *J-Link Debug Probes*. 2018. URL: https://www.segger.com/products/debug-probes/j-link/ (cit. 01. 05. 2018).

[10] Lpctools. *j-link JTAG Debugger*. 2006. URL: http://www.lpctools.com/ProductImages/images/iar_jlink_arm_yellow.jpg (cit. 01. 05. 2018).

[11] IAR Systems. *I-Jet Debugger*. 2018. URL: https://www.iar.com/iar-embedded-workbench/add-ons-and-integrations/in-circuit-debugging-probes/#!?currentTab=i-jet (cit. 01. 05. 2018).

[12] Cypress Semiconductor. *Getting Started with Traveo Family S6J3300 Series MCUs*. 2018. URL: http://www.cypress.com/file/207071/download (cit. 22. 03. 2018).

[13] Cypress Semiconductor. *How to Set Up the Clock System for the Traveo Family*. 2018. URL: http://www.cypress.com/file/236456/download (cit. 22. 03. 2018).

[14] Cypress Semiconductor. *S6J33xx, S6J34xx, S6J35xx Series Hardware Manual Platform Part*. 2018. URL: http://www.cypress.com/file/219516/download (cit. 22. 03. 2018).

[15] Cypress Semiconductor. *Traveo Family Hardware Manual*. 2018. URL: http://www.cypress.com/file/221496/download (cit. 22. 03. 2018).

[16] Cypress Semiconductor. *How to Use CAN FD for Traveo Family*. 2018. URL: http://www.cypress.com/file/354851/download (cit. 22. 03. 2018).

[17] Miroslav Bureš. *Přednáška Zajištění kvality softwaru – Příprava kombinací testovacích dat*. 2016. URL: `http://www.satisfice.com/tools.shtml` (cit. 01. 05. 2018).

[18] Satisfice. *ALLPAIRS Test Case Generation Tool*. 2016. URL: `http://www.satisfice.com/tools.shtml` (cit. 01. 05. 2018).

Příloha A

Ukázky kódu

Ukázka A.1 Algoritmus pro přenos zprávy dle pravidel.

```
void FDCAN_GW_xfer_message(FDCAN_TypeDef * FDCANx_target,
FDCAN_RxMessage * RxMessage)
{
uint8_t ret = 0;
uint32_t id;
if (RxMessage->Identifier == FORBIDDEN_ID) return;
id = RxMessage->Identifier;
if ((void *)FDCANx_target == (void *)&FD2) FD1_traffic++;
else FD2_traffic++;

for (uint32_t i = 0; i < NUM_ITEMS; i++)
{
        if (id != active_config[i].ID) continue;
        GW_item_t * item = &active_config[i];
        #ifdef EMULATE
        FDCAN_GW_print_rule(item);
        #endif // EMULATE

        switch (item->state)
        {
        case STATE_BLOCK_PASS:
                /*if there is passing reset,
                this message should be blocked,
                if there is blocking reset,
                this message should be passed*/
                ret = (active_reset == RESET_PASSING) ? 0 : \
                FDCAN_GW_pass_msg(FDCANx_target, RxMessage);
                return;
        case STATE_BLOCK_PASS_TIME:
                /*this rule is removed by timer,
                 in emulated version by thread*/
                ret = (active_reset == RESET_PASSING) ? 0 : \
                FDCAN_GW_pass_msg(FDCANx_target, RxMessage);
                return;
        case STATE_BLOCK_PASS_COUNT:
                if (item->length == 0) {
                        item->ID = FORBIDDEN_ID;
                        /*in this case the ternary is opposite, we need
                        to send the message after count reaches zero*/
                        ret = (active_reset == RESET_PASSING) ? \
                        FDCAN_GW_pass_msg(FDCANx_target, RxMessage) : 0;
                }
                else {
                        item->length--;
```

```
                    ret = (active_reset == RESET_PASSING) ? 0 : \
                    FDCAN_GW_pass_msg(FDCANx_target, RxMessage);
            }

            return;
        case STATE_MODIFY:
            FDCAN_GW_modify_msg(FDCANx_target, RxMessage, i);
            return;
        case STATE_MODIFY_COUNT:

            if (item->length == 0) {
                    item->ID = FORBIDDEN_ID;
            }
            else {
                    FDCAN_GW_modify_msg(FDCANx_target, RxMessage, i);
                    item->length--;
            }
            return;
            case STATE_MODIFY_TIME:
            FDCAN_GW_modify_msg(FDCANx_target, RxMessage, i);
            return;


        default:
            /*if there was a special rule with undefined state
             - that is an error*/
            if ((void *)FDCANx_target == (void *)&FD2) {
                    ack_stat[0] |= (active_reset == RESET_PASSING) ? \
                    ERR_LIST_FD1_PASSING : ERR_LIST_FD1_BLOCKING;
            }
            else {
                    ack_stat[0] |= (active_reset == RESET_PASSING) ? \
                    ERR_LIST_FD2_PASSING : ERR_LIST_FD2_BLOCKING;
            }
            return;
        }
}
/*lastly, if there is a message without a special rule,
we handle it according to reset state*/
ret = (active_reset == RESET_PASSING) ? \
FDCAN_GW_pass_msg(FDCANx_target, RxMessage) : 0;
}
```

Ukázka A.2 Inicializace LED (na pinu P0_27) pomocí registrů.

```
//LEDs are on pins 55-58 -- P0_27-30
//PPC_PCFGR027(0x0036) P0_27  GPIO_PODR0:POD27 SCS23_0 M_SDATA1_0 - M_RWDS
//              increment  word access  offset
PPC_KEYCDR = (0x0 << 30) | (0x2 << 28) | 0x36;
PPC_KEYCDR = (0x1 << 30) | (0x2 << 28) | 0x36;
PPC_KEYCDR = (0x2 << 30) | (0x2 << 28) | 0x36;
PPC_KEYCDR = (0x3 << 30) | (0x2 << 28) | 0x36;
//P0_27 POF[2:0] = 0 - GPIO_PODR0:POD27, POD[14]=0 (Low)PortOutputDataBit
PPC_PCFGR027 = 0x0;
//              increment  word access  offset (DDSR reg offset)
GPIO_KEYCDR = (0x0 << 30) | (0x2 << 28) | 0x8;
GPIO_KEYCDR = (0x1 << 30) | (0x2 << 28) | 0x8;
GPIO_KEYCDR = (0x2 << 30) | (0x2 << 28) | 0x8;
GPIO_KEYCDR = (0x3 << 30) | (0x2 << 28) | 0x8;
GPIO_DDSR0 = 0x08000000;//27 th bit direction to output

GPIO_PODR0 = 0x0 << 27;
```

Ukázka A.3 Inicializace CAN FD periferie uvnitř MCU pomocí struktury.

```
/* CAN-FD - configuration */
stc_canfd_config_t stcCanfdConfig = {
        /* use callback for interrupt */
        .pfnTxCallback = NULL,
        /* use callback for interrupt */
        .pfnRxCallback = &CanFDCallbackRx,
         /* use callback for interrupt */
        .pfnStatusCallback = NULL,
        /* use callback for interrupt */
        .pfnErrorCallback = NULL,
         /* TRUE: CAN-FD mode, FALSE: CAN mode */
        .bCanFDMode = TRUE,

        //at CAN 40Mhz, divided by 4, 10MHz,
        // we need 20TQ for 500kbit/s (500kbit - 2us)/(10Mhz - 100ns) = 20
        //20 = 1sync + 15tseg1 + 4tseg2 [values are inserted as (val-1)]

        /* PRESCALER for BTP */
        .stcBitrate.u16Prescaler = 3,
        /* TSEG1 for BTP */
        .stcBitrate.u8TimeSegment1 = 14,
         /* TSEG2 for BTP */
        .stcBitrate.u8TimeSegment2 = 3,
        /* SYNC_JUMP_WIDTH for BTP */
        .stcBitrate.u8SyncJumpWidth = 4,

        //for bit rate switch, same num of TQs - 20,
        //but 4 times faster clock, so div by 1 -- gives 2Mbit/s

        /* PRESCALER for FBTP */
        .stcFastBitrate.u16Prescaler = 0,
        /* TSEG1 for FBTP */
        .stcFastBitrate.u8TimeSegment1 = 14,
        /* TSEG2 for FBTP */
        .stcFastBitrate.u8TimeSegment2 = 3,
        /* SYNC_JUMP_WIDTH for FBTP */
        .stcFastBitrate.u8SyncJumpWidth = 2
};

// CAN Prescaler Configuration
// CAN prescaler division setting : we have 40MHz going to CAN periph,
// so div1 - 1
// CAN prescaler source clock selection : FALSE (Select the PLL clock)

Can_PrescalerInit((stc_canp_t*)&CANP, 0, FALSE); // CAN clock = 40MHz

// CAN-FD Initialize

CanFD_Init(CANFD0_Type, &stcCanfdConfig);
CanFD_Init(CANFD3_Type, &stcCanfdConfig);
CanFD_Init(CANFD5_Type, &stcCanfdConfig);
```