UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Robotický systém pro dálkové ovládání

Bc. Ota Kober

Diplomová práce
2017

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2015/2016**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Ota Kober**
Osobní číslo:	**I13377**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Komunikační a řídicí technologie**
Název tématu:	**Robotický systém pro dálkové ovládání**
Zadávající katedra:	**Katedra elektrotechniky**

Zásady pro vypracování:

V rámci teoretické části proveďte kritickou rešerši podobných systémů. Proveďte návrh komunikačního systému dálkového ovládání včetně návrhu vhodného protokolu.
V praktické části realizujte ovladač motorků, ovládací prvek a BlueTooth modul pro připojení "chytrého" mobilního telefonu. Vytvořte odpovídající programové vybavení ovladače motorků a ovládacího prvku. Vytvořte aplikaci pro mobilní telefon.

Rozsah grafických prací:

Rozsah pracovní zprávy: **60 stran A4**

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

ST Microelectronics [online]. [cit. 2014-10-26]. Dostupné z: www.st.com
MATOUŠEK, David. Aplikace mikrokontrolérů ATmega644. 1. vyd. Praha: BEN - technická literatura, 2013, ca 200 s. v různém stránkování. ISBN 978-80-7300-492-7.

Vedoucí diplomové práce: **Ing. Bc. David Matoušek**
Katedra elektrotechniky

Datum zadání diplomové práce: **31. října 2015**

Termín odevzdání diplomové práce: **13. května 2016**

Univerzita Pardubice Česká republika

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

Ing. Zdeněk Němec, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2015

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 16. 8. 2017 Ota Kober

Poděkování

Tímto bych rád poděkoval panu Ing. Bc. Davidu Matouškovi za vedení této práce a za cenné rady, které mi poskytl. Dále také rodině a přítelkyni, kteří mne podporovali po celou dobu studia.

Anotace

Cílem této práce je návrh systému pro dálkové ovládání polohy motorků. Jsou zde rozebrány komunikační protokoly používané v různých odvětvích a zejména proveden návrh vlastního komunikačního protokolu pro daný systém. Dále je proveden návrh, fyzické sestavení a vytvoření softwaru jednotlivých prvků daného systému. Součástí práce je též aplikace pro OS Android, pomocí které probíhá komunikace s daným systémem.

Klíčová slova

robotický systém, ovládání polohy, sběrnice, protokol, krokové motorky, android

Title

The robotic system for remote control

Annotation

The aim of this thesis is to design the system for remote position control of motors. The communication protocols used in the various sectors will be analyzed here. Mainly the design of own communication protocol for the given system will be developed. In addition, the design, physical build and creation of the software for each element of the system is carried out here. Another part of the work is an Android application that is also used for communication with the system.

Keywords

robotic system, position control, bus, protocol, stepper motors, android

Obsah

Seznam zkratek

UART	Universal Asynchronous Receiver/Transmitter
SPI	Seriál Peripheral Interface
I^2C	Inter-Integrated Circuit
PLC	Programmable Logic Controller
RO	Read Only
R/W	Read/Write
CAN	Controller Area Network
OD	Object Dictionary
EDS	Electronic Data Sheet
DCF	Device Configuration File
SDO	Service Data Object
PDO	Process Data Object
AC	Alternating Current
DC	Direct Current
RC	Radio Control
MOSI	Master Out Slave In
MISO	Master In Slave Out
SCK	Serial Clock
PWM	Pulse Width Modulation
API	Application Programming Interface
OS	Operační Systém
SNR	Signal-to-Noise Ratio
RMS	Root Mean Square

Seznam symbolů

τ	časová konstanta
l	délka
D	měrné průchozí zpoždění
x	počet
V	napětí
P	Výkon

Seznam obrázků

Seznam tabulek

Úvod

Původní myšlenkou pro návrh daného systému byla možnost dálkového ovládání potenciometrů na kytarových aparátech. Mnoho kytaristů v dnešní době nedá dopustit na elektronkové zesilovače, díky jejich charakteristickému zkreslení. Velmi oblíbené jsou mimo jiné tzv. „vintage“ modely zesilovačů, které kopírují tradiční schémata již z cca. 60. let minulého století. Jedná se například o různé modely kytarových zesilovačů značek Vox, Fender, Marshall a podobně. Jelikož se v mnoha případech jedná o jednokanálové zesilovače, vzniká zde problém s rychlým přepínáním zvukových nastavení během právě hrané skladby. To je obecně řešeno různými zkreslovacími krabičkami, které jsou zapojeny do signálové cesty, přičemž je na zesilovači implicitně nastaven „čistý“ zvuk. Tímto způsobem je sice docíleno možnosti rychlého přepínání mezi čistým a zkresleným zvukem, bohužel zde již není přítomno kýžené zkreslení vytvářené právě elektronkami v zesilovači, tedy jeden z hlavních důvodů vůbec pro koupi elektronkového aparátu.

Tento problém mě přivedl na nápad s mechanickým nastavováním poloh potenciometrů externím systémem, pomocí například krokových motorků. Uživatel (hráč na kytaru) bude mít možnost rychlého přepínání mezi různými zvukovými nastaveními aparátu pomocí nožního přepínače. Dále bude možná komunikace s mobilními zařízeními s OS Android pomocí Bluetooth.

Je nutno zmínit, že realizace podobných nápadů již proběhla a to například už v roce 1978, kdy Neil Young používal speciálně pro něj sestavené zařízení nazvané „Whizzer“. V té době bylo toto zařízení schopno mechanicky ovládat pouze jeden potenciometr (hlasitost) a bylo omezeno na dvě možná nastavení. Později v roce 1991 byl opět pro Neila Younga sestaven další „Whizzer“ ovšem jiným konstruktérem. Ten již ovládal tři potenciometry a měl více možných zvukových nastavení (Davis, 2014). Dalším příkladem je kytarové kombo Fender Cyber Twin, jehož součástí je právě mechanické polohování potenciometrů a obdobně fungující T-Rex Spin Doctor, což je efektový pedál. Nakonec lze uvést také nedávno proběhlý projekt na serveru Kickstarter, nazvaný „Rig Master“, který si klade za cíl jistou univerzálnost a možnost připojení na jakékoli zařízení, čímž je podobný mnou navrhovanému systému (Kickstarter, 2017).

Celý systém se tedy fyzicky sestává z nejméně tří částí a to tzv. ovladače motorků, který zajišťuje řízení polohy daným motorkům, nožního přepínače, sloužícího pro ovládání a nakonec bluetooth modulu, který bude jakýmsi prostředníkem mezi OS Android a navrhovaným systémem. V této práci tedy provedu návrh jednotlivých prvků a příslušného komunikačního protokolu, přičemž teorie týkající se daného problému bude vždy probírána před jednotlivými tématy.

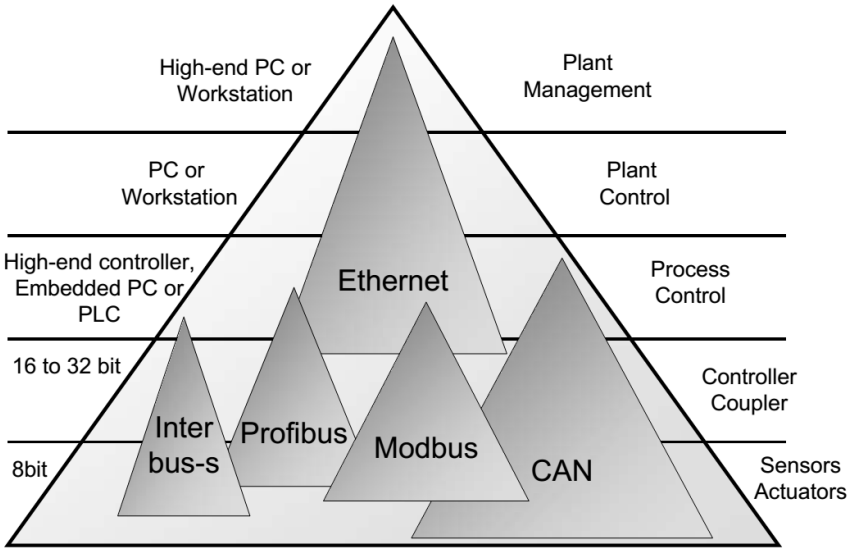
1 Návrh systému

Nejprve bude pojednáno obecně o komunikaci ve vestavěných systémech a v průmyslu. Poté bude zmíněn přístup k řešení navrhovaného systému.

1.1 Komunikace ve vestavěných systémech

V současné době, díky rozmachu jednočipových procesorů, je svět zahlcen vestavěnými (jednoúčelovými) zařízeními. V těchto systémech obvykle vznikají požadavky na různé druhy komunikace. Může se jednat o komunikaci mezi mikroprocesory a jinými integrovanými obvody, zpravidla na krátké vzdálenosti, kde jsou typicky využity sběrnice UART, SPI nebo I^2C. Dále existují systémy vyžadující vzájemnou komunikaci mezi oddělenými prvky i na větší vzdálenosti. Příkladem mohou být různé systémy pro ovládání výtahů, řídicí systémy v automobilech a nespočet průmyslových aplikací ve výrobních linkách a podobně.

Pro popis průmyslových řídicích systémů lze sestavit tzv. komunikační pyramidu (Obrázek 1), která znázorňuje různé úrovně celkového systému. Pyramida může například popisovat nějaký výrobní proces.



Obrázek 1 – Komunikační pyramida s příklady použitých komunikačních technologií (Pfeiffer, a další, 2008)

Úplný spodek pyramidy tvoří senzory a akční členy (Sensors, Actuators). Senzory mohou být v závislosti na aplikaci opravdu různé, kdy příkladem jsou snímače teploty, otáček, polohy a podobně. Akční členy mohou reprezentovat například hydraulické, či elektrické pohony. O úroveň výše se vyskytují ovladače (Controller), které tvoří jakési přemostění mezi nejspodnější vrstvou a prvkem řídicím daný proces. V průmyslové aplikaci si to lze nejlépe představit například na měniči, který dle požadavků řídicího prvku, typicky PLC, ovládá polohu elektrického pohonu. S tímto měničem je tedy spojen na jedné straně jednak elektrický pohon a navíc například resolver určující polohu daného pohonu a na straně druhé

je měnič pomocí komunikační sběrnice propojen s PLC. Jak bylo zmíněno, další vrstvu tvoří prvek řídicí daný proces (Process control), což je v průmyslových aplikacích nejčastěji právě PLC. Vrchní dvě vrstvy zajišťují management, tedy zejména rozdělení a plánování dílčích procesů.

Ve vestavěných systémech je situace téměř obdobná, až na zmíněné vrchní dvě vrstvy, které jsou zřídkakdy zapotřebí. Bude-li se ale zaměřeno na vrstvy zbývající, je zřejmé, že s jejich pomocí je možné popsat komunikaci právě i ve vestavěných systémech. Jediný rozdíl je tedy v implementaci daných zařízení, kdy například místo PLC je použit mikroprocesor starající se o požadovaný proces a podobně.

S ohledem na komunikační pyramidu je vhodné také zmínit fakt, že čím je vrstva nižší, tím více se zde nachází prvků a navíc tyto prvky obvykle plní elementární úlohy, tudíž zde nejsou tak vysoké nároky na výpočetní výkon. Je zde tedy snaha použít jednoduché technologie jednak z toho důvodu, že pro plnění daného úkolu plně dostačují, navíc je spodní vrstva více „cenově citlivá“ právě díky většímu počtu prvků. V případě situace, kdy je třeba zjistit například 16bitový údaj o poloze určitého senzoru a pro komunikaci by bylo využito Ethernetového spojení, bylo by dané řešení jednak naprosto zbytečně složité a navíc s ohledem na fakt, že v celkovém systému může být více senzorů i velmi drahé. Pro tyto účely tedy existují různé průmyslové protokoly, které budou probrány v další kapitole (Protokoly rodiny Fieldbus) této práce. (Pfeiffer, a další, 2008)

1.2 Navrhovaný systém

Původní myšlenkou bylo vytvořit systém, pomocí kterého by bylo možné mechanicky ovládat potenciometry kytarových zesilovačů. Uživatel, v tomto případě hráč na kytaru, by tedy mohl měnit různá zvuková nastavení svého aparátu pomocí několika motorků mechanicky spojených s danými potenciometry. Nastavení poloh motorků by uživatel prováděl například pomocí nožního přepínače, nebo zařízení s OS Android.

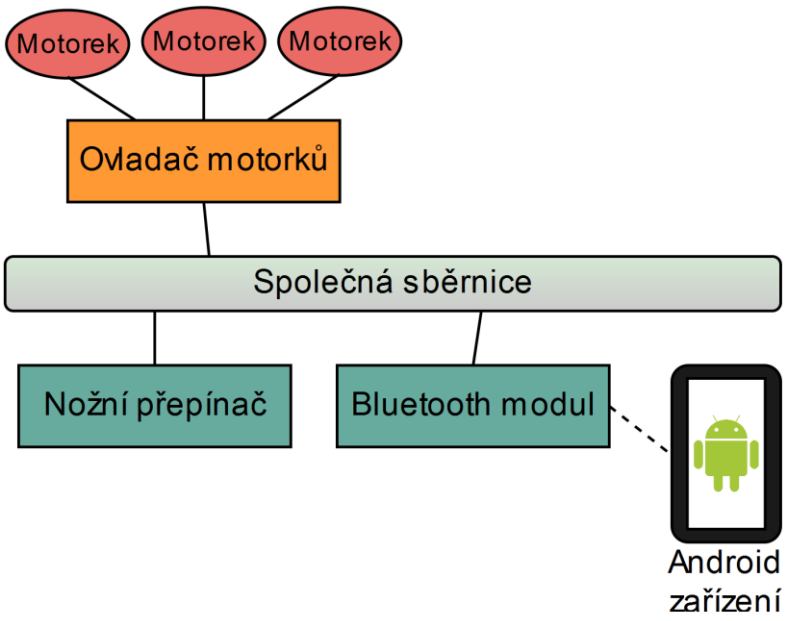
Blokové schéma navrhovaného systému je na obrázku níže (Obrázek 2), kde je mimo jiné znázorněno použití společné sběrnice pro vzájemnou komunikaci. Právě použití jedné společné sběrnice činí systém flexibilním co do počtu připojených prvků. Je zde předpoklad, že veškeré informace vyslané na sběrnici jsou okamžitě k dispozici všem prvkům.

Po určitém zobecnění lze prvky připojené na společnou sběrnici rozdělit podle jejich funkce na dvě skupiny, a to:

- na akční prvky,
- a na ovládací prvky.

Akčním prvkem je v tomto případě ovladač motorků včetně oněch samotných motorků. Obecně to může být jakýkoli člen, který je třeba nějakým způsobem ovládat. Oproti tomu ovládací prvek slouží právě pro ovládání akčních prvků. Konkrétně je zde realizován nožním

přepínačem. Do kategorie ovládacích prvků lze ale zařadit i bluetooth modul spojený se zařízením s OS Android.



Obrázek 2 - Navrhovaný systém

Při návrhu systému tedy bude brán v potaz požadavek na určitou modulárnost. S tím dále souvisí požadavek na možnost připojení vícero ovládacích a akčních prvků na danou sběrnici.

V dalším textu budou navrženy a teoreticky rozebrány jednotlivé části navrhovaného systému. Jedná se v první řadě o návrh společné sběrnice. Tento problém bude rozdělen do dvou kategorií a to:

- řešení protokolu (=linková a aplikační vrstva referenčního modelu ISO/OSI)
- a řešení sběrnice (=fyzická vrstva referenčního modelu ISO/OSI).

Dále bude následovat návrh akčního prvku s ohledem na danou aplikaci a samozřejmě návrh ovládacího prvku, nejdříve v podobě nožního přepínače a poté v podobě bluetooth modulu. S tímto souvisí další úkol, a to napsat software pro OS Android, pomocí kterého bude probíhat komunikace mezi například „chytrým“ telefonem a zmíněným bluetooth modulem.

2 Protokoly rodiny Fieldbus

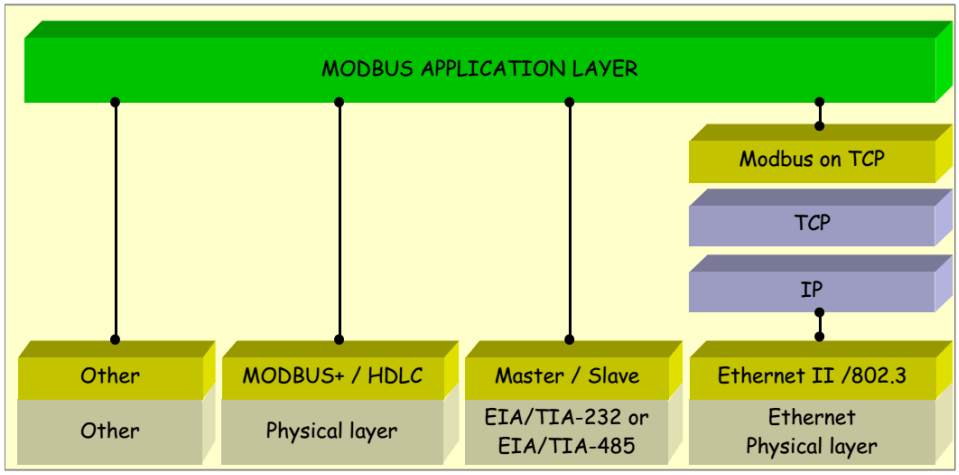
Než bude proveden samotný návrh protokolu, budou zde nejprve zmíněny některé v různých oblastech používané komunikační protokoly.

V průmyslových řídicích systémech byl zpočátku trend propojení akčních prvků a senzorů s prvky řídicími samostatnými kabely. Tím je myšleno takové zapojení, kdy určitý řídicí prvek (např. PLC), je s daným senzorem spojen jedním kabelem, zatímco s prvkem akčním je spojen kabelem jiným, připojeným do jiného konektoru. Po jednom konkrétním kabelu byla tedy možná komunikace pouze mezi dvěma prvky. Počet možných připojených zařízení k jednomu prvku je zde dán maximálním podporovaným počtem konektorů onoho prvku.

Od principu výše se liší myšlenka použití jedné společné sběrnice pro komunikaci mezi vícero prvky v reálném čase. Právě tato věc je význačná pro skupinu protokolů typu Fieldbus. Propojení prvků může být realizováno různými způsoby, jako například použitím stromových, hvězdicových či kruhových síťových topologií. Počet připojených zařízení k řídicímu prvku již není omezen fyzickým počtem konektorů, jelikož je zde možné využít konektor jediný. Dále tedy budou probrány protokoly spadající pouze do této kategorie. (Kumar, 2014)

2.1 MODBUS

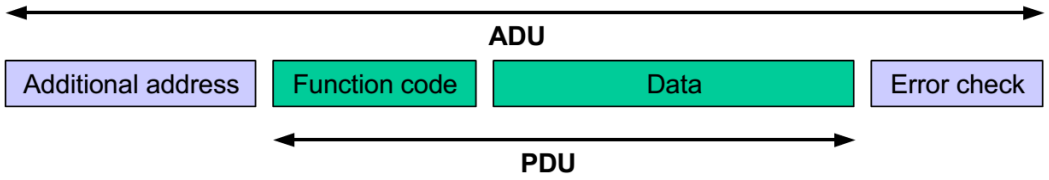
Možná vůbec nejznámějším protokolem pro průmyslové řízení je MODBUS. Nejdříve se jednalo o proprietární protokol vyvinutý firmou Modicon (dnes Schneider Electric), který byl ovšem později uvolněn a díky své jednoduchosti a robustnosti se stal velice rozšířeným. MODBUS definuje aplikační vrstvu modelu ISO/OSI a poskytuje klient/server komunikaci mezi prvky připojenými k různým sběrnicím či sítím.



Obrázek 3 - MODBUS implementace (Studios, 2012)

Jak je vidět na obrázku výše (Obrázek 3), je MODBUS momentálně implementován protokolem TCP/IP přes Ethernet, různými podobami asynchronních sériových komunikací (např. RS-232, RS-422, RS-485) nebo protokolem MODBUS PLUS.

Rámec protokolu (Obrázek 4) se skládá z tzv. application data unit (ADU) a protokol data unit (PDU). Funkční kód (Function code), který je realizován jedním bajtem, předává serveru informaci o tom, jakou akci má vykonat. Při kladném vyřízení požadované akce je zpráva se stejným funkčním kódem poslána zpět klientovy i s případnými daty. Rozsah hodnot 128-255 je vyhrazen pro oznámení záporné odpovědi (chyby). (Studios, 2012)



Obrázek 4 - MODBUS rámec (Studios, 2012)

2.2 Profibus

Profibus představuje jeden z nejvíce používaných standardů pro komunikační síť typu Fieldbus v oblasti průmyslového řízení. Možnosti využití tohoto standardu jsou opravdu široké, pokrývající například automatizaci výrobních linek, domovní automatizaci, řízení výroby a distribuce energie a podobně. Existují celkem tři typy protokolu Profibus a to:

- Profibus DP,
- Profibus PA,
- a Profibus FMS.

Vzhledem k referenčnímu síťovému modelu ISO/OSI definuje standard Profibus pouze fyzickou, linkovou a aplikační vrstvu.

Fyzická vrstva definuje použité fyzické spojení a danou síťovou topologii. Fyzické spojení může být realizováno rozhraním RS-485 (FMS, DP), optickým vláknem (FMS, DP) a proudovou smyčkou dle IEC 1158-2 (PA). Při použití rozhraní RS-485 lze dosáhnout přenosových rychlostí od 9,6 kbps (pro vzdálenost do 1200 m) po 1,2 Mbps (pro vzdálenost do 100 m). Doporučená síťová topologie je zde sběrnice. Přenosová rychlost proudové smyčky je 31,25 kbps.

Linková vrstva řeší především přístup účastníků na společné přenosové médium. Toto je zde řešeno dvěma způsoby a to metodou „master-slave“ a „token-passing“. Princip první, obecně známé metody, spočívá v určení jednoho „master“ zařízení, které dotazuje všechny ostatní zařízení připojené ke sběrnici. Dané „master“ zařízení tedy řídí veškerou komunikaci a ostatní „slave“ zařízení pouze odpovídají na dotazy. Druhá metoda „token-passing“ funguje tím způsobem, že po zařízeních připojených na sběrnici koluje tzv. „token“, který dané zařízení opravňuje k vysílání. Zařízení tedy vytváří logický kruh, po kterém kolují tokeny.

Obě výše zmíněné metody mimochodem spadají do skupiny deterministických metod přístupu na společné médium. Zmíněné deterministické metody se na rozdíl od stochastických vyznačují tou vlastností, že je zde garantován přenos informace v konečném čase. Výše zmíněné metody lze také ve standardu Profibus zkombinovat tím způsobem, že tokeny kolují pouze po master zařízeních, které při získání oprávnění k vysílání dotazují slave zařízení připojené ke sběrnici. (Kumar, 2014)

2.3 CAN

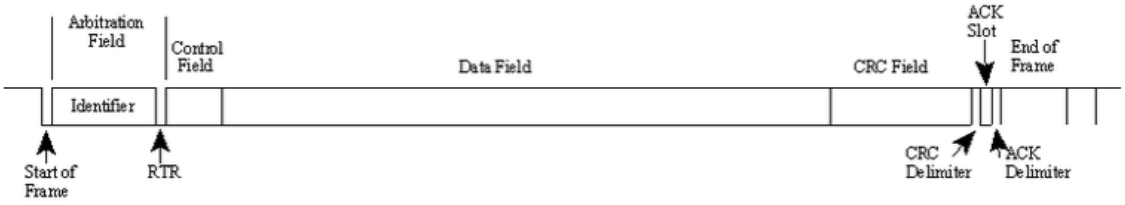
CAN (control area network) je protokol původně vyvinutý pro potřeby komunikace v rámci společné sběrnice mezi zařízeními v automobilu, později pronikající i do dalších, mimo jiné, také automatizačních odvětví. Ve skutečnosti existuje mnoho standardů tohoto protokolu, z nichž nejznámější jsou protokol linkové vrstvy ISO/OSI modelu dle standardu ISO 11898-1 a protokol fyzické vrstvy dle standardu ISO 11898-2.

CAN využívá NRZ kódování s tzv. „bit stuffing“, což lze do češtiny přeložit jako „plnění bity“. Princip onoho plnění bity spočívá v tom, že pokud po sobě následuje 5 bitů stejné logické úrovně, je následující vysílaný bit opačný a nepočítá se do výsledné zprávy. Protokol rozlišuje mezi dominantními bity, dále uvedené jako logické „0“, a nedominantními bity, jako logické „1“. Pokud jedno z připojených zařízení vyšle logickou „0“, tak bez ohledu na to, zda někdo jiný pošle logickou „1“, sběrnice bude vždy v „0“. Tohoto faktu se využívá při přístupu na společné médium tím způsobem, že zařízení ve zprávě nejprve posílá svůj identifikátor a zároveň kontroluje při vysílání stav sběrnice. Pokud stav sběrnice neodpovídá posílanému kódu, pravděpodobně vysílá i někdo jiný, který sběrnici stahuje do „0“ a proto zařízení přeruší své vysílání. Je zřejmé, že nejvyšší prioritu mají zařízení s nejnižší hodnotou identifikátoru (v případě vysílání MSB jako prvních), jelikož obsahují od začátku vysílání nejvíce „0“ a tak vždy vyhrají „souboj“ se zařízeními obsahující ve významných bitech „1“.

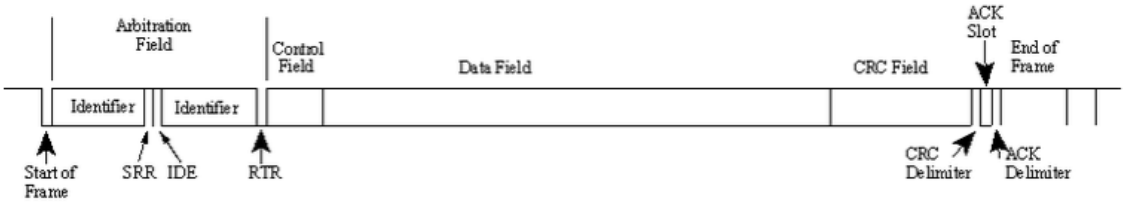
Existují čtyři typy posílaných zpráv probrané níže.

2.3.1 Data Frame

Datový rámec je nejběžnější typ zprávy, jehož struktura je uvedena dále (Obrázek 5). Po Start-bitu je vyslán identifikátor, který obsahuje buď 11 bitů (CAN 2.0A) nebo 29 bitů (CAN 2.0B), oddělených dvěma bity. Po identifikátoru následuje RTR bit, který indikuje, zda se jedná o vyslanou datovou zprávu nebo o požadavek na vyslání zprávy. Kontrolní část má 6 bitů, přičemž první dva jsou vždy nulové a zbývající 4 určují délku datové části v bajtech, která je omezena na maximální počet 8 bajtů. Po datové části následuje kontrolní suma a dále „ACK slot“, který slouží jako potvrzení přijetí zprávy alespoň jedním jiným zařízením.



A CAN 2.0A ("standard CAN") Data Frame.

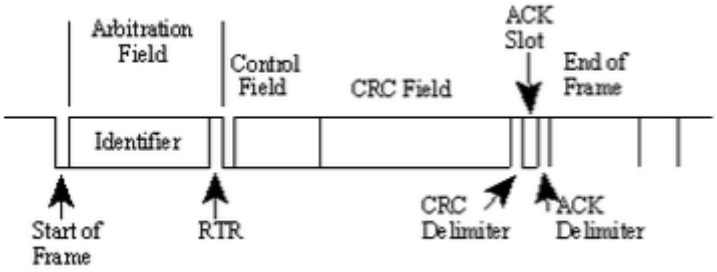


A CAN 2.0B ("extended CAN") Data Frame.

Obrázek 5 – CAN Data Frame (Kvaser Inc., 2014)

2.3.2 Remote Frame

Tento rámec se od datového liší tím, že hodnota RTR bitu je „1" a rámec neobsahuje žádná data. Rámec se používá pro vynucení si datového rámce s patřičným identifikátorem. Výsledkem je tedy odpověď se stejným identifikátorem obsahující odpovídající data. RTR bit zde určuje, že datový rámec má přednost



Obrázek 6 – CAN Remote Frame (Kvaser Inc., 2014)

2.3.3 Error Frame

Při detekci chyby na sběrnici je vyslán chybový rámec. CAN protokol dokáže detekovat celkem pět různých typů chyb:

- Každý uzel při vysílání zároveň skenuje stav sběrnice, a pokud se na sběrnici objeví jiný než požadovaný bit, je signalizována bitová chyba. Toto neplatí pro „Arbitration Field".

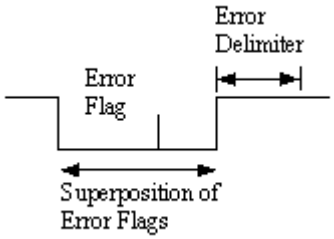
- Při porušení plnění bity, tedy detekcí více než pěti po sobě jdoucích stejných bitů.

- Některé části CAN zprávy mají pevný formát (CRC delimiter, ACK delimiter atd.), při jehož porušení vzniká další typ chyby.

- Pokud uzel na konci svého vysílání nezaznamená aktivaci ACK slotu, nastává ACK chyba.

- Posledním typem chyby je chyba CRC součtu.

Chybový rámce se skládá ze dvou částí dle obrázku níže (Obrázek 7). První část v případě aktivního chybového příznaku obsahuje 6 až 12 dominantních bitů (pasivní chybový příznak používá nedominantní bity). Při vyslání chybového rámce je tedy narušeno plnění bity. Ostatní zařízení mohou zaznamenat určitou chybu ve stejný moment, nebo v důsledku vysílaní chybového rámce zaznamenají jinou chybu, v krajním případě právě porušení plnění bity, na což zareagují dalším vysláním chybového rámce.



Obrázek 7 – CAN Error Frame (Kvaser Inc., 2014)

2.3.4 Overload Frame

Rámec přetížení je svou strukturou podobný chybovému rámci, kdy jeho první část obsahuje příznak přetížení a druhá část oddělovač. Jak název napovídá, tento rámec je vyslán, pokud je zařízení přetíženo. V dnešní době jsou ovšem CAN ovladače natolik chytré, že se tento rámec téměř nepoužívá.

Fyzická vrstva může být realizována různými způsoby, jako zřejmě nejznámější standardem ISO 11898-2, dále ISO 11898-3, či SAE J2411. (Kvaser Inc., 2014)

2.4 CANopen

Jak bylo zmíněno, výše popsaný protokol CAN definuje pouze linkovou a fyzickou vrstvu. Pro aplikaci je ovšem potřeba definovat, jak se výše zmíněné CAN rámce použijí, to tedy znamená definovat aplikační vrstvu. Pro tento účel a daný protokol bylo vyvinuto několik aplikačních protokolů jako Isobus, DeviceNet, NMEA 2000 a v neposlední řadě právě CANopen.

CANopen vychází z protokolu CAL (CAN application layer), přičemž využívá jeho služeb, ale navíc je i konkretizuje a dává jim určitý význam. Celkový popis daného protokolu, včetně vysvětlení všech jeho funkcí a principů by byl relativně rozsáhlý, proto se zde spokojíme

pouze se základními myšlenkami CANopen. Při větším zájmu o tento protokol je možno použít literaturu (Boterenbrood, 2000) nebo (Pfeiffer, a další, 2008), ze kterých bylo čerpáno pro stručný popis uvedený níže.

2.4.1 Object Dictionary

Hlavním konceptem CANopen protokolu je využití tzv. Object Dictionary (OD), což lze přeložit jako slovník objektů. Jedná se o uspořádanou skupinu objektů, kde každý je adresovatelný 16bitovým indexem, navíc je zde k dispozici 8bitový subindex, kterým lze přistupovat k jednotlivým elementům daného objektu. Pro každé zařízení na sběrnici je definován slovník objektů popisující jeho chování a parametry. Vlastní slovník objektů musí mít nějak definovanou svou podobu, aby bylo jasné, jaké typy objektů se nachází pod danými indexy. Je tedy třeba každému objektu ve slovníku nejdříve definovat jeho parametry, mezi které patří jeho funkce, jméno, datový typ, možnosti čtení/zápisu a podobně. Tohoto popisu daného slovníku objektů je dosaženo pomocí tzv. Electronic Data Sheet (EDS) v případě popisu obecných zařízení a v případě parametrů konkrétního jednoho zařízení pomocí tzv. Device Configuration File (DCF).

Slovník objektů tvoří určité přemostění mezi komunikačními objekty (níže) a chováním daného zařízení. Při nějakém požadavku na dané zařízení se tedy typicky pomocí komunikačních objektů nastaví hodnota určitého objektu ve slovníku objektů.

2.4.2 CANopen komunikace

Komunikace dle CANopen je řešena pomocí tzv. komunikačních objektů. Jedná se ve své podstatě o zprávy, rozdělené do několika kategorií, kdy každé kategorii je přiřazena určitá priorita. Ony priority jsou definovány prvními 4 bity arbitračního pole použitého CAN protokolu, dalších 7 bitů je bráno jako tzv. Node ID, definující unikátní ID daného zařízení. Všimněme si, že CANopen je definován pro standard CAN 2.0A, který používá 11bitové arbitrační pole.

2.4.3 Service Data Object (SDO)

Tento komunikační objekt se používá pro přímý přístup do slovníku objektů. Je založen na metodě klient-server, kdy zařízení, jehož slovník objektů nás zajímá, se chová jako server. Komunikace vždy probíhá s potvrzením přijatého SDO, proto existují celkem čtyři typy těchto zpráv a to:

- vyvolání SDO uploadu (žádost o data ze serveru),
- uploadování SDO segmentu (poslání požadovaných dat ze serveru),
- vyvolání SDO downloadu (poslání požadovaných dat serveru),
- a downloadování SDO segmentu (potvrzení příjmu dat).

Pokud se konkrétní data nevejdou do jednoho rámce, poskytuje SDO metodu pro rozdělení dat do více rámců.

2.4.4 Process Data Object (PDO)

PDO je používán pro přenos dat v reálném čase. Data jsou generována právě jedním zařízením, přičemž mohou být přijímána jedním či více zařízeními. Délka dat je na rozdíl od SDO limitována pouze jedním rámcem, čili 8 datovými bajty. PDO zprávy se nijak nepotvrzují. Před použitím těchto zpráv je nutné vytvořit ve slovníku objektů tzv. mapování PDO, což definuje význam dat v jednotlivých PDO vzhledem k objektům ve slovníku. Generace těchto zpráv může být řízena způsoby jako:

- změna stavu přiřazeného objektu ve slovníku,
- na vyžádání,
- periodicky s určitým časovým intervalem,
- a synchronně na základě „SYNC“ komunikačního objektu.

2.4.5 Další komunikační objekty

Výše byly popsány dva základní komunikační objekty používané v CANopen protokolu, nicméně existuje i několik dalších plnících různé funkce. Jedná se například o zmíněný SYNC objekt, sloužící pro synchronizaci událostí provedených danými zařízeními (například sejmutí dat ze senzorů všemi zařízeními ve stejný okamžik a jejich následné vyslání pomocí PDO). EMERGENCY objekt je používaný při výskytu vnitřních chyb zařízení. Dále můžeme uvést skupinu NETWORK MANAGEMENT objektů (NMT), do které patří zprávy řídící operační stav daných zařízení (při spuštění je zařízení v tzv. předoperačním stavu, ve kterém nemůže komunikovat pomocí PDO a čeká na uvedení do provozu právě pomocí NMT zprávy) nebo například NODE GUARDING, monitorující stav připojených zařízení. (Boterenbrood, 2000)

3 Navrhovaný protokol

O něco větší pozornost byla v předešlé kapitole věnována protokolu CAN a jeho aplikační nadstavbě CANopen. Mnou navrhovaný protokol se ukazuje jako velmi podobný těmto dvěma výše popsaným protokolům a tak bylo zvoleno i podobné názvosloví, což se týká například typu komunikačních zpráv (objektů) a podobně. Jednou z hlavních odlišností je ovšem větší provázanost linkové a aplikační vrstvy, kde struktura i délka rámců se výrazně liší v závislosti na typu vysílané zprávy. Toto je zapříčiněno zejména vývojem a optimalizací protokolu pro navrhovaný systém, přičemž CAN s CANopen jsou obecné protokoly pro průmyslové systémy, které je možné použít v mnoha různých odvětvích.

V dalším textu bude popsán návrh komunikačního protokolu. Jedná se zejména o linkovou a aplikační vrstvu. Fyzická vrstva bude popsána v kapitole „Společná sběrnice“.

3.1 Základní parametry

Jednou z prvních věcí při návrhu protokolu pro daný systém je vyřešit metodu přístupu na sdílené médium. V textu výše bylo již několik metod popsáno, ale samozřejmě jich existuje mnohem více[1]. Pro danou aplikaci jsem zvolil **metodu prioritního přístupu**, použitou ve výše popsaném CAN protokolu. Touto metodou je za určitých okolností možné data odeslat okamžitě, což je požadováno z důvodu rychlé odezvy akčních prvků na povely ovládacích prvků. Prvek před započetím vysílání ověří, zda neprobíhá vysílání někoho jiného. Pokud je sběrnice v klidu, což je reprezentováno nedominantními bity (zvolené jako logická „1“), začne s vysíláním zprávy. Při vysílání daný prvek zároveň skenuje stav sběrnice, a pokud se objeví neočekávaný agresivní bit (zvolen jako logická „0“), okamžitě přeruší své vysílání.

Generované zprávy nebudou synchronizovány žádným hodinovým kmitočtem. Pokud bude chtít daný prvek uskutečnit přenos, vyšle na sběrnici „Start-bit“, označující začátek rámce, za nímž bude následovat požadovaná zpráva. Z důvodu absence synchronizačního signálu je použito plnění bity, což má i další výhodu týkající se určování chyb (popsáno v podkapitole Detekce chyb). Pokud se ve zprávě objeví 6 po sobě jdoucích stejných bitů, bude další vyslaný bit opačný a nebude se započítávat do celkové zprávy. Toto opatření tedy zajistí synchronizaci.

Dále je vhodné zmínit, že doba jednoho bitu byla zvolena 16 µs, což odpovídá rychlosti 62500 Baud.

3.2 Struktura rámců

Než se budou probrány běžně užívané rámce, představím zde tzv. Error frame, který je znázorněn na obrázku níže (Obrázek 8 – Error frame). Jedná je pouze o 10 po sobě jdoucích logických „0“.

[1] Řízení přístupu na společné médium řeší v ISO/OSI modelu linková vrstva. Lze využít dělení na deterministické a stochastické metody, případně na metody s centrálním řízením a metody distribuované. Více informací a konkrétní metody je možné dohledat v použité literatuře (Kurose, a další, 2013).

klid na sběrnici			Error frame = 10 bitů (160μs)										klid na sběrnici		
1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1

Obrázek 8 – Error frame

Na dalším obrázku (Obrázek 9 - Základní struktura rámce) je znázorněno obecné uspořádání vysílaných rámců, kde vždy v poli nahoře je popis daného bloku a dole je buď přímo hodnota, nebo délka daného bloku. Pokud je popis bloku v závorce, znamená to, že je volitelný a nenachází se ve všech typech zpráv.

začátek rámce	typ zprávy	(typ SDO)	adresa zdroje	(adresa cíle)	(parametry SDO)	(data)	CRC součet	CRC delimiter	ack bit	konec rámce
"0"	2b	(2b)	8b	(8b)	(4b)	(0-8B)	8b	"1"	1b	"01111111"

Obrázek 9 - Základní struktura rámce

Dále budou vysvětleny jednotlivé části bod po bodu.

- **Začátek rámce:** Každý rámec začíná jedním Start-bitem, jehož hodnota je samozřejmě logická „0“. V momentu příjmu Start-bitu zažínají ostatní prvky příjem.

- **Typ zprávy:** Toto pole je dlouhé 2 bity a jeho hodnota může tedy nabývat 4 různých hodnot. Díky použité metodě přístupu na sdílené médium se priorita odvysílaného rámce dělí nejdříve právě dle typu zprávy. Prvním typem, s hodnotou „00“ je tzv. Emergency zpráva, dále s nižší prioritou („01“) zpráva PDO, s ještě nižší prioritou („10“) SDO a nakonec Heartbeat („11“). Všechny tyto zprávy budou podrobněji rozebrány v textu níže.

- **Typ SDO:** Pole je přítomno pouze v případě vysílání SDO zprávy a je dlouhé 2 bity, které dále určují typ SDO zprávy.

- **Adresa zdroje:** Pokud se stane, že dva, či více prvků začnou ve stejný okamžik vysílat stejný typ zprávy, bitovou arbitráž vyhraje samozřejmě ten s nižší adresou. Tímto polem veškerá soutěž končí a dále již probíhá vysílání jednoho jediného prvku. Obecně je vhodné, aby ovládací prvky měli přidělenu nižší adresu než prvky akční. Pole obsahuje 8 bitů, přičemž hodnota 0 je použita pro broadcast. Maximální počet prvků na sběrnici je tedy roven 255 s nejnižší možnou adresou 1.

- **Adresa cíle:** Toto pole je přítomno pouze u SDO zprávy a logicky obsahuje stejně jako pole předešlé 8 bitů.

- **Parametry SDO:** Pole je opět přítomno pouze v případě SDO zprávy a určuje její parametry.

- **Data:** Datové pole je také závisle na typu vysílané zprávy a může být různé délky. Například v případě Emergency zprávy je zde obsažen 8 bitů dlouhý kód určující typ nouzového stavu, zatímco při vysílání Heartbeat je toto pole prázdné.

- **CRC součet:** Od počátku vysílání určeného Start-bitem je každý vyslaný bit zahrnut do počítání CRC součtu. Toto pole je dlouhé 8 bitů a je obsaženo opět již v každém rámci, stejně jako všechna pole následující.

- **CRC delimiter:** Jeden bit obsahující logickou „1“ je zde z toho důvodu, aby měla přijímací strana čas dopočítat CRC součet a v případě nesouladu vyslat Error frame. Tímto polem dále přestává platit plnění bity.

- **Ack bit:** Tento potvrzující bit je vysílaný přijímací stranou v případě korektního přijmutí dané zprávy čímž vysílací strana dostává zpětnou vazbu.

- **Konec rámce:** Na úplném konci každého rámce musí být sběrnice v klidu (v logické „1“) alespoň po dobu 7 bitů, které obyčejně poruší plnění bity (6 bitů). V případě že by vysílací strana dále vysílala, předpokládá se, že bude dodržovat plnění bity a v nejhorším případě by nejpozději sedmý bit musela invertovat. Toho by si všimla přijímací strana a vyslala by Error frame. Tato podoba ukončení rámce dokáže tedy mimo jiné poznat, že vysílač skutečně přestal vysílat. Logická „0“ vyslaná ještě před posloupností „1“ je zde zejména ze synchronizačních důvodů.

Takto tedy vypadá obecná struktura rámců, přičemž jak již bylo zmíněno, existují čtyři typy zpráv rozdělených podle priority. Toto rozdělení mimo jiné zvýší efektivitu komunikace. Při provozu systému je totiž žádoucí, aby například příkazy nastavující polohu akčních prvků byly co nejkratší, z důvodu co možná nejnižší časové prodlevy a měli přednost například před zprávami informativního charakteru. Dále tedy budou popsány jednotlivé typy zpráv a struktura jejich rámců.

3.2.1 Emergency zpráva

Tato zpráva má nejvyšší prioritu a je vyslána v případě nouzové situace. V rámci je v datovém poli přenesen kód určující typ nouzového stavu. Tabulka možných kódu je uvedena níže (Tabulka 1 – Emergency kódy). Od kódu 21 jsou chybové stavy přídavných modulů. Pod těmito moduly si lze například v konkrétním případě ovladače motorků představit integrovaný obvod řídící krokové motorky.

Start bit	typ zprávy		adresa zdroje	emergency kód	CRC součet	CRC del	ack bit	konec rámce
0	0	0	(8b)	(8b)	(8b)	1	(1b)	01111111

Obrázek 10 – Emergency zpráva

Kód	**Jméno**
1	Obecná chyba
2	Napětí
3	Teplota
4	Proud
5	Hardware
..	..
21	Přídavný modul (PM) obecná chyba
22	PM napětí
23	PM teplota
24	PM proud
25	PM hardware
26	PM komunikace

Tabulka 1 – Emergency kódy

3.2.2 PDO zpráva

Nejzákladnějším požadavkem na daný systém je nastavení motorku do určité polohy. Intuitivně by příslušná zpráva obsahovala například mimo jiné adresu cílového prvku a hodnotu požadované polohy. Tento přístup je ovšem relativně omezující, jelikož je vyžadováno rychlé nastavení poloh více motorků (i u více prvků) naráz a odesílatel by musel každému prvku zvlášť poslat požadovanou polohu. Mohl by zde tedy vzniknout problém s časovou prodlevou dané akce a zahlcením sběrnice.

Lepším způsobem jak docílit výsledku je definovat u akčních prvků určité stavy, které budou reprezentovat polohu daných motorků. Konkrétní stav může pro každý motorek znamenat jinou hodnotu polohy. Takto tedy stačí odvysílat jednu jedinou zprávu obsahující pouze hodnotu stavu a každý prvek připojený na sběrnici se podle této zprávy nějakým předem definovaným způsobem zařídí a případně nastaví motorky do určité polohy.

Typ zprávy, který bude přenášet tyto hodnoty stavů, bude nazván PDO (Process Data Object). Podobný typ zprávy se stejným názvem je obsažen u CANopen protokolu, kde je zvolen ale obecnější přístup. Díky konkrétním požadavkům na navrhovaný systém jsou obsahem PDO pouze data reprezentující požadované stavy.

Ze struktury rámce znázorněné na obrázku níže (Obrázek 11 – PDO zpráva) je zřejmé, že priorita PDO je druhá nejvyšší hned po Emergency zprávě. Tato zpráva dále ze své podstaty

obsahuje pouze adresu zdroje, nikoli cíle a přenášená data. Poslední čtyři pole rámce jsou opět obdobné jako u ostatních typů zpráv.

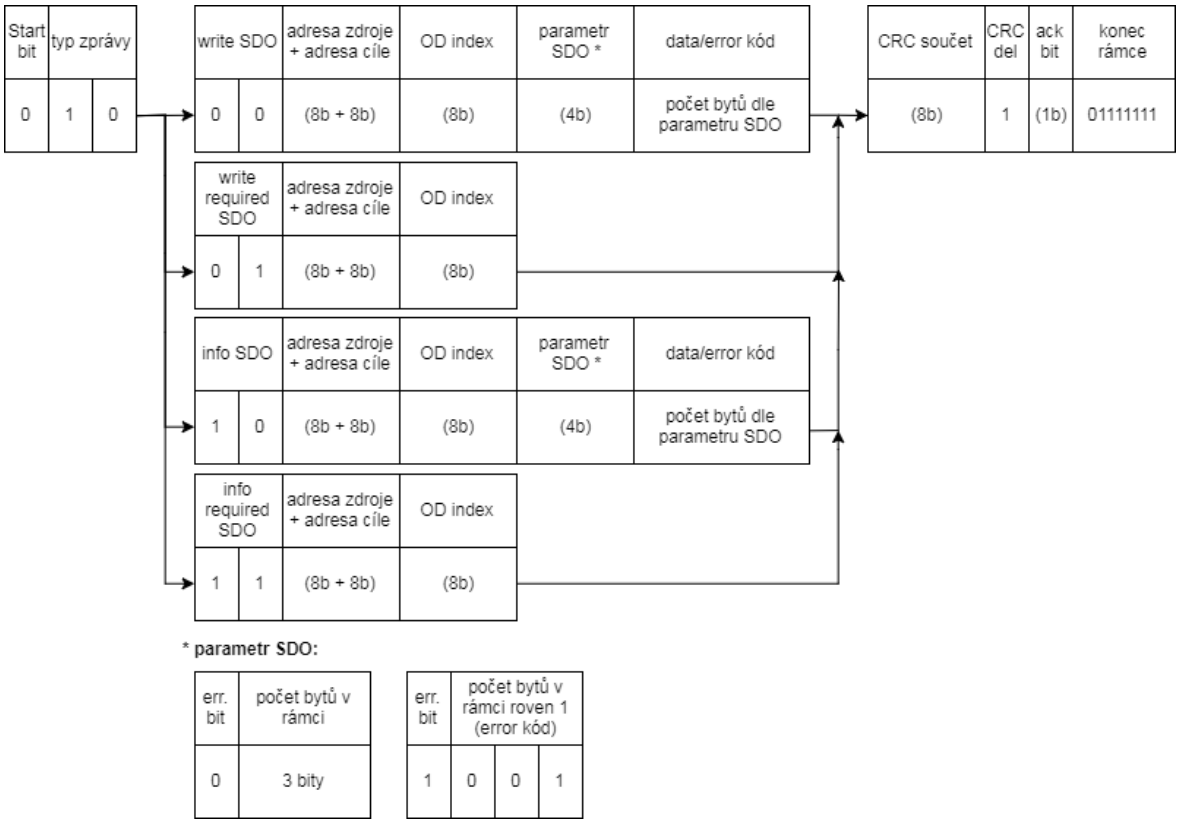
Start bit	typ zprávy		adresa zdroje	data	CRC součet	CRC del	ack bit	konec rámce
0	0	1	(8b)	(4B)	(8b)	1	(1b)	01111111

Obrázek 11 – PDO zpráva

3.2.3 SDO zpráva

Kromě výše popsaného způsobu nastavení motorků do určité polohy při provozu, je zde další požadavek na nastavení i jiných parametrů prvků připojených na sběrnici. Toto nastavení parametrů je realizováno pomocí SDO (Service Data Object) zprávy.

Již o trochu složitější strukturu daného typu zprávy zachycuje obrázek níže (Obrázek 12 – SDO zpráva), který zde bude rozebrán. Z obrázku je patrné, že délka a struktura SDO zprávy se typ od typu odlišuje.



Obrázek 12 – SDO zpráva

SDO zpráva se tedy dále dělí na čtyři typy a jelikož jsou příslušné reprezentující bity umístěny stále v bitové arbitráži, jsou tyto typy rozdělené podle priority na:

- **Write SDO:** Zpráva s nejvyšší prioritou určená k zapisování parametrů. Zprávu odesílá prvek, který chce nastavit parametr jinému prvku.
- **Write required SDO:** Tato SDO zpráva reprezentuje požadavek na zápis parametru.
- **Info SDO:** Touto zprávou daný prvek informuje o svých parametrech.
- **Info required SDO:** Toto je požadavek o sdělení informací o určitém parametru.

Další pole obsahuje již adresu odesílatele, při níž samozřejmě stále přetrvává bitová arbitráž a dále logicky následuje adresa příjemce.

Pole „OD index“ značí odkaz na požadovaný parametr, se kterým se má pracovat (více v podkapitole níže - Aplikační vrstva). V případě Write required SDO a Info required SDO dále následuje již klasické zakončení zprávy, jelikož nic dalšího než informace „od koho/komu?“ a „co?“ není potřeba.

Při bližším pohledu na zprávu Info SDO jsou v dalším poli dlouhém čtyři bity obsaženy parametry SDO zprávy. Prvním bitem je příznak chyby, přičemž v případě, že jeho hodnota je „0“, tak další tři bity určují délku datového pole v bajtech (hodnota „000“ značí délku 8 bajtů). Datové pole potom přenáší již samotnou hodnotu parametru. V případě, že je první bit parametru SDO roven „1“, znamená to, že došlo k nějaké chybě. Jelikož typ zprávy Info SDO slouží zejména jako odpověď na Write SDO, či Info required SDO (podkapitola Aplikační vrstva), může zde nastat situace, kdy se někdo pokusí zapsat například nesmyslná data, nebo data určená pouze pro čtení a podobně. Info SDO v tomto případě v parametru SDO nastaví err. bit (chybový bit) do „1“ a v datech přenese chybový kód. Tento kód je dlouhý 1 bajt a proto další tři bity parametru SDO mají pevně nastavenou hodnotu „001“.

Analogicky ke zprávě Info SDO funguje Write SDO, kdy přenášená data jsou samozřejmě ta, která se mají do daného prvku zapsat.

3.2.4 Heartbeat zpráva

Posledním typem přenášených zpráv je Heartbeat. Tato zpráva má jednoduché poslání a to informovat o přítomnosti daného prvku na sběrnici. Její priorita je tedy nejnižší. Jak si lze povšimnout na obrázku (Obrázek 13 – Heartbeat zpráva), struktura je velmi jednoduchá a zpráva krátká.

Start bit	typ zprávy		adresa zdroje	CRC součet	CRC del	ack bit	konec rámce
0	1	1	(8b)	(8b)	1	(1b)	01111111

Obrázek 13 – Heartbeat zpráva

3.3 Shrnutí Linkové vrstvy

Výše bylo popsáno řešení Linkové vrstvy pro daný protokol. Jednalo se o základní parametry, jako je bitový čas, plnění bity, struktura vysílaných rámců a podobně. Dále je vhodné doplnit ještě některé informace týkající se zejména procesu vysílání a přijímání zpráv.

Každá zpráva tedy začíná Start-bitem a končí posloupností sedmi logických „1“. Pokud se v jakémkoli okamžiku mezi těmito dvěma body objeví na sběrnici Error frame, považuje se daná zpráva za neplatnou. To znamená, že i v případě, že příjemce potvrdil příjem stažením Ack bitu do „0“, ale konec rámce by se odlišoval, vyšlou zařízení na sběrnici Error frame a zpráva přestává být validní.

Minimální rozestup mezi koncem jedné a začátkem další zprávy je 8 bitů, během kterých musí být sběrnice v klidovém stavu. Dohromady s koncem rámce to tedy dělá 15 po sobě jdoucích logických „1“. Při bitovém času 16µs po vynásobení dostaneme 240µs, což je doba, po kterou musí být sběrnice v klidu, aby byla další přijatá zpráva platná.

Co se procesu komunikace týče, jsou důležité i další parametry, týkající se například různých odpovědí na dané zprávy a tak dále. Tento mechanismus ale již nespadá do řešení Linkové vrstvy, a proto bude popsán v jedné z dalších kapitol (Aplikační vrstva).

3.4 Detekce chyb

Aby byl popis Linkové vrstvy protokolu kompletní, zbývá ještě vyřešit otázku detekce chyb a jejich následné zpracování. Z pohledu fungování systému jako celku mohou nastat obecně následující krizové situace:

- Prvek zaznamenal vnitřní chybu, o které chce informovat.
- Příjemce porozuměl zprávě, ale nemůže vyhovět.
- Příjemce neporozuměl odvysílané zprávě.

První případ je řešen již výše popsaným vysláním tzv. Emergency zprávy, která v sobě skrývá kód popisující chybu. Jedná se o využití rámce Linkové vrstvy vrstvou Aplikační, definující určené kódy (již popsány - Tabulka 1 – Emergency kódy).

Druhý případ řeší rámce typu Info SDO a Write SDO tím způsobem, že nastaví tzv. chybový bit do logické „1“ a v datovém poli je dále opět přenesen kód, který určuje, proč není možné provedení požadovaného příkazu. Tento proces je věcí Aplikační vrstvy a mechanismu přenosu SDO zpráv a odpovědí na ně.

Z pohledu Linkové vrstvy je stěžejní případ poslední, kdy v průběhu přenosu rámce došlo k nějaké chybě. K detekci chyb v přenosu rámců je tedy využito několik níže popsaných mechanismů:

- **Monitorování bitů:** Každý vysílající prvek současně skenuje stav sběrnice, přičemž pokud se bit vysílaný na sběrnici liší od skutečného stavu, je vyslán Error frame. Logicky může nastat pouze případ, kdy prvek vysílá logickou „1“, ale na sběrnici je „0“. Monitorování bitů však není aktivní v průběhu arbitráže, kdy v případě neshody prvek pouze přeruší vysílání a přejde na příjem.
- **Plnění bity:** Pokud je při přenosu rámce porušeno plnění bity, je příjemcem vyslán Error frame. Plnění bity přestává být v rámci aktivní po odvysílání CRC součtu.
- **Kontrola rámce:** V každém rámci jsou místa s předem určenými hodnotami bitů, jejichž rozdílná hodnota vede opět k odeslání Error frame. Jedná se o následující případy:
    - Start-bit („0“),
    - CRC delimiter („1“),
    - Konec rámce („01111111“),
    - Parametr SDO při přenosu Write/Info SDO, kdy je chybový bit roven „1“ (Potom má parametr hodnotu „1001“, jelikož se v datech přenáší pouze jeden bajt chybového kódu).
- **Kontrola CRC:** Od začátku příjmu počítá příjemce tzv. CRC součet. Pokud vypočítaná hodnota nesouhlasí s hodnotou přijatou, je vyslán Error frame.

3.4.1 CRC součet

Pro detekci špatně přijatých bitů je možno obecně použít několik metod. Mezi nejzákladnější patří například sudá, či lichá parita, používaná v různých typech jednoduché sériové komunikace. O něco sofistikovanější metodou je použití tzv. cyklického redundantního součtu. Základní myšlenka spočívá v tom, že si lze odeslanou posloupnost bitů představit jako jedno velké číslo. Toto číslo se posléze dělí jiným, pro všechny zúčastněné prvky předem známým číslem, přičemž zbytek po dělení je použit jako požadovaná kontrolní suma. Pokud přijatá kontrolní suma souhlasí s vypočtenou, je vše v pořádku.

Dané dělení je ovšem prováděno v aritmetice modulo 2. To například znamená, že sčítání i odčítání lze provést logickou funkcí XOR a nerealizuje se přenášení hodnot do vyšších řádů (binárně 1+1=0, nikoli 10). Tyto vlastnosti dané aritmetiky dovolují velice zjednodušit proces dělení pod sebe. Je možné si vystačit pouze s posuvným registrem, který je dle výsledku dílčího dělení známou hodnotou buď pouze doplněn jedním bitem hodnoty dělence zprava, nebo je ještě před doplněním navíc provedena operace XOR se onou známou hodnotou. Je vhodné poznamenat, že se v literatuře spíše pracuje s pojmem „polynom“. Posloupnost bitů může být právě jako polynom reprezentována, proto zde tedy dělíme polynomy. Mnohem podrobněji a zřetelněji je princip CRC kontrolního součtu popsán například v (Williams, 1993).

Ještě je dobré se pozastavit nad jedinou věcí, kterou lze parametrizovat CRC algoritmus a tou je podoba dělitele. Obecně je dělitel nazýván pouze „polynomem“. V konkrétním případě navrhovaného systému, kdy je zbytek po dělení dlouhý 8 bitů, je tedy zvolen polynom 9. řádu. Přesná podoba polynomu se volí dle charakteru datového přenosu v závislosti na pravděpodobnostech výskytu bitových chyb. V tomto konkrétním případě budou uvažovány konstantní a navzájem nezávislé pravděpodobnosti výskytu bitových chyb. Proto bude použit jeden z obecně doporučených polynomů. (Koopman, 2015)

3.5 Aplikační vrstva

Jedním z úkolů aplikační vrstvy je definovat pravidla přenosu daných rámců. Tabulka níže (Tabulka 2 – Potvrzování zpráv) zachycuje mechanismus odesílání zpráv a odpovědí na ně. Lze si povšimnout, že jediné zprávy vyžadující odpověď jsou Write SDO, Write required SDO a Info required SDO. Při odvysílání ostatních typů zpráv se na ně neočekává žádná odpověď příslušného prvku. Pro zaslání odpovědi není určen žádný časový limit, a odpověď je odeslána až po zpracování přijaté zprávy. Jako příklad lze uvést nastavení polohy motorku pomocí Write SDO. Příjemce danou zprávu nejdříve zpracuje, což znamená, že nastaví například daný motorek do požadované polohy, a teprve po tomto odešle zpět odesílateli zprávu Info SDO. Celý proces tedy může zabrat poměrně dlouhou dobu, ve které mohlo na sběrnici dojít k další komunikaci. Pokud bylo prvku adresováno v tomto čase více zpráv, bude reagovat pouze na poslední přijatou.

Tabulka 2 – Potvrzování zpráv

Odeslaná zpráva	**Odpověď**
Emergency	-
PDO	-
Write SDO	Info SDO
Write required SDO	Write SDO
Info SDO	-
Info required SDO	Info SDO
Heartbeat	-

Z hlediska významu přenášených dat je zde využit koncept tzv. Object Dictionary (česky slovníku objektů, zkratka OD), použitého v protokolech jako CANopen, Profibus nebo Interbus-S. Jedná se o uspořádanou skupinu objektů (parametrů, informací apod.), kde každý objekt je reprezentován svým indexem. Při popisu zprávy typu SDO (podkapitola SDO zpráva) bylo zmíněno pole s názvem „Index“. Toto pole obsahuje 8 bitů a proto celkový počet možných objektů ve slovníku je 256. Dále je známa maximální délka dat v SDO rámci, která je 8 bajtů. Jelikož navrhovaný protokol nepodporuje na rozdíl od CANopen více rámcový přenos, je i maximální objem dat jednotlivých objektů právě 8 bajtů.

Počet povinných objektů ve slovníku byl navržen minimální možný (Tabulka 3 – Povinné objekty). Jak vidíme, každému objektu přísluší index, název, délka v bajtech, datový typ a přístup k objektu, který může být pouze pro čtení (RO), nebo pro čtení i zápis (R/W).

Význam prvního objektu je patrný z názvu a reprezentuje jméno daného prvku. Dle objektu „Typ prvku“ je odvozena další podoba slovníku objektů a tedy i možné chování. Pro navrhovaný systém jsou definovány tedy tři možné typy prvků a to:

1) Ovladač motorků,
2) Nožní přepínač,
3) Bluetooth modul.

Dalším povinným objektem je „Adresa prvku“, která musí být pochopitelně pro každý prvek unikátní. Jako poslední je v tabulce uveden „Heartbeat čas“, který v desítkách milisekund definuje periodu vysílání Heartbeat zprávy.

Tabulka 3 – Povinné objekty

Index	Název objektu	Počet bajtů	Datový typ	Přístup
1	Jméno	8	char	R/W
2	Typ prvku	1	uint8_t	RO
3	Adresa prvku	1	uint8_t	R/W
4	Heartbeat čas	1	uint8_t	R/W

V další tabulce níže (Tabulka 4 – Výběr některých objektů prvku typu „Ovladač motorů“) jsou uvedeny některé důležité objekty Ovladače motorků. Celý slovník objektů mimo jiné samozřejmě obsahuje i výše popsané povinné prvky na nižších adresách. Jako první v tabulce jsou tři objekty nesoucí informace o poloze motorků. Tyto objekty je možné číst i do nich zapisovat, což vede k nastavení motorků do požadované polohy. Počet kroků udává rozsah polohování motorků. Ovladač motorků bude dále podporovat několik stavů, které budou definovat polohu motorků v závislosti na přijatém PDO. Objekt „Stav 1 PDO“ tedy charakterizuje hodnotu přijatého PDO k dosažení daného stavu, přičemž další objekt „Stav 1 parametry“ v prvních třech bajtech definuje polohu každého motorku. Čtvrtý bajt je využit jako maska, určující, kterých motorků se bude polohování týkat. Je tedy možné změnit polohu jen u některých motorků.

Je vhodné, aby prvek nožního přepínače obsahoval i pedál, pomocí něhož by bylo možné plynule ovládat polohu motorků. Toto bude prováděno opět pomocí PDO zpráv. Aby Ovladač motorků správně reagoval na přijaté PDO zprávy musí mít ke každé přiřazen určitý stav. Zde se ale vyskytuje problém, jelikož různých PDO zpráv může být poměrně veliké množství. Při použití způsobu popsaného v předešlém odstavci by nakonec nestačil celý Slovník objektů. Proto je definován pouze určitý rozsah hodnot PDO, kterému náleží jiný rozsah poloh motorků (objekty „Rozsah stavů 1 PDO“ a „Rozsah stavů 1 parametry“). V prvním objektu je v prvních čtyřech bajtech definováno minimální přijaté PDO a v dalších čtyřech bajtech maximální přijaté PDO. Druhý objekt v prvních třech bajtech definuje minimální hodnotu poloh motorků a pochopitelně v dalších třech bajtech maximální hodnotu

poloh. Sedmý bajt je použit jako maska, která opět definuje, kterých motorků se polohování týká.

Poslední zmíněné objekty v tabulce slouží pro uložení současných poloh motorků do určitých stavů, případně do rozsahů stavů.

Tabulka 4 – Výběr některých objektů prvku typu „Ovladač motorů“

Index	Název objektu	Počet bajtů	Datový typ	Přístup
51	Poloha 1	1	uint8_t	R/W
52	Poloha 2	1	uint8_t	R/W
53	Poloha 3	1	uint8_t	R/W
54	Počet kroků 1	1	uint8_t	RO
55	Počet kroků 2	1	uint8_t	RO
56	Počet kroků 3	1	uint8_t	RO
…	…	…	…	…
61	Stav 1 PDO	4	uint8_t	R/W
62	Stav 1 parametry	4	uint8_t	R/W
…	…	…	…	…
81	Rozsah stavů 1 PDO	8	uint8_t	R/W
82	Rozsah stavů 1 parametry	7	uint8_t	R/W
…	…	…	…	…
101	Ulož současné polohy	6	uint8_t	R/W
102	Ulož současné polohy jako minimum	6	uint8_t	R/W
103	Ulož současné polohy jako maximum	6	uint8_t	R/W

V další tabulce (Tabulka 5 - Výběr některých objektů prvku typu „Nožní přepínač“) jsou na prvních místech definovány hodnoty PDO, které se vyšlou po stisku daného tlačítka na nožním přepínači. Objekt „Pedál PDO minimální“ definuje minimální PDO vyslané při minimální poloze pedálu. Při polohování pedálu budou dále odesílány vyšší a vyšší hodnoty PDO, až do maximální hodnoty dané součtem hodnoty minimální a hodnoty v objektu „Pedál PDO rozsah“. Prvek typu „Bluetooth modul“ nemá ve svém Slovníku objektů žádné další vstupy, krom těch povinných, definovaných v tabulce výše (Tabulka 3 – Povinné objekty).

Tabulka 5 - Výběr některých objektů prvku typu „Nožní přepínač“

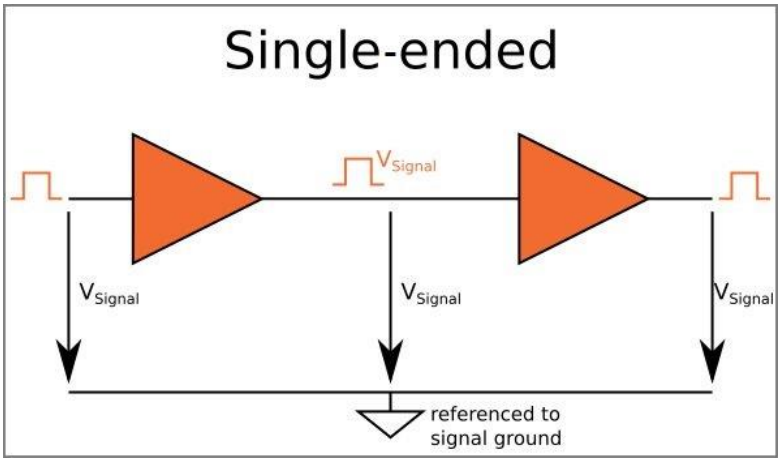
Index	Název objektu	Počet bajtů	Datový typ	Přístup
151	Tlačítko 1 PDO	4	uint8_t	R/W
152	Tlačítko 2 PDO	4	uint8_t	R/W
153	Tlačítko 3 PDO	4	uint8_t	R/W
…	…	…	…	…
161	Pedál PDO minimální	4	uint8_t	R/W
162	Pedál PDO rozsah	1	uint8_t	RO

4 Společná sběrnice – trendy a příklady

V této kapitole budou nejdříve popsány některé základní přístupy a metody používané ve fyzických vrstvách sériových komunikačních protokolů. Dále zde budou uvedeny některé již používané konkrétní sběrnice.

4.1 Signalizace se společnou zemí a diferenciální signalizace

Signalizace se společnou zemí je běžná metoda přenosu elektrického signálu. Signál je reprezentován napětím vztaženým k zemi. Jeden vodič slouží právě pro přenos napětí, zatímco druhý funguje jako referenční zem. Proud generovaný signálem prochází signálovým vodičem k příjemci, od něhož se vrací zpět po zemním vodiči. V případě přenosu více signálů nám stačí pouze jeden společný zemní vodič. Situaci názorně zobrazuje obrázek níže (Obrázek 14 – Signalizace se společnou zemí).



Obrázek 14 – Signalizace se společnou zemí (Pinkle, 2016)

Diferenciální signalizace (Obrázek 15 – Diferenciální signalizace) obsahuje dva doplňující se napěťové signály použité pro přenos jednoho informačního signálu. Druhý napěťový signál je oproti prvnímu invertovaný. Příjemce detekuje rozdíl mezi prvním a druhým (invertovaným) napěťovým signálem, z čehož určí signál informační. Proudy generované napěťovými signály jsou tedy též vzájemně invertované a ve výsledku se vyruší, proto zemním spojením (které tedy není nutné) neteče v ideálním případě žádný proud.

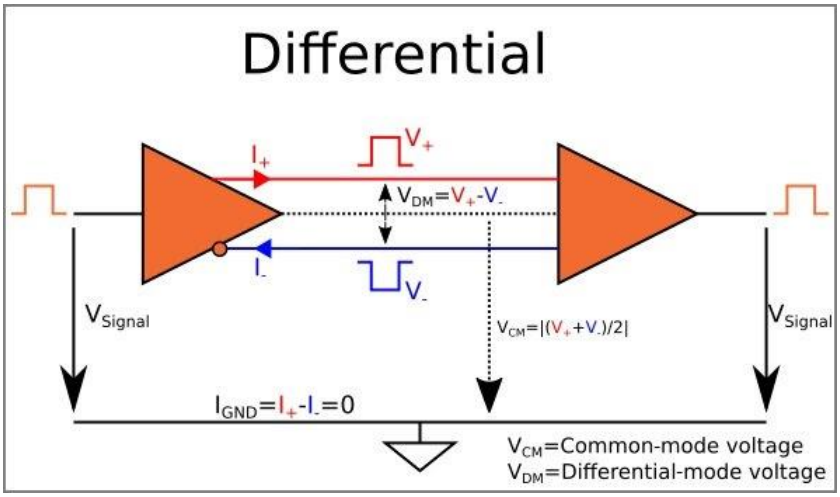
V případě přenosu více signálu je tedy logicky nutný dvojnásobný počet vodičů. Navíc je v tomto případě vhodné využít i jeden zemní vodič. Tato, dá se říci jediná nevýhoda, je oproti signalizaci se společnou zemí kompenzována několika lepšími vlastnostmi:

- **Žádný zpětný proud –** Pokud zpět k odesílateli neputuje žádný proud, zemní potenciál přestává být podstatný a může být rozdílný pro oba komunikační prvky. Nicméně stejně je vhodné využít zemní spojení pro zajištění toho, aby souhlasné signálové napětí bylo pro oba prvky v akceptovatelných mezích.

- **Odolnost proti elektromagnetické interferenci (EMI) a přeslechům –** Pokud je při přenosu přítomno nějaké rušení, předpokládá se, že se na oba signály naindukuje stejná hodnota napětí. Jelikož příjemce vyhodnocuje rozdílové napětí a souhlasné ignoruje, tak se v ideálním jakékoli rušení potlačí.

- **Redukce vyzařované EMI –** Strmé náběžné a sestupné hrany digitálních signálů generují rušení. V případě diferenciální signalizace jsou ovšem oba generované elektromagnetické signály shodné v amplitudě ale opačné v polaritě. Při dodržení těsné blízkosti mezi oběma vodiči (například v kroucené dvojlince) se velká část generovaného elektromagnetického pole vzájemně vyruší.

- **Možnost pracovat s nižším napětím –** Signalizace se společnou zemí musí používat relativně vyšší napětí (typicky 3,3 V nebo 5V) pro udržení adekvátního poměru signálu k šumu (SNR). Při udržení stejného SNR může diferenciální signalizace díky vlastnostem popsaným výše použít pro komunikaci nižší napětí. To vede k redukci vyzařovaného EMI, nižší spotřebě energie a možnosti použití vyšších frekvencí (Pinkle, 2016).

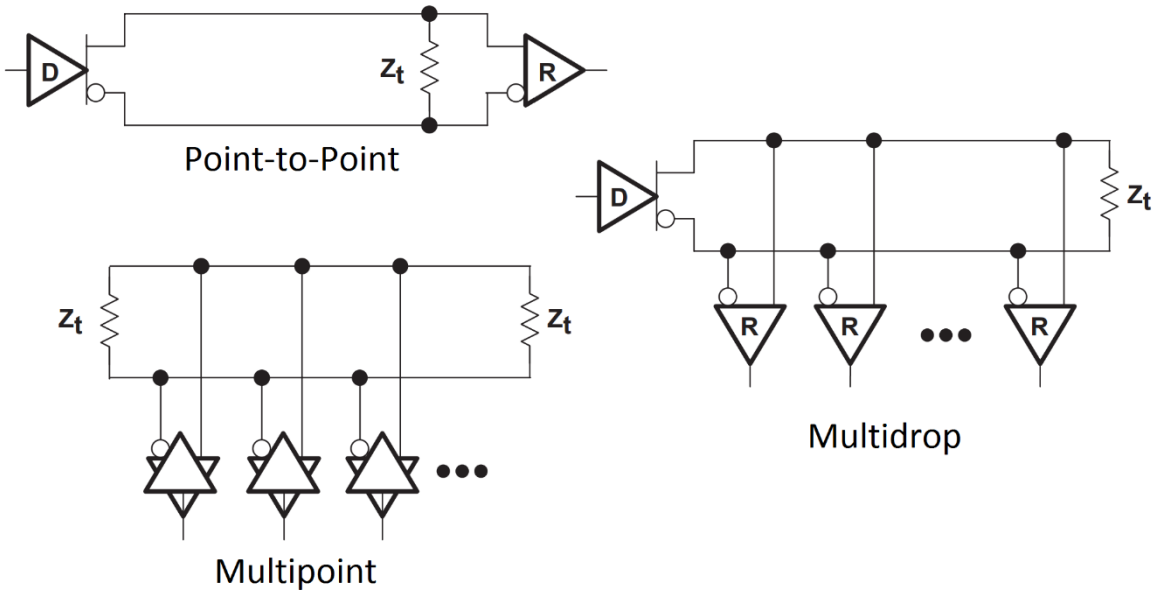


Obrázek 15 – Diferenciální signalizace (Pinkle, 2016)

Výše popsané způsoby signalizace se samozřejmě nevztahují pouze k digitálním signálům, ale běžně se výhod diferenciální signalizace používá například při kvalitním přenosu zvukových analogových signálů.

4.2 Základní schémata pro sériovou komunikaci

Na obrázku níže (Obrázek 16 – Sériové topologie) jsou znázorněny topologie používané v sériových sběrnicích.



Obrázek 16 – Sériové topologie (Peffers, 2013)

4.2.1 Point to point

Toto zapojení se skládá z jednoho vysílače a jednoho přijímače spojeného přenosovým médiem, kde je na konci signál terminován. Z pohledu kvality signálu představuje toto zapojení optimální volbu a umožňuje nejvyšší možné přenosové rychlosti. Nevýhodou je pouze jednosměrný tok dat.

4.2.2 Multidrop

Další možností je konfigurace jednoho vysílače a více přijímačů na sběrnici, kde terminace signálu probíhá opět na konci linky. Tímto způsobem se data odesílají opět jednosměrně, ale je možný příjem více prvky.

4.2.3 Multipoint

V této konfiguraci jsou na sběrnici navíc připojeny alespoň dva nebo více vysílačů. To zapříčiňuje možné konflikty ve vysílání, které nemusí být řešeny u předchozích topologií. Multipoint konfigurace tedy umožňuje obousměrnou poloduplexní komunikaci na jedné společné lince. V tomto případě je nutná terminace signálu na obou koncích dané linky.

4.3 RS232, RS423, RS422, RS485

V tabulce níže (Tabulka 6 – Přehled vlastností RS232, RS423, RS422 a RS485) jsou shrnuty vlastnosti daných sériových protokolů. Lze si například povšimnout, že protokoly RS422 a RS485 používají diferenciální signalizaci, na rozdíl od signalizace se společnou zemí používanou RS232 a RS423, což samozřejmě vede k možnosti použití vyšších přenosových rychlostí. Dále protokol RS232 jako jediný používá Point to point topologii, čili je zde možný pouze jeden odesílatel a jeden příjemce. Protokoly RS423 a RS422 používají Multidrop, což umožnuje přítomnost jednoho odesílatele ale už více příjemců. Protokol

RS485 se dá tedy považovat za nejpokročilejší díky použité diferenciální signalizaci a zároveň topologii Multipoint, umožňující přítomnost více odesílatelů.

Tabulka 6 – Přehled vlastností RS232, RS423, RS422 a RS485 (Lammert, 2015)

	RS232	**RS423**	**RS422**	**RS485**
Diferenciální signalizace	ne	ne	ano	ano
Max. počet vysílačů Max. počet přijímačů	1 1	1 10	1 10	32 32
Druh spojení	plný duplex, poloviční duplex	poloviční duplex	poloviční duplex	poloviční duplex
Topologie	point-to-point	multidrop	multidrop	multipoint
Max. vzdálenost	15 m	1200 m	1200 m	1200 m
Max. rychlost při 12 m Max. rychlost při 1200 m	20 kbs (1 kbs)	100 kbs 1 kbs	10 Mbs 100 kbs	35 Mbs 100 kbs
Max. rychlost přeběhu	30 V/μs	nastavitelná	n/a	n/a
Vstupní impedance přijímače	3..7 kΩ	≧ 4 kΩ	≧ 4 kΩ	≧ 12 kΩ
Impedance zatížení vysílače	3..7 kΩ	≧ 450 Ω	100 Ω	54 Ω
Vstupní citlivost přijímače	±3 V	±200 mV	±200 mV	±200 mV
Vstupní rozsah napětí přijímače	±15 V	±12 V	±10 V	–7..12 V
Max. výstupní napětí vysílače	±25 V	±6 V	±6 V	–7..12 V
Min. výstupní napětí vysílače (při zatížení)	±5 V	±3.6 V	±2.0 V	±1.5 V

4.4 CAN dle ISO 11898-2

O protokolu CAN bylo již pojednáno v kapitole věnující se návrhu linkové vrstvy (CAN). Nyní budou zmíněny některé doplňující parametry nejvíce rozšířené fyzické vrstvy, tzv. High speed CAN, definované standardem ISO 11898-2.

Sběrnice je na první pohled podobná RS485, jelikož využívá diferenciální signalizace a multipoint topologie. Datová propustnost je definována po krocích až do 1 Mbps, přičemž maximální délka na této rychlosti činí 40 m. Standard definuje dva vodiče, tzv. CAN_H a CAN_L, které logicky přenášejí vzájemně invertované signály. Souhlasné napětí se pohybuje od -2 V na CAN_L do +7 V na CAN_H. Při nedominantním bitu jsou typicky CAN_H i CAN_L shodné a mají hodnotu 2,5 V. Při dominantním bitu je potom CAN_H 3,5 V a CAN_L 1,5 V. Charakteristická impedance sběrnice činí dle standardu 120 Ω (splňuje kroucená dvojlinka), přičemž je sběrnice na obou koncích zakončena terminačními

rezistory s hodnotou pochopitelně také 120 Ω. Nominální průchozí zpoždění je specifikováno na 5 ns/m. (Pfeiffer, a další, 2008)

Co se týče CAN protokolu obecně, je zde vhodné ještě zmínit alespoň princip metody synchronizace. Každý bit je zde rozdělen na tzv. časová kvanta, kde určitý počet kvant definuje různé segmenty bitu (včetně okamžiku vzorkování). Dále je znám nominální bitový čas a jakýkoli příjem jedné logické úrovně musí být jeho celistvým násobkem. Pokud tedy příjem neproběhne přesně dle představ, může si příjemce bitový čas patřičně upravit.

5 Podoba společné sběrnice

V této kapitole provedu návrh fyzické vrstvy protokolu odpovídající řešenému problému.

5.1 Základní parametry

Navrhovaná sběrnice bude v mnoha ohledech téměř shodná s výše popsaným standardem ISO 11898-2. Pro upřesnění tedy bude použita diferenciální signalizace a s ohledem na požadavky samozřejmě multipoint topologie (dá se považovat obecně za sběrnicovou topologii sítě). Charakteristická impedance bude opět 120 Ω, přičemž budou na obou koncích použity terminační rezistory 120 Ω. Napěťové hodnoty signálu jsou opět obdobné, a to v klidu oba signály 2,5 V a při dominantní logické „0“ první 1,5 V a druhý 3,5 V. Stejně tak průchozí zpoždění bude definováno shodně a to 5 ns/m. Dále si je dobré si připomenout, že byla zvolena bitová rychlost 62500 Baud, což odpovídá bitovému času 16 µs.

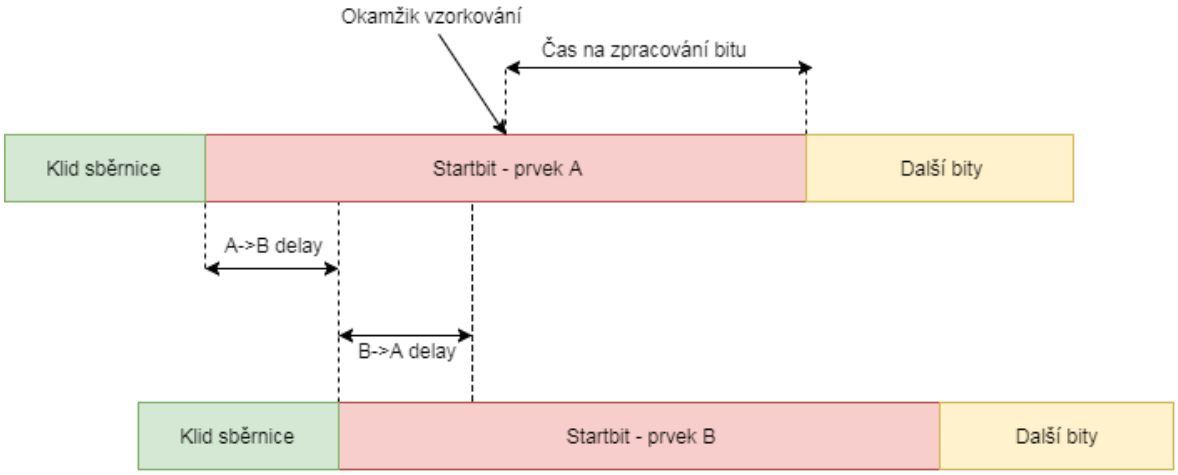
5.2 Bitové časování a synchronizace

Aby se každý prvek shodnul na hodnotě právě vysílaného bitu na sběrnici, je nutno zajistit synchronizaci a definovat okamžik vzorkování bitové hodnoty. Tento okamžik byl stanoven na 8 µs (což je polovina bitového času) a zajišťuje dostatek času na zpracování přijatého bitu. Synchronizace zde oproti protokolu CAN probíhá jednodušeji, bez případné adaptace nominálního bitového času příjemcem. Způsobem je takový, že přijímač používá vlastní časovač, který při dosažení poloviny bitového času způsobí vzorkování sběrnice. Hodnota časovače je přitom při každém přechodu z jedné logické úrovně do druhé opravena. Z důvodů synchronizace je tedy zavedeno plnění bity, který zajišťuje, aby i při dlouhém sledu stejných logických úrovní po sobě nedošlo k chybnému vzorkování. Předpokládá se, že alespoň po dobu šesti bitů bude časovač fungovat spolehlivě.

Dále se bude řešena otázka dopadu průchozího zpoždění a jeho vlivu na maximální délku sběrnice.

Na obrázku níže (Obrázek 17 – Bitové časování) je znázorněn případ, kdy dva prvky začínají s vysíláním. Bude uvažován nejhorší možný scénář, a to takový, že se oba nachází na opačných koncích sběrnice. Prvek „A“ vyšle Start-bit, který se k prvku „B“ dostane až po uplynutí průchozího zpoždění. V ten samý okamžik ale prvek „B“ začne s vysíláním, jelikož z jeho pohledu byla sběrnice v klidovém stavu. V tuto chvíli je všude na sběrnici samozřejmě logická „0“, ale obecně stejné zpoždění nastane i při vysílání z „B“ do „A“. Pro správné fungování sběrnice je důležité, aby vzorkování bitů probíhalo až po uplynutí těchto dvou zpoždění. Problém by nastal například v případě, kdyby dále ve vysílání prvek „A“ vysílal logickou „1“ a prvek „B“ logickou „0“ (předchozí bit by musel být u obou „1“). Pokud by skenování sběrnice[2] u prvku „A“ proběhlo dříve, než by dorazila „0“, každý z prvků by vyhodnotil stav sběrnice odlišně.

[2] Každý prvek při vysílání současně skenuje stav sběrnice, viz. podkapitola Detekce chyb.



Obrázek 17 – Bitové časování

Co se týče zmíněných zpoždění, je třeba kromě průchozího, způsobeného délkou vedení, vzít v potaz i jiná. Jedná se o zpoždění mezi okamžikem požadavku procesoru na překlopení sběrnice a jejím skutečným překlopením vykonaným příslušným hardwarem. Dále naopak o prodlevu mezi skutečným fyzickým překlopením sběrnice a zaznamenáním dané změny procesorem. První skutečnost bude označena jako τ_{mcu_bus}, druhá jako τ_{bus_mcu} a bude předpoklad, že prvky „A“ a „B“ pracují se stejným zpožděním (do těchto zpoždění budou započítány i délky náběžných a sestupných hran). Dále bude celkové zpoždění od začátku vysílání prvku „A“ až do příjmu bitu vyslaného prvkem „B“ označeno jako τ_{total} a nakonec průchozí zpoždění jako $\tau_{propagation}$. Pro hodnotu celkového zpoždění tedy platí následující rovnice (1):

$$\tau_{total} = 2\tau_{mcu\_bus} + 2\tau_{bus\_mcu} + 2\tau_{propagation}. \quad (1)$$

Pro navrhovanou sběrnici bude vznesen požadavek na maximální hodnoty τ_{mcu_bus} a τ_{bus_mcu} následovně (2):

$$\tau_{max\_IO} = \tau_{max\_mcu\_bus} = \tau_{max\_bus\_mcu} \leq 800[\text{ns}], \quad (2)$$

kde τ_{max_IO} je zpoždění mezi akcí v mikroprocesoru a akcí na sběrnici.

Aby byla tedy zajištěna funkčnost, musí být τ_{total} menší než okamžik vzorkování τ_{sample}. Lze tedy jednoduše dopočítat maximální délku sběrnice l_{max} s přihlédnutím na typické měrné průchozí zpoždění $D_{nom} = 5ns/m$ z následujících vzorců:

$$\tau_{total} = 4\tau_{max\_IO} + 2D_{nom}l_{max} < \tau_{sample}, \quad (3)$$

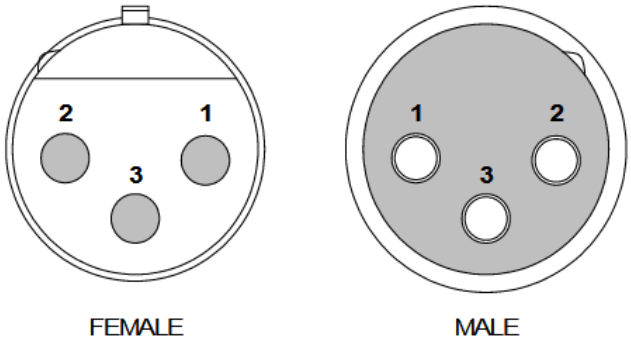
$$l_{max} < \frac{\tau_{sample} - 4\tau_{max\_IO}}{2D_{nom}}. \quad (4)$$

Po dosazení tedy vychází:

$$l_{max} < 480[m]. \quad (5)$$

5.3 Fyzické provedení

Pro fyzické připojení na sběrnici byly použity tří pinové XLR konektory. Stejné konektory jsou použity v hudebním průmyslu jednak pro přenos analogových, například mikrofonních signálu, ale dále jejich použití definuje i standard DMX512. Tento standard popisuje digitální komunikaci s osvětlovací technikou. Pro tento případ je samozřejmě v kombinaci s XLR konektory použit příslušný komunikační kabel pro diferenciální signalizaci. Dané fyzické provedení vyhovuje i požadavkům pro navrhovanou sběrnici, tudíž bude použit obdobný kabel a již zmíněné XLR konektory. (Professional, 2008)



Obrázek 18 – Tří pinové XLR konektory (Professional, 2008)

6 Akční prvek - ovladač motorků – hardwarové řešení

V této kapitole vyřeším implementaci daného protokolu, nejprve v podobě hardwaru. Ze všeho nejdříve budou zvoleny vhodné motorky pro polohování. Dále uvedu některé použité integrované obvody a nakonec provedu návrh již samotného schématu a desky plošných spojů.

6.1 Volba motorků

Stěžejním úkolem je vybrat vhodný motorek pro danou aplikaci přesného polohování. Většina motorků pro účely polohování se dá rozdělit do dvou kategorií a to servo motory a krokové motory.

6.1.1 Servo motory

Servo motory jsou elektrické motorky doplněné o servo mechanismus, umožňující jejich nastavení do přesné polohy. Dle typu použitého motoru se rozlišují AC (samozřejmě asynchronní) a DC motory. Aplikace servo motorů je široká, kde příkladem jsou různé hračky, RC modely, robotika a podobně.

Servo mechanismus se skládá z řízeného motorku, senzoru polohy a řídicího systému. Jedná se o systém s uzavřenou zpětnou vazbou, který využívá informace o poloze k danému polohování. Motorek dále obsahuje převody, které zjemňují výsledný pohyb výstupní hřídele.

Typicky servo motory pracují v určitém rozsahu stupňů (většinou 0°-180°), jelikož je jejich výstup většinou mechanicky spojen s hřídelí potenciometru, udávající informaci o poloze. Řízení polohy probíhá pomocí pulsně šířkové modulace.

6.1.2 Krokové motorky

Název motorků je odvozen od způsobu jejich polohování, které probíhá v diskrétních krocích. Obvykle je to řešeno ovladačem motorků často v podobě integrovaného obvodu a mikroprocesorem řídícím daný ovladač. Mikroprocesor tedy posílá ovladači signály ohledně provedení kroku a směru. Pokud se nebude předpokládat přetěžování motorku, tak není potřeba zpětná vazba o poloze a lze pouze počítat kroky. Krokové motorky mají obecně vysoký moment při nízkých rychlostech a naopak při vyšších rychlostech na momentu ztrácejí.

Nejčastěji jsou využívány dvoufázové motorky, které lze rozdělit na unipolární a bipolární. Unipolární motorky mají z každé fáze vyvedený střed, proti kterému se spíná každá fázová sekce. Změna polarity magnetického pole v cívce je tedy provedena použitím druhé poloviny vinutí. Výhodou je jednoduché řízení motorku, jelikož není třeba měnit polaritu. Nevýhoda ovšem spočívá v nižším momentu, z důvodu použití pouze poloviny cívek v daném čase. Bipolární motorky nemají vyvedený střed a pro jejich krokování je nutné střídat polaritu ve fázích. Nevýhoda tedy spočívá ve složitějším ovládacím obvodu (typicky dvojitý H můstek). Výhodou je ovšem vyšší moment daných motorků, jelikož se používá celá cívka.

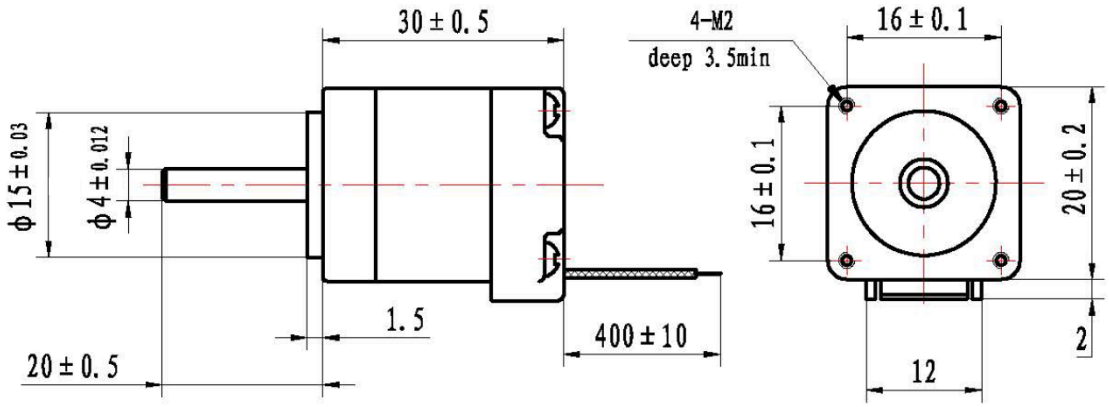
Pro danou aplikaci bylo nakonec zvoleno řešení s krokovými motorky, kdy jedním z hlavních důvodů bylo omezené úhlové natočení běžně dostupných servo motorků. Krokové motorky s tímto problém nemají, navíc jsou velmi přesné, jednoduše se řídí a nepotřebují zpětnou vazbu o poloze. (Instruments, 2016)

6.1.3 Použitý krokový motorek

Pro danou aplikaci byl zvolen bipolární hybridní krokový motorek standardu NEMA8, který se vyznačuje parametry uvedenými v tabulce níže (Tabulka 7 – Parametry krokového motorku). Motorek je rozměrově velmi malý, což je pro danou aplikaci stěžejní (Obrázek 19 – Rozměry krokového motorku).

Tabulka 7 – Parametry krokového motorku (RobotDigg)

Úhel kroku	1,8 [°] (+-5%)
Počet fází	2
Izolační odpor	100 [MΩ] (500 VDC)
Hmotnost	60 [g]
Jmenovité napětí	3,9 [V]
Jmenovitý proud	600 [mA]
Odpor na fázi	6,5 [Ω] (+-10%)
Indukčnost na fázi	1,7 [mH] (+-20%)
Moment při proudu ve vinutí	18 [mNm]
Moment při nulovém proudu	2 [mNm]



Obrázek 19 – Rozměry krokového motorku (RobotDigg)

6.2 Použité integrované obvody

Dále budou popsány zvolené integrované obvody. Jedná se zejména o obvod sloužící pro komunikaci se sběrnici, mikroprocesor a v neposlední řadě také obvod řídící krokové motorky.

6.2.1 MCP2551

Jelikož navržená fyzická vrstva protokolu je v mnoha ohledech téměř shodná s protokolem CAN, byl jako prostředník mezi mikroprocesorem a sběrnicí použit integrovaný obvod MCP2551. Jedná se High-Speed CAN vysílač/přijímač plně kompatibilní s ISO-11898 standardem, takže s možností bitové rychlosti až do 1 Mbps. Jeho funkce spočívá v převodu logických úrovní mikroprocesoru na diferenciální signál používaný sběrnicí, navíc doplněný o dominantní/recesivní chování v závislosti na vysílaném bitu. Obvod dokáže správně pracovat s nejvyšší možnou zátěží na sběrnici 45 Ω. Nejmenší hodnota impedance MCP2551 mezi CAN_H a CAN_L je 20 kΩ. Když se vezmou v úvahu ještě terminační rezistory o hodnotách 120 Ω, tak vychází, že nejvyšší možný počet těchto obvodů připojitelných na jednu sběrnici je přibližně 111ks. To je provedeno v rovnicích níže (6) a (7), kde jsou sčítány převrácené hodnoty zmíněných odporů. Výsledek je označen jako x. Výrobce uvádí hodnotu 112, implikující zátěž na sběrnici rovnou 44,91 Ω, což je samozřejmě v toleranci.

$$\frac{1}{45} > \frac{x}{20 \cdot 10^3} + \frac{2}{120} \tag{6}$$

$$x < \frac{10^3}{9} \tag{7}$$

Dále jsou důležité časové prodlevy mezi piny určenými k připojení na sběrnici a piny pro komunikaci s mikroprocesorem. Nejhorší případ nastane u příjmu přechodu z dominantního do recesivního stavu a to s maximálním zpožděním 400 ns. Daný integrovaný obvod dále disponuje ochranou proti trvalému dominantnímu stavu, kdy po uplynutí 1,25 ms je vysílání vyřazeno. Tato skutečnost implikuje maximální bitový čas, který v případě vyslání dvaceti dominantních bitů (kombinace Error frame od více uživatelů) činí 62,5 µs, což tedy nepředstavuje problém. (Microchip Technology Inc. , 2010)

6.2.2 Atmega328P

Atmega328P je 8bitový mikroprocesor od firmy Atmel. Jeho použití bylo zvoleno díky některým jeho charakteristikám, jako jsou:

- 32KBajtová flash paměť programu,
- dva 8bitové a jeden 16bitový čítač/časovač,
- integrované SPI a USART rozhraní,
- 10bitový ADC převodník,
- pouzdra s 28, či 32 piny/ploškami.

Tento mikroprocesor bude použit i pro další dva navrhované obvody (nožní přepínač a bluetooth modul). (Atmel Corporation, 2016)

Jako pouzdro zde bylo zvoleno PDIP s 28 piny.

6.2.3 L9942

Pro řízení zvolených krokových motorků byl zvolen obvod L9942 od ST Microelectronic, jehož hlavními charakteristikami jsou:

- dva H-můstky s maximální proudovou zátěží 1,3 A,
- programovatelné proudové profily s 9 možnými vstupy a 5bitovým rozlišením,
- plné, poloviční, mini a mikro krokování,
- programovatelný způsob maření energie z vinutí při spínání,
- detekce přetížení motorku (stojící motorek).

Tento integrovaný obvod je určen pro bipolární krokové motorky. Komunikace mezi mikroprocesorem probíhá pomocí SPI komunikace, kdy je obvod v roli Slave zařízení. Veškeré parametry se tedy nastavují tímto způsobem, zatímco pro samotné krokování je použit oddělený logický vstup reagující na náběžnou hranu. (STMicroelectronics, 2013)

6.3 Schéma a DPS

Návrh plošného spoje byl proveden v programu Eagle, verze 7.7.0.

Navrhovaný obvod bude schopen řídit polohu celkem tří krokových motorků. Vstupní napětí 9 VDC je přivedeno přes pojistku F2A na vypínač. Je využito doporučení výrobce obvodu L9942 a v jednom bodě je zem rozdělena na výkonovou a signálovou. Dále je napětí 9 VDC přes diodu 1N5400 (pro souvislý proud až 3 A), která slouží jako ochrana proti přepólování, přivedeno k jednotlivým obvodům L9942 pro výkonové účely, a na lineární stabilizátor, jehož výstupní napětí je 5 VDC. Toto napětí slouží jako napájecí pro mikroprocesor, obvod MCP2551 a obvody L9942. Pro hodinový kmitočet mikroprocesoru je použit externí krystal 14,7456 MHz.

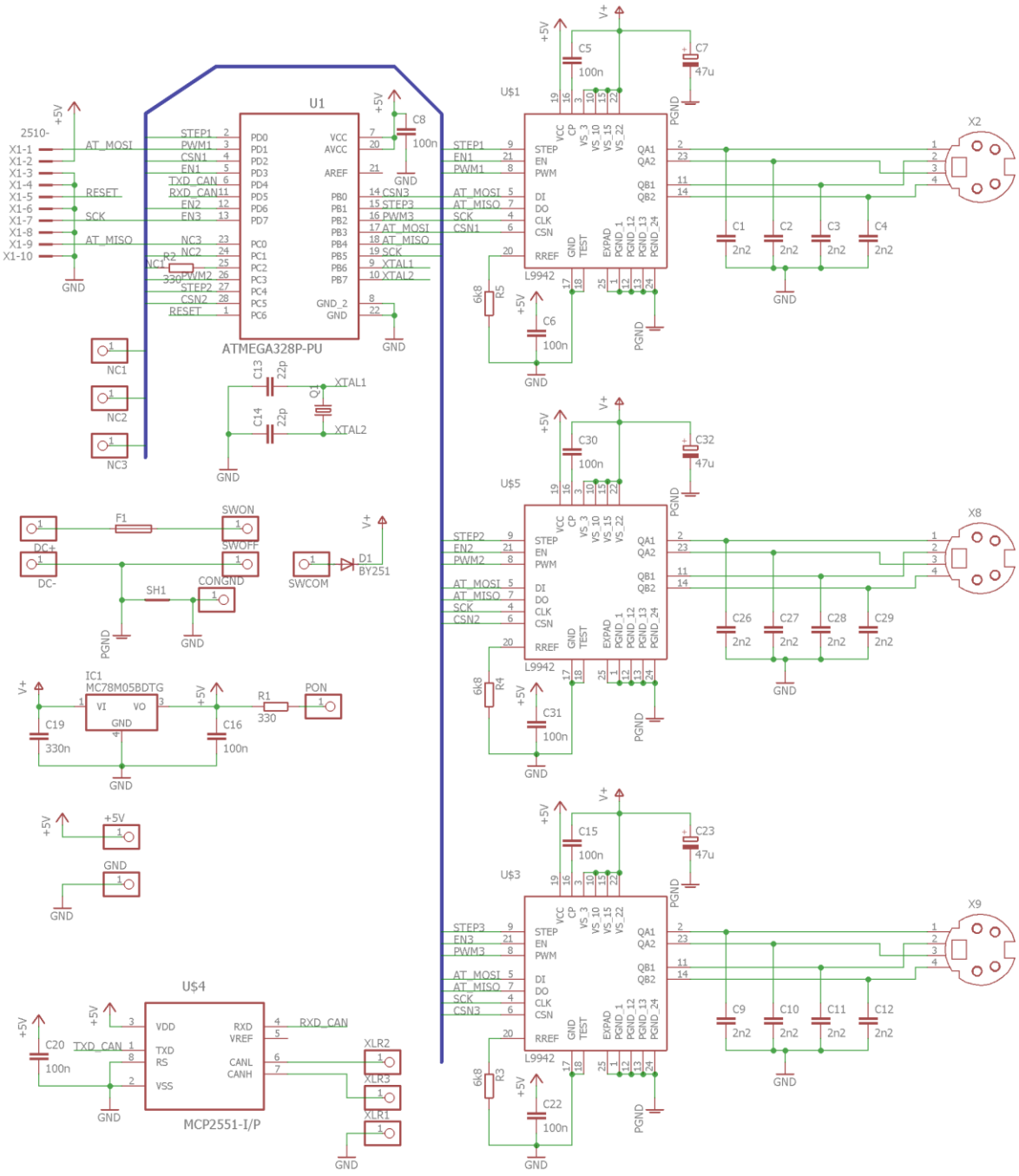
Komunikace mezi MCP2551 a mikroprocesorem je dvouvodičová, s označením RXD_CAN a TXD_CAN. Mikroprocesor dále komunikuje s obvody L9942 pomocí SPI sběrnice. Tato komunikace je typu master-slave, a ke každému obvodu je zvlášť přiveden tzv. „CSN“ signál. Tento signál stažením do logické „0“ určuje, který obvod bude s mikroprocesorem komunikovat. SPI sběrnice dále využívá signály MOSI (master out, slave in), MISO (master in, slave out) a SCK (hodinový signál). Tyto signály, spolu s resetovým pinem mikroprocesoru a napájením se zemí, jsou vyvedeny na 10pinový ISP konektor. Tento je použit pro připojení programátoru pro mikroprocesor.

Mimo SPI komunikace mezi L9942 a mikroprocesorem je zde dále signál pro řízení krokování (STEPx), povolení funkce L9942 (ENx) a signál s názvem PWM, který je naopak výstupem obvodu L9942. Signál reprezentuje současnou střídu PWM (pulsně šířkové

modulace) signálu ovladače můstku „A“ a může mu být přiřazena alternativní funkce detekující například přetížení motorku.

Napájecí přívody všech integrovaných obvodů jsou opatřeny blokovacími kondenzátory 100 nF. K obvodům L9942 byly dále dle doporučeného zapojení výrobce připojeny kondenzátory 100 nF k pinu CP a vyrovnávací kondenzátor 47 µF. Výstup pro připojení na vinutí motorků je přiveden na konektor typu mini-din. Zbylé hodnoty součástek vycházejí z doporučení výrobců, až na rezistory R1 a R2. Ty slouží jako předřadné rezistory pro připojení indikačních LED diod. Uvažuje se zde proud LED diodou 10 mA, při napětí na diodě 2 V. Pro výpočet hodnoty předřadného rezistoru lze použít následující vzorec (8), kdy reálně byla použita hodnota 330 Ω.

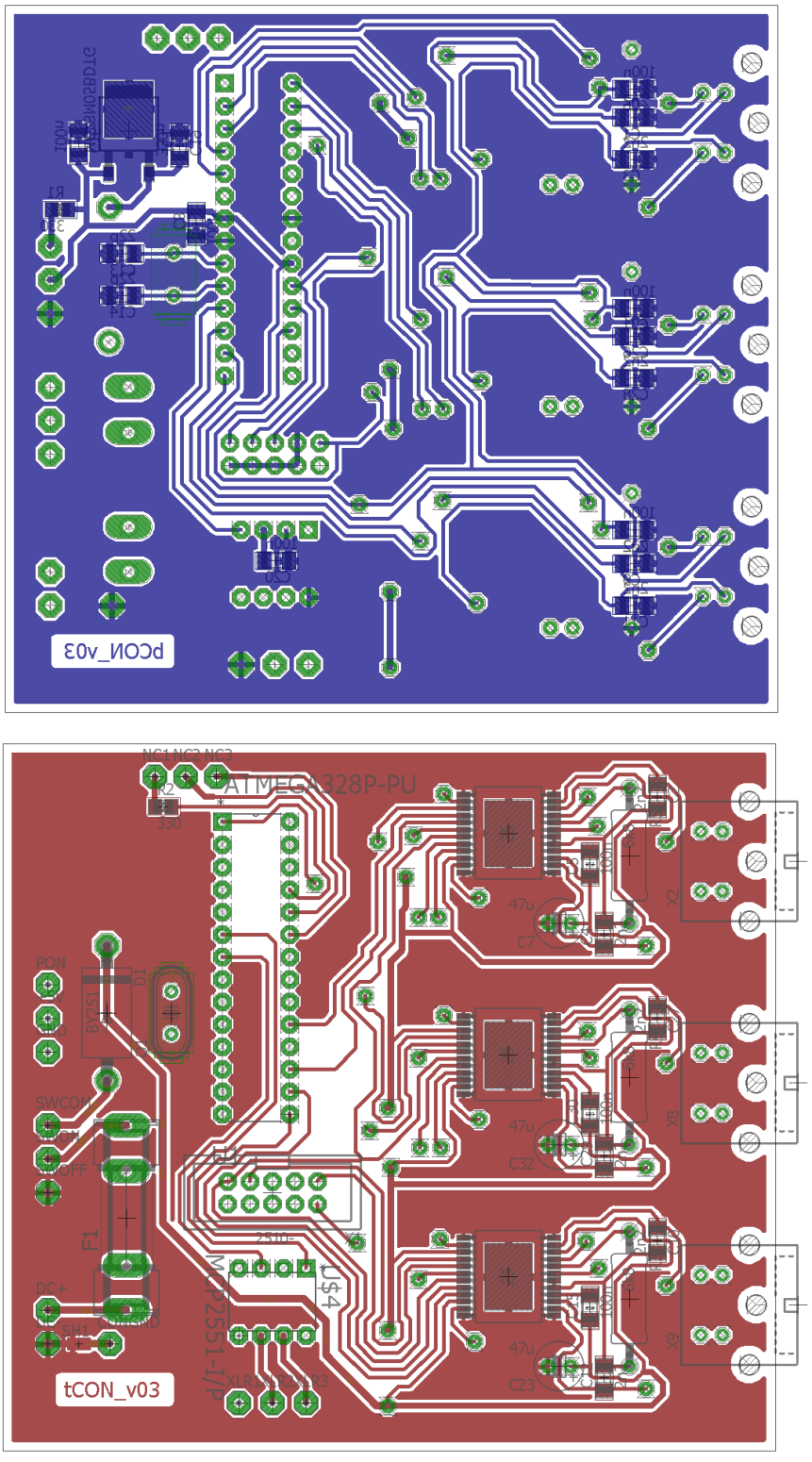
$$R = \frac{U_{cc} - U_D}{I_D} = \frac{5 - 2}{10^{-3}} = 300\ [\Omega] \qquad (8)$$



Obrázek 20 – Schéma ovladače motorků

Na dalším obrázku (Obrázek 21 – DPS ovladače motorků) je znázorněno rozmístění součástek a trasy desky plošných spojů. Výsledné fyzické provedení desky má rozměry 8,5x8cm. Vidíme, že deska je realizována jako dvouvrstvá, kdy na horní části obrázku jsou modře znázorněny spoje spodní části, zatímco níže jsou červenou barvou vyobrazeny spoje vrchní části desky. Pro mikroprocesor Atmega328P bylo zvoleno PDIP pouzdro, díky možnosti využití pinů k propojení spodní a vrchní vrstvy (deska byla vyráběna v domácích podmínkách). Obvody L9942 jsou v pouzdře pro povrchovou montáž. Napájecí konektor, vypínač, LED dioda a XLR konektory pro připojení na společnou sběrnici jsou fyzicky umístěny na panelu, proto jsou na DPS pouze pájecí plošky pro jejich připojení pomocí

vodičů. Na horní části DPS si lze povšimnout ještě tří vývodů, které jsou označeny jako „NC1“, „NC2“ a „NC3“. Jedná se o nevyužité vstupy/výstupy mikroprocesoru, sloužící jako rezervy.



Obrázek 21 – DPS ovladače motorků

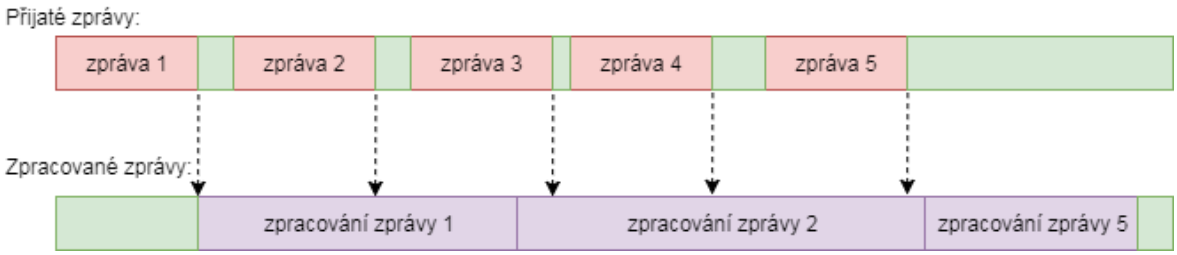
7 Akční prvek - ovladač motorků – softwarové řešení

Pro vývoj softwaru daného mikroprocesoru bylo využito prostředí Atmel Studio 6 a celý program je psaný v programovacím jazyce C.

7.1 Logická struktura programu

Pokud probíhá komunikace na sběrnici, je třeba spolehlivě určit každý přijatý bit. Proto je skenování sběrnice provedeno pomocí přerušení programu. Ostatní akce jsou brány jako vedlejší a běží v hlavní smyčce programu, která může být kdykoli přerušena právě nutností zpracování příchozího bitu.

Ve zmíněné stále dokola běžící hlavní smyčce programu jsou vždy nejdříve přečtena případná příchozí data a posléze jsou data zpracovávána. Dané zpracování například představuje nastavení motorků do určité polohy, nebo vysílání odpovědi na sběrnici a podobně. Do této struktury zasahují případná přerušení vyvolaná komunikací na sběrnici, která zapisují přijaté bity. Během výkonu procesů je tedy prvek schopen také přijímat zprávy, které se ukládají do vyrovnávacích proměnných. Po zpracování dat, prvek přečte další data z vyrovnávacích proměnných, která začne dále zpracovávat. Může se stát, že během zpracovávání bylo přijato více zpráv. V tomto případě jsou stará data nahrazena novými a proběhne zpracování nejčerstvější zprávy, přičemž veškeré dříve přijaté a nezpracované zprávy jsou dále ignorovány. Situace je znázorněna na obrázku níže (Obrázek 22 – Zpracovávání přijatých zpráv).



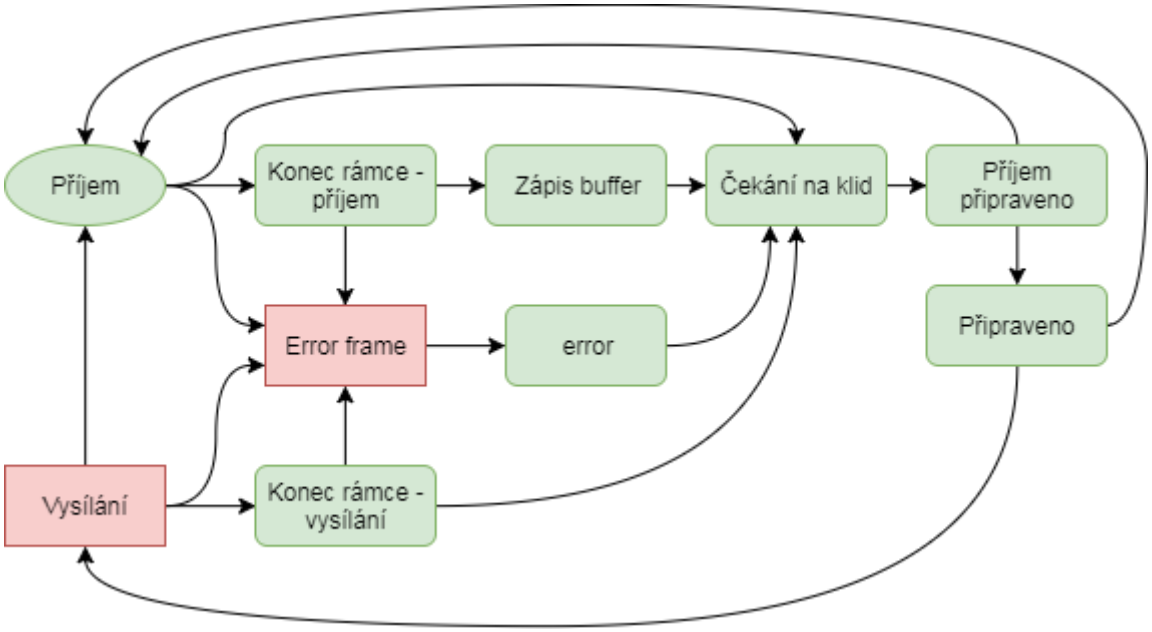
Obrázek 22 – Zpracovávání přijatých zpráv

Příjem dat je řešen pomocí stavového automatu, kdy je každý přijatý bit reprezentován určitým stavem přijímače. Podle hodnoty daného bitu se určí další stav a tedy význam dalšího přijatého bitu. Situaci znázorňuje obrázek níže (Obrázek 23 – Grafické znázornění komunikace). Zeleně jsou na obrázku vyznačeny určité stavy, přičemž „Příjem“ reprezentuje množinu různých stavů v rámci, začínající typem zprávy a končící Ack bitem (viz. kapitola Struktura rámců). Červeně jsou na obrázku vyznačeny akce, při kterých neprobíhá příjem, ale vysílání. Šipky znázorňují možné přechody mezi stavy a přechody z případného vysílání do určitých stavů.

Stav „Příjem připraveno“ a „Připraveno“ reprezentují Start-bit, který je rozdělen na dva stavy, kdy z prvního je již možný další příjem, zatímco pokud chce prvek začít vysílat, musí

počkat ještě jeden bitový čas. Pokud by byl umožněn přechod do stavu „Příjem“ a povolení dalšího vysílání současně, mohlo by to vést k problémům. Důvod je takový, že v praxi nejsou hodinové kmitočty přijímače a vysílače naprosto totožné, což může zapříčinit situaci (zejména při rozestupu 15 bitů mezi zprávami), kdy vysílač již začne vysílat, ale přijímač ještě nestihl dojít do stavu kdy je připraven na další příjem. Komunikaci na sběrnici bude tedy dále ignorovat a neustále čekat na klid na sběrnici.

Přechod do příjmu z klidového stavu je spuštěn stáhnutím sběrnice do „0“. V tomto případě je význam daného bitu „Start-bit“ a je nastaven další stav reprezentující první bit typu zprávy, který bude naskenován za jeden bitový čas. Přehledně jsou veškeré možné situace znázorněny na daném obrázku.



Obrázek 23 – Grafické znázornění komunikace

7.2 Implementace jednotlivých částí

V dalším textu bude rozebrán mnou naprogramovaný zdrojový kód. Budu se postupně zabývat vždy určitými logickými celky programu, které patřičně okomentuji.

7.2.1 Hlavní smyčka

Po prvotní inicializaci, sloužící k nastavení veškerých vstupních a výstupních pinů, časovačů, SPI komunikace a podobně, začne program běžet v nekonečné smyčce. Tato smyčka obsahuje pouze čtení dat z vyrovnávacích proměnných a jejich následné zpracování. Třetí bit proměnné „dataBuffer[13]“ obsahuje příznak čerstvých dat, který je nastaven při přijmu každé nové zprávy. Pokud dojde ke čtení, je nejdříve nastaven semafor v podobě prvního bitu dané proměnné. To zamezí případné změně dat při příjmu další zprávy

v průběhu čtení. Po zkopírování přijatých dat dojde k vynulování příznaku čerstvých dat a semaforu čtení, načež následuje nastavení příznaku čerstvých dat ke zpracování. Data jsou dále zpracovávána ve funkci „performBusAction“. Parametrem dané funkce je ukazatel na pole, proto pro zpracování dat „dataToProcess“ je funkci předána adresa tohoto pole (tzn. prvního prvku pole).

Zdrojový kód 1 – Hlavní smyčka

```
while(1)
    {
            /* --cteni dat z bufferu-- */
            if((dataBuffer[13]>>2&0b1)==1)    // jsou k dispozici nova data?
            {
                  dataBuffer[13] |= (0b00000001);    // semafor cteni
                  for(uint8_t i=0; i<13;i++)         // zkopirovani dat
                  {
                        dataToProcess[i] = dataBuffer[i];
                  }
                  dataBuffer[13] &= ~(0b00000101);   // zrusit priznak cerstych
                                                     // dat a semafor cteni
                  dataToProcess[13] |= (0b00000100); // nova data ke zpracovani
            }
            /* --zpracovani prijatych dat-- */
            if((dataToProcess[13]>>2&0b1)==1)
            {
                  performBusAction(&dataToProcess);
                  dataToProcess[13] &= ~((0b00000100));    // data zpracovana
            }
    }
```

7.2.2 Příjem rámce

Příjem začíná stažením sběrnice do „0“ (Start-bit), což vyvolá přerušení změny pinu. V obsluze tohoto přerušení (Zdrojový kód 2 – Pinchange přerušení) je upravena hodnota čítače takovým způsobem, aby vzorkování proběhlo v polovině bitového času (vzorkování je řešeno přerušením od daného čítače). Dále je zde provedeno samozřejmě povolení přerušení od čítače, což v případě příjmu dalších bitů již není potřeba. Nakonec je zde zakázáno další přerušení od změny pinu, které bude dále povoleno až po provedení vzorkování (důvod je zřejmý z kapitoly Bitové časování a synchronizace, probrané v předešlém textu).

Zdrojový kód 2 – Pinchange přerušení

```
ISR(PCINT2_vect)
{
      TCNT0 = 165;               // nastaveni hodnoty citace
      TIFR0 |= (1<<OCF0A);
      TIMSK0 |= (1<<OCIE0A);
      PCMSK2 &= ~((1<<PCINT21)); // zruseni preruseni od pinchange
}
```

Vlastní okamžik vzorkování tedy probíhá v okamžiku obsluhy přerušení daného čítače (Zdrojový kód 3 – Timer přerušení). Nejdříve proběhne vynulování hodnoty daného čítače, jehož porovnávací hodnota je nastavena na jeden bitový čas. Po přečtení stavu na sběrnici

následuje kontrola plnění bity, které je platné pouze do příjmu CRC delimiteru (globální proměnná „receivedBitCount“ definuje současný stav přijímače). V případě příjmu šesti stejných bitů po sobě je buď vyslán Error frame (pokud je i sedmý bit shodný) makrem „TRANSMITERROR“, nebo je další přijatý bit přeskočen a přerušení ukončeno (pokud se sedmý bit od předchozích liší). Další věcí je zpracování přijatého bitu, které se odvíjí od současného stavu přijímače. Některé bloky tohoto zpracování budou popsány níže. Po zpracování následuje ještě výpočet CRC, který probíhá pokaždé, ale jeho hodnota je brána v potaz pouze ve stavu CRC delimiter. Poslední věcí před návratem z přerušení je povolení přerušení od změny pinu.

Zdrojový kód 3 – Timer přerušení

```
ISR(TIMER0_COMPA_vect)
{
	TCNT0 = 0;
	if(ISSET(RXD_CAN))  // zjisteni stavu sbernice
	{
		stav = 1;
	}
	else
	{
		stav = 0;
	}
	if (receivedBitCount<crcDel) // kontrola plneni bity
	{
		if (dataRaw[11]>=BITSTUFFAMOUNT)
		{
			if (previousBit==stav)
			{
				TRANSMITERROR
			}
			previousBit = stav;
			dataRaw[11] = 1;
			return;
		}
		if (previousBit==stav)
		{
			dataRaw[11]++;
		}
		else
		{
			dataRaw[11] = 1;
		}
		previousBit = stav;
	}
	switch (receivedBitCount)  // zpracovani prichoziho bitu
	{
		.
		.
		.
	}
	if ((dataRaw[15]>>7)&0b1)  // vypocet crc
	{
		dataRaw[15] = dataRaw[15]<<1;
		dataRaw[15] = dataRaw[15]|stav;
		dataRaw[15] = dataRaw[15]^CRCPOLY;
	}
```

```
        else
        {
                dataRaw[15] = dataRaw[15]<<1;
                dataRaw[15] = dataRaw[15]|stav;
        }
        PCMSK2 |= (1<<PCINT21);    // povoleni preruseni od zmeny pinu
}
```

Co se týče zpracování jednotlivých stavů, bude nejprve popsáno, jak probíhá začátek příjmu rámce (Zdrojový kód 4 – Ukázky zpracování některých stavů - 1). Stav „recReady“ reprezentuje připravenost prvku na další příjem. Tento stav je nastaven v závěru stavu „waitForStill“ (čekání na klid na sběrnici), a jeho zpracování od tohoto okamžiku nastane nejpozději do jednoho a půl bitového času (v případě změny pinu těsně před okamžikem vzorkování). Pokud nebylo zaznamenáno stažení sběrnice do „0“, je zakázáno další přerušení od čítače a bude se dále tedy čekat pouze na změnu pinu. Také je nastaven stav „ready“, který umožňuje případné provedení vysílání. Za předpokladu příjmu „0“ je nastaven momentální počet stejných po sobě jdoucích bitů na hodnotu 1, vynulován CRC posuvný registr a nastaven význam dalšího přijatého bitu.

Stav „ready“ tedy reprezentuje příjem Start-bitu, který je vždy vyvolaný změnou pinu z „1“ do „0“. Pokud je však bit v okamžiku vzorkování roven „1“, je vyslán Error frame. Při dalším příjmu (například stav „typZpr0“) se již bity ukládají do prvků pole „dataRaw“, kde například prvních 8 prvků pole je rezervováno pro datové pole, devátý prvek (s indexem 8) pro typ zprávy, typ SDO a délku zprávy a podobně. Po uložení bitu následuje opět nastavení dalšího příchozího stavu.

Zdrojový kód 4 – Ukázky zpracování některých stavů - 1

```
.
.
case recReady:
              if (stav)
              {
                     TIMSK0 &= ~(1<<OCIE0A);    // zakazani preruseni od timeru
                     TIFR0 |= (1<<OCF0A);       // zrusit priznak zpracovani prerus.
                     receivedBitCount = ready;  // další stav bude „ready“
              }
              else
              {
                     dataRaw[11] = 1;           // pocet stejnych bitu po sobe
                     dataRaw[15] = 0;           // vynulovat CRC registr
                     receivedBitCount = typZpr0;
              }
break;
case ready:
              dataRaw[11] = 1;
              dataRaw[15] = 0;
              receivedBitCount = typZpr0;
              if (stav)                         // pokud pri start bitu „1“, error
              {
                     TRANSMITERROR;
              }
break;
case typZpr0:
              dataRaw[8] ^= (stav<<(117-receivedBitCount)^dataRaw[8]) // zapis bitu
```

```
                        &(1<<(117-receivedBitCount));
            receivedBitCount = typZpr1;
break;
.
.
```

V další ukázce zdrojového kódu (Zdrojový kód 5 - Ukázky zpracování některých stavů - 2) je znázorněn nejdříve příjem posledního bitu prvního datového bajtu, kdy se rozhoduje, zda pokračovat v příjmu dat dále, či přejít na příjem CRC součtu. Podmínka je daná dříve zjištěnou hodnotou délky zprávy uložené v prvních 3 bitech prvku pole „dataRaw[8]“.

Po přijetí CRC součtu následuje stav „crcDel“. V tomto stavu je nejdříve vyhodnocen přijatý CRC kód. Jelikož aktualizace vypočítávaného CRC posuvného registru probíhá při každém přerušení od čítače, byl výpočet aplikován i na přijatý CRC kód. Proto při bezchybném příjmu by hodnota v registru měla být nulová. (Williams, 1993)

Pokud nesouhlasí CRC součty, je vyslán Error frame. V opačném případě se vyšle „Ack bit“, potvrzující správné přijetí zprávy. Nejdříve se musí zakázat přerušení od čítače a počkat na konec CRC delimiteru. Poté je tedy sběrnice stažena na jeden bitový čas do „0“. Po tomto okamžiku je opět povoleno přerušení od čítače a nastavena počáteční hodnota, zajišťující další přerušení v polovině bitového času dalšího stavu „ackDel“. Další postup je takový, že je hlídáno dodržení správného vysílání konce rámce, tedy bitové posloupnosti „01111111“. V případě neshody je opět vyslán Error frame a zpráva se bere jako neplatná. Pokud je struktura dodržena, následuje aktualizace přijatých dat, což je provedeno zkopírováním pole „dataRaw“ do pole „dataBuffer“. Tato operace je provedena pouze, pokud není prováděno čtení pole „dataBuffer“, což je řešeno semaforem. Čtení dat v hlavní smyčce programu trvá kratší časový okamžik, než je doba mezi zpracováním jednotlivých přerušení od čítače. V nejhorším případě se tedy data do pole „dataBuffer“ zapíší na druhý pokus. Po tomto stavu již následuje čekání na klid na sběrnici a přechod do stavů „recReady“ a „ready“.

Zdrojový kód 5 - Ukázky zpracování některých stavů - 2

```
.
.
case 7:
            dataRaw[0] ^= (stav<<(7-receivedBitCount)^dataRaw[0]) // zapis bitu
                        &(1<<(7-receivedBitCount));
            if ((dataRaw[8] & (0b111))==1)    // dosazen pocet prijimanych bajtu?
            {
                  receivedBitCount = crc0;
            }
            else
            {
                  receivedBitCount++;
            }
break;
.
.
case crcDel:
            if (stav&&(dataRaw[15]==0))       // souhlasi CRC?
            {
```

```
                    TIMSK0 &= ~(1<<OCIE0A);    // zakazani preruseni od timeru
                    receivedBitCount = ackDel;
                    _delay_us(6);
                    CLEARBIT(TXD_CAN);         // vyslani „Ack“ bitu
                    _delay_us(16);
                    SETBIT(TXD_CAN);           // sbernice do „1“
                    TIMSK0 |= (1<<OCIE0A);     // povoleni timer preruseni
                    TIFR0 |= (1<<OCF0A);
                    TCNT0 = 135;
             }
             else
             {
                    TRANSMITERROR;
             }
break;
.
.
```

7.2.3 Vysílání rámců

Před samotným započetím vysílání se nejdříve musí provést nastavení požadovaných hodnot do pole „dataToTransmit“. Pro každý typ zprávy je vytvořena daná funkce nastavující správné hodnoty týkající se typu zprávy, délky dat a podobně. Níže (Zdrojový kód 6 – Příprava dat pro vyslání Heartbeat zprávy) je zobrazen příklad nastavení dat pro přenos Heartbeat zprávy. Kromě nastavení adresy odesílatele a typu zprávy zde dále probíhá i výpočet CRC součtu, jehož výsledek je uložen do prvku pole „dataToTransmit[14]“.

Zdrojový kód 6 – Příprava dat pro vyslání Heartbeat zprávy

```
void setHeartBeat(uint8_t *dataTrans)
{
       uint8_t crcRegister = 0;
       int8_t j = 0;
       dataTrans[13] = objectData[3][1]; // adresa vysilani moje/zdrojova
       dataTrans[8] ^= (0b01100<<3 ^ dataTrans[8]) & (0b11111 << 3);  // typ zpravy
       crcCalc(&crcRegister, 1);  // vypocet zacit při prvnim nenulovem bitu
       crcCalc(&crcRegister, 1);  // druhy bir typu zpravy
       for (j=7;j>=0;j--)         // vypocet pro odeslanou adresu
       {
              crcCalc(&crcRegister, (dataTrans[13]>>j & 1));
       }
       for (j=0;j<8;j++)          // dodeleni konce
       {
              if ((crcRegister>>7)&0b1)
              {
                     crcRegister = crcRegister<<1;
                     crcRegister^= CRCPOLY;
              }
              else
              {
                     crcRegister = crcRegister<<1;
              }
       }
       dataTrans[14] = crcRegister; // zapis vypoctene hodnoty CRC do promenne
}
```

Když jsou data nastavena, může se pro započetí vysílání zavolat funkce „transmission“ (Zdrojový kód 7 – Ukázka funkce transmission). Jejími parametry jsou data k vysílání (v tomto konkrétním program to je vždy již zmíněné globální pole „dataToTransmit“), počet pokusů pro vysílání a možnost zrušit snahu o vysílání v případě příjmu čerstvých dat. Jelikož při započetí vysílání může arbitráž vyhrát jiný prvek, jsou nejdříve data týkající se právě arbitráže zkopírována do pole reprezentující případný příjem zprávy. Dále je zde nastaven čítač do správného módu a zvoleny dvě hodnoty pro porovnávání, využívané hlavně v makru „TRANSMITBITWCHECK“ (Zdrojový kód 8 – Makro pro vyslání bitu). Před samotným započetím vysílání je třeba nejdříve ověřit, zda neprobíhá již přenos jiného rámce. To znamená, že se prvek musí nacházet ve stavu „ready“ a sběrnice musí být v „1“. Tato podmínka je testována ve smyčce, která trvá nejdéle 140 bitových časů (odpovídá nejdelší možné zprávě i s ohledem na plnění bity a dobu mezi zprávami). Proběhnutí jedné této smyčky je bráno jako jeden pokus o vysílání. Je zde také testováno, zda nejsou k dispozici nová data, která by případně vedla k přerušení pokusů o vysílání.

Pokud je tedy možné vysílání, je zakázáno přerušení od změny pinu, resetován čítač a odvysílán Start-bit. Dále je určena hodnota předchozího bitu a současný počet stejných po sobě následujících bitů, zejména pro účely plnění bity. Další vysílání bitů se určí dle hodnot v poli „dataToTransmit“. Po odvysílání CRC pole následuje logická „1“ reprezentující CRC delimiter. Dále je skenována sběrnice pro určení Ack bitu od příjemce. V tomto okamžiku je hodnota Ack bitu uložena do pomocné proměnné, protože ještě musí proběhnout přenos konce rámce. Pokud i ten proběhne v pořádku je nastaven stav „transEOF“ reprezentující přechod z konce rámce po odvysílání a je povoleno přerušení od čítače a změny pinu. Jako poslední věc při bezproblémovém odvysílání rámce je funkcí vrácena hodnota přijatého Ack bitu.

Zdrojový kód 7 – Ukázka funkce transmission

```
enum transmissionSucces transmission(uint8_t *dataTrans, uint8_t attempts, uint8_t
ignore)
{
	enum transmissionSucces pomReturn;
	int8_t j = 0;
	uint8_t portPom = 0, delkaDat = 0, i = 0, k = 0, crcReg = 0;
	dataRaw[8] = dataTrans[8];  // v pripade prechodu na prijem zapis typu zpravy
	dataRaw[13] = dataTrans[13];  // adresa prijem
	dataTrans[11] = 1;
	delkaDat = (dataTrans[8] & 0b111);
	if (delkaDat==0)
	{
		delkaDat = 8;
	}
	TCCR2B ^= (0b001<<0 ^ TCCR2B) & (0b111<<0);  // citac pro vysilani
	OCR2A = 227;  // nastaveni hodnot bitove delky a skenovani
	OCR2B = 100;
	TIFR2 |= (1 << OCF2B);
	TIFR2 |= (1 << OCF2A);
	TCNT2 = 0;
	while (portPom!=1)  // smycka pro provadeni pokusu o vysilani
	{
		if (i>=attempts)    // vycerpany pokusy?
```

```
        {
                return timeExpired;
        }
        i++;
        k = 0;
        while (k<=140)      // nejdelsi mozny ramec trva 140 bitu
        {
                if ((TIFR2 & (1 << OCF2A))) // citac delky bitu
                {
                        k++;
                        TIFR2 |= (1 << OCF2A);
                        TCNT2 = 0;
                }
                if (((dataBuffer[13]>>2&0b1)==1)&&(!ignore)) // jsou nova data?
                {
                        return newDataToProcess;
                }
                if ((receivedBitCount == ready)&&(ISSET(RXD_CAN))) // mozne
                {                                                  // vysilani?
                        PCMSK2 &= ~((1<<PCINT21)); // zakazat prerus. zmeny pinu
                        TCNT2 = 0;
                        TRANSMITBIT(0)      // vyslat start-bit
                        portPom = 1;        // pryc ze smycky provadeni pokusu
                        break;
                }
        }
}
dataTrans[11] = 1;
previousBit = 0;
switch ((dataTrans[8]>>5)&0b11)   // akce dle typu zpravy
{
        case 0b00:
        TRANSMITBITWCHECK(0,err)
        TRANSMITBITWCHECK(0,err)
        for (j=7;j>=0;j--)
        {
                TRANSMITBITWCHECK((dataTrans[13]>>j & 1),(adrZdr7-j))
        }
        delkaDat = 1;
        break;
        .
        .
        .
}
for (i=0;i<delkaDat;i++)   // odvysilani dat
{
        for (j=7;j>=0;j--)
        {
                TRANSMITBITWCHECK((dataTrans[i]>>j & 1),err)
        }
}
for (j=7;j>=0;j--)  // vysilani CRC
{
        TRANSMITBITWCHECK((dataTrans[14]>>j & 1),err)
}
dataTrans[11] = 0;
TRANSMITBITWCHECK(1,err)   // CRC delimiter
while(!(TIFR2 & (1 << OCF2A)));   // cekani na odvysilani CRC delimiteur
TIFR2 |= (1 << OCF2B);
TIFR2 |= (1 << OCF2A);
TCNT2 = 0;
```

```
        while(!(TIFR2 & (1 << OCF2B)));   // skenovani „Ack“ v polovine bit. casu
        if (ISCLEAR(RXD_CAN))
        {
                pomReturn = ack;
        }
        else
        {
                pomReturn = notAck;
        }
        TRANSMITBITWCHECK(0,err);  // konec ramce zacinajici „0“
        for (j=0;j<BITSTUFFAMOUNT;j++)    // zbytek konce ramce „1“
        {
                dataTrans[11] = 0;
                TRANSMITBITWCHECK(1,err)   // kdyz na sbernici „0“, vyslat error
        }
        while(!(TIFR2 & (1 << OCF2B)));   // od poloviny bit. casu pote prijem
        TCNT0 = 15;                       // stavu cekani na klid
        TIFR0 |= (1<<OCF0A);
        TIMSK0 |= (1<<OCIE0A);
        PCMSK2 |= (1<<PCINT21);
        receivedBitCount = transEOF;
        return pomReturn;
}
```

V právě popsané funkci „transmission“ je pro odvysílání každého bitu použito makro „TRANSMITBITWCHECK“ (Zdrojový kód 8 – Makro pro vyslání bitu). Makro obsahuje dva parametry a to hodnotu bitu pro odvysílání a stav, který daný bit reprezentuje. Nejdříve je zde řešeno plnění bity, kdy je v případě potřeby odvysílán opačný bit od předchozích, který se do výsledné zprávy nepočítá. Při případném vysílání opačného bitu je v polovině bitového času provedeno vzorkování, přičemž když nesouhlasí vysílaná hodnota s hodnotou na sběrnici, je odeslán Error frame, nastaven stav přijímače na „err“ a funkcí „transmission“ je vrácena hodnota „transmittedError“. Zde je vhodné zmínit fakt, že popisované makro nesmí být použito jinde než právě ve funkci „transmission“. Po vyřešení případného plnění bity je na řadě vyslání požadovaného bitu, kdy v polovině bitového času je opět provedeno vzorkování stavu sběrnice. Zde přichází do role hodnota stavu onoho bitu. Pokud probíhá arbitráž, například při vysílání prvního bitu adresy zdroje, měla by být druhým parametrem makra právě hodnota tohoto stavu. V případě neshody hodnoty bitu a skutečného stavu sběrnice, se přejde z vysílání na příjem právě od tohoto stavu a funkcí je vrácena hodnota „lostArbitration“. Pokud vysílání již za bitovou arbitráží, druhým parametrem makra je stav „err“, kdy neshoda vyvolá vyslání Error frame. V celém makru si lze povšimnout speciálních znaků „\“, které jsou nutné z toho důvodu, že je makro napsáno na více řádků.

Zdrojový kód 8 – Makro pro vyslání bitu

```
#define TRANSMITBITWCHECK(x,bitCount)    \
{ \
        if (dataTrans[11]>=BITSTUFFAMOUNT) \
        {       \
                portPom = CANPORT_LOC^(((((~previousBit)&1)<<CANPORT)
                        ^CANPORT_LOC)&(1<<CANPORT)); \
```

```
            while(!(TIFR2 & (1 << OCF2A))); \
            CANPORT_LOC = portPom; \
            TIFR2 |= (1 << OCF2B); \
            TIFR2 |= (1 << OCF2A); \
            TCNT2 = 0; \
            while(!(TIFR2 & (1 << OCF2B))); \
            if (ISCLEAR(RXD_CAN) && (portPom>>CANPORT)&0b1) \
            { \
                  receivedBitCount = err;\
                  TRANSMITERROR \
                  return transmittedError; \
            } \
            dataTrans[11] = 1; \
            previousBit = (~previousBit)&1; \
      } \
      if (previousBit==x) \
            { \
                  dataTrans[11]++; \
            }  \
      else \
            { \
                  dataTrans[11] = 1; \
            } \
      portPom = CANPORT_LOC ^ ((((x)<<CANPORT) ^ CANPORT_LOC) & (1<<CANPORT)); \
      while(!(TIFR2 & (1 << OCF2A))); \
      CANPORT_LOC = portPom; \
      TIFR2 |= (1 << OCF2B); \
      TIFR2 |= (1 << OCF2A); \
      TCNT2 = 0; \
      previousBit = x; \
      while(!(TIFR2 & (1 << OCF2B))); \
      if (ISCLEAR(RXD_CAN) && (portPom>>CANPORT)&0b1) \
      { \
            dataRaw[11] = dataTrans[11];\
            dataRaw[15] = crcReg; \
            receivedBitCount = (bitCount);\
            if (receivedBitCount == err) \
            { \
                  TRANSMITERROR \
                  return transmittedError; \
            } \
            TIFR0 |= (1<<OCF0A); \
            TIMSK0 |= (1<<OCIE0A); \
            TCNT0 = 200; \
            PCMSK2 |= (1<<PCINT21); \
            return lostArbitration; \
      } \
} \
```

7.2.4 Zpracování dat

Klíčovým motivem pro reprezentaci přenesených dat je již zmiňovaný Slovník objektů. Každý objekt je reprezentován polem, kdy v prvním prvku jsou informace o délce dat v objektu (nejnižší 4 bity) a možnosti přístupu (když je nejvýznamnější bit roven 1, tak je objekt pouze pro čtení, jinak i pro zápis). Další prvky pole již obsahují charakteristická data. Celý Slovník objektů je implementován ve formě pole ukazatelů, kdy každý ukazatel odkazuje na první prvek pole daného objektu. Pořadí v tomto poli určuje hodnotu indexu.

Pokud pro určitý index neexistuje objekt, je ukazatel v poli typu NULL (Zdrojový kód 9 – Object Dictionary).

Zdrojový kód 9 – Object Dictionary

```
uint8_t       .
              .
              pos1[2] = {1,0},
              pos2[2] = {1,0},
              pos3[2] = {1,0},
              ranPos1[2] = {0b10000001,0xff},
              ranPos2[2] = {0b10000001,0xff},
              ranPos3[2] = {0b10000001,0xff},
              .
              .
uint8_t *objectData[256] = {NULL,
                            name,
                            type,
                            address,
                            heartBeatTime,
                            [5 ... 50] = NULL,
                            pos1,
                            pos2,
                            pos3,
                            ranPos1,
                            ranPos2,
                            ranPos3,
                            .
                            .
```

Zpracování přijatých dat probíhá ve funkci „performBusAction“ (Zdrojový kód 10 – Zpracování přijatých dat), volané z hlavní smyčky programu při obdržení příznaku čerstvých dat. Nejdříve je ve switch-case bloku zjištěn typ obdržené zprávy. Typy Emergency a Heartbeat ovladač motorků nijak nezpracovává. Při obdržení PDO zprávy je volána funkce „performPDOAction“, která dle hodnoty přijatého PDO provede nastavení motorků do určité polohy. Pokud je přijata zpráva typu SDO, v dalším switch-case bloku probíhá vyhodnocení dle typu. Při příjmu info SDO není vykonána žádná činnost, dále při write required SDO je odesláno zpět write SDO s příznakem chyby a příslušným kódem. Pokud je přijato info required SDO, proběhne prozkoumání platnosti daného indexu, kdy v případě, že index nereprezentuje žádný prvek, je vysláno info SDO opět s příznakem chyby a kódem. V opačném případě je odesláno info SDO s parametry požadovaného objektu.

Nejvíce kódu zde ovšem zabírá zpracování write SDO. Ze všeho nejdříve jsou otestovány podmínky platnosti indexu, délky dat, a zda není daný objekt určen pouze pro čtení. Při nesplnění některé z podmínek je odesláno info SDO samozřejmě s příznakem chyby a příslušným kódem. Pokud jsou podmínky splněny, přistupuje se k případnému vykonání nějaké akce spojené s daným objektem. Pro příklad je zde nastavení polohy prvního motorku, kdy je nejdříve otestována požadovaná hodnota polohy s maximální možnou. Pokud požadovaná hodnota není v rozsahu, je vysláno info SDO s chybovým příznakem a kódem definujícím nevalidní data. Když je hodnota v pořádku, vykoná se funkce „setPos“. Dalším příkladem je uložení současných pozic motorků do jednoho z pěti možných stavů,

do nichž se lze dostat příslušným PDO. Ne každý zápis do Slovníku objektů musí nutně znamenat nějakou akci, může se jednat pouze o přepsání nějakého parametru. Toto je v daném switch-case bloku ošetřeno případem „default“, kde proběhne pouze zápis do požadovaného objektu. V případě bezproblémového provedení výše uvedených příkazů, následuje odpověď ve formě info SDO.

Zdrojový kód 10 – Zpracování přijatých dat

```
void performBusAction(uint8_t *dataAction)
{
	switch((dataAction[8]>>5)&0b11)
	{
	.
	.
	case 0b01:		// PDO
		performPDOAction(dataAction);
	break;
	case 0b10:		// SDO
	switch((dataAction[8]>>3)&0b11)
	{
		case 0b00:		// SDO write
		{
			uint8_t i = (dataAction[8]&0b111);
			if(objectData[dataAction[10]]==NULL) // nespravny index?
			{
				setSDOinfoErr(dataAction[11],
					dataAction[10],1,&dataToTransmit);
				transmission(&dataToTransmit,10,1);
				return;
			}
			if(i==0) i = 8;
			if(((objectData[dataAction[10]][0])&0b1111)!=i)// delka dat ok?
			{
				setSDOinfoErr(dataAction[11],
					dataAction[10],3,&dataToTransmit);
				transmission(&dataToTransmit,10,1);
				return;
			}
			if((((objectData[dataAction[10]][0])>>7)&0b1)==1) // zkousen
			{					// zapis do RO objektu?
				setSDOinfoErr(dataAction[11],
					dataAction[10],2,&dataToTransmit);
				transmission(&dataToTransmit,10,1);
				return;
			}

			switch(dataAction[10])
			{
				case 51:
				{
					if (dataAction[0]>(objectData[54][1]-1)) // jsou
					{				// data v rozsahu?
						setSDOinfoErr(dataAction[11],
							dataAction[10],4,&dataToTransmit);
						transmission(&dataToTransmit,10,1);
						return;
					}
					setPos(dataAction[0], 0, 0,0b100);
				}
```

```
                        break;
                        .
                        .
                        case 101:
                        {
                               for(i = 1;i<=4;i++)
                               {
                                      objectData[61+2*(dataAction[4]-1)][i]
                                             = dataAction[i-1];
                               }
                               objectData[62+2*(dataAction[4]-1)][1]
                                             = objectData[51][1];
                               objectData[62+2*(dataAction[4]-1)][2]
                                             = objectData[52][1];
                               objectData[62+2*(dataAction[4]-1)][3]
                                             = objectData[53][1];
                               objectData[62+2*(dataAction[4]-1)][4]
                                                    = dataAction[5];
                        }
                        break;
                        default:      // zapis do objektu bez vyzadovane akce
                               for(i = 1;i<=((objectData[dataAction[10]][0])
                                             &0b1111);i++)
                               {
                                      objectData[dataAction[10]][i]
                                             = dataAction[i-1];
                               }
                        break;
                 }
                 setSDOinfo(dataAction[11],dataAction[10],
                               ((objectData[dataAction[10]][0])&0b1111),
                               &objectData[dataAction[10]][1],&dataToTransmit);
                 transmission(&dataToTransmit,10,1);      // vyslani info SDO
          }
          break;
          case 0b01:           // SDO write required
                 setSDOwriteErr(dataAction[11],dataAction[10],
                               4,&dataToTransmit);
                 transmission(&dataToTransmit,10,1);
          break;
          .
          .
          case 0b11:           // SDO info required
          {
                 if(objectData[dataAction[10]]==NULL) // nespravny index?
                 {
                        setSDOinfoErr(dataToProcess[11],dataToProcess[10],
                               1,&dataToTransmit);
                        transmission(&dataToTransmit,10,1);
                        return;
                 }
                 setSDOinfo(dataAction[11],dataAction[10],
                               ((objectData[dataAction[10]][0])&0b1111),
                               &objectData[dataAction[10]][1],&dataToTransmit);
                 transmission(&dataToTransmit,10,1);
          }
          break;
   }
   break;
   .
   .
```

```
	}
}
```

7.2.5 SPI komunikace

Obvod L9942 pro řízení motorků disponuje 8 registry, každý s kapacitou 13 bitů, v nichž jsou uloženy nejrůznější parametry. Jako příklad lze uvést směr krokování, krokovací mód (plné, poloviční, mini a mikro krokování), proudové profily, způsob maření energie cívek (tzv. Decay mode) a tak dále. Nastavené hodnoty registrů všech tří integrovaných obvodů jsou uloženy v proměnných v mikroprocesoru, které jsou datového typu bezznaménkový 16bitový integer (celočíselná hodnota). Níže (Zdrojový kód 11 – Nastavení parametrů obvodům L9942) je znázorněna ukázka modifikace daných proměnných v závislosti na parametru. Při každé změně je nastaven příznak, určující, do kterého registru se bude následně zapisovat.

Zdrojový kód 11 – Nastavení parametrů obvodům L9942

```
uint8_t l9942_set_reg_par(enum l9942Par par, uint8_t val)
{
	switch (par)
	{
		case dirCh1:
			registersCh1[0] ^= (val<<0 ^ registersCh1[0]) & (0b1 << 0);
			regChangeFlagCh1 |= (1<<0); // priznak zmeny registru
			break;
		case stepModeCh1:
			registersCh1[0] ^= (val<<1 ^ registersCh1[0]) & (0b11 << 1);
			regChangeFlagCh1 |= (1<<0);
			break;
		.
		.
		case clr7Ch3:
			registersCh3[7] ^= (val<<12 ^ registersCh3[7]) & (0b1 << 12);
			regChangeFlagCh3 |= (1<<7);
			break;

	}
}
```

Jak již bylo zmíněno, každý registr v obvodu L9942 obsahuje 13 bitů. Při zápisu do registru pomocí SPI komunikace, proběhne ovšem výměna 16 bitů, přičemž první tři bity mají význam adresy daného registru. Z toho důvodu jsou přiřazené proměnné 16bitové, s pevně danými třemi nejvyššími bity obsahujícími adresu. V ukázce dalšího zdrojového kódu níže (Zdrojový kód 12 – Zápis pomocí SPI) je zobrazen způsob zápisu do registrů. Pro každý ze tří obvodů L9942 je postupně testován příznak změny proměnných, podle něhož proběhne zápis pouze do určitých registrů. Při zápisu je pin příslušného obvodu „Chip Select Not“ stažen do logické „0“. Poté proběhne přenos pomocí integrovaného SPI rozhraní, kterým je vybaven použitý mikroprocesor. Jelikož toto rozhraní podporuje výměnu pouze 8 bitů, je přenos proveden dvakrát po sobě, kdy je nejdříve přenesena horní polovina bitů dané proměnné (obsahující adresu) a posléze spodní polovina. Po přenosu je pinu „Chip Select

Not“ opět přiřazena logická „1“. Dále je zrušen příznak změny parametrů a další případný přenos je proveden s určitým zpožděním, které vyžaduje obvod L9942.

Zdrojový kód 12 – Zápis pomocí SPI

```
void l9942_write_par()
{
	uint8_t i;

	for (i=0;i<8;i++)
	{
		if ((regChangeFlagCh1>>i) & 1)// zmenen nejaky registr v prvnim L9942?
		{
			CLEARBIT(CSN1);	// pozadovany slave 1
			SPDR = registersCh1[i]>>8;
			while(!(SPSR & (1<<SPIF) ));
			recRegistersCh1[i] = SPDR<<8;
			SPDR = registersCh1[i] & 0xFF;
			while(!(SPSR & (1<<SPIF) ));
			recRegistersCh1[i] |= SPDR;
			SETBIT(CSN1);
			regChangeFlagCh1 &= ~(1<<i);
			_delay_us(4);
		}
	}
for (i=0;i<8;i++)
	{
		if ((regChangeFlagCh2>>i) & 1)
		{
	.
	.
	.
}
```

7.2.6 Polohování motorků

Nastavení motorků do určitých poloh zajišťuje funkce „setPos“ (Zdrojový kód 13 – Funkce setPos, část 1), jejímiž parametry jsou samozřejmě požadované polohy a navíc maska, určující pro které motorky je požadovaná poloha platná.

Po deklaraci proměnných následuje výpočet fázového čítače. Jedná se o parametr obvodu L9942, od kterého se odvíjí proud vinutími a jehož hodnota je určena 5 bity. To znamená rozsah hodnot 0 až 31. V případě mikro krokování se jeho hodnota s každým krokem inkrementuje o jedničku. V navrhovaném případě je ovšem použito plné krokování a s každým krokem je proto přičtena hodnota osm. To určuje čtyři možné inkrementace, než se bude cyklus opakovat. Proudových profilů je celkem devět, přičemž v tomto případě jsou použity pouze první (s hodnotou proudu 0 mA) a poslední (s hodnotou proudu 615 mA). Cyklus krokování je znázorněn v tabulce níže (Tabulka 8 – Proudy vinutími v závislosti na provedených krocích), kde si lze všimnout, že v krocích tři a čtyři je použita stejná hodnota proudu, ovšem opačné polarity.

Tabulka 8 – Proudy vinutími v závislosti na provedených krocích

Číslo kroku	Hodnota fázového čítače	Proudový profil vinutí A	Proud vinutím A [mA]	Proudový profil vinutí B	Proud vinutím B [mA]
0	0	0	0	8	615
1	8	8	615	0	0
2	16	0	0	8	-615
3	24	8	-615	0	0

Výpočet fázového čítače v popisovaném kódu je tedy proveden jako hodnota polohy (poloha se uvažuje jako počet kroků od nulového bodu) modulo čtyřmi, následně vynásobená osmi. Toto je provedeno samozřejmě pro všechny tři obvody. Dále pokud je maska pro daný motorek rovna nule, je pro novou požadovanou polohu použita hodnota současná. Po případném omezení maximálních poloh je provedeno nastavení fázových čítačů. Dalším modifikovaným parametrem je „DAC scale“, což určuje škálování výstupního proudu. Za klidového stavu, jelikož není předpokládán žádný moment působící na hřídel, je vhodné snížit hodnotu proudu cívkami, hlavně z teplotních důvodů. Tato snížená hodnota je nastavena již během inicializace a činí 280 mA. Pokud probíhá polohování motorku, je nutné zvýšit škálování a nastavit nominální hodnotu proudu, což je zde provedeno. Ještě než jsou zapsány modifikované parametry do obvodů, je třeba také zjistit směr krokování (otáčení) motorků. To je provedeno zároveň s výpočtem počtu kroků pro každý motorek, načež tedy následuje již samotný zápis parametrů do registrů.

Zdrojový kód 13 – Funkce setPos, část 1

```
uint8_t setPos(uint8_t position1, uint8_t position2, uint8_t position3,
uint8_t mask)
{
	uint8_t steps1, steps2, steps3, pinLittle, portLittle, stepLittle,
		loopLittle, pinMed, portMed, stepMed, loopMed, pinLot, portLot,
		stepLot, loopLot;
	uint8_t phaseCount1, phaseCount2, phaseCount3;

	phaseCount1 = (objectData[51][1]%4)*8;   // vypocet fazoveho citace
	.
	.
	if (!((mask>>2)&1)) // se kterymi motorky se bude hybat?
	{
		position1 = objectData[51][1];
	}
	.
	.
	if (position1>(objectData[54][1]-1)) // omezeni maximalni polohy
	{
		position1 = (objectData[54][1]-1);
	}
	.
	.
	l9942_set_reg_par(phaseCounterCh1,phaseCount1);
```

```
l9942_set_reg_par(phaseCounterCh2,phaseCount2);
l9942_set_reg_par(phaseCounterCh3,phaseCount3);
l9942_set_reg_par(dacScaleCh1,0b100);
l9942_set_reg_par(dacScaleCh2,0b100);
l9942_set_reg_par(dacScaleCh3,0b100);
.
.
if (position1>objectData[51][1])  // jakym smerem a kolik se bude krokovat?
{
       steps1 = position1 - objectData[51][1];
       l9942_set_reg_par(dirCh1,0);
}
else
{
       steps1 = objectData[51][1] - position1;
       l9942_set_reg_par(dirCh1,1);
}
.
.
l9942_write_par();
.
.
```

Při polohování je žádoucí, aby se motorky dostavili do nových poloh v ideálním případě současně. To například znamená, že malá změna polohy jednoho motorku se bude provádět pomaleji než změna polohy o více kroků motorku jiného. Nejkratší čas mezi dvěma kroky je zvolen 8 ms (4 ms daného výstupu pro logickou „1“ a další 4 ms pro „0“). Při pomalejším krokování jsou použity pouze celistvé násobky tohoto času. To znamená, že pokud nejvyšší počet kroků nebude zároveň celistvým násobkem středního počtu kroků a nejmenšího počtu kroků, zmiňovaný ideální případ nenastane. Nejlepší bude si algoritmus polohování uvést na konkrétním případě. Nejprve je předpokládán nejvyšší počet kroků 17 (motorek 1), střední počet kroků 11 (motorek 2) a nejmenší 4 (motorek 3). Na každý krok motorku 3 se provedou 2 kroky motorku 2, a na každý krok motorku 2 se provedou 2 kroky motorku 1. Po provedení ještě zbývají provést 3 kroky motorku 2 a 1 krok motorku 1. Provede se tedy na každý krok prvního motorku dva kroky motorku 2. Nakonec zbývá provést jediný krok motorku 2.

V algoritmu je pro nejmenší počet kroků použit vnější cyklus, do něhož jsou vnořeny cykly pro střední a nejvyšší počet kroků. Před provedením je ale nejprve nutné zjistit seřazení motorků v závislosti na počtu požadovaných kroků. Tím je určeno, kterému konkrétnímu motorku odpovídá nejvyšší, střední a nejmenší počet kroků. Dále je přistoupeno ke zjištění počtu vykonání daných smyček a určení zbývajících počtu kroků (Zdrojový kód 14 - Funkce setPos, část 2).

Zdrojový kód 14 - Funkce setPos, část 2

```
.
.
if (steps1<steps2)
{
       if (steps2<steps3)  // 1<2<3
       {
```

```
                pinLittle = STEPPIN1;
                portLittle = STEPPORT1;
                stepLittle = steps1;
                pinMed = STEPPIN2;
                portMed = STEPPORT2;
                stepMed = steps2;
                pinLot = STEPPIN3;
                portLot = STEPPORT3;
                stepLot = steps3;
        }
        else
        {
                if (steps1<steps3)  // 1<3<=2
                {
                .
                .
                .
        }
loopLittle = stepLittle;
if (loopLittle==0)
{
        loopMed = stepMed;
        stepMed = 0;
}
else
{
        loopMed = stepMed/loopLittle;
        stepMed = stepMed%(loopMed*loopLittle);
}
if (loopMed==0)
{
        loopLot = stepLot;
        stepLot = 0;
}
else
{
        loopLot = stepLot/(loopMed);
        if (loopLittle==0)
        {
                stepLot = stepLot%(loopLot*loopMed);
        }
        else
        {
                loopLot = loopLot/(loopLittle);
                stepLot = stepLot%(loopLot*loopMed*loopLittle);
        }
}
.
.
```

V dalším zdrojovém kódu (Zdrojový kód 15 - Funkce setPos, část 3) je vidět již způsob provedení jednotlivých smyček. Vnější a prostřední smyčka se musejí provést vždy, i když je jejich počet nulový (žádná změna polohy motorků). Proto zde jsou podmínky určující vyslání STEP signálu. Provedení smyčky pro nejvyšší počet kroků je již podmíněno. Po provedení této části polohování dojde k testování počtu zbývajících kroků, a pokud do této doby motorku s nejvyšším počtem kroků zbývá menší počet kroků než druhému zbývajícímu motorku, je provedeno prohození přiřazených výstupních pinů a porovnávaných počtů

kroků. Dále následuje zjištění počtu dalších dvou smyček a počtu zbývajících kroků posledního motorku. Po provedení další fáze polohování je provedena již poslední smyčka v závislosti na počtu zbývajících kroků. Poslední věcí je aktualizace současných poloh a nastavení klidového proudu vinutími.

Zdrojový kód 15 - Funkce setPos, část 3

```
.
.
for (uint8_t i = 0; i<=loopLittle; i++)
{
	if (i>0)
	{
		SETBITALT(_SFR_IO8(portLittle), pinLittle);
	}
	for (uint8_t j = 0; j<=loopMed; j++)
	{
		if (((j>0)&&(i>0))||((loopLittle==0)&&(j>0)))
		{
			SETBITALT(_SFR_IO8(portMed), pinMed);
		}
		if (((j>0)&&(i>0))||((loopLittle==0)&&(j>0))||(loopMed==0))
		{
			for (uint8_t k = 1; k<=loopLot; k++)
			{
			SETBITALT(_SFR_IO8(portLot), pinLot);
			_delay_ms(4);
			CLEARBITALT(_SFR_IO8(portLot), pinLot);
			CLEARBITALT(_SFR_IO8(portMed), pinMed);
			CLEARBITALT(_SFR_IO8(portLittle), pinLittle);
			_delay_ms(4);
			}
		}
	}
}
if (stepLot<stepMed) // vypocet pro druhou fazi
{
	pinLittle = pinMed;
	portLittle = portMed;
	stepLittle = stepMed;
	pinMed = pinLot;
	portMed = portLot;
	stepMed = stepLot;
	pinLot = pinLittle;
	portLot = portLittle;
	stepLot = stepLittle;
}
loopMed = stepMed;
if (loopMed==0)
{
	loopLot = stepLot;
	stepLot = 0;
}
else
{
	loopLot = stepLot/loopMed;
	stepLot = stepLot%(loopLot*loopMed);
}
for (uint8_t j = 0; j<=loopMed; j++)
```

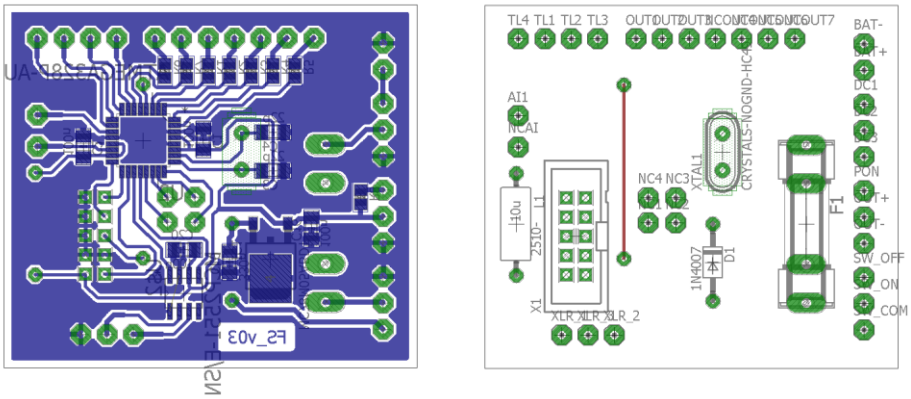
```
	{
		if (j>0)
		{
			SETBITALT(_SFR_IO8(portMed), pinMed);
		}
		if ((j>0)||(loopMed==0))
		{
			for (uint8_t k = 1; k<=loopLot; k++)
			{
				SETBITALT(_SFR_IO8(portLot), pinLot);
				_delay_ms(4);
				CLEARBITALT(_SFR_IO8(portLot), pinLot);
				CLEARBITALT(_SFR_IO8(portMed), pinMed);
				_delay_ms(4);
			}
		}
	}
	loopLot = stepLot; // pripadne dokrokovani posledniho motorku
	for (uint8_t k = 1; k<=loopLot; k++)
	{
		SETBITALT(_SFR_IO8(portLot), pinLot);
		_delay_ms(4);
		CLEARBITALT(_SFR_IO8(portLot), pinLot);
		_delay_ms(4);
	}
	objectData[51][1] = position1;    // zapis novych pozic do slovniku objektu
	objectData[52][1] = position2;
	objectData[53][1] = position3;
	l9942_set_reg_par(dacScaleCh1,0b010); // nastaveni nizsich proudu pro klidovy
	l9942_set_reg_par(dacScaleCh2,0b010); // stav
	l9942_set_reg_par(dacScaleCh3,0b010);
	l9942_write_par();
}
```

8 Ovládací prvek – nožní přepínač

V této kapitole provedu návrh ovládacího prvku, který je reprezentován formou nožního přepínače. Mnoho věcí je podobných s prvkem akčním, proto bude popis již stručnější a v jedné kapitole bude probráno jak hardwarové, tak softwarové vybavení.

8.1 Hardwarové řešení

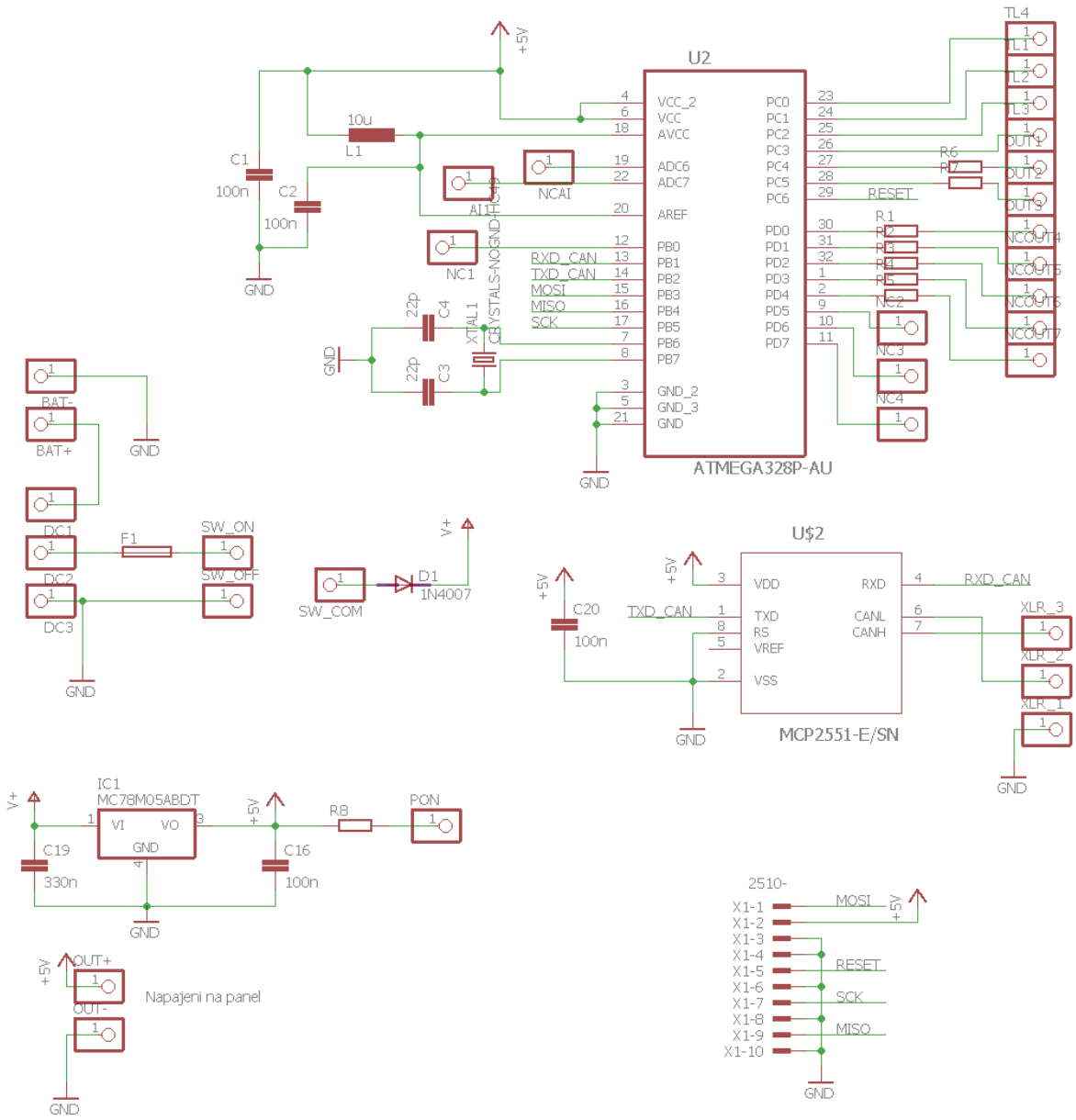
Na schématu (Obrázek 25 – Schéma nožního přepínače) si lze povšimnout, že pro nožní přepínač byl použit stejný mikroprocesor Atmega328P jako pro ovladač motorků. Procesor je zde ovšem v 32pinovém TQFP pouzdře. Stejně tak pro obvod MCP2551 bylo použito 8pinové SOIC pouzdro. Celý nožní přepínač je možno napájet z baterie 9 V. Plusový pól z baterie je přiveden na napájecí kontakt, který po připojení konektoru mechanicky odpojí bateriové napájení. Kladné napětí je přivedeno přes pojistku na vypínač. Dále je zde ochranná dioda proti přepólování, za kterou je napětí přivedeno na lineární stabilizátor 5 VDC. Jelikož je použit analogový vstup pro ovládání pomocí pedálu, je napětí pro napájení ADC převodníku přivedeno přes tlumivku, pro snížení šumu. Jinak je pro hodinový kmitočet mikroprocesoru opět použit externí krystal 14,7456MHz. Samozřejmostí je zde opět použití 10pinového ISP konektoru pro programování mikroprocesoru. Hodnoty předřadných rezistorů jsou obdobné jako u dříve popsaného ovladače motorků. Všechny ostatní hodnoty kondenzátorů a tlumivky (10 µH) vycházejí z doporučení výrobce.



Obrázek 24 – DPS nožního přepínače

Dále nožní přepínač reálně obsahuje tři vstupy z tlačítek, jimž jsou přiřazeny výstupy v podobě stavových LED diod. Použit je i jeden analogový vstup pro získání hodnoty z pedálu a jeden přepínač sloužící pro povolení či zakázání snímání onoho analogového vstupu. Veškerá tlačítka a LED diody jsou umístěny na panelu a na samotné desce plošných spojů (Obrázek 24 – DPS nožního přepínače) jsou pouze vyvedeny piny pro dané vodiče. Ve skutečnosti je na desce vyvedeno více pinů, které slouží ovšem jako rezerva. Dále si lze povšimnout vyvedení potenciálů 0 VDC a 5 VDC. Tyto potenciály jsou použity pro

připojení k tlačítkům, LED diodám a potenciometru pro snímání analogové hodnoty. Zmíněný potenciometr je samozřejmě mechanicky spojen s pedálem.



Obrázek 25 – Schéma nožního přepínače

8.2 Softwarové řešení

Struktura programu je obdobná jako u ovladače motorků, popsaného v předešlé kapitole. Program opět běží v hlavní smyčce a komunikace probíhá pomocí vykonávaných přerušení. Promítlo se zde ale samozřejmě rozdílné hardwarové zapojení, a to například ve výkonu přerušení od změny pinu PB1, místo PD5 a podobně. Hlavní smyčka programu obsahuje kromě načtení dat z vyrovnávacího pole a jejich následného zpracování, také neustálé

dotazování na stav tlačítek (Zdrojový kód 16 – Část hlavní smyčky). Toto dotazování je vyřešeno ve funkci „userAction“ (

Zdrojový kód 17 – Funkce userAction), která vrací výčtový datový typ dle provedené akce. Danými akcemi jsou pro každé ze tří tlačítek dlouhý či krátký stisk, a dále vypnutí či zapnutí pedálu. Při krátkém stisku tlačítka je vysláno příslušné PDO dle parametrů ve slovníku objektů, načež pokud není povoleno snímání analogové hodnoty z pedálu, je rozsvícena LED dioda odpovídající danému tlačítku (ostatní diody jsou zhasnuty). Dlouhý stisk tlačítka vyvolá vyslání zprávy SDO write, která provede zápis do objektu „Ulož současné polohy“ ovladače motorků (Tabulka 4 – Výběr některých objektů prvku typu „Ovladač motorů“). První čtyři bajty objektu reprezentují hodnotu PDO, která je zvolena dle hodnoty PDO vyslané při krátkém stisku konkrétního tlačítka. Další bajt reprezentuje hodnotu stavu v ovladači motorků a posledním bajtem je určeno maskování. To je zde zvoleno jako binární hodnota „111“, která aktivuje všechny tři motorky. Dalšími typy uživatelských akcí, kromě výše zmíněných, je vypnutí či zapnutí snímání hodnot z analogového vstupu (pedálu). To je provedeno jednoduše zápisem příslušného bitu do ovládacího registru ADC převodníku. Tento zápis je spojen buď s rozsvícením všech tří LED diod při zapnutí snímání, nebo naopak se zhasnutím všech tří LED diod v opačném případě.

Po vykonání příslušné akce v hlavní smyčce programu následuje případné snímání analogové hodnoty. Použitý analogově digitální převodník v daném mikroprocesoru Atmega328P je 10bitový. Pro dané účely a i z hlediska omezení šumu zde bude použito pouze 8 nejvyšších bitů (dva nejméně významné bity jsou ignorovány). Proto je do pomocné proměnné zkopírován pouze jeden registr obsahující právě 8 nejvyšších bitů. Vyslání příslušné hodnoty PDO je podmíněno změnou předcházející navzorkované hodnoty.

Zdrojový kód 16 – Část hlavní smyčky

```
switch (userAction())            // vykonani uzivatelskych akci
{
	case dIn1Short:              // kratky stisk tlacitka 1
	{
		setPDO(&objectData[151][1], &dataToTransmit);
		transmission(&dataToTransmit, 10, 1);
		if (!((ADCSRA>>ADEN)&1))
		{
		CLEARBIT(DO1);
		SETBIT(DO2);
		SETBIT(DO3);
		}
	}
	break;
	case dIn1Long:               // dlouhy stisk tlacitka 1
	{
		uint8_t byteArray[6];
		for (uint8_t i = 0; i<=3; i++)
		{
			byteArray[i] = objectData[151][i+1];
		}
```

```
                byteArray[4] = 1;
                byteArray[5] = 0b00000111;
                setSDOwrite(ADDRESS_CONTR,101,6,byteArray,&dataToTransmit);
                transmission(&dataToTransmit, 10, 1);
        }
        break;
        .
        .
        case dIn4OffToOn:   // prechod z vypnuteho do zapnuteho stavu, povolit ADC
        {
                ADCSRA |= (1<<ADEN);
                CLEARBIT(DO1);
                CLEARBIT(DO2);
                CLEARBIT(DO3);
        }
        break;
        case dIn4OnToOff:   // prechod ze zapnuteho do vypnuteho stavu, zakazat ADC
        {
                ADCSRA &= ~((1<<ADEN));
                SETBIT(DO1);
                SETBIT(DO2);
                SETBIT(DO3);
        }
        break;
}
if ((ADCSRA>>ADEN)&1)      // snimani analogoveho vstupu pokud povoleno
{
        ADCSRA|=(1<<ADSC);                // spustit prevod
        while(!(ADCSRA & (1<<ADIF)));     // cekani na dokonceni
        ADCSRA|=(1<<ADIF);                // snulovani priznaku
        anaInp = ADCH;                    // zapis nejvyssich 8 bitu
        if (anaInp!=anaInpOld)            // prenos PDO
        {
                pdoVal[3] = anaInp;
                setPDO(pdoVal,&dataToTransmit);
                transmission(&dataToTransmit,10,1);
        }
        anaInpOld = anaInp;
        _delay_ms(10);
}
```

V další ukázce zdrojového kódu níže (

Zdrojový kód 17 – Funkce userAction) je zobrazen způsob zjišťování uživatelských akcí. Probíhá zde postupné dotazování na stav vstupních pinů. Pokud je stav vstupního pinu pro tlačítko logická „0“, proběhne další vyhodnocování. Nejdříve je zde zpoždění díky možným zákmitům na tlačítku. Po předpokládaném ustálení stavu následuje měření doby stisku tlačítka ve „while“ smyčce. V čase 1,55 s LED dioda odpovídající danému tlačítku problikne, což znázorňuje dosažení dlouhého stisku tlačítka. Po následném uvolnění následuje vyhodnocení délky stisku a zápis do proměnné, která je na konci vrácena funkcí. Než je možné pokračovat v kódu dále, je nejprve nutné ošetřit případné zákmity spojené s uvolněním tlačítka. K tomu musí být splněna podmínka dvaceti naskenovaných po sobě jdoucích logických „1“ při periodě 1 ms.

Pro zjištění zapnutí či vypnutí přepínače pro povolení pedálu je třeba znát nejdříve předchozí stav. Ten je uložen v globální proměnné „din4PrevState“. Pokud je hodnota proměnné „0“ a zároveň je zaznamenána na daném vstupním pinu logická „1“, je to vyhodnoceno jako zapnutí pedálu. Při skenování pinu s periodou 1 ms je navíc vyhodnocována pomocná proměnná, která chrání program před nechtěným zacyklením v případě krátkého pulsu na vstupu. Po provedení smyčky se předpokládá ustálená hodnota na pinu, která v případě logické „1“ vede k nastavení návratové hodnoty do stavu „dIn4OffToOn“. Obdobně je logicky provedena akce související s vypnutím přepínače.

Zdrojový kód 17 – Funkce userAction

```
enum typyAkci tlacitkoAkce = none;
uint8_t tlac = 0, pom = 0;
if (ISCLEAR(DIN1))                // stav tlacitka 1
{
       _delay_ms(150);            // ignorovani zakminu pri sepnuti
       tlac = 0;
       while (ISCLEAR(DIN1))
       {
              _delay_ms(20);
              if(tlac<255) tlac++;
              if(tlac==70)
              {
                     FLIPBIT(DO1);
                     _delay_ms(100);
                     FLIPBIT(DO1);
              }
       }
       if (tlac<70)
       {
              tlacitkoAkce = dIn1Short;  // stisk kratsi nez 1,55 sekundy
       }
       else
       {
              tlacitkoAkce = dIn1Long;
       }
       tlac = 0;
       do                                // osetreni zakmitu pri uvolneni
       {
              _delay_ms(1);
              if (ISCLEAR(DIN1))
              tlac = 0;
              else
              tlac++;
       }
       while(tlac<20);
}
.
.
if (ISSET(DIN4)&&(din4PrevState==0))     // prechod z 0 do 1
{
       tlac = 0;
       while ((tlac<20)||(pom>250))      // pom promenna proti zacykleni
       {
              _delay_ms(1);
              if(ISCLEAR(DIN4))
```

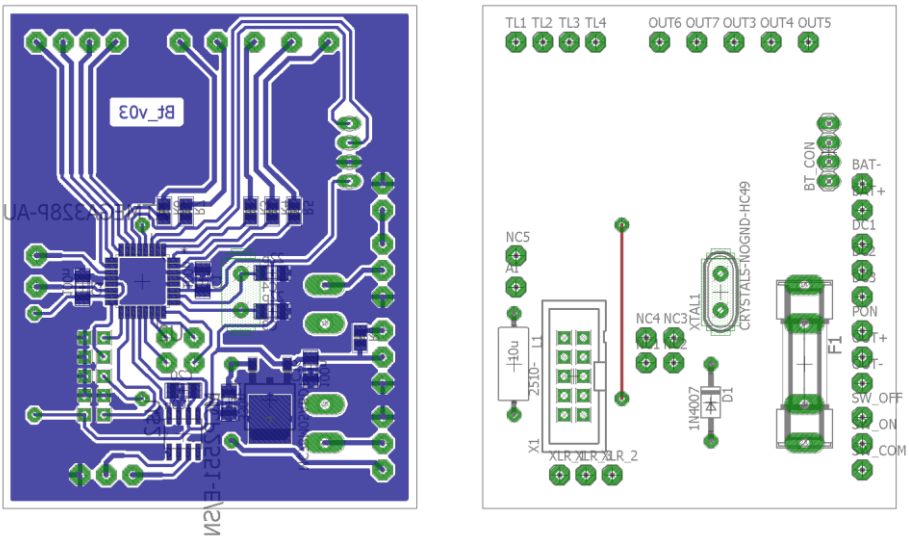
```
                {
                        tlac = 0;
                }
                tlac++;
                pom++;
        }
        if (ISSET(DIN4))    // pokud to nebylo zakolisani tak vrat danou akci
        {
                tlacitkoAkce = dIn4OffToOn;
                din4PrevState = 1;
        }
}
if (ISCLEAR(DIN4)&&(din4PrevState==1))   // prechod z 1 do 0
{
        tlac = 0;
        while ((tlac<20)||(pom>250))      // pom proti zacykleni
        {
                _delay_ms(1);
                if(ISSET(DIN4))
                {
                        tlac = 0;
                }
                tlac++;
                pom++;
        }
        if (ISCLEAR(DIN4))  // pokud to nebylo zakolisani tak vrat danou akci
        {
                tlacitkoAkce = dIn4OnToOff;
                din4PrevState = 0;
        }
}
return tlacitkoAkce;
```

9 Bluetooth modul

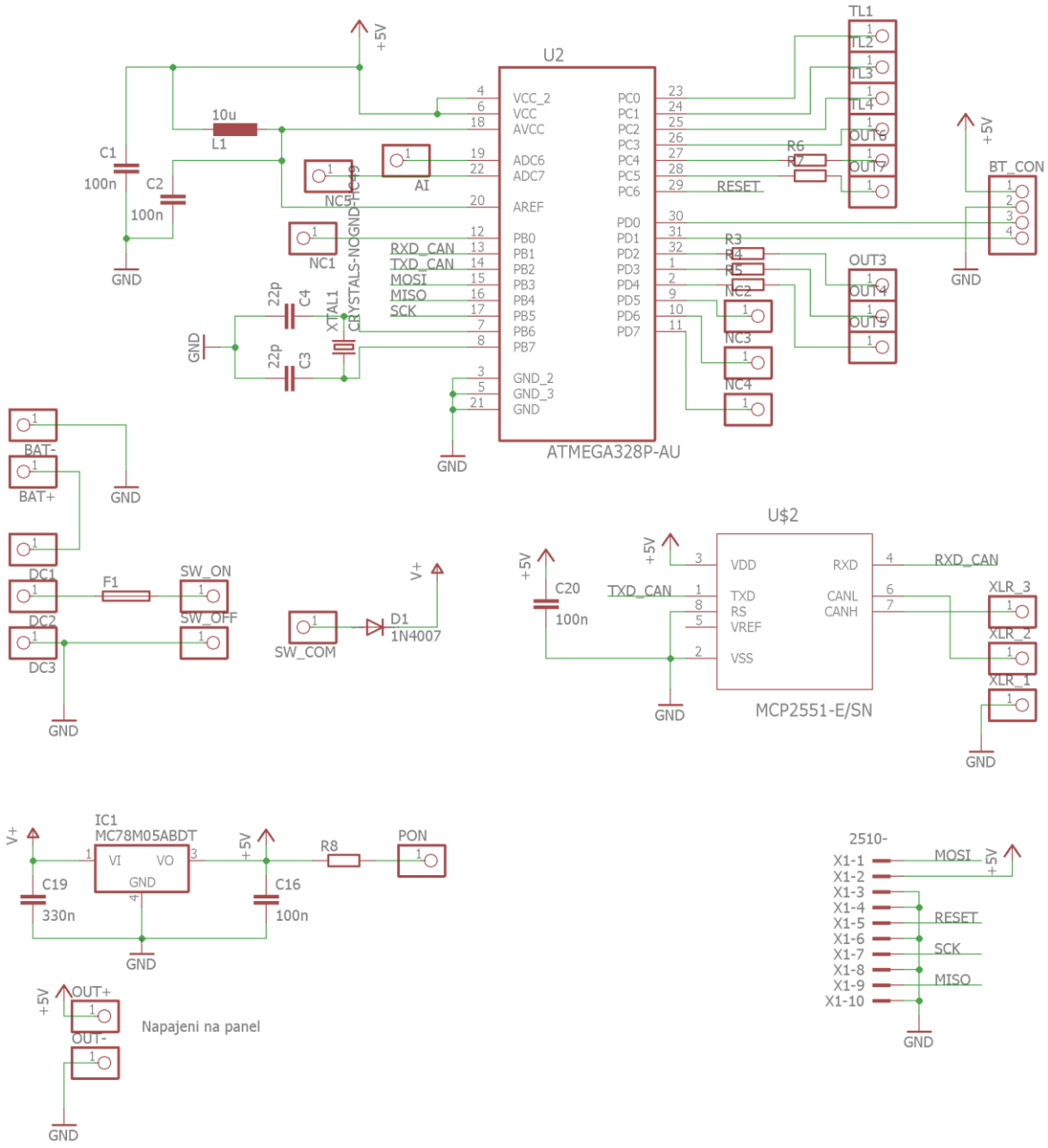
Tato kapitola obsahuje popis bluetooth modulu, který slouží jako jakýsi prostředník mezi komunikací na sběrnici a android aplikací. Nejdříve bude pojednáno o hardwarovém řešení daného prvku a poté bude popsána část týkající se softwaru. Výklad bude proveden již stručně, jelikož je řešení obdobné jako u předchozích prvků. Budou zde vyzdviženy hlavně rozdíly oproti prvkům popsaným v předešlých kapitolách.

9.1 hardwarové řešení

Na obrázku níže (Obrázek 27 – Schéma bluetooth modulu) je znázorněno mnou navržené schéma. Jedná se prakticky o téměř totožné schéma jako u nožního přepínače a i všechny hodnoty součástek jsou obdobné. Je tu ale jeden zásadní rozdíl, a to že místo rezervních výstupů je na pinech PD0 a PD1 mikroprocesor spojen s SPP bluetooth modulem HC-06. Jedná se o zařízení, které veškerou komunikaci přes bluetooth převádí na sériovou komunikaci typu RS232. Mikroprocesor je vybaven periferií UART, kdy zmíněné piny PD0 a PD1 plní právě alternativní funkci přijímání a odesílání dat. Na použitém modulu HC-06 lze provést různá nastavení týkající se bitové rychlosti sériové komunikace, použití paritního bitu, hodnoty čtyřmístného pinu pro spárování a podobně. Tato nastavení jsou přístupná, pokud modul není spárován s žádným zařízením (blikající LED) a provádějí se pomocí tzv. AT příkazů (komunikace opět po sériové lince). Co se týče parametrů komunikace, tak byl použitý modul nastaven na 8bitovou komunikaci bez parity s bitovou rychlostí 38400 Baud. Stejně je samozřejmě nastavena i UART periferie mikroprocesoru.



Obrázek 26 – DPS bluetooth modulu



Obrázek 27 – Schéma bluetooth modulu

9.2 Softwarové řešení

Program stejně jako u předchozích prvků běží v hlavní smyčce, která je přerušována případným příjmem ze sběrnice. Hlavní smyčka obsahuje opět nejprve načítání dat z vyrovnávacích proměnných a jejich následné zpracování. Funkce „performBusAction“ v tomto případě obsahuje pouze možnost zápisu a čtení parametrů slovníku objektů, a dále přeposílání všech přijatých dat přes sériovou linku komunikující s bluetooth modulem HC-06. Dále je v hlavní smyčce umístěno dotazování se na stav přijatých dat po sériové lince (Zdrojový kód 18 – Přijímání dat z HC-06). Nejdříve je nastaven čítač, jehož vstup je

hodinový signál mikroprocesoru, vydělený hodnotou 32. Porovnávací hodnota je nastaven tím způsobem, že odpovídá časové hodnotě 325 µs. Tento čítač je dále využíván ve smyčce, která přijímá „balík“ 15 bajtů od Android zařízení. Přenos jednotlivých bajtů nesmí být od sebe vzdálen právě více než 325 µs, jinak jsou data prohlášena za neplatná. V případě platnosti dat je nutné provést navíc výpočet CRC, proto je zde switch-case blok, který dle přijatých dat provede nastavení čistých dat k odeslání.

Zdrojový kód 18 – Přijímání dat z HC-06

```
if((UCSR0A & (1<<RXC0)))   // prisla data?
{
	TCCR2B ^= (0b011<<0 ^ TCCR2B) & (0b111<<0); // nastaveni citace clock/32
	OCR2A = 150;                      // odpovida 325 us – vice nez UART ramec
	TIFR2 |= (1 << OCF2A);
	TCNT2 = 0;
	uint8_t valid = 1;
	for(uint8_t i=0; i<15;i++) // prijem 15ti bajtu
	{
		while(!(UCSR0A & (1<<RXC0)))
		{
			if ((TIFR2 & (1 << OCF2A))) // pokud vyprsel cas, data neplatna
			{
				valid = 0;
			}
		}
		dataToTransmit[i] = UDR0;
		TCNT2 = 0;    // resetovani citace
	}
	if (valid==1)
	{
		dataToTransmit[13] = objectData[3][1];   // zdrojova adresa
		switch ((dataToTransmitPom[8]>>5)&0b11)  // jaky typ zpravy
		{
			case 0b00:    // EMERGENCY
			etEmcy(dataToTransmitPom[0],&dataToTransmit);
			break;
			case 0b01:    // PDO
			setPDO(&dataToTransmitPom,&dataToTransmit);
			break;
			case 0b10:    // SDO
				switch ((dataToTransmitPom[8]>>3)&0b11)  // jaky typ SDO
				{
				case 0b00:    // SDO write
				if ((dataToTransmitPom[10]>>7)&0b1)// error priznak?
				{      // SDO write error
				setSDOwriteErr(dataToTransmitPom[9],
					dataToTransmitPom[12],dataToTransmitPom[0],
					&dataToTransmit);
				}
				else
				{      // SDO write
				setSDOwrite(dataToTransmitPom[9],dataToTransmitPom[12],
					(dataToTransmitPom[8]&0b111),&dataToTransmitPom,
					&dataToTransmit);
				}
				break;
				case 0b01:    // SDO write required
				setSDOwriteReq(dataToTransmitPom[9],
					dataToTransmitPom[12],&dataToTransmit);
```

```
                        break;
                        case 0b10:    // SDO info
                        if ((dataToTransmitPom[10]>>7)&0b1) // error priznak?
                        {      // SDO info error
                        setSDOinfoErr(dataToTransmitPom[9],dataToTransmitPom[12,
                               dataToTransmitPom[0],&dataToTransmit);
                        }
                        else
                        {      // SDO info
                        setSDOinfo(dataToTransmitPom[9],dataToTransmitPom[12],
                               (dataToTransmitPom[8]&0b111),
                               &dataToTransmitPom,&dataToTransmit);
                        }
                        break;
                        case 0b11:    // SDO info required
                        setSDOinfoReq(dataToTransmitPom[9],dataToTransmitPom[12]
                                      ,&dataToTransmit);
                        break;
                        }
                  break;
                  case 0b11:    // HEARTBEAT
                        setHeartBeat(&dataToTransmit);
                  break;
            }
            transmission(&dataToTransmit,10,1);      // odvysilani nastavenych dat
      }
}
```

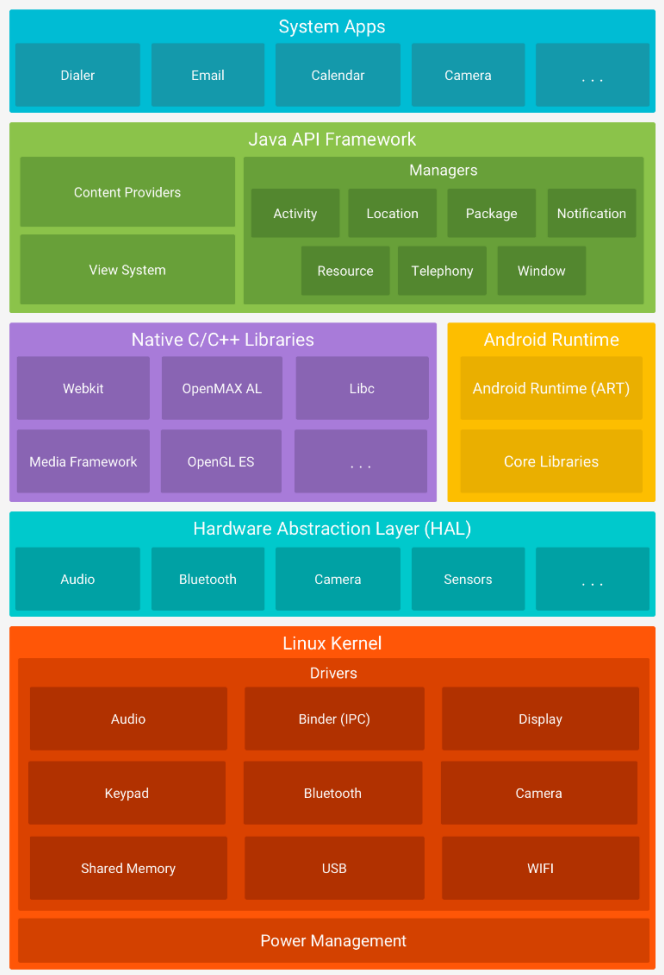
10 Software pro Android

Pro vývoj aplikace pro OS Android bylo využito vývojové prostředí Android Studio s použitím programovacího jazyku Java. V dalším textu bude nejprve rozebrán princip vývoje aplikací pro Android spolu s jeho vlastnostmi. Dále již bude popsána implementace konkrétní navrhované aplikace.

10.1 Obecně o vývoji pro OS Android

Android je otevřený Linuxový operační systém, určený pro přenosná zařízení, jako jsou telefony, tablety a podobně. V poslední době je používán například i u „chytrých“ televizí. Různé verze Androidu jsou vzestupně kódovány podle počátečních písmen abecedy (Aestro, Blender, Cupcake, Donut atd.), a je s nimi spjato určité číslo definující tzv. API level (application programming interface, česky rozhraní pro programování aplikací).

Pro vývoj Android aplikací existují různé možnosti, od použití různých grafických nástrojů určených zejména pro začátečníky (např. MIT App Inventor) až po použití profesionálních vývojových prostředí jako jsou Eclipse, či Android Studio. Právě Android Studio je společností Google, která vede vývoj OS Android, určeno jako oficiální vývojové prostředí.



Obrázek 28 – Komponenty OS Android (APP DEV)

OS Android je soubor softwarových komponent, které lze rozdělit do několika vrstev dle jejich funkčnosti (Obrázek 28 – Komponenty OS Android (APP DEV)). Vývoj požadované aplikace bude probíhat právě v nejvyšší vrstvě, přičemž bude využíváno zejména funkcionalit vrstvy nižší, která definuje rozhraní pro programování aplikací v Javě (Java API Framework).

Aplikace se v Androidu skládají z tzv. aplikačních komponent, které je možné dále využívat i v jiných aplikacích. Toto řešení je výhodné, pro svou reprodukovatelnost stejného kódu různými aplikacemi. Díky tomu se stejná funkcionalita nemusí programovat vícekrát a vývojář má ulehčenou práci. Mezi hlavní aplikační komponenty patří

- „Activity“,
- „Service“,
- „Broadcast Receiver“,
- „Content Provider“.

Komponenta „Activity“ (česky jako Aktivita) reprezentuje obrazovku s uživatelským prostředím. Dá se říci, že tato komponenta vykonává uživatelské akce spojené s danou obrazovkou. Příkladem může být u aplikace pro zasílání e-mailů jedna aktivita pro zobrazování seznamu e-mailů, další pro psaní a další pro čtení e-mailů. Pokud má aplikace více aktivit, musí být jedna označena jako aktivita, která se zobrazí po spuštění.

Další komponenta „Service“ (česky Služba) běží v pozadí a provádí dlouho trvající operace. Jako příklad lze uvést přehrávání hudby v pozadí, zatímco uživatel může interagovat s jinou aplikací. Služba tedy neposkytuje žádné uživatelské rozhraní.

Aplikační komponenta „Broadcast Receiver“ slouží pro „naslouchání“ různých oznámení od aplikací, či systému. Sama navrhovaná aplikace může také určitá oznámení vysílat, čímž dává ostatním možnost reakce. Je to tedy „Broadcast Receiver“ který zachytí tuto komunikaci a vyvolá potřebnou akci.

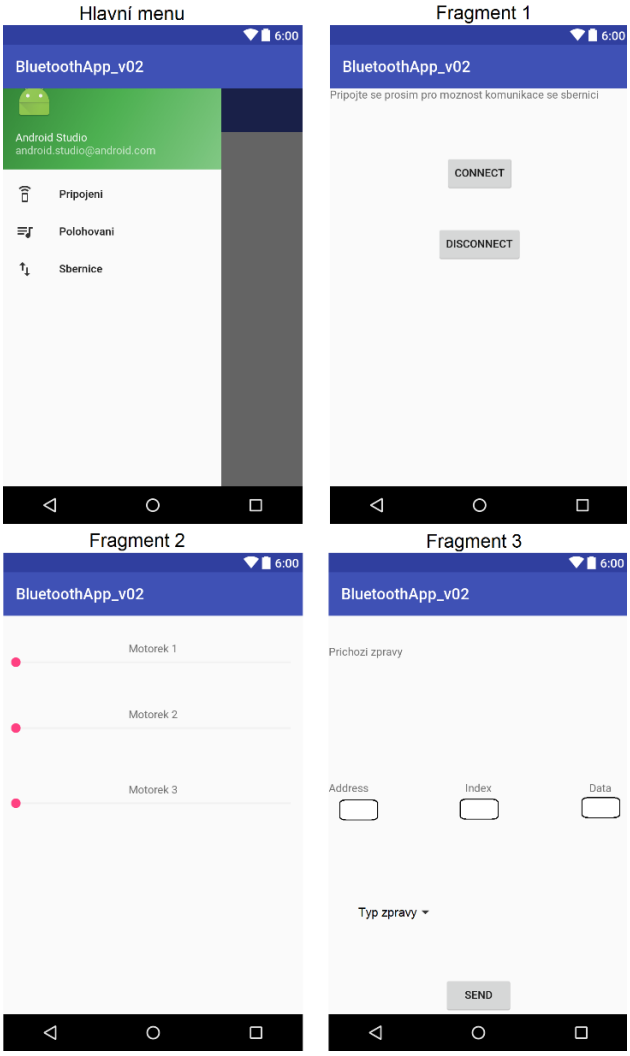
Poslední popisovanou komponentou je „Content Provider“, která poskytuje rozhraní pro sdílení dat mezi aplikacemi.

Kromě výše zmíněných, existují i jakési přídavné aplikační komponenty, které doplňují, či vzájemně svazují tyto hlavní. Příkladem jsou „Fragments“ (česky Fragmenty), které reprezentují určitou část zobrazované aktivity. Ještě je vhodné se zmínit o souboru „AndroidManifest.xml“, který se musí povinně nacházet v kořenovém adresáři každé aplikace. Obsahem tohoto souboru je popis základních informací o aplikaci (jako je například seznam použitých komponent) pro OS Android.

Doposud byla tedy velmi stručně popsána problematika vývoje aplikací pro OS Android, přičemž podrobný výklad je možné dohledat v použité literatuře (Developers, 2017). Dále bude probrána implementace navrhované aplikace.

10.2 Implementace programu

Navrhovaná aplikace se skládá z komponenty služby, která běží na pozadí a obstarává bluetooth komunikaci s modulem HC-06. Dále je zde hlavní aktivita, sloužící jako uživatelské rozhraní. V této hlavní aktivitě dochází k přepínání mezi jednotlivými fragmenty, které jsou celkem tři. První fragment je spuštěn implicitně po startu a obsahuje obrazovku s tlačítky pro připojení či odpojení od daného modulu. V dalším fragmentu je možno nastavovat polohu motorků pomocí grafického znázornění (objekt „SeekBar“). V posledním fragmentu lze posílat obecné zprávy na sběrnici, přičemž jsou zde také zobrazovány všechny zprávy, které navržený bluetooth modul odposlechne ze sběrnice. Pro celkové uspořádání byl použit tzv. „Navigation Drawer“, což je model uspořádání zahrnující postranní panel, kde je v tomto případě možné volit mezi zobrazovanými fragmenty. Šablona kódu pro toto uspořádání byla použita z použitého zdroje (Developers, 2017). Z tohoto zdroje vychází i použitý kód pro bluetooth spojení a komunikaci s daným modulem. Na obrázku níže (Obrázek 29 – Podoba Android aplikace), je zobrazen výsledný vzhled aplikace.



Obrázek 29 – Podoba Android aplikace

Celý kód je poměrně obsáhlý, a je možné ho dohledat v přiloženém CD (ostatně jako všechny ostatní zdrojové kódy a navržená schémata). V dalším textu bude popsán tedy hlavní princip s pouze několika ukázkami.

Při spuštění aplikace se vytvoří hlavní aktivita. V tzv. „Callback" funkci „onCreate", která je volána OS Android při vytvoření (instancializaci) aktivity, jsou vytvořeny potřebné instance fragmentů, třídy BluetoothAdapter a podobně. Zároveň je zobrazen fragment s možností provedení připojení k modulu. Po stisku tlačítka „Connect" (připojit), je spuštěna tzv. „Bound Service" (Svázaná služba), obstarávající připojení k modulu a následnou komunikaci. Druhým tlačítkem je možné se ze služby naopak odhlásit, čímž je instance služby zlikvidována. Zastavení svázané služby proběhne vždy teprve po odhlášení všech aktivit svázaných s danou službou. V tomto případě je ke službě přistupováno pouze z jedné, hlavní aktivity, proto je po odhlášení služba zastavena. Ve zdrojovém kódu níže (Zdrojový kód 19 – Akce tlačítek připojení/odpojení) je znázorněno ošetření stisku zmíněných tlačítek.

Zdrojový kód 19 – Akce tlačítek připojení/odpojení

```
@Override
public void onClick(View v) {
switch (v.getId()) {
	case R.id.StartBConnect:
		if(mainActivity.mBounded) {
		MainActivity.toast.setText("Jiz pripojeno");
		MainActivity.toast.show();
		}else {
		if (!mainActivity.mBluetoothAdapter.isEnabled()) {
		MainActivity.toast.setText
					("Zapnete Bluetooth a zkuste znovu");
		MainActivity.toast.show();
		Intent enableBtIntent = new
					Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
		startActivityForResult(enableBtIntent,
					mainActivity.REQUEST_ENABLE_BT);
			}
		else{
		Intent i = new Intent(mainActivity,
					BluetoothService.class);
		mainActivity.bindService(i, mainActivity.mConnection,
					Context.BIND_AUTO_CREATE);
		}
		}
	break;
	case R.id.StartBDisconnect:
		if(mainActivity.mBounded) {
		mainActivity.unbindService(mainActivity.mConnection);
		mainActivity.mBounded = false;
		mainActivity.mBluetoothService = null;
		}
		else{
		MainActivity.toast.setText("Jiz odpojeno");
		MainActivity.toast.show();
		}
	break;
	}
}
```

Při svázání dané služby „BluetoothService“ je OS Android automaticky volána funkce „onBind“ ze které je spuštěna níže popsaná funkce „connectToBus“ (Zdrojový kód 20 – Funkce „connectToBus“). V této funkci probíhá případné vytvoření vlákna, sloužícího pro připojení k modulu. Pokud se ve vlákně nepodaří připojit k modulu, je provedeno odhlášení se od služby. Dané odhlášení lze ale provést pouze z hlavní aktivity, proto je ve službě vytvořeno veřejné rozhraní, obsahující funkce „unbindBluetoothService“ a „updateClient“. Toto rozhraní je implementováno právě v hlavní aktivitě. První zmíněná funkce slouží pro odhlášení ze služby a druhá pro aktualizaci uživatelského rozhraní v závislosti na přijatých datech.

Zdrojový kód 20 – Funkce „connectToBus“

```
public void connectToBus() {
Set<BluetoothDevice> pairedDevices =
MainActivity.mBluetoothAdapter.getBondedDevices();
if (pairedDevices.size() > 0) {
	for (BluetoothDevice device : pairedDevices) {
		String deviceName = device.getName();
		if(deviceName.equals("HC-06")){
			mBluetoothDevice = device;
			mConnectThread = new ConnectThread(mBluetoothDevice);
			mConnectThread.start();
			Message readMsg = MainActivity.mHandler.obtainMessage
				(MainActivity.MessageConstants.MESSAGE_PROGRESS,1 ,1,
				"Pripojovani");
			readMsg.sendToTarget();
			MainActivity.toast.setText("Pokus o pripojeni");
			continue;
		}
		else{
			MainActivity.toast.setText
					("Sparuje zarizeni HC-06 a opakujte znovu");
		}
	}
}
else{
	MainActivity.toast.setText("Nesparovana zadna zarizeni");
}
MainActivity.toast.show();
}
```

Po navázání spojení může probíhat komunikace se sběrnicí ve dvou zbylých fragmentech. V prvním fragmentu „ContFragment“ je možné pouze nastavování poloh motorků, přičemž je zde určité grafické znázornění požadované polohy. Druhý fragment „BusCommFragment“ využívá služby „BluetoothService“ a její funkce „write“ pro posílání libovolných zpráv na sběrnici. Příjem je řešen již zmíněnou implementací funkce „updateClient“, která aktualizuje textové pole daného fragmentu v závislosti na přijaté zprávě. Ve zdrojovém kódu níže (Zdrojový kód 21 – Akce tlačítka pro odeslání zprávy) je znázorněna akce spjatá s odesláním požadované zprávy.

Zdrojový kód 21 – Akce tlačítka pro odeslání zprávy

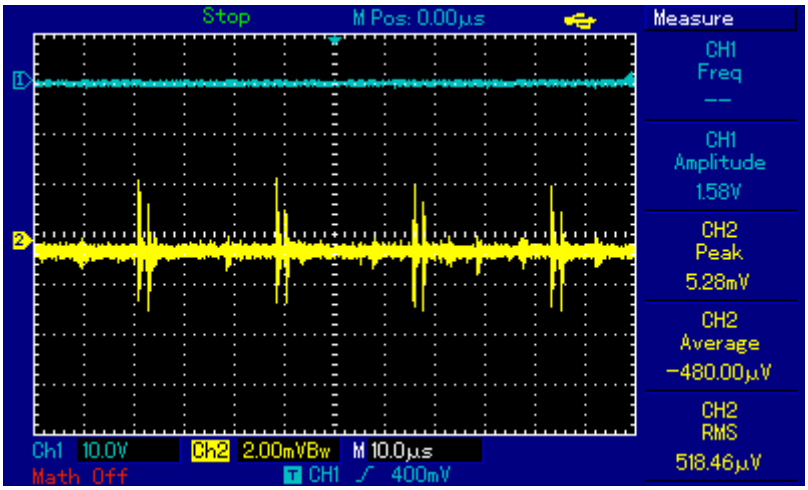
```
@Override
public void onClick(View v) {
switch (v.getId()) {
	case R.id.BusCommBSend:
		if(mainActivity.mBounded){
			String multiLines1 = editText_address.getText().toString();
			String multiLines2 = editText_index.getText().toString();
			String multiLines3 = editText_data.getText().toString();
				if((multiLines1 != null && !multiLines1.isEmpty())
				&&(multiLines2 != null && !multiLines2 .isEmpty())
				&&(multiLines3 != null && !multiLines3 .isEmpty())){
				byte[] bytes = busComPrepareData();
				mainActivity.mBluetoothService.write(bytes);
				MainActivity.toast.setText("Zprava odeslana");
				MainActivity.toast.show();
				}else{
				MainActivity.toast.setText
						("Vsechna pole musi byt vyplnena");
				MainActivity.toast.show();
			}
			}else{
				MainActivity.toast.setText("Nepripojeno");
				MainActivity.toast.show();
			}
	break;
	}
}
```

11 Měření přeslechů

Při provozu navrženého systému může docházet k nechtěnému ovlivňování užitečného kytarového signálu. To je způsobeno přeslechy buďto z probíhající komunikace po sběrnici, či z důvodu polohování motorků, jelikož je pro řízení proudu vinutími použit PWM signál. Dále provedu měření, zaměřující se právě na tyto rušivé signály.

Princip měření je takový, že do vstupu zesilovače je připojena elektrická kytara. To je provedeno standardním stíněným, nesymetrickým nástrojovým kabelem. Dále na onom vstupu zesilovače je osciloskopem prováděno měření naindukovaných signálů, z čehož se vychází při výpočtu poměru signál/šum. Geometrické uspořádání vzájemné polohy kabelů a všech zařízení bylo zvoleno náhodně, nicméně v průběhu celého měření shodně. Pro měření byl použit osciloskop UNI-T UTD2102CEL (výrobní číslo 5120009425).

Na obrázku níže (Obrázek 30 – Průběh z osciloskopu při klidovém stavu) je znázorněn průběh vstupního signálu (kanál 2 - žlutě) v klidovém stavu, kdy neprobíhá komunikace ani polohování motorků a vše je vypnuté. Lze si povšimnout přítomnosti neznámého zdroje rušení, jehož efektivní hodnota činí asi 518 µV.



Obrázek 30 – Průběh z osciloskopu při klidovém stavu

Pro výpočet poměru signál/šum platí následující rovnice:

$$SNR = 10 \cdot \log_{10} \frac{P_{signal}}{P_{noise}} = 20 \cdot \log_{10} \frac{V_{signal_{RMS}}}{V_{noise_{RMS}}} \quad , \qquad (9)$$

kde: SNR – je poměr signálu k šumu,

P_{signal} – je výkon signálu,

P_{noise} – je výkon šumu,

$V_{signal_{RMS}}$ – je efektivní hodnota signálu,

$V_{noise_{RMS}}$ – je efektivní hodnota šumu.

Efektivní hodnota užitečného kytarového signálu závisí samozřejmě na mnoha faktorech, jako je například síla úderu nebo použitý snímač. Tato hodnota bude dle provedeného měření dále uvažována jako $V_{signal_{RMS}}$ = 80 mV. Výsledek poměru signál/šum v klidovém stavu (označen jako SNR_{still}) je vypočten níže (10):

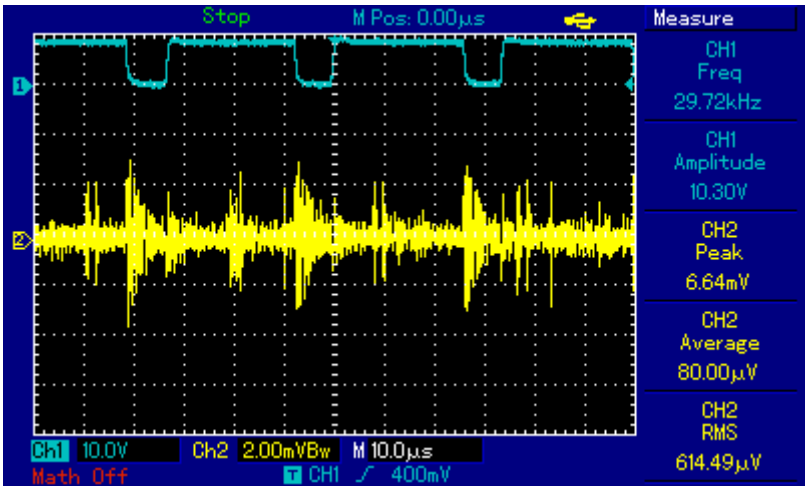
$$SNR_{still} = 20 \cdot \log_{10} \frac{V_{signal_{RMS}}}{V_{noise_{RMS}}} = 20 \cdot \log_{10} \frac{80}{0{,}518}\ , \tag{10}$$

$$SNR_{still} \approx 43{,}775\ [dB]. \tag{11}$$

Na dalším obrázku (Obrázek 31 – Průběh z osciloskopu při polohování motorků) jsou znázorněny průběhy při polohování motorků. V prvním kanálu je zobrazeno napětí na vinutí, které je realizováno PWM modulací. Druhý kanál znázorňuje opět měřený vstup. Lze si povšimnout vyšší hodnoty $V_{motorNoise_{RMS}} =$ 614 µV, která reprezentuje efektivní hodnotu šumu při polohování motorků. Výpočet poměru signál/šum (v tomto případě označeném jako SNR_{motor}), je proveden v rovnici níže (12):

$$SNR_{motor} = 20 \cdot \log_{10} \frac{V_{signal_{RMS}}}{V_{motorNoise_{RMS}}} = 20 \cdot \log_{10} \frac{80}{0{,}614}\ , \tag{12}$$

$$SNR_{motor} \approx 42{,}299\ [dB]. \tag{13}$$



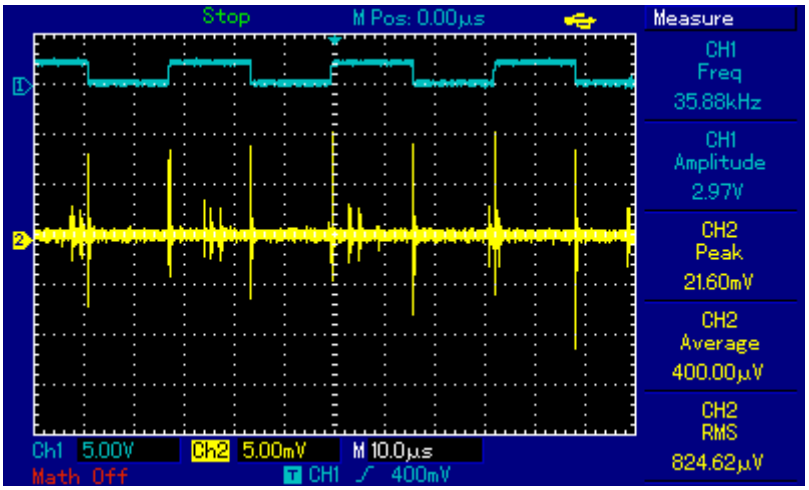
Obrázek 31 – Průběh z osciloskopu při polohování motorků

Při měření přeslechů způsobených komunikací na sběrnici byla sběrnice buzena obdélníkovým signálem s délkou pulsu 16 µs, což odpovídá jednomu bitovému času. Situace je znázorněna na obrázku níže (Obrázek 32 – Průběh z osciloskopu při buzení sběrnice). První kanál znázorňuje napětí mezi CAN_H a CAN_L, zatímco na druhém je samozřejmě opět měřený vstup zesilovače. Z měření vychází ještě vyšší hodnota

$V_{busNoise_{RMS}} = 825$ µV, znázorňující efektivní hodnotu šumu při buzení sběrnice. Výpočet poměru signál/šum (v tomto případě označeném jako SNR_{bus}), je proveden v rovnici níže (14):

$$SNR_{bus} = 20 \cdot \log_{10} \frac{V_{signal_{RMS}}}{V_{busNoise_{RMS}}} = 20 \cdot \log_{10} \frac{80}{0{,}825}\ , \qquad (14)$$

$$SNR_{bus} \approx 39{,}733\ [dB]. \qquad (15)$$



Obrázek 32 – Průběh z osciloskopu při buzení sběrnice

Z měření tedy vychází, že nejvíce je užitečný signál ovlivňován komunikací na sběrnici. Jelikož je nicméně jak bitová frekvence sběrnice, tak PWM signál od polohování motorků frekvenčně za slyšitelným spektrem, lze předpokládat na vstupu zesilovače jeho odstranění případnou dolní propustí.

Závěr

Cílem této práce byl návrh a fyzické sestavení systému pro dálkové ovládání poloh motorků. Tento požadavek byl splněn a výsledkem je fungující systém, který se skládá ze tří fyzicky vyrobených prvků a naprogramované aplikace pro OS Android. Mezi prvky je umožněna vzájemná komunikace na společné sběrnici, do které lze mimo jiné zasahovat i pomocí zařízení s OS Android pomocí vytvořené aplikace.

Pro účel zmíněné komunikace jsem navrhnul zvláštní protokol, umožňující připojení až 255 prvků (případné omezení je dané použitým řídicím obvodem fyzické vrstvy). Právě návrh protokolu byl jedním z hlavních témat této práce, proto mu byl věnován patřičný prostor. Podoba protokolu se odvíjí zejména od zvolené metody přístupu na sdílené médium. Daná metoda byla zvolena jako tzv. prioritní přístup, který využívá i známý CAN protokol. Právě s CAN a jeho aplikační nadstavbou CANopen sdílí mnou navržený protokol několik vlastností, nicméně jsou zde mimo jiné rozdíly hlavně ve větší provázanosti Linkové a Aplikační vrstvy (struktura a délka rámců se v navrženém protokolu výrazně liší dle typu vysílané zprávy). Díky zmiňovanému prioritnímu přístupu ke sdílenému médiu jsou zprávy děleny dle priority, a je možné za stanovených podmínek odvysílat zprávu okamžitě, bez časové prodlevy. Tato vlastnost je velmi žádoucí z důvodu co možná nejrychlejší reakce na požadovaný příkaz. Dále co se týče aplikační vrstvy tak protokol obsahuje koncept tzv. slovníku objektů, kdy jsou aplikační data reprezentována různými objekty, kde je každému objektu přiřazen index. Přístup k těmto datům probíhá právě na základě daného indexu.

Při návrhu protokolu byl brán ohled i na jeho provedenou implementaci, která je řešena pomocí 8bitových mikroprocesorů Atmega328P použitých v každém ze tří navržených prvků. Lze si povšimnout, že tedy například adresy prvků jsou 8bitové. Stejně tak zmiňovaný index sloužící k přístupu k objektům je také 8bitový. Tato skutečnost samozřejmě usnadňuje implementaci, ale je i s ohledem na danou aplikaci naprosto dostačující.

Pro samotné polohování byly po uvážení vybrány krokové motorky standardu NEMA8. V kontrastu se servo motorky, které mají většinou omezený rozsah natočení od 0° do 180°, krokové motorky tímto problémem netrpí a alespoň z tohoto pohledu se více hodí pro danou aplikaci. Navíc se velmi snadno řídí, a jelikož není předpokládáno přetěžování tak nepotřebují zpětnou vazbu.

Jelikož zpracovávané téma práce bylo relativně široké, je zde dost prostoru pro další rozšíření a vylepšení v různých konkrétních částech. Zajímavým námětem by mohlo být například rozšíření funkčnosti o možnost komunikace mezi samostatnými odlehlými sběrnicemi, kde by vznikalo určité časové zpoždění. To by mohlo být řešeno dalším protokolem, zapouzdřujícím protokol stávající. Kromě námětu týkajícího se komunikace zde jsou dále možnosti v podobě implementace více objektů v daných prvcích, nebo rozšíření aplikace pro OS Android.

Literatura

Atmel Corporation. 2016. *8-bit AVR Microcontrollers ATmega328/P DATASHEET COMPLETE.* [Dokument] San Jose : Atmel Corporation, 2016.

Boterenbrood, Danne. 2000. *CANopen high-level protocol for CAN-bus.* [Dokument] Amsterdam : Nikhef, 2000.

Davis, Rick. 2014. Neil Young's Whizzer Designed and Built by Rick Davis. *RDLX, LLC Worldwide Media Asset Brokers.* [Online] RDLX, LLC All Rights Reserved, 2014. [Citace: 15. 8 2017.] http://www.rdlx.com/neil_young.htm.

Developers, Android. 2017. Android Developers. [Online] Creative Commons Attribution 2.5., 2017. [Citace: 13. 8 2017.] https://developer.android.com/index.html.

Instruments, National. 2016. Motor Fundamentals - National Instruments. *National Instruments: Testovací, měřicí a integrované systémy - National Instruments.* [Online] © 2017 National Instruments Corporation, 24. Srpen 2016. [Citace: 29. 7 2017.] http://www.ni.com/white-paper/3656/en/.

Kickstarter. 2017. Rig Master - Guitar Bass Analog Gear Control by Novel Tones. *Kickstarter.* [Online] 2017. [Citace: 15. 8 2017.] https://www.kickstarter.com/projects/rigmaster/rig-master-guitar-and-bass-analog-gear-control?ref=nav_search.

Koopman, Philip. 2015. Best CRC Polynomials. *Checksum and CRC Central.* [Online] Carnegie Mellon University, 2015. [Citace: 29. 7 2017.] https://users.ece.cmu.edu/~koopman/crc/.

Kumar, Sunit Sen. 2014. *Fieldbus and Networking in Process Automation.* Boca Raton : Taylor & Francis Group, 2014. 978-1-4665-8677-2.

Kurose, James a Ross, Keith. 2013. *Computer Networking: A Top-Down Approach.* Boston : Pearson Education, 2013. 9780132856201.

Kvaser Inc. 2014. CAN Protocol Tour by Kvaser. *Kvaser | Advanced CAN Solutions.* [Online] 2014. [Citace: 3. 5 2016.] https://www.kvaser.com/can-protocol-tutorial/.

Lammert, Bies. 2015. Lammert Bies - Computer Interfacing. [Online] Lammert Bies © 1997-2015, 2015. [Citace: 29. 7 2017.] https://www.lammertbies.nl/comm/info/RS-485.html.

Microchip Technology Inc. . 2010. *High-Speed CAN Transceiver.* [Dokument] místo neznámé : Microchip Technology Incorporated, 2010.

Peffers, Michael. 2013. *Introduction to M-LVDS (TIA/EIA-899).* [Dokument] Dallas : Texas Instruments Incorporated, 2013.

Pfeiffer, Olaf, Ayre, Andrew a Keydel, Christian. 2008. *Embedded Networking with CAN and CANopen.* Greenfield : Copperhill Technologies Corporation, 2008. 978-0-9765116-2-5.

Pinkle, Carsten. 2016. The Why and How of Differential Signaling. *All About Circuits - Electrical Engineering & Electronics Community.* [Online] EETech Media, LLC., 16. Listopad 2016. [Citace: 29. 7 2017.] https://www.allaboutcircuits.com/technical-articles/the-why-and-how-of-differential-signaling/.

Professional, Elation. 2008. *DMX 101: A DMX 512 HANDBOOK.* [Dokument] Los Angeles : Elation Professional ®, 2008.

RobotDigg. *RobotDigg, Delta Robot, Cartesian XYZ, SCARA, 3D Printer, OpenPnP, CNC Laser, Injection and Extrusion.* [Dokument] místo neznámé : RobotDigg.

STMicroelectronics. 2013. *Integrated stepper motor driver for bipolar stepper motors.* [Dokument] místo neznámé : STMicroelectronics, 2013.

Studios, Hazelwood. 2012. *Modbus Application Protocol V1 1b3.* [Dokument] Hopkinton : Modbus Organization, 2012.

Williams, Ross. 1993. *A Painless Guide to CRC Error Detection Algorithms.* [Dokument] Adelaide : Rocksoft^tm Pty Ltd., 1993.

Příloha A – Obsah přiloženého CD

Na přiloženého CD je zkomprimovaný soubor KoberO_RobotickySystem_DM_2017.zip, který obsahuje:

- tuto práci v .pdf formátu:
    - KoberO_RobotickySystem_DM_2017.pdf,
- projekty z vývojového prostředí Atmel Studio 6 v samostatných složkách:
    - Atmel_Ovladac_motorku,
    - Atmel_Nozni_prepinac,
    - Atmel_Bluetooth_modul,
- navržená schémata a desky plošných spojů v souborech .sch a .brd kompatibilních se softwarem Eagle 7.7.0 v samostatné složce „Eagle“ která obsahuje:
    - Ovladac_motorku_v03.sch,
    - Ovladac_motorku_v03.brd,
    - Bluetooth_modul_v03.sch,
    - Bluetooth_modul_v03.brd,
    - Nozni_prepinac_v03.sch,
    - Nozni_prepinac_v03.brd,
- projekt z vývojového prostředí Android Studio ve složce:
    - Android_BluetoothApp_v02.