Západočeská univerzita v Plzni

Fakulta elektrotechnická
Katedra výkonové elektroniky a strojů

Diplomová práce

Monitor průmyslové komunikace mezi řídícím systémem a senzory s využitím SSI sběrnice

Autor práce: **Bc. Jan Kohout**
Vedoucí práce: **Ing. Petr Kropík, Ph.D.**
Konzultant práce: **Ing. Petr Štětka**

2022

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta elektrotechnická
Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Jan KOHOUT**
Osobní číslo:	**E20N0043P**
Studijní program:	**N0713A060013 Výkonové systémy a elektroenergetika**
Specializace:	**Výkonové elektronické technologie a pohony**
Téma práce:	**Monitor průmyslové komunikace mezi řídícím systémem a senzory s využitím SSI sběrnice**
Zadávající katedra:	**Katedra výkonové elektroniky a strojů**

Zásady pro vypracování

1. Analyzujte možnosti využití SSI sběrnice v průmyslovém prostředí pro řízení pohonů s využitím BPS (Barcode Positioning Systems).
2. Analyzujte stávající dodané zařízení a navrhněte rozšíření zařízení v závislosti na požadavcích testovacího procesu.
3. Zohledněte synchronizaci přijímaných dat z více SSI kanálů a navrhněte možnosti uložení a odeslání naměřených dat pro další zpracování.
4. Ověřte návrh praktickou realizací na dodaném zařízení.

Rozsah diplomové práce: **40-60**
Rozsah grafických prací: **dle doporučení vedoucího**
Forma zpracování diplomové práce: **elektronická**

Seznam doporučené literatury:

- [online] http://www.alphaint.cz/absolutni-rotacni-snimace/inkrementalni-rotacni-snimace/.
- [online] https://www.seeurodrive.cz/vyrobky/prevodove_motory/prevodove_servomotory/prislusenstvi_a_doplnky_motor/snimace/snimace.html.
- [online] https://www.profess.cz/cs/pci/produkty/mereni_polohy/prislusenstvi_temposonics/digitalni-indikator-pro-snimace-s-vystupem-ssi.
- [online] https://www.tr-electronic.com/products/linear-encoders/barcode-positioning-systems.html.
- [online] https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z0000019MgLSAU&l=cs-CZ.
- [online] https://worldwide.espacenet.com/patent/search/family/006240539/publication/EP0171579A1q=pn3DEP0171579.
- [online] https://www.kunbus.com/ssi.html.

Vedoucí diplomové práce: **Ing. Petr Kropík, Ph.D.**
Katedra elektrotechniky a počítačového modelování

Datum zadání diplomové práce: **8. října 2021**
Termín odevzdání diplomové práce: **26. května 2022**

Západočeská univerzita v Plzni
Fakulta elektrotechnická

L.S.

Prof. Ing. Zdeněk Peroutka, Ph.D.
děkan

Prof. Ing. Václav Kůs, CSc.
vedoucí katedry

V Plzni dne 8. října 2021

Abstrakt

Předkládaná diplomová práce se zabývá analýzou a návrhem rozšíření monitorovacího zařízení průmyslové sběrnice SSI. V práci je řešena analýza základní funkce dodaného zařízení a detailní analýza příslušného zdrojového kódu. Následuje rozšíření původního zařízení o možnost synchronizace pomocí časových razítek. Dále se práce zabývá hardwarovým a softwarovým návrhem emulátoru zařízení pracujícího na zmiňované sběrnici, které pomocí uživatelského rozhraní umožňuje uživateli testovat funkci SSI sběrnice a SSI monitoru.

Klíčová slova

Průmyslové sběrnice, synchronní sériová sběrnice, SSI monitor, senzory, měření polohy.

Abstract

This thesis deals with the analysis and development of an extension for the device which is intended for monitoring the SSI interface. The work also deals with the analysis of the supplied device and a detailed analysis of the source code with which the device works. This is followed by an extension of the original device with the possibility of synchronization using time stamps. Furthermore, the work deals with the hardware and software design of the device emulator working on the mentioned bus, which allows the user to test the function of the SSI bus and SSI monitor using the user interface.

Keywords

Industrial communication interface, synchronous serial interface, SSI monitor, sensors, position measurement.

Poděkování

Touto cestou bych chtěl moc poděkovat vedoucímu diplomové práce panu Ing. Petru Kropíkovi, Ph.D. za jeho laskavý přístup, cenné rady a metodické vedení této práce.

Velký dík patří i konzultantovi diplomové práce panu Ing. Petru Štětkovi, nejen za jeho trpělivost a ochotu, bez které bych tuto práci pravděpodobně nedokončil, ale hlavně za všechno to, co mě během relativně krátké doby naučil a také za to, že na mě přenesl své nadšení pro programování a elektroniku.

Dále děkuji všem svým kolegům z firmy Leuze Engineering Czech s.r.o. za přijetí a vytvoření perfektního prostředí pro vypracování této práce.

Závěrem bych chtěl vyjádřit velké díky mé rodině, partnerce a nejbližším přátelům, kteří mi byli podporou po celou dobu mého studia.

Obsah

Seznam použitých symbolů a zkratek

Značka	**Popis**
ARC	Absolutní čidlo polohy
ARM	Advanced RISC Machine - označení architektury procesorů
BPS	Bar code positioning system
BNC	Bayonet Neil Concelman connector - označení typu konektorů
CAN	Controller Area Network - typ komunikační sběrnice
CLK	Zkratka pro hodinový signál
COM	Hardwarové rozhraní sériového portu
CPU	Centrální procesorová jednotka
csv	Comma-separated values - souborový formát pro tabulková data
DMA	Direct memory access - přímý přístup do paměti
DPS	Deska plošných spojů
FBPS	Fail-Safe bar code positioning system
GDB	GNU Project debbuger
GPIO	Univerzální vstupní/výstupní pin
HMI	Human Mahcine Interface
IP	Internet Protocol
JTAG	Joint Test Action Group
LCD	Displej z kapalných krystalů
LED	Elektroluminiscenční dioda
OOP	Objektově orientované programování
OPC UA	Open Platform Communications - Unified Architecture
OSAC	Operation system abstraction layer
OSI	Open System Interconnection
PISO	Posuvný registr s paralelními vstupy a sériovým výstupem
PLC	Programovatelný logický automat
PT	Parameter Transmit
PWR	Napájení
RTC	Hodiny reálného času
SCADA	Dispečerské řízení a sběr dat
SIL	Safety Integrity Level
SPI	Sériové periferní rozhraní
SRAM	Static random-access memory
SSI	Synchronní sériová sběrnice
SIPO	Posuvný registr se sériovým vstupem a paralelními výstupy
TCP	Transmission Control Protocol
UART	Unversální asynchroní přijímač/vysílač
UDP	User Datagram Protocol
UI	Uživatelské rozhraní
USB	Univerzální sériová sběrnice

Seznam obrázků

Seznam tabulek

Úvod

V současné době narůstá využití automatizace a robotizace v mnoha různých odvětvích. Veškeré manipulátory, roboty, výrobní stroje, pohony a polohovací systémy jsou řízeny mikroprocesory. Proto, aby bylo možné řídit tyto systémy pomocí zavedení zpětné vazby od senzorů, je použití komunikačních sběrnic prakticky nevyhnutelné. Jednou ze základních sběrnic, které se v průmyslu hojně využívají, je „synchronní sériová sběrnice“ (SSI). Tato sběrnice je v průmyslu často využívána mimo jiné díky její odolnosti vůči elektromagnetické interferenci.

V praxi se ovšem mohou vyskytnout problémy, které vedou k přenášení chybného typu informací. Problematikou monitorování chyb a poruch SSI sběrnice se zabývá práce [1], ve které autor řeší návrh a realizaci monitoru pro SSI sběrnici. Jím navržený koncept umožňuje zachytávání datových rámců na sběrnici a detekci chybových signálů. Jelikož se pro zvýšení provozní bezpečnosti používají redundantní senzory, byl monitor vyvinut pro dva kanály synchronní sériové sběrnice.

Dodaný koncept bude potřeba kompletně analyzovat z hlediska navrženého softwaru i hardwaru a otestovat jeho základní chování. Dále bude navrženo několik rozšíření. Nejprve bude realizována synchronizace časovými razítky jednotlivých SSI kanálů, s jejíž pomocí bude možné přesně monitorovat datové rámce na dvou nezávislých kanálech sběrnice. Implementace časových razítek umožní porovnávat průběhy a fázový posun dvou SSI kanálů, kdy tato informace bude v budoucnu využita pro testování a následně pro vyhodnocování polohy, s dodanou časovou informací, různých typů manipulátorů, které v rámci funkční bezpečnosti pracují se dvěma SSI kanály.

Samotné testování SSI sběrnice a dodaného SSI monitoru je z ekonomického hlediska (ceny jednotlivých zařízení) relativně náročné, proto se jeví jako vhodné rozšíření navrhnout levnější zařízení - emulátor a k němu vytvořit řídící software. Emulátor bude postaven na levnějších leč dostatečně kvalitních hardwarových komponentách a zároveň bude díky navrženému softwaru věrně simulovat funkci zařízení pracujícího na SSI sběrnici. Tím bude uživateli umožněno testovat ostatní zařízení na SSI sběrnici, bez nutnosti výrazné finanční investice.

1 Synchronní sériová sběrnice

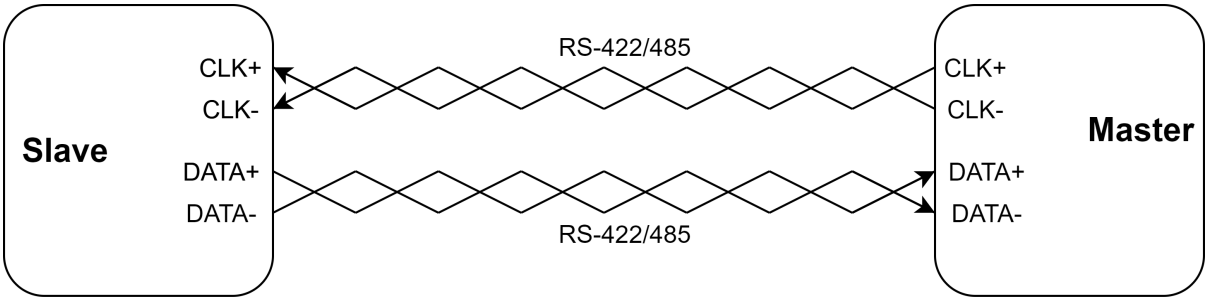
Synchronní sériová sběrnice (dále už jen SSI) je velmi často používanou sběrnicí v průmyslu pro absolutní senzory polohy nebo rychlosti. Tato sběrnice byla vyvinuta Maxem Stegmannem[1] v roce 1984 pro přenos dat absolutních enkodérů a byla patentována německým patentem DE 34 45 617 v roce 1990. [2] Pro základní přehled vlastností a funkcí sběrnice je níže uveden OSI model sběrnice SSI.

1.1 Referenční model sběrnice SSI

Sběrnice SSI zaujímá pouze dvě nejnižší vrstvy referenčního OSI (Open System Interconnection) modelu. Jedná se o vrstvu fyzickou a vrstvu linkovou/datovou. Ostatní vrstvy[2] tato sběrnice neobsahuje. Obě zmíněné vrstvy SSI sběrnice jsou popsány níže.

1.1.1 Fyzická vrstva

Fyzická vrstva této sběrnice je definována standardem RS-422 (popřípadě RS-485). Tento standard umožňuje přenášet data rychlostí 10 Mbps a to do vzdálenosti cca 15 metrů. Nicméně, pokud snížíme rychlost přenášení dat, RS-422 nám umožní přenášet data až do vzdálenosti cca 1200 metrů o rychlosti 10 kbps. Avšak hlavním důvodem pro použití standardu RS-422 je možnost využití diferenciálních vstupů a výstupů, tedy zvýšení odolnosti vůči elektromagnetické interferenci, tzn. zvýšení elektromagnetické odolnosti přenášeného signálu[3]. Co se týče napěťových úrovní, které tato vrstva taktéž definuje, pohybujeme se v rozmezí dvou až šesti voltů [5].
Základní konfigurace systému komunikující po SSI při využití standardu RS-422 je zobrazena na obrázku 1.



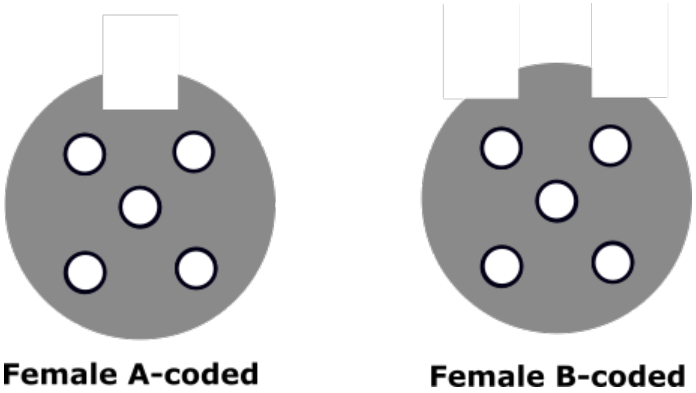
Obrázek 1: Komunikace mezi mastrem a slavem pomocí SSI a standardu RS-422

[1] Je možné se setkat s označením „Stegmann Interface“.
[2] Síťová, Transportní, Relační, Prezentační, Aplikační
[3] Více informací o problematice diferenciálních signálů je možné nalézt například v publikacích [3] anebo [4].

Fyzická vrstva je realizována kabely pro standard RS-422 nebo RS-485. Na propojovacím kabelu využíváme 5 vodičů, dva vodiče pro diferenciální hodinové signály (*CLK+, CLK-*), další dva vodiče pro diferenciální datové signály (*DATA+, DATA-*) a poslední vodič pro zem. V průmyslu se pro tuto komunikaci používají kabely M12. Příklad tohoto kabelu je na obrázku 2.

Obrázek 2: Kabel typu M12 používaný v průmyslu pro sběrnici SSI [6]

Při práci s těmito kabely je nutné dávat pozor na jejich zakončení. Existuje několik typů kódování[4] koncovek kabelů M12. Namátkou například A-coded a B-coded (viz obr 3), kdy koncovka typu A-coded je používána pro připojení napájení a koncovka typu B-coded pak slouží pro realizaci komunikace.



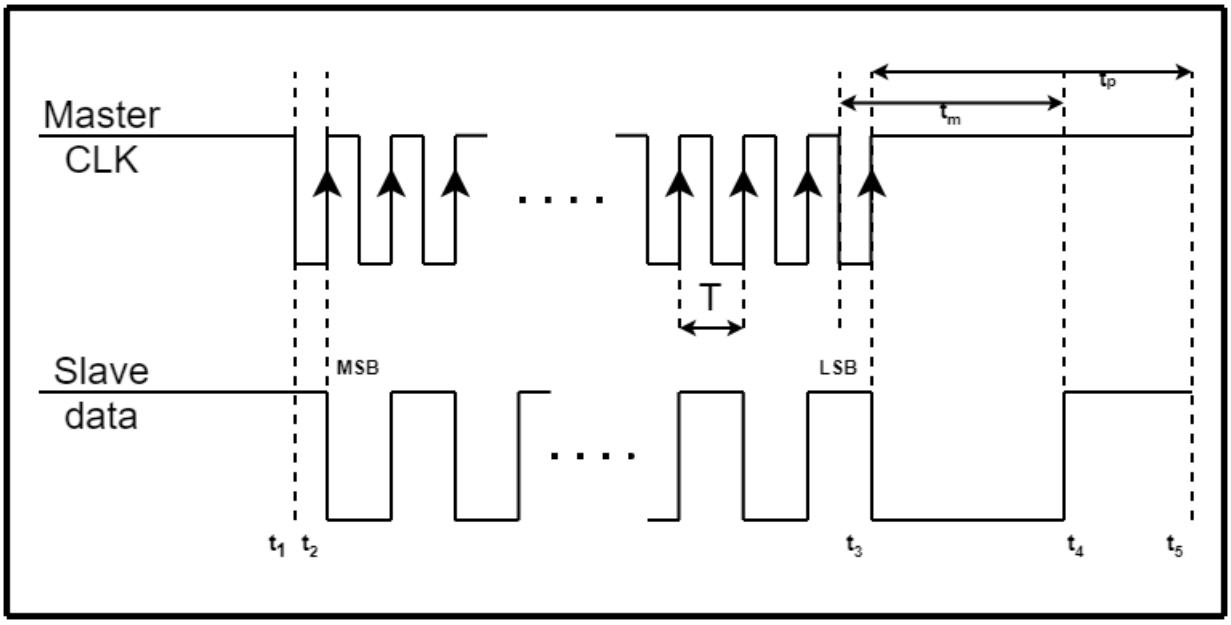
Obrázek 3: Dva základní typy kódování koncovek kabelů M12

1.1.2 Linková vrstva

Linková, nebo taktéž datová vrstva, uspořádává jednotlivá data do rámců, které jsou následně přenášeny po fyzické vrstvě. Z hlediska linkové vrstvy je SSI simplexní point to point sběrnice, u které přenos jednotlivých rámců závisí na hodinovém signálu generovaným masterem. Formát jednoho datového rámce je znázorněn na obrázku 4.
Obrázek zobrazuje všechny tři stavy, ve kterých se může sběrnice nacházet. První stav, tzv. idle (klidový stav), je na sběrnici od počátku do času t_1 a taktéž od času t_4 do času t_5. V tomto stavu čeká slave zařízení na hodinový signál od master zařízení. V čase t_1 začíná master generovat

[4] Každé kódovaní má jiné využití.

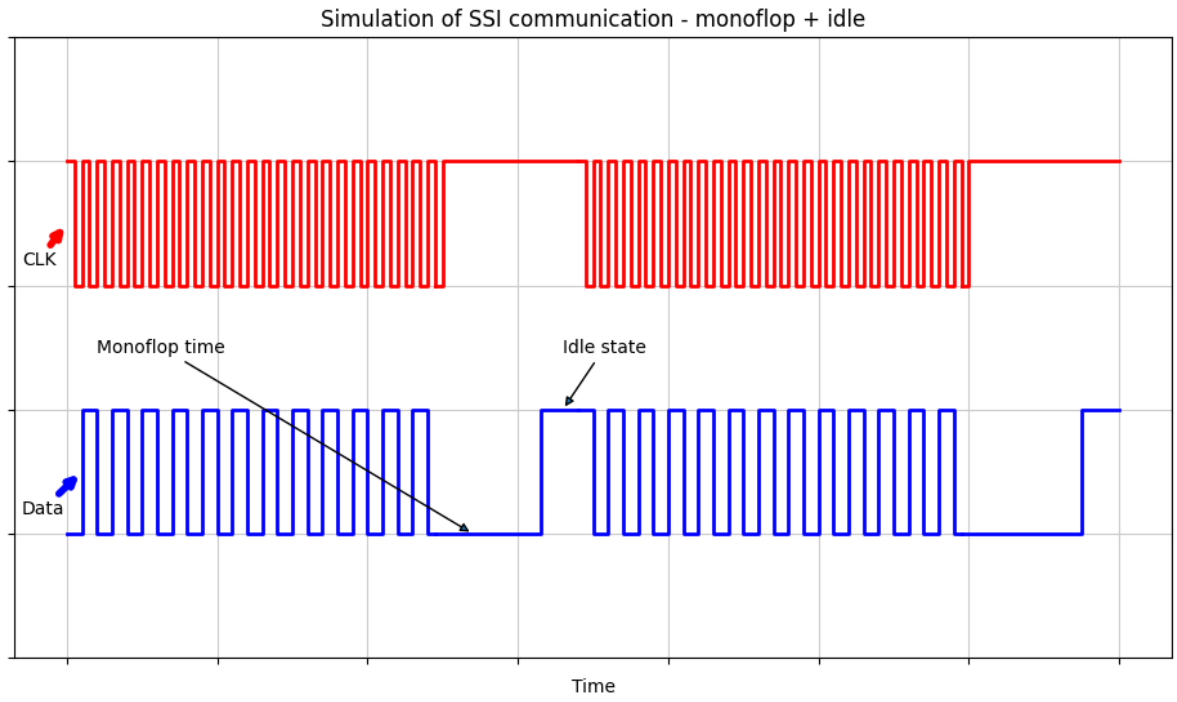
hodinový signál a posílat ho po fyzické vrstvě. Slave zařízení pak v čase t_2 reaguje na náběžnou hranu hodinového signálu a začíná odesílat data. Data jsou ze slave zařízení odesílána s každou náběžnou hranou hodinového signálu a vzorkování dat je prováděno s každou sestupnou hranou hodinového signálu. Po odeslání všech datových bitů[5] přechází sběrnice do stavu, který se nazývá „monoflop time“ (dále jen monoflop). V tomto stavu je hodinový signál v logické jedničce a datový signál v logické nule. Pokud by v tomto čase master zařízení začalo znovu generovat hodinový signál, slave zařízení by nestihlo načíst do svých registrů nová data a odeslalo by masteru stejný datový rámec. Slave zařízení potřebuje ideálně monoflop dlouhý 20-30 μs v závislosti na frekvenci hodinového signálu na to, aby načetl další data.



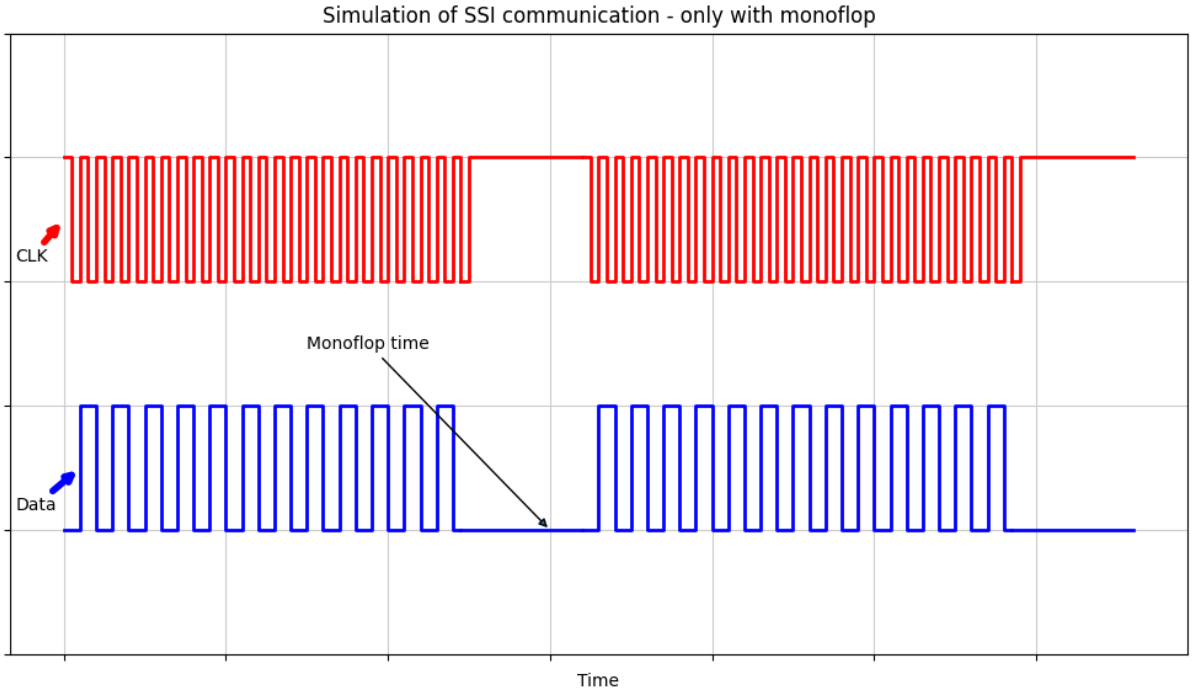
Obrázek 4: Datový rámec na sběrnici SSI

Po odeznění monoflopu může na sběrnici opět nastat klidový stav, nebo může master zařízení ihned generovat hodinový signál, načež slave zařízení začne odpovídat odesíláním dat.
Pro přehlednost a pochopení fungování této sběrnice, jsem vytvořil jednoduchou simulaci v jazyce Python za využití modulů *Matplotlib* a *NumPy*, která zobrazuje jednotlivé stavy sběrnice. Z obrázku 5 jsou evidentní všechny tři stavy. Stav vyčítání dat při každé náběžné hraně hodinového signálu, poté monoflop, při kterém senzor sbírá data a následuje klidový režim, za kterým opět nastává vyčítání dat. Klidový režim na sběrnici může být de facto libovolně dlouhý. Na dalším obrázku 6 sběrnice pracuje bez klidového stavu tj. po odeznění monoflopu začnou být data ihned odesílána.
Z obrázků 5 a 6 vidíme, že simulovaná data jsou pořád stejná, ve skutečnosti tomu tak samozřejmě není, střídání dat logické jednotky a nuly jsem volil z důvodu přehlednosti. Hlavním cílem simulací bylo přibližné zobrazení datového rámce s monoflopem, klidovým stavem a bez klidového stavu. V praxi se taktéž mohou lišit časy, frekvence hodinových signálů a trvání monoflopů. Uvedené simulace zobrazují pouze principiální funkci přenosu datového rámce po sběrnici. Zdrojové kódy jsou dostupné v příloze A.

[5] V průmyslu se velikost datového rámce obvykle pohybuje na 25 bitech, z čehož je 24 bitů datových a 1 bit funguje jako stop bit, ale je možné využívat větší množství bitů.



Obrázek 5: Simulace přenosu datových rámců po sběrnici SSI s vloženým klidovým stavem



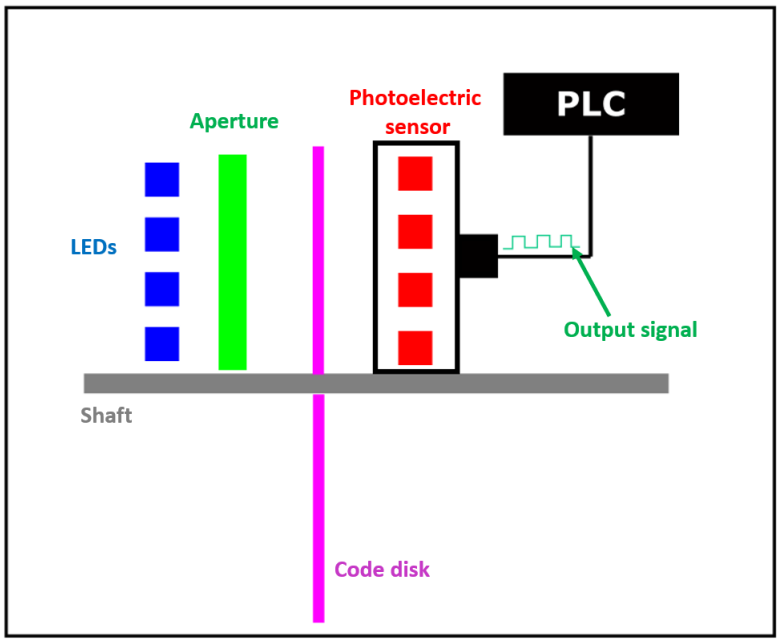
Obrázek 6: Simulace přenosu datových rámců po sběrnici SSI bez klidového stavu mezi rámci

1.2 Využití sběrnice SSI

Sběrnice SSI se díky své odolnosti vůči elektromagnetické interferenci a své jednoduchosti hojně využívá v průmyslu. Nejčastější využití této sběrnice je pro přenos informace o snímané poloze lineárních polohovacích systémů a pro komunikaci s čidly polohy rotoru elektrických strojů.

1.2.1 Absolutní rotační čidla ARC

Významnou výhodou ARC čidel je jejich odolnost vůči výpadkům napájení a tedy jejich schopnost „pamatovat si“ aktuální polohu. Toho je docíleno jejich základním principem, který je založen na osvěcování kotouče. Princip je pro přehlednost jednoduše ilustrován na obrázku 7.

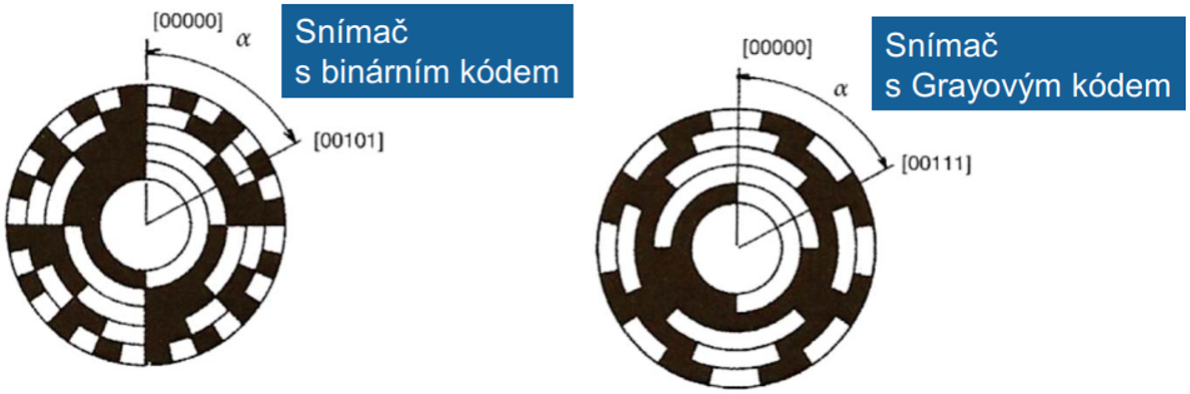


Obrázek 7: Princip ARC čidla

LED vyzařují světelný tok, který prochází clonkou a přes kódovaný kotouč, kdy samotný kotouč má různá rozlišení. Příklad čidla na obrázku 7 má rozlišení čtyř bitů, kdy každá LED reprezentuje právě jeden z bitů. Toto čidlo tedy může zobrazovat 16 pozic. Každá pozice má svůj „vzorek“, například nulová pozice čidla bude reprezentována binárním číslem 0b0000, natočení o 22,5° pak například 0b0001, následovala by pozice 45°, která by byla reprezentována binárním číslem 0b0010, a takto bychom postupovali dále až do 365°. Právě zmíněný postup reprezentace polohy pomocí binárních čísel, se nazývá binární kódování a běžně se v průmyslu používá. Taktéž se zde používá kódování v tzv. Grayově kódu, které má vyšší odolnost vůči rušení[6]. Hlavní rozdíl mezi Grayovým a binárním kódem je popsán následující poučkou:
„Řetězce v Grayově a binárním kódu se liší na pozicích takových, že na pozici vlevo od porovnávané se v binárním kódu nachází znak 1“ [7].

[6] O typech kódování a jejich zpracování bude tato práce pojednávat až v pozdějších částech.

Pro tato kódování se samozřejmě liší i samotná mechanická konstrukce kotouče. Příklady těchto kotoučů jsou zobrazeny na obrázku 8.



Obrázek 8: Typy kotoučů pro ARC čidla [7]

Výstup tohoto čidla je realizován dvěma možnými způsoby. Prvním ze způsobů je tzv. paralelní výstup čidla. Hlavní myšlenkou paralelního výstupu je, připojení paralelních vodičů na univerzální vstupní piny mikroprocesoru, kdy následuje paralelní výčet dat. Tento způsob je efektivní a jednoduchý, ale jeho nevýhodou je velké množství paralelních vodičů.
Druhý způsob je založen právě na principu SSI sběrnice. Čidlo se v tomto případě chová jako slave zařízení a při reakci na hodinový signál generovaný master zařízením, odesílá datový rámec s informací o poloze rotoru. Aplikací sběrnice SSI pro absolutní čidla polohy se zabývá článek [8], který taktéž podává velmi stručný přehled implementace a využití této sběrnice právě v součinnosti s rotačními čidly.

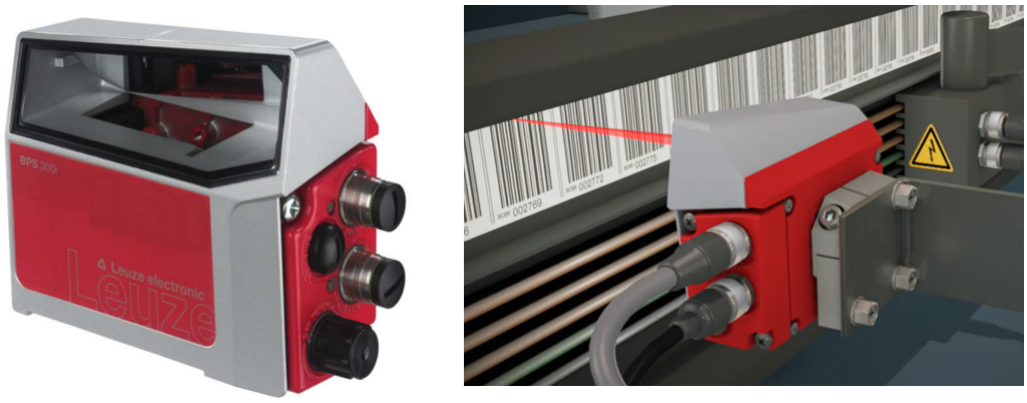
1.2.2 Lineární senzory polohy

Sběrnice SSI se také využívá pro snímaní polohy lineární pohonů, které pracují jako polohovací pohony v regálech velkých skladů, nebo jako lineární pohony poháněj́ící výrobní linky popř. jiná průmyslová zařízení. Díky senzorům polohy se pak tato zařízení mohou orientovat a bezpečně pohybovat po předem definovaných pozicích.
V praxi existuje několik lineárních senzorů polohy. Osobně jsem se průběhu svých studií a praxe setkal se senzory polohy vyvíjené a vyráběné firmou Leuze, ve které toho času působím.

BPS307i

Senzory typu BPS (Barcode Positioning System), známé také pod zkratkou BCL (Barcodeleser) a vyráběné firmou Leuze, využívají pro určování své polohy čárové kódy, které jsou rozmístěné po průmyslových plochách, po kterých se polohovací systémy pohybují. Komunikační sběrnice pro tyto senzory pracují pomocí různých komunikačních protokolů, kterými jsou například EtherCAT, Ethernet, EtherNet IP, PROFIBUS DP a PROFINET. V závislosti na použitém protokolu se liší označení daného senzoru.
BPS pracující na sběrnici SSI nesou označení BPS307i. Přesně tyto senzory byly využívány při

praktické realizaci této diplomové práce. Ukázka tohoto senzoru a možnost jeho implementace v průmyslovém prostředí, například pro snímání polohy manipulátoru, je zobrazena na obrázku 9.

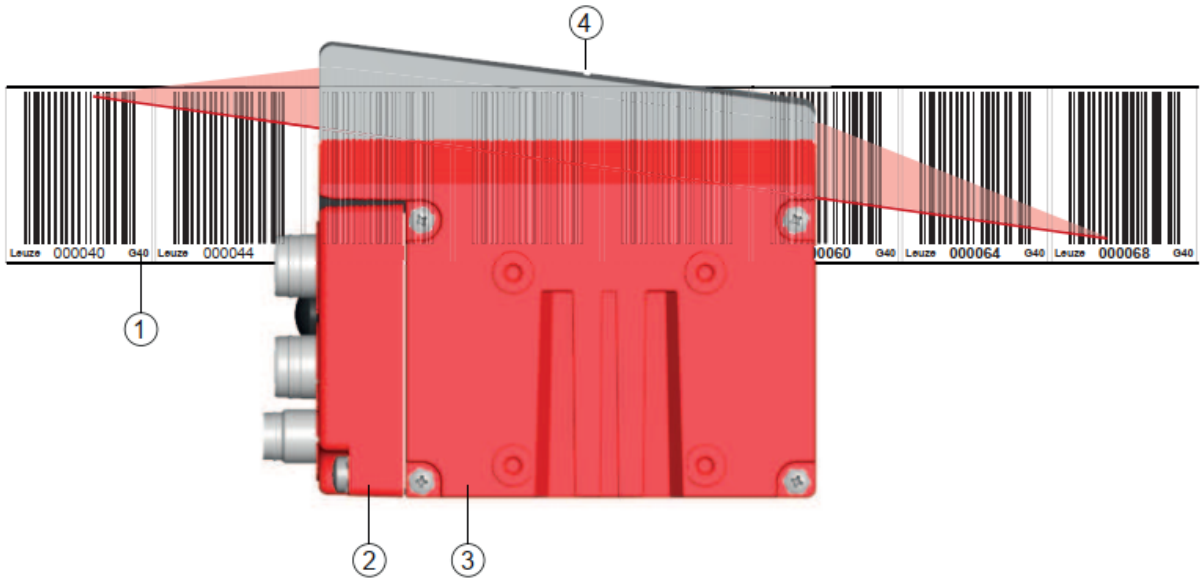


Obrázek 9: Senzor BPS307i (vlevo) a systém s čidlem BPS (vpravo) [9] [10]

Základní funkce senzoru BPS307i spočívá ve využití laserového signálu, díky kterému čidlo vyhodnocuje pozici a rychlost v závislosti na poloze vůči pásce s čárovým kódem. Vše probíhá v následujících krocích:

- čtení čárového kódu umístěného na pásce,
- určení pozice přečteného kódu ve skenovacím paprsku,
- vypočítání polohy z informace získané z čárového kódu v závislosti ke „středu“ senzoru,
- odesílání informace o pozici pomocí průmyslové sběrnice.

Tento proces je pak zobrazen na obrázku 10.

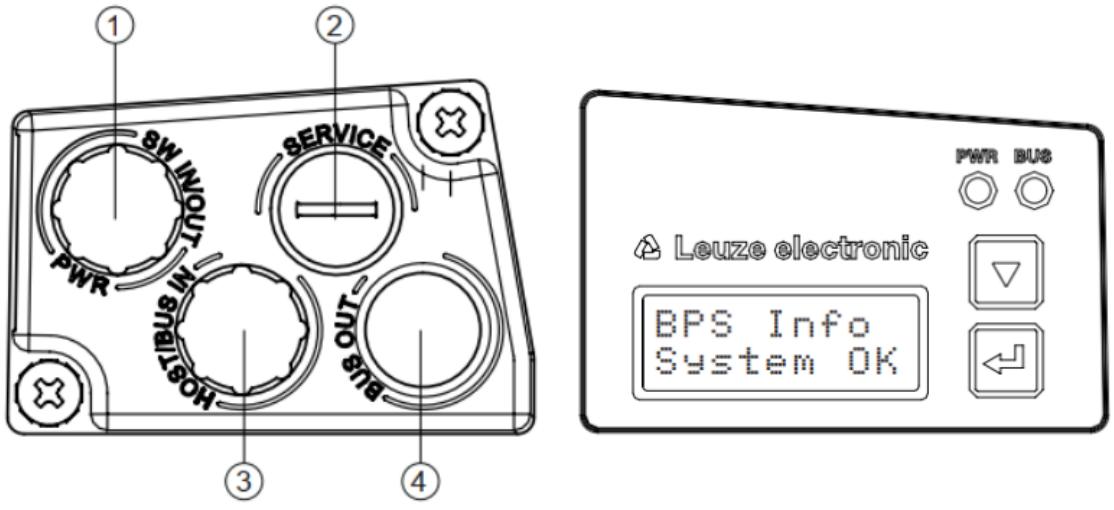


Obrázek 10: Základní uspořádání senzoru při čtení čárového kódu [9]

Na obrázku 10 je vidět několik částí, které jsou popsány číslem. Jednotlivá čísla popisují následující.

1. Čárový kód[7]
2. Připojovací modul, který slouží k připojení napájení a komunikace[8]
3. Ochranný kryt senzoru
4. Střed skenovacího paprsku

Samotná konfigurace senzoru probíhá přes tzv. webConfig[9]. Aby bylo možné senzor konfigurovat, musíme se k němu připojit. Připojení je zajištěno již zmíněným připojovacím modulem. Připojovací modul obsahuje několik konektorů. Jeden z konektorů typu M12 s kódováním typu A slouží k připojení napájení a tento konektor je označen zkratkou *„PWR“*. Druhým z konektorů je již zmiňovaný konektor pro sběrnici. Pokud je použita sběrnice SSI, je tento konektor typu M12 s kódováním typu B a je označený zkratkou *„BUS“*. Další konektor, který se na tomto modulu nachází je označen jako *„SERVICE“* konektor, ke kterému se lze připojit pomocí USB mini. Po připojení je možné komunikovat se serverem senzoru, tedy se serverem webConfig, pomocí počítače a „libovolně“ konfigurovat nastavení senzoru (parametry, rychlost čtení atp.), nebo je možné sledovat aktuálně čtená data
V základní formě lze informace nalézt i na displeji tohoto senzoru, který se nachází na jeho druhé straně. Tento malý displej poskytuje opravdu pouze základní informace o aktuální poloze, rychlosti, a přesnosti měření. U displeje jsou ještě umístěna dvě tlačítka určená k přepínání mezi zobrazením jednotlivých informací, a dvě LED pro signalizaci napájení a dění na sběrnici.



Obrázek 11: Připojovací modul (vlevo), informační displej (vpravo) [9]

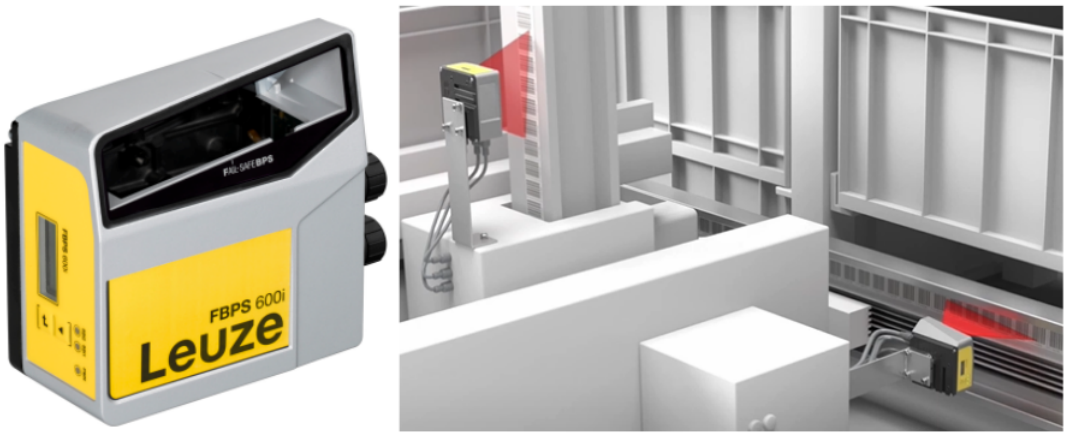
[7]Na obrázku 10 je vidět více čárových kódů vedle sebe, které jsou od sebe odděleny mezerou. Každý z těchto čárových kódů obsahuje informaci. Při vypracovávání této práce jsem se setkal se dvěma druhy těchto kódů, které se lišily pouze svoji „šířkou“ a to třícentimetrové a čtyřcentimetrové. To, jaký typ čárových kódů senzor čte je možné nastavit při samotné konfiguraci senzoru.

[8]Tato část je vyměnitelná, například pokud dojde k nějaké poruše v připojovacím modulu, nebo pokud je potřeba použít jiný typ komunikační sběrnice.

[9]Server, na kterém lze konfigurovat téměř všechna zařízení z portfolia firmy Leuze. Detailnější informace o webConfigu budou rozebrány v pozdější části této práce.

FBPS607i

Další senzor, pracující se sběrnicí SSI je FBPS607i (Fail-safe Bar Code Positioning System). Tento senzor byl navržen, vyvinut a testován tak, aby splňoval standardy funkční bezpečnosti[10]. FPBS607i slouží pro absolutní měření aktuální pozice pohybujících se systémů a konstrukcí, stejně jako jeho předchůdce BPS307i.



Obrázek 12: Senzor FBPS607i (vlevo), systém dvou FBPS v High-bay úložném prostoru s bezpečnou detekcí polohy v ose x a y (vpravo) [11]

Funkční bezpečností se musíme zabývat u všech elektronických zařízení, jejichž selhání by mohlo způsobovat zdravotní újmy, ohrožení životů či poškození životního prostředí. Pro safety zařízení používáme označení SIL (Safety Integrity Level), které udává dovolenou pravděpodobnost hazardního stavu za hodinu. V praxi se setkáváme se čtyřmi stupni SIL [12].

SIL 1 Pravděpodobnost hazardního stavu za hodinu 10^{-5} až 10^{-6}

SIL 2 Pravděpodobnost hazardního stavu za hodinu 10^{-6} až 10^{-7}

SIL 3 Pravděpodobnost hazardního stavu za hodinu 10^{-7} až 10^{-8}

SIL 4 Pravděpodobnost hazardního stavu za hodinu 10^{-8} až 10^{-9}

Senzor FBPS607i splňuje požadavky na funkční bezpečnost, dle norem IEC/EN 62061 a EN 61508 SIL 3 [11].
Toto zařízení disponuje dvěma redundantními SSI kanály, kdy oba kanály pracují s jiným kódováním. Běžně senzor pracuje s kódováním dle Grayova kódu v prvním SSI kanálu a s binárním kódováním ve druhém SSI kanálu. Díky redundantní informaci jsme schopni vyhodnocovat informace ze dvou kanálů. Je nutné zmínit, že každý z kanálů pracuje nezávisle na sobě jako samostatné slave zařízení. Pro přenesení dat do master zařízení je tedy nutné použít, pro generování hodinových signálů, dvě nezávislé master zařízení.
Co se týče samotného čtení čárového kódu, funguje senzor totožně jako jeho předchůdce BPS307i. Senzor se taktéž konfiguruje pomocí webConfigu a taktéž obsahuje připojovací modul s několika

[10] V praxi se převážně setkáváme s pojmem „Functional Safety“, který se uchytil i v českém jazyce.

konektory. Tentokrát se jedná opět o připojení napájení *„PWR“* pomocí kabelu M12 s kódováním typu A. Dále dva konektory pro SSI sběrnici označené *„X1 SS1“* a *„X2 SSI2“* opět pro kabel M12 s kódováním typu B. Poslední konektor je opět *„SERVICE“* konektor, který slouží k propojení s PC a konfiguraci senzoru pomocí serveru.
Na zařízení nalezneme informační displej pro zobrazení základních informací a tři LED pro signalizaci stavů jednotlivých SSI kanálů a napájení.

2 Monitor synchronní sériové sběrnice

Pro monitorování SSI sběrnice a následné vyhodnocování jednotlivých dat, bylo vytvořeno a navrženo zařízení, které nese název SSI monitor. Následující kapitola se primárně zabývá analýzou, zapojením, zprovozněním a testováním dodaného zařízením. Nejprve budou vypsány informace získané ze závěrečné práce Felixe Heila, který se touto problematikou zabýval [1]. Ve druhé části této kapitoly bude popsána analýza dodaného zařízení a popis jeho zprovoznění.

2.1 Návaznost na předchozí práci

Jak již bylo zmíněno, tato diplomová práce volně navazuje na práci Felixe Heila [1]. V této části jsou uvedeny a zpracovány základní informace získané z detailní analýzy dodaného dokumentu, které vnímám jako klíčové pro implementaci následujících rozšíření.

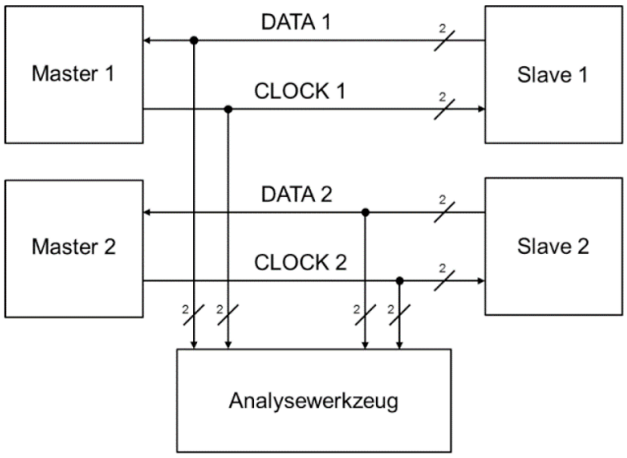
2.1.1 Základní informace o zařízení

SSI monitor byl vytvořen za účelem analýzy dvou SSI kanálů. Tyto dva kanály měří a vyhodnocují dění na sběrnicích nezávisle na sobě. Tento nápad vznikl z důvodu použití dvou BPS307i senzorů, které jsou použity pro měření stejných parametrů. Díky použití dvou BPS senzorů vzniká redundace a tím pádem větší provozní bezpečnost zařízení[1].
Hlavními úkoly SSI monitoru je ověřit, zda-li při přenosu rámce na jednom z kanálů nenastala chyba. Pokud chyba na sběrnici nastane, monitor by měl uložit hodnoty informace zakódované v jednotlivých rámcích před výskytem chyby a poté, co se chyba vyskytla[2]. Samotné chybové rámce jsou uloženy na externí USB disk pro následující analýzu. Jednotlivé telegramy jsou pak čteny v přerušovacích rutinách procesoru.
Mezi další chyby, které monitor musí rozeznat, patří následující:

- Nesprávná doba monoflopu,
- špatná perioda sekvence pulsů hodinového signálu,
- přerušení hodinového signálu,
- porušení počtu datových bitů,
- přeskočení pozice mezi SSI telegramy (jitter).

[1] V současnosti je na trhu k dispozici zmiňovaný FBPS senzor, který obsahuje dva SSI kanály, SSI monitor tedy je i vhodný pro kombinaci s FBPS.
[2] Například výskyt error bitu, což je poslední bit datového rámce.

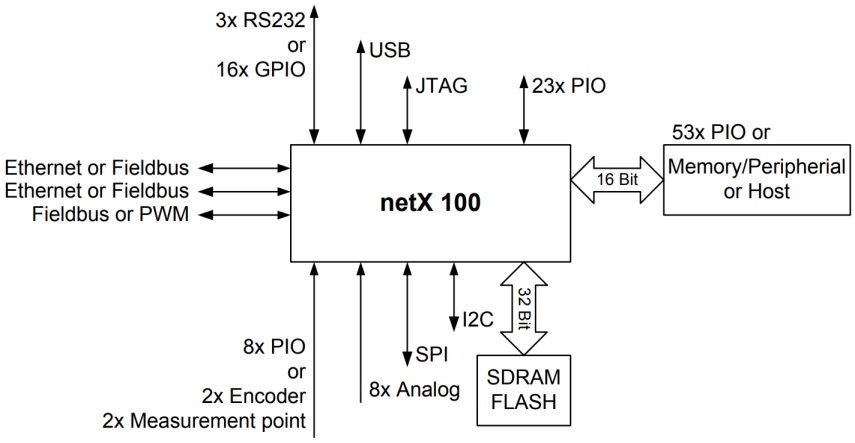
Monitor tedy čte a vyhodnocuje dění na sběrnici v reálném čase. Zařízení musí umožňovat vyhodnocování po delší dobu, jelikož chyba se může objevit velice sporadicky a v průmyslu se chyba může projevit až po delší době jejího výskytu.
Na obrázku 1 je zobrazeno základní principiální zapojení SSI monitoru ke dvěma SSI kanálům.



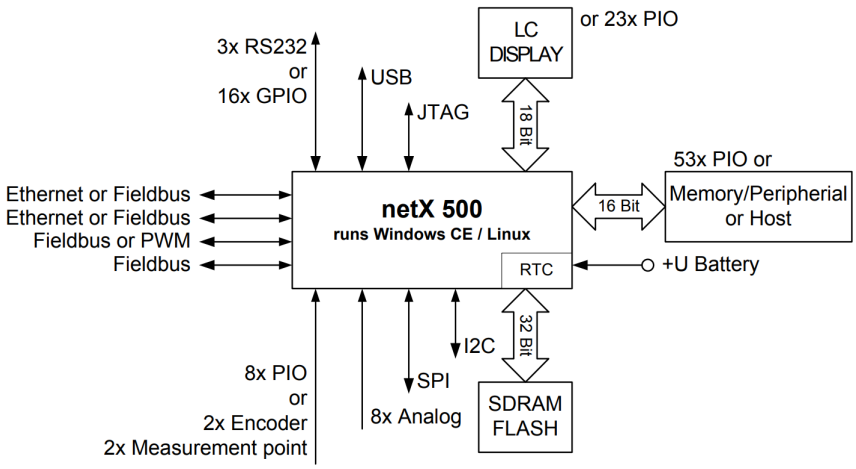
Obrázek 1: Principiální zapojení SSI monitoru se dvěma master zařízeními a dvěma senzory [1]

Hardware [1], [13]

Hardwarová část realizace SSI monitoru byla založena na mikroprocesoru netX od firmy Hilscher, konkrétně na vývojové desce NXDB500-SYS, která obsahuje dva mikroprocesory netX100 a netX500 s 32-bitovým jádrem ARM926 s rychlostí 200 MHz.
Typické aplikace procesorů netX jsou například pro aplikaci komunikačního rozhraní mezi PLC, meniči a HMI, pro systémy založené na operačních systémem Windows nebo Linux s Ethernetovou nebo fieldbusovou komunikací.
Blokový diagram procesoru netX500 je uveden v příloze B na obrázku 2.
Základní rozdíl mezi procesorem netX500 a netX100 je, že netX100 je pouhým derivátem procesoru netX500 a neobsahuje funkcionality, jako například LCD kontrolér nebo RTC. Rozdíly jsou taktéž patrné z přiložených obrázků 2 a 3.

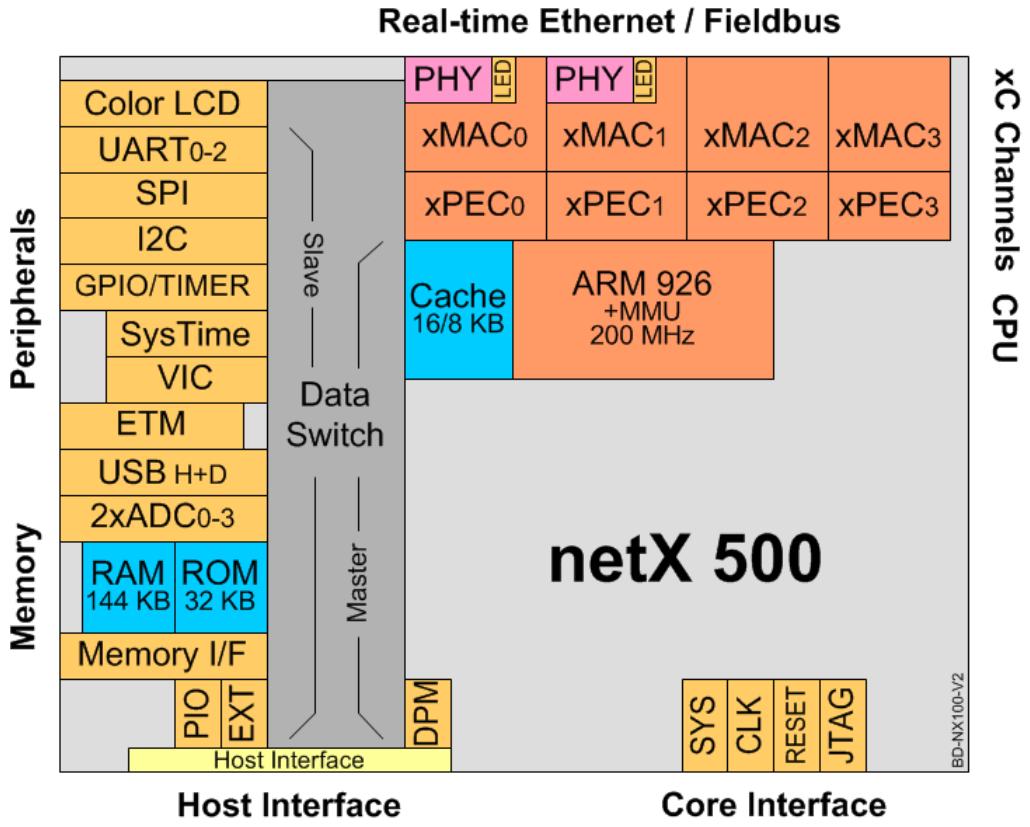


Obrázek 2: Základní funkcionality procesoru netX100 [13]



Obrázek 3: Základní funkcionality procesoru netX500 [13]

Oba ze zmíněných procesorů dále obsahují plně programovatelné sub-procesory, netX500 obsahuje čtyři a netX100 obsahuje tři. Samotné sup-procesory se nazývají xPEC (Flexible Protocol Execution Control) a xMAC (Flexible Media Access Control), popřípadě zkráceně pro obě verze XC (Flexible Communication).
Sub-procesory xMAX odesílají nebo přijímají sériové nebo datové toky podle příslušného procesu přístupu ke sběrnici a šifrují je nebo převádějí z bitových na bajtové bloky.
Sub-procesory xPEC pak kompilují bajty dodané z xMAC do datových paketů a řídí provoz telegramů. Datové pakety jsou pak dále přenášeny do vnitřní paměti po blocích pomocí DMA.
Samozřejmostí pak je například ladící rozhraní pro JTAG s boundary skenem. Více informací je samozřejmě k nalezení v datových listech procesoru [13].
Z hlediska komunikací, použitá vývojová deska podporuje různá rozhraní, jako například, CANopen, PROFIBUS, AS-Interface, DeviceNet, InterBus, CC-Link, Ethernet, USB a SPI. Schéma mikroprocessoru netX500 je zobrazeno obrázku 4.
Jak je vidět z obrázku 4, samotná SSI sběrnice není základní součástí podporovaných rozhraní, byla tedy realizována studentem. Pro realizaci student použil diferenciální receiver DS26LV32AT od firmy Texas Instruments, který podporuje rozhraní RS-422 a RS-485 a slouží k příjímání diferenciálních signálů.
Z receiveru jsou pak signály hodin a dat připojeny na samotné piny, ze kterých jsou tyto signály dále vyčítány. Schéma připojení pinů je uvedeno v příloze B na obrázku 1. Samozřejmostí je i využití dalších pinů například pro signalizaci, trigger a podobně.
Další součástí hardware je interní osciloskop PicoScope 3043D [14], jehož úkolem je zobrazovat signály v aplikaci dodané výrobcem tohoto osciloskopu. SSI monitor taktéž disponuje displayem pro snazší konfiguraci a signalizační LED, které slouží k zobrazení základních stavů monitoru. Na přední části monitoru se taktéž nachází zdířka pro externí USB disk a tři připojovací konektory M12. Dva ze zmíněných konektorů disponují kódováním typu B, které slouží k připojení SSI kanálů. Vstup s A kódováním pak slouží k připojení napájení. Dále je zde několik BNC konektorů. BNC konektory na horní straně slouží pro připojení externího, nebo zabudovaného osciloskopu a BNC konektory, umístěné ze strany SSI monitoru představují vstupy pro interně instalovaný osciloskop.



Obrázek 4: Schéma procesoru netX500 [13]

Software

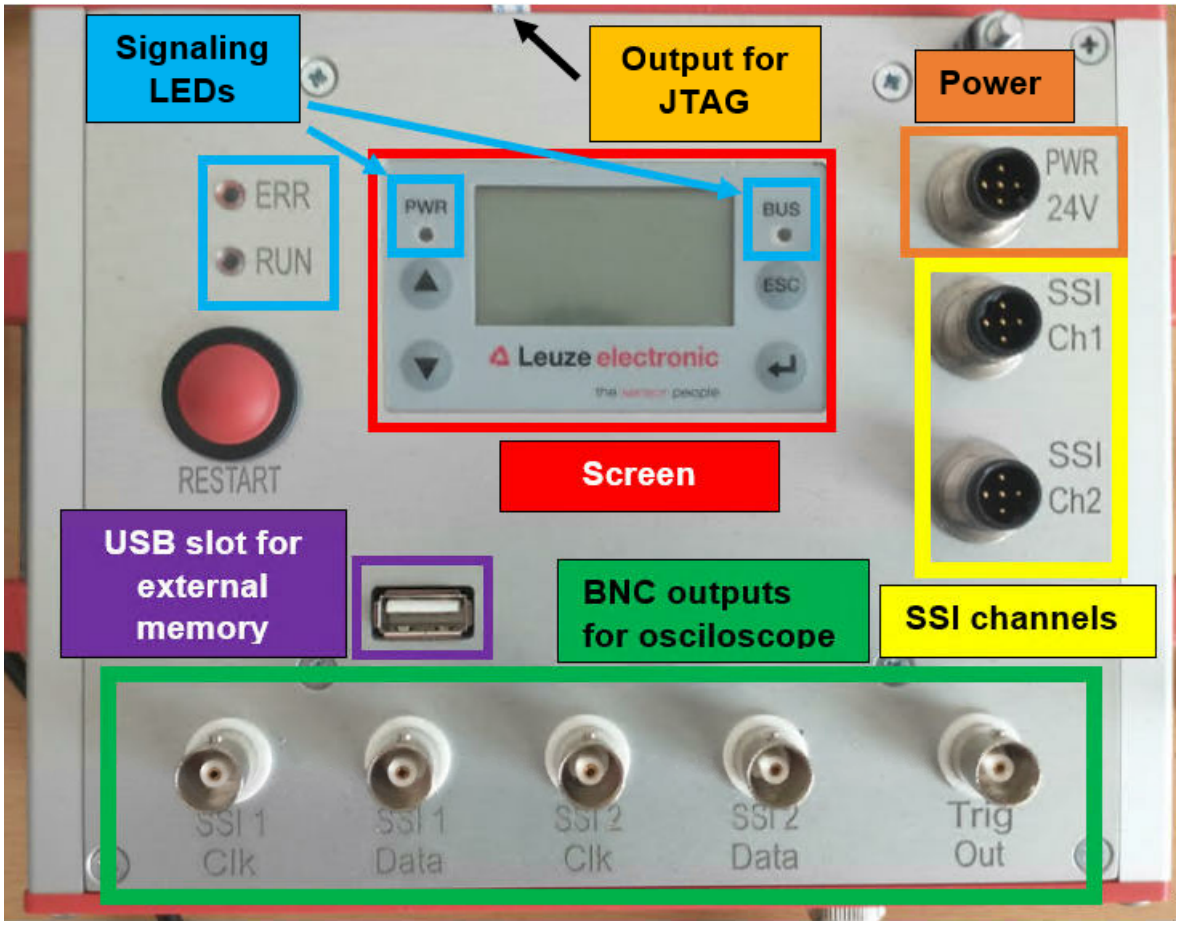
Software využívá komponenty interní implementace SSP platformy. Tato platforma pracuje s jazykem C++ a je založena na OSAC (Operation System Abstraction Layer). Jednotlivé komponenty mezi sebou komunikují pomocí vlastního komunikačního mechanismu, který je založen na principech operačního systému. Z hlediska složitosti a obsáhlosti kódu, zde nebudou uvedeny další informace, které jsou dostupné v práci [1]. Problematikou aplikace a výsledné funkcionality dodaného kódu, principu kompilace atd., se zabývá i tato práce, kdy nejdůležitější aspekty budou rozebrány později.

2.2 Analýza dodaného zařízení

Tato podkapitola se věnuje analýze dodaného zařízení, jeho zapojení, testování funkčnosti monitorování jednoho a dvou kanálů. Dále je zde zmíněno, jaké komponenty používáme pro testování a jak lze se zařízením komunikovat prostřednictvím OPC UA nebo telnetu. Taktéž zde bude zmíněno zprovoznění a konfigurace nejen SSI monitoru, ale také master zařízení a použitých senzorů.

2.2.1 SSI monitor

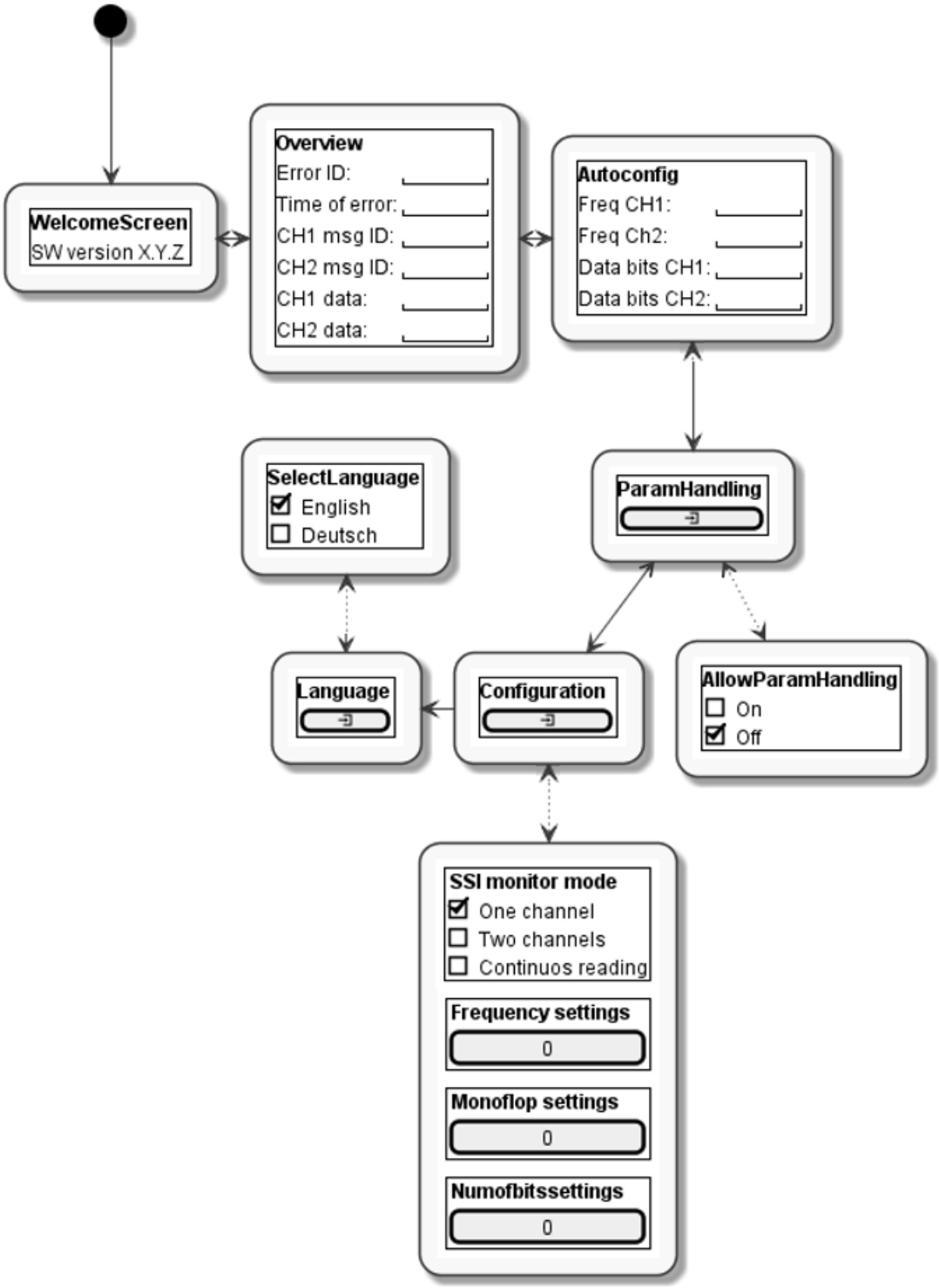
Na následujícím obrázku 5 je zobrazena horní strana SSI monitoru s popisem jednotlivých částí.



Obrázek 5: Fotografie horní strany SSI monitoru s popisem jednotlivých částí

Jak již bylo zmíněno v předchozí části, monitor disponuje jedním napájecím 24V vstupem, který je realizován pomocí konektorů M12 s kódováním A. Monitor pak obsahuje dva SSI kanály, které jsou realizovány konektory M12 s kódováním B. V dolní části jsou umístěny BNC konektory, které slouží k připojení k osciloskopu, ať už k internímu, či externímu. Dále monitor obsahuje USB slot pro externí paměť, do které zapisuje jednotlivá data, za splnění určitých podmínek[3]. Následuje displej, který slouží k zobrazení základních parametrů, a k jednoduché konfiguraci pomocí tlačítek umístěných vedle displeje. Dále monitor obsahuje signalizační LED určené pro hrubé zobrazení nejzákladnějších stavů. V horní části je pak vývod na JTAG, který bude použit pro debug a flashování softwaru do mikroprocesoru.
Při spuštění SSI monitoru se dostaneme na uvítací obrazovku, kde je zobrazena verze nahraného firmwaru. Pomocí tlačítek nahoru [↑] a dolů [↓] je možné přepínat mezi jednotlivými okny. Každé okno má jinou funkcionalitu. Pro přiblížení jsem vytvořil diagram 6, který ukazuje, co monitor zobrazuje a co lze nakonfigurovat.

[3]Původním účelem této paměti je, ukládat datové rámce, které byly přenášeny po sběrnici bezprostředně před vznikem erroru, v průběhu chybového stavu na sběrnici a popř. bezprostředně po jeho odeznění.



Obrázek 6: Zjednodušený diagram zobrazující možnosti konfigurace SSI monitoru pomocí LCD displeje

Poznámka: V diagramu 6 jsou použity dva druhy šipek. Přerušované ↔ ukazují možnost vstoupit do daného okna pomocí ⏎ a vystoupit z něj pomocí ESC. Plné, nepřerušované ↔ naopak zobrazují možnost pohybu mezi obrazovkami pomocí šipek ↑ a ↓.

Obrazovka [Overview] zobrazuje jednotlivá data, které jsou na sběrnici, stejně tak i ID zprávy. Pokud dojde k chybě, zobrazí se typ chyby, který je reprezentován hexadecimálním číslem a monitor zároveň vytvoří časové razítko s informací, kdy chyba vznikla. Toho je docíleno pomocí vnitřního RTC s krystalem o frekvenci 32,768 kHz. Zobrazí se čas a zároveň se rozsvítí LED signalizující ERROR. Přehled chyb s přiřazeným hexadecimálním číslem je uveden v příloze B na obrázku 3. Na obrazovce [Autoconfig] se zobrazuje frekvence přenášených signálů na SSI kanálech, kterou SSI monitor automaticky detekuje, popř. lze frekvenci, se kterou zařízení na sběrnici komunikují, nastavit ručně v sekci [Configuration] [Frequency settings], čemuž se práce věnuje o několik řádek níže. Monitor lze obdobně nakonfigurovat například pro detekci počtu bitů.
Následuje okno s názvem [ParamHandling], do kterého je možné vstoupit pomocí tlačítka [↵] na SSI monitoru. Vystoupit pak lze pomocí tlačítka [ESC]. V okně [ParamHandling], lze zapnout či vypnout režim změny parametrů, jako výchozí možnost je pak vybrána možnost *OFF* ☑ [4].
Následuje sekce [Configuration], do které se opět lze dostat pomocí tlačítka [↵] na monitoru. Zobrazí se možnosti konfigurace jednotlivých částí. Například lze vybrat, v jakém režimu se zařízení nachází, tzn. zdali měří na jednom či více kanálech. Taktéž zde lze nastavit požadovanou frekvenci, o které víme, že na sběrnici je[5]. Stejně tak můžeme změnit i počet bitů zprávy, délku monoflopu a podobně[6]. Z diagramu je zřejmé, že hodnoty frekvence, délky monoflopu a počtu bitů, jsou nastaveny do nuly. Pokud jsou monitoru zadány všude samé nuly, znamená to, že se monitor sám pokusí hodnoty detekovat. To může způsobit menší prodlevu na začátku měření, kdy se monitor snaží „naladit“ na danou frekvenci, počet bitů a monoflop. V této sekci lze také nastavit vzdálenostní offset jednotlivých kanálů. Ten se nastavuje, pokud měříme dvěma senzory, které jsou od sebe vzdálené, to si asi každý dokáže představit, ale pro úplnost, je v příloze B na obrázku 4 uvedena fotografie této skutečnosti. Dalším parametrem, který lze nastavit je tzv. jitter, který značí rozkolísání hodinové frekvence oscilátoru. Z hlediska jitteru se tedy nastavuje tolerance. Dále je nutné poznamenat, že pokud budeme chtít jednotlivé parametry měnit ručně, musíme mít zaškrtnuté ☑ u možnosti *ON* v sekci [ParamHandling].
Posledním krokem je nastavení jazyka, kdy je angličtina nastavena jako výchozí.
Výše uvedená konfigurace, je prováděna přímo na SSI monitoru pomocí tlačítek a LCD displeje. Velkou výhodou monitoru je, že obsahuje vnitřní server, ke kterému se lze připojit za použití ethernetového kabelu RJ65. Komunikace je realizována na základě protokolu TCP/IPV4. Výchozí IP adresa serveru SSI monitoru je 192.168.60.101. Na tuto adresu se lze připojit například přes aplikaci PuTTY, kde je nutné nastavit přidělenou IP adresu a správný port 23, což je port pro telnet. Obrázek 5 s nastavením programu PuTTY pro připojení k SSI monitoru pomocí telnetu se opět nachází v příloze B. K serveru se pak jednoduše připojíme pomocí [Open]. Konfigurace parametrů pak probíhá pomocí PT sekvencí. Detailnější analýza PT sekvencí bohužel nebyla realizována z důvodu absence potřebné dokumentace. O PT sekvencích bude ještě v této práci pojednáváno, jako o jednom z možných rozšíření komunikace monitoru s uživatelem.

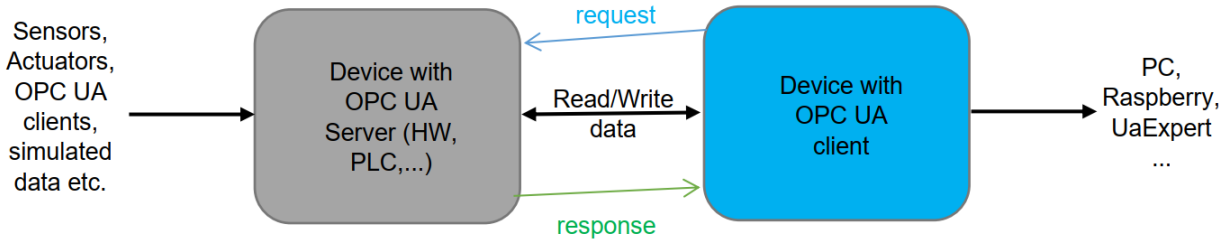
[4] Značka „☑“ v diagramu 6 značí výchozí stavy.
[5] Například, když je frekvence manuálně nastavena v master zařízení.
[6] V diagramu 6 jsou zobrazeny na ukázku pouze tři základní možnosti, reálně lze konfigurovat více parametrů.

OPC UA

Další možnost jak nakonfigurovat SSI monitor je pomocí OPC UA[7]. OPC UA je platforma používaná pro komunikaci mezi zařízeními. Základní struktura OPC UA se skládá z tzv. uzlů[8], kdy každý údaj má svůj uzel se specifickým ID a každé ID má specifický název/syntax. Důležitou součástí OPC UA je tzv. *Secure model*, který umožňuje větší zabezpečení přenášených informací. OPC UA se skládá z několika částí:

- **OPC DA Server** definuje určitou výměnu dat včetně jejich hodnot, informace o čase a kvalitě informací — mluvíme zde tedy pouze o datech,
- **OPC A&E Server** pracuje se zprávami typu alarm či jinými událostmi,
- **OPC HDA Server** obsahuje metody, které umožňují vytvářet časová razítka.

Základní ukázka principu OPC UA je zobrazena na obrázku 7.

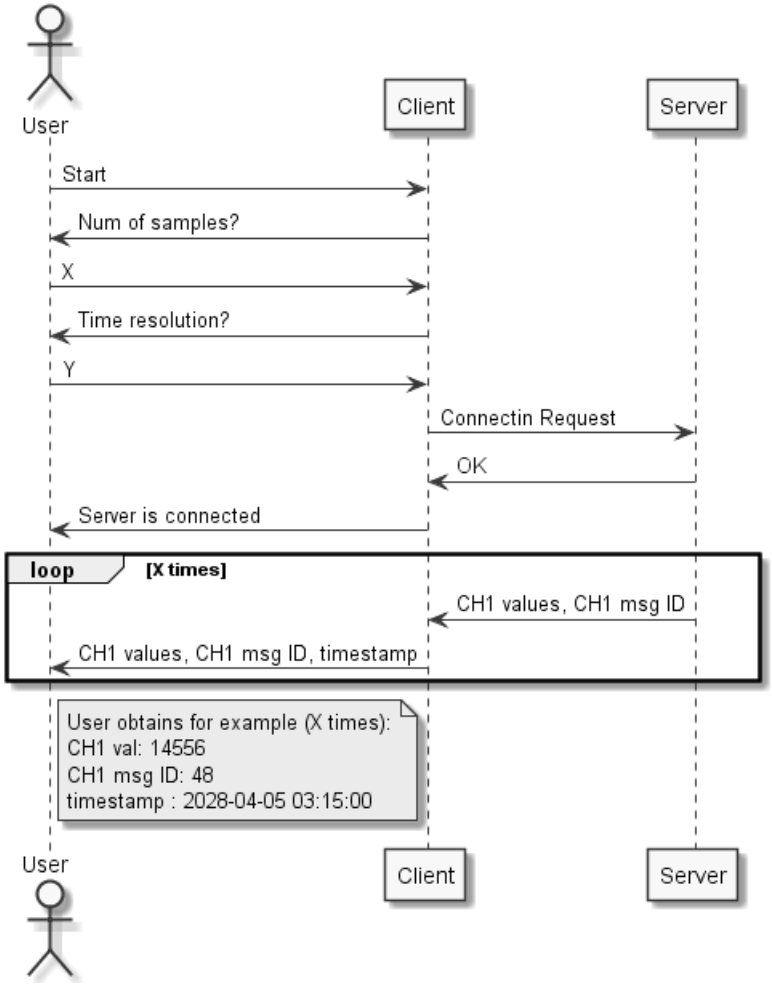


Obrázek 7: Zjednodušený diagram zobrazující princip OPC UA

Zařízení s OPC UA serverem je pro nás v tuto chvíli samotný SSI monitor[9], který má svůj server na již zmíněné IP adrese. SSI monitor je pak připojen k master a slave zařízení a sbírá data o dění na SSI sběrnici.
Modrý blok reprezentuje klienta OPC UA, což je v našem případě počítač, který je na server SSI monitoru připojen a sbírá data ze serveru a dále je zpracovává, či zobrazuje. Klient taktéž může data na server zapisovat, v našem případě to znamená konfiguraci parametrů SSI monitoru zmíněnou na obrázku 6. Samotné propojení mezi serverem a klientem je realizováno pomocí ethernetového kabelu.
Pro ilustraci a pochopení fungování komunikace server↔klient jsem vytvořil jednoduchou aplikaci v podobě klienta, který čte ze simulačního serveru. Jako simulační server jsem použil aplikaci Prosys OPC UA Simulation Server [16]. Simulační server pro ukázku funguje jako SSI monitor, který čte dění na sběrnici a zároveň odesílá přečtená data na OPC UA server. V aplikaci Prosys OPC UA Simulation Server jsem tedy vytvořil jeden uzel, který obsahuje dva indexy. Jednomu indexu odpovídá simulovaná hodnota dat na sběrnici a druhému odpovídá ID odesílané zprávy. Po nastavení a spuštění serveru, je třeba z něj data vyčítat. Pro vyčítání jsem napsal klient aplikaci v programovacím jazyce Python. Při psaní kódu jsem využil třídu *Client* z modulu

[7] Základní informace o OPC UA (Open Platform Communication Unified Architecture) zmíněné v této části, byly převzaty ze zdroje [15].
[8] Velice často se setkáváme s pojmem node = uzel.
[9] SSI monitor je reprezentován šedým blokem na obrázku 7.

opcua. Dále pak moduly *time* a *datetime*. Zdrojový kód je k dispozici v příloze A 5 pod názvem „Simulation of OPC UA client“.
Výsledná klient aplikace pak funguje následujícím způsobem, viz obrázek 8.



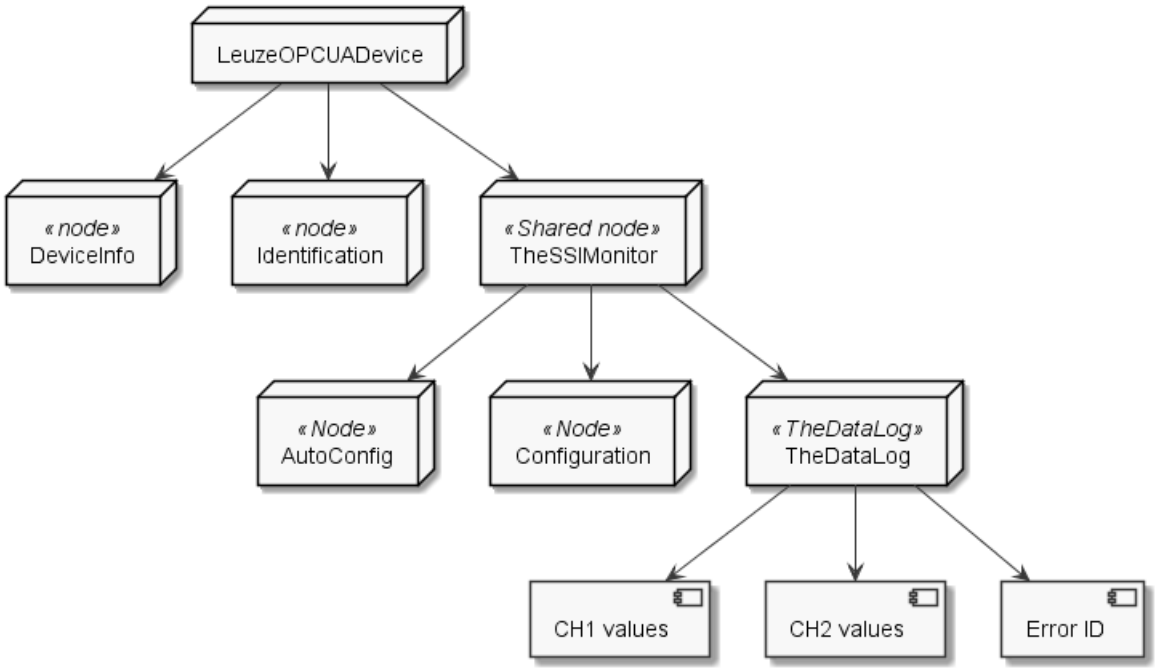
Obrázek 8: Sekvenční diagram zobrazující princip aplikace klienta OPC UA (čistě z pohledu uživatele)

Ukázka skutečného výstupu ze simulovaného serveru je zobrazena níže.

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19044.1586]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Jan Kohout\OneDrive\Škola\Diplomová práce>python OPC_UA_client.py
Client connected
Set how many samples:
5
Set how time stamp resolution:
1
Channel 1 values:  1265 Channel 1 msg ID:  397.0 Timestamp 2022-04-03 19:46:43.715223
Channel 1 values:  1128 Channel 1 msg ID:  398.0 Timestamp 2022-04-03 19:46:44.719597
Channel 1 values:  1173 Channel 1 msg ID:  399.0 Timestamp 2022-04-03 19:46:45.734864
Channel 1 values:  1044 Channel 1 msg ID:  400.0 Timestamp 2022-04-03 19:46:46.736981
Channel 1 values:  1089 Channel 1 msg ID:  401.0 Timestamp 2022-04-03 19:46:47.749258
Client disconnected

C:\Users\Jan Kohout\OneDrive\Škola\Diplomová práce>
```

Obrázek 9: Výstup aplikace v termínálu při čtení ze simulačního serveru

Zdrojový kód pro tuto aplikaci funguje tak, že nejprve načte adresu serveru. Poté požádá uživatele o dva vstupy. Jakmile je uživatel zadá, klient se připojuje k serveru, vykoná čtení hodnot a hodnoty zobrazí uživateli.
Vytvořená aplikace slouží pouze pro jednoduché seznámení se s funkcionalitou OPC UA, není v ní tedy žádné ošetření špatných vstupů nebo nepřipojení k serveru, jde opravdu jen o ukázku, jak lze k OPC UA přistupovat. Tyto znalosti pak mohou být aplikovány například pro připojení se ke skutečnému OPC UA serveru SSI monitoru, nebo jakémukoliv jinému PLC, které tímto serverem disponuje. Pokud by čtenáře zajímalo, jak vypadá prostředí systému pro simulaci serveru v aplikaci Prosys, přikládám pro ilustraci obrázky 6, 7 z tohoto prostředí do přílohy B.
Samotnou konfiguraci SSI monitoru pomocí OPC UA je možné realizovat v programu UaExpert [17], který funguje jako klient pro připojení PC k serveru SSI monitoru a disponuje možností zapisovat/číst hodnoty z jednotlivých uzlů. Nejprve je nutné server vyhledat, s čímž si aplikace sama poradí a server nalezne. Následuje připojení se na server. Po připojení je již možné procházet jednotlivé položky, které server obsahuje. Velmi zjednodušenou verzi složek a uzlů, ukazuje následující diagram.



Obrázek 10: Zjednodušený diagram zobrazující jednotlivé možnosti konfigurace v aplikaci UaExpert

Při tvorbě diagramu[10] 10, jsem chtěl primárně poukázat, jak se v programu UaExpert „proklikat“ k jednotlivým proměnným, které lze číst/přepisovat. Příkladem může být proměnná „CH1 values“, ke které je možné se dostat LeuzeOPCUADevice » TheSSIMonitor » TheDataLog » CH1Values a následně lze číst její hodnotu.
Konfigurace SSI monitoru je pak stejná, jak je uvedeno na obrázku 6. Musíme ovšem do LeuzeOPCUADevice » TheSSIMonitor » Configuration » ....., kde proměnné přepisujeme. Přepisování v aplikaci Ua Expert provádíme přímo na obrazovce *„Data Access View“* do sloupce Value.

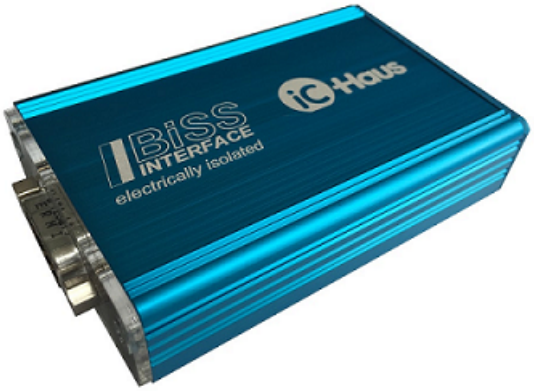
[10]Pro informaci je vhodné zmínit, že pro tvorbu všech diagramů byl používán jazyk PlantUML za doprovodu literatury [18], která pojednává o správném návrhu UML diagramů pro popis software.

2.2.2 Konfigurace master zařízení

Následující podkapitola se zabývá základním popisem a konfigurací master zařízení, které je používáno pro generování hodinových signálů.

MB5U

Jako master používáme zařízení MB5U od firmy icHaus [19], což je zařízení, které dokáže generovat hodinové signály pro SSI až do frekvencí 4 MHz. Zařízení je propojeno s počítačem pomocí USB mini. Ukázka tohoto zařízení je zobrazena na obrázku 11.

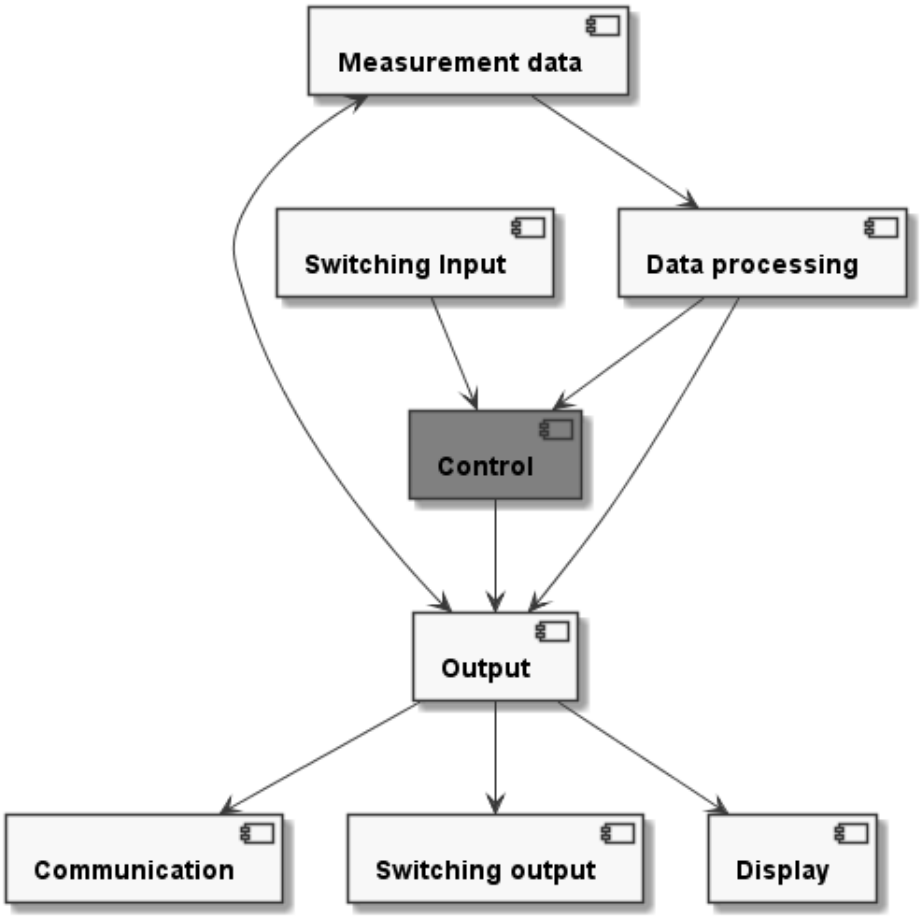


Obrázek 11: Master zařízení pro sběrnici SSI — MB5U [19]

Dále zařízení obsahuje aplikaci pro Windows *BiSS-Interface DLL*, ve kterém lze master nakonfigurovat.
Základní konfigurace je velmi jednoduchá. Po připojení master zařízení přes USB do PC v GUI vybereme k jakému zařízení se chceme připojit, pro nás tedy Connection » MB5U - Biss. Tlačítko Disconnected značí, že nejsme připojeni k master zařízení. Po kliknutí na něj, se k master zařízení připojíme. Pokud připojení proběhne úspěšně, tlačítko se změní na Connected. Dále je třeba nastavit požadovaný počet datových bitů a error bit. Následuje zapsání konfiguračního nastavení do master zařízení pomocí tlačítka Write Master.
Po konfiguraci už je možné hodinový signál generovat a odesílat do slave zařízení. Pro spuštění generace hodinového signálu je třeba přejít do Windows » Fast Reader. Dále lze v sekci Connection setup nastavit frekvence hodinových signálů a frame repetition rate. Ten je nastaven defaultně a není třeba jej měnit. Následuje nastavení počtu vzorků v sekci Sample Count. Generování hodinových signálů do slave zařízení spustíme tlačítkem · Read SCD Frames.
Ukázka prostředí SW je zobrazena v příloze B na obrázku 8.

2.2.3 Konfigurace senzoru

Pro ověření funkčnosti SSI monitoru je potřeba slave a master zařízení. V této podkapitole se zaměřím konfiguraci slave zařízení, což je již zmiňovaný senzor BPS307i[11]. Pro konfiguraci je potřeba se k zapnutému senzoru připojit. Pro připojení využíváme USB mini konektor který je umístěn na připojovacím modulu. Po připojení se připojíme k webConfig serveru přes IP adresu. Server automaticky detekuje zařízení a zobrazí možnosti konfigurace. Senzor se ve výchozím stavu nachází v procesním režimu, ve kterém dochází ke čtení dat a konfigurace není možná. Pro možnost konfigurace zařízení je třeba přejít do servisního režimu.
Po přechodu do servisního režimu nám aplikace dovolí provádět konfigurační změny v zařízení. Co vše lze konfigurovat a s čím vším je možné pracovat je naznačeno na následujícím obrázku 12. Koncepce diagramu na obrázku odpovídá uspořádání, které se nachází přímo ve webConfigu.



Obrázek 12: Základní možnosti konfigurace BPS307i pomocí webConfig

V sekci Measurement data je uživateli zpřístupněno základní nastavení formátu používané pásky s čárovým kódem, jako například rozměry použité pásky, šířka čárového kódu a podobně. Následuje sekce Data processing, ve které uživatel může nastavit parametry jako jednotky, ve kterých senzor měří, meze pro zavolání erroru při špatné kvalitě čtení a další různá nastavení týkající se chování senzoru při práci s daty.

[11] Pro FBPS je konfigurace v základu podobná, jelikož se taktéž jedná o produkt firmy Leuze.

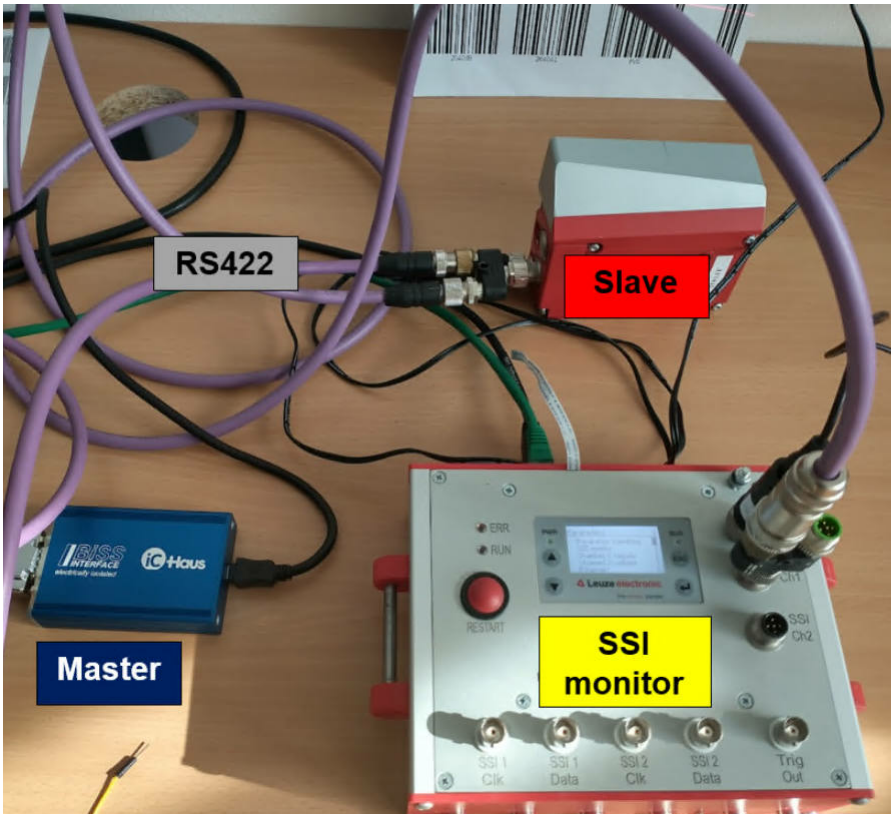
V sekci Switching input/output může uživatel pracovat s I/O módy senzoru. Následuje sekce Control, do které nemá běžný uživatel přístup, z toho důvodu je v diagramu na obrázku 12 vyznačena šedou barvou.
Asi nejdůležitější část práce s webCongigem se nachází v sekci Output. V této sekci uživatel nastavuje důležité parametry SSI komunikace, jakými jsou například rozlišení čtené pozice v milimetrech, typ kódování odesílané zprávy, počet datových bitů a podobně. Toto nastavení musí, pro správnou funčnost samozřejmě korespondovat s nastavením SSI monitoru.
V sekci Communication pak uživatel nastavuje očekávanou frekvenci z master zařízení, proto, aby mohl generovat požadovaný monoflop. Pro frekvence hodinového signálu, které jsou menší než 80 kHz, je používán monoflop o délce 30 μs. Pro frekvence vyšší se používá monoflop o délce 20 μs.

2.2.4 Testování funkce dodaného zařízení

Tato část práce pojednává o zprovoznění, prvotním testování a analýze chování dodaného zařízení. Nejprve bude popsán režim práce SSI monitoru v režimu jednoho kanálu. Následovat bude testování a analýza chování SSI monitoru při práci se dvěma aktivními SSI kanály.
Co se týče zapojení, vychází se z obrázku 1, který je zobrazen na začátku této kapitoly.

Jeden aktivní SSI kanál

Následuje obrázek 13, který zobrazuje fotografii se zapojením a popisem jednotlivých komponent.



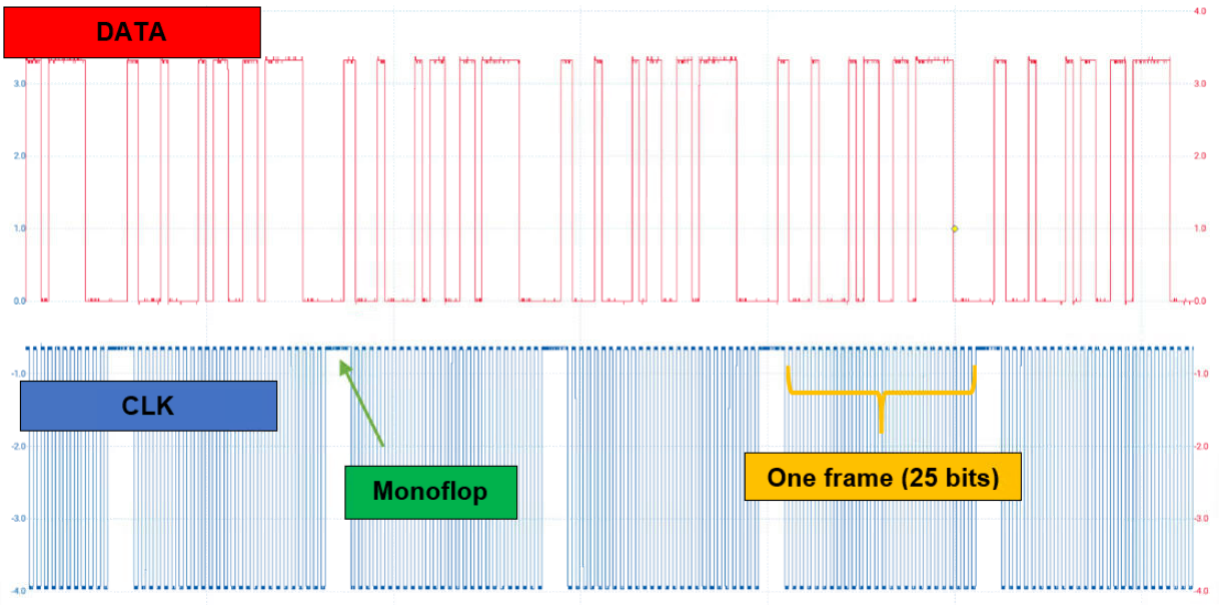
Obrázek 13: Zapojení pro testování SSI monitoru v režimu jednoho aktivního kanálu

Po zapojení je potřeba nakonfigurovat všechna zařízení. Postup, jak se jednotlivá zařízení konfigurují, je popsán v předchozích částech této kapitoly. Master i slave zařízení byla pro první testy nakonfigurována dle tabulky 1.

-	Master zařízení	BPS307i	Monitor
Monitoring mode	-	-	One Channel
Délka SSI zprávy (bit)	25 bitů	25bitů	0
Hodnota pozice (bit)	24 bitů	24 bitů	24 bitů
Rozlišení pozice (mm)	0.1 mm	0.1 mm	0.1 mm
Kódování	-	Grey	Grey
Frekvence (kHz)	500 kHz	$>$ 80 kHz	0
Monoflop (μs)	20	20	0
Šířka čárového kódu (mm)	-	40 mm	-

Tabulka 1: Tabulka s hodnotami pro konfiguraci

Nutno poznamenat, že pomlčky v tabulce 1 značí, že tato část není v daném zařízení konfigurována. Nula naopak značí, že se zařízení nakonfiguruje automaticky[12].
Po zapojení, konfiguraci následuje měření a kontrola naměřených dat. Pro kontrolu funkčnosti a správnosti přenášených zpráv, byly použity čárové kódy, které jsou označeny hodnotou, která je v nich zakódována. Příklady čárových kódů, převzatých z [9], jsou dostupné v příloze C na obrázcích 9 a 10.
Pro měření jednotlivých signálů byl použit interní osciloskop, který je integrován přímo v SSI monitoru. Výstup z osciloskopu, zachycený během provozu, je zobrazen níže 14.

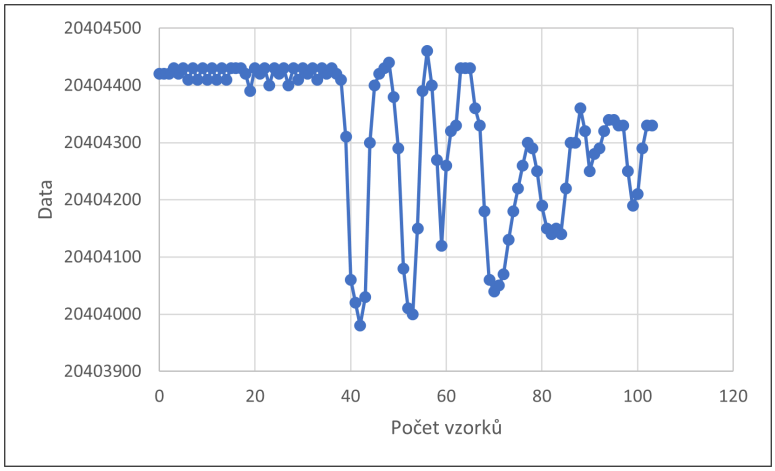


Obrázek 14: Výstup z osciloskopu zabudovaného v SSI monitoru

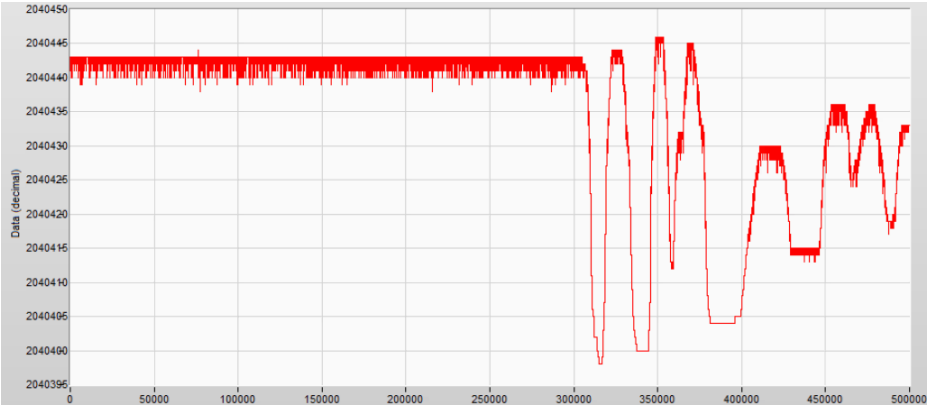
Na obrázku 14 je modře zobrazen přenos hodinového signálu. Červeně jsou pak zobrazena přenášená data. Dále je zde vyznačen čas, ve kterém na sběrnici probíhá již zmiňovaný monoflop.

[12]Pro nás toto platí pouze u SSI monitoru, který disponuje funkcí autoConfig, která je zmíněna na obrázku 6. Autokonfigurace proběhne na základně počtu bitů a frekvence generované master zařízením.

Samotná naměřená data se pak zobrazují na displeji SSI monitoru. Data je možné ověřit porovnáním s hodnotami uvedenými pod čárovými kódy, popř. s hodnotami zobrazující se na informačním displeji senzoru, nebo pomocí webConfigu.
Průběh dat, která SSI monitor po dobu měření nasbíral, je možné si uložit pomocí aplikace Ua Expert[13] ve formě .csv, což následně umožňuje jednoduché zobrazení získaných dat do grafu. Pro ověření správnosti průběhu získaného SSI monitorem, je zde možnost zobrazení průběhu uloženého v master zařízení. Následuje porovnání obrázků průběhů naměřených hodnot senzorem BPS.



Obrázek 15: Průběh hodnot vyčtených z .csv souboru (SSI monitor)



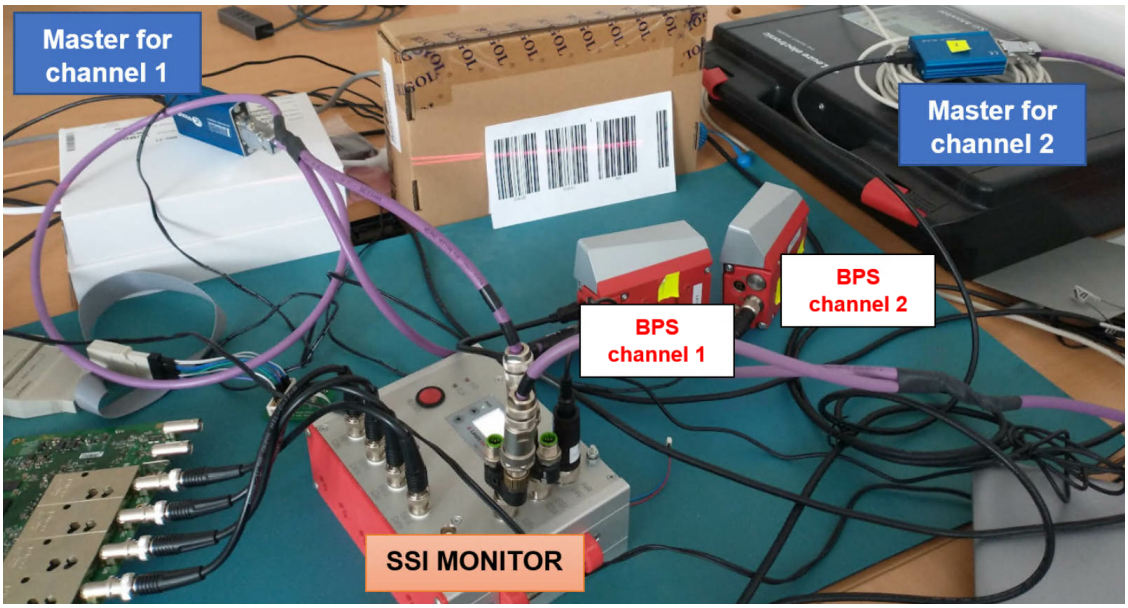
Obrázek 16: Průběh hodnot zobrazený master zařízením

Výstup z obrázků může na první pohled vypadat jako rozdílný, ale po detailnější analýze lze zjistit, že si získané průběhy odpovídají. Na obrázku 15 vidíme průběh bez značného šumu, taktéž si povšimněme výrazně menšího počtu vzorků, než na obrázku 16. Master zařízení zaznamenávalo hodnoty od první chvíle odesílání hodinových signálů, naopak data ze serveru byla ukládána s menším časovým prodlením. V principu ale vidíme, že průběhy si odpovídají a měření lze považovat za úspěšné.

[13] Pro uložení do .csv souboru pomocí Ua Expert, je nutné použít tzv. „Data logger“, ke kterému je možné se dostat cestou: Project » Documents » Add » Document Type » Data Logger View » Add, následně už stačí jen vybrat proměnnou, kterou chce uživatel ukládat.

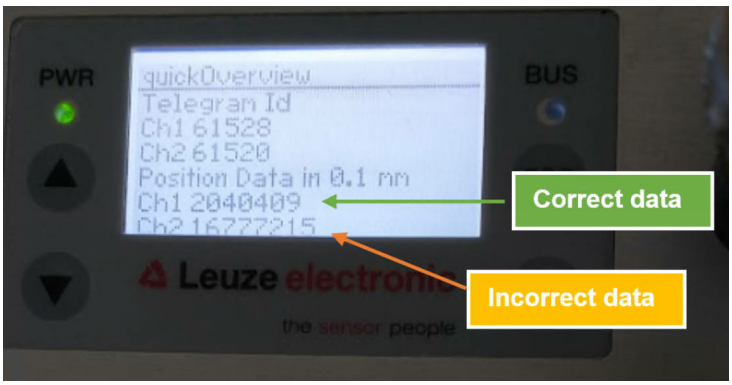
Dva aktivní kanály

Po ověření a analýze funkčnosti SSI monitoru v režimu jednoho aktivního kanálu, přecházíme na testování funkce dvou aktivních kanálů. Na obrázku 17 níže je fotografie daného uspořádaní.



Obrázek 17: Zapojení, pro testování monitoru v režimu dvou aktivních kanálů

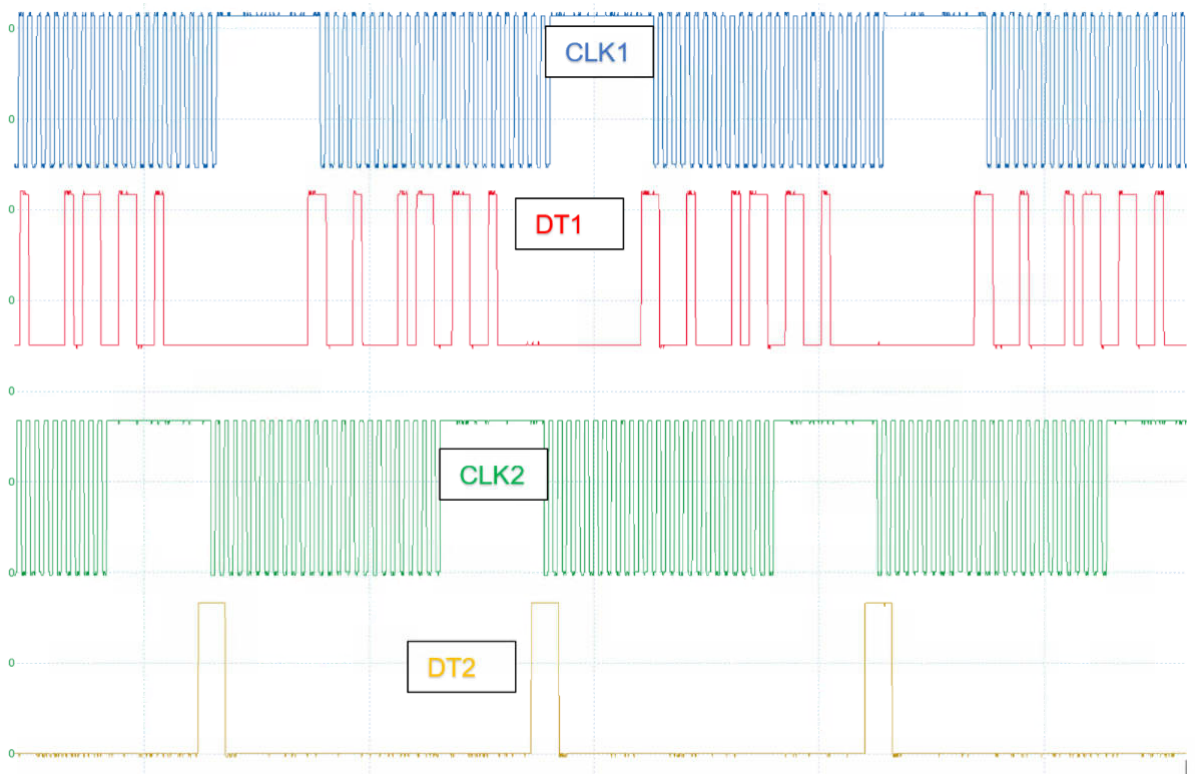
Z hlediska konfigurace jednotlivých komponent opět vycházíme z tabulky 1. Jediné, co se v tabulce nyní liší, je mód ve kterém SSI monitor pracuje — „Two Channel“. Dále je samozřejmě potřeba dvou master a dvou slave zařízení.
Zde je nutné poznamenat, že během zprovozňování této konfigurace, nebylo možné ovládat dvě stejná master zařízení z jednoho počítače. V bakalářské práci kolegy Felixe Heila [1], byl tento problém vyřešen připojením dvou rozdílných master zařízení[14]. V případě této práce, byly k dispozici pouze dva mastery MB5U, jejichž aktivování bylo tedy nutné provést odděleně, tedy každý z jiného PC.
Při zprovoznění a testování chování jsem narazil na problém s druhým kanálem, který zobrazoval nesprávná data.



Obrázek 18: Zobrazovaná data SSI monitorem — chybná data kanálu 2

[14] V práci [1] jsou použity mastery MB5U a MB4U.

Následovalo zobrazení dat pomocí interního osciloskopu, kdy bylo zjištěno, že není chyba na straně SSI monitoru, ale na straně BPS zařízení, které odesílá chybné hodnoty. Výstup osciloskopu je zobrazen níže.



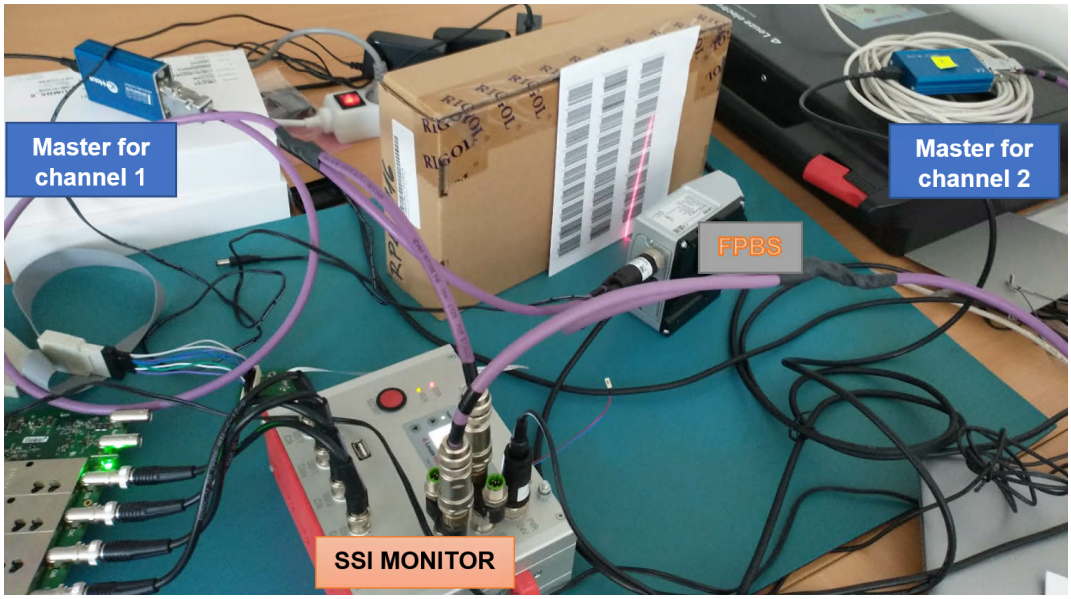
Obrázek 19: Průběhy z osciloskopu — chybná data kanálu 2

Jelikož bylo jedno slave zařízení nefunkční, bylo nutné najít jinou variantu, jak SSI monitor otestovat. Pro tyto testy nám posloužilo zařízení FBPS607i, které disponuje dvěma SSI kanály. Více informací o FBPS607i je uvedeno v první části této práce.
Konfigurace byla provedena dle tabulky 2.

-	Master zařízení (1 i 2)	FBPS607i	Monitor
Monitoring mode	-	-	Two Channel
Délka SSI zprávy (bit) - kanál 1	25 bitů	25 bitů	0
Délka SSI zprávy (bit) - kanál 2	25 bitů	25 bitů	0
Hodnota pozice (bit) - kanál 1	24 bitů	24 bitů	24 bitů
Hodnota pozice (bit) - kanál 2	24 bitů	24 bitů	24 bitů
Rozlišení pozice (mm) - kanál 1	0.1 mm	0.1 mm	0.1 mm
Rozlišení pozice (mm) - kanál 2	0.1 mm	0.1 mm	0.1 mm
Kódování - kanál 1	-	Grey	Grey
Kódování - kanál 2	-	Binary	Binary
Frekvence (kHz) - kanál 1	500 kHz	$>$ 80 kHz	0
Frekvence (kHz) - kanál 2	500 kHz	$>$ 80 kHz	0
Monoflop (μs) - kanál 1	20	20	0
Monoflop (μs) - kanál 2	20	20	0
Šířka čárového kódu (mm)	-	30 mm	-

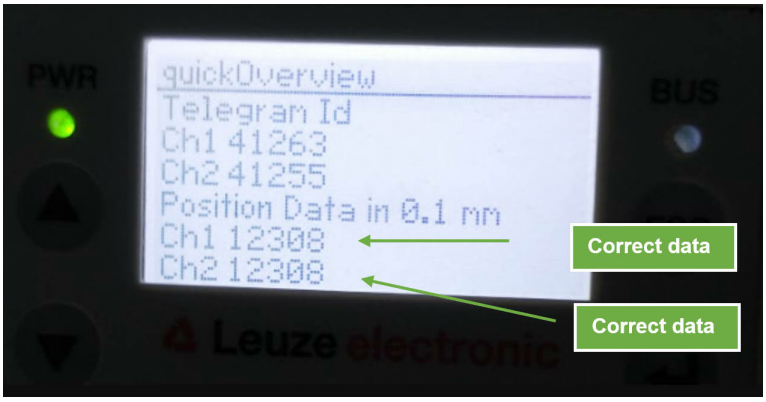
Tabulka 2: Tabulka s hodnotami pro konfiguraci

Samotné zapojení se zařízením FPBS607i je zobrazeno na obrázku 20.



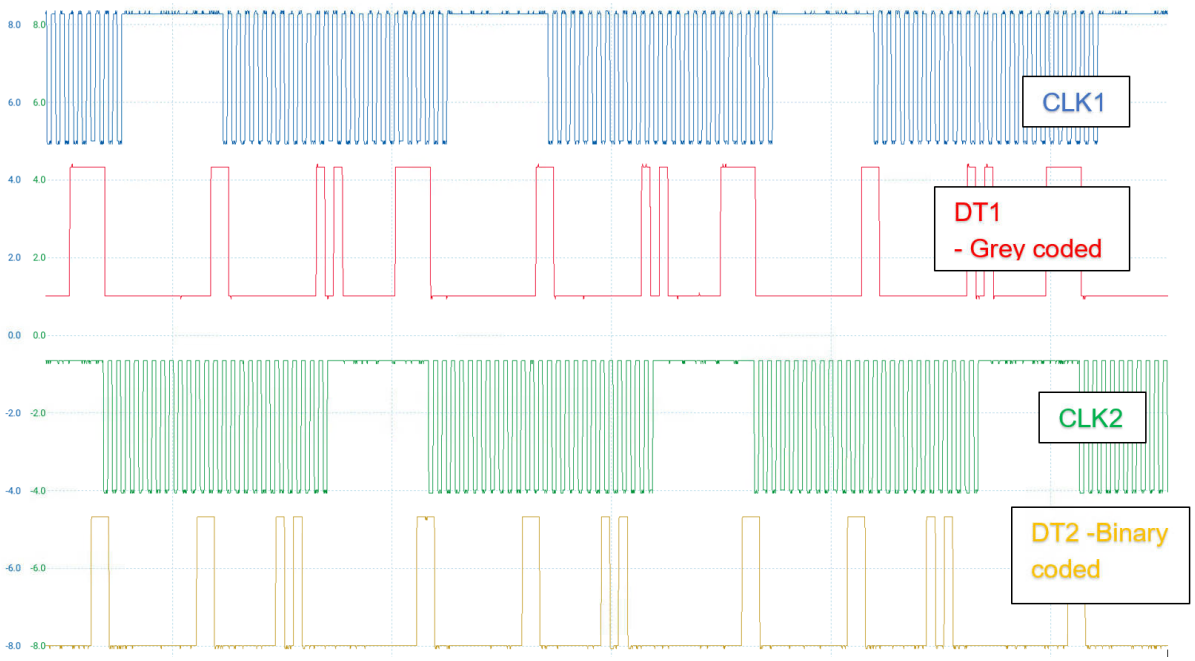
Obrázek 20: Zapojení pro testování monitoru v režimu dvou aktivních kanálů se senzorem FPBS607i

Následují obrázky získané z osciloskopu 22 a z displeje SSI monitoru 21.



Obrázek 21: Zobrazovaná data SSI monitorem — oba kanály funkční

Měření, testování a ověření funkčnosti SSI monitoru v režimu jednoho i dvou aktivních kanálů lze tuhle chvíli považovat za splněné. Dodaný SSI monitor funguje dle předpokladů, stejně tak je funkční i verze softwaru *T.0.8.0*. V dalších částech této práce, bude pojednáváno o jednotlivých rozšířeních SSI monitoru. Pro rozšíření samotných funkcionalit SSI monitoru, je nutno detailně analyzovat dodaný zdrojový kód a z analýzy pak navrhnout možnosti rozšíření, která budou implementována jako nástavba k dosavadnímu SW. Dále je možnost rozšířit monitor pomocí externího zařízení, které bude umožňovat testování bez nutnosti použití senzorů či master zařízení.



Obrázek 22: Průběhy z osciloskopu — oba kanály funkční

3 Analýza SW a návrh na rozšíření

V této kapitole se práce zabývá hlubší a detailnější analýzou dodaného kódu. Taktéž je zde popsán proces a postup nahrání a debuggování dostupných verzí softwaru. Nutno poznamenat, že při analýze zařízení, vznikaly časté problémy s dostupnostmi jednotlivých komponent, které jsou v kódu používány. Taktéž byl problém s verzemi jednotlivých komponent[1].

3.1 Verze softwaru

Jak již bylo zmíněno v předchozí kapitole, aktuální verze, nahraná přímo v zařízení, je verze *T.0.8.0*. Tato verze je funkční a disponuje všemi základními funkcionalitami, které jsou uvedeny v práci [1]. Bohužel všechny pokusy o nalezení aktuální verze, která je nahrána v zařízení, byly neúspěšné. Během práce bylo otestováno několik různých verzí SW, ale žádná z nich se neshodovala s verzí nahranou v zařízení.
V závěru však byla nalezena verze, která pracuje, alespoň částečně, jako verze *T.0.8.0*. Verze, která byla použita pro analýzu zdrojového kódu a bude pravděpodobně použita i pro budoucí rozšíření, je „trunk“ verze SW *U.0.8.0*.

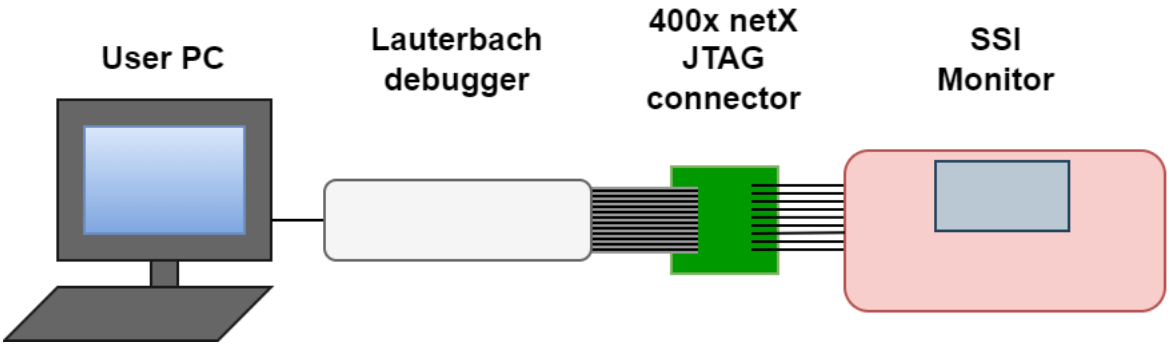
Kompilace a sestavení zdrojového kódu

Pro nahrání verze do SSI monitoru bylo potřeba stáhnout zdrojové kódy z SVN repozitáře daného projektu. Ke kompilaci a sestavení zdrojového kódu je využíván ARM toolchain a multiplatformní systém Gradle[2].
Po stažení zdrojového kódu, je nutno přejít do složky s aktuální verzí „platform“. Cesta k této složce může vypadat například ssiMon › trunk › software › application › platform. Zde je pak pomocí příkazového řádku spuštěn soubor `setToolchain.bat` a dále pak je volán příkaz `gradle`. Po zadání těchto příkazů je uživateli zobrazena tabulka s dalšími příkazy, které Gradle nabízí. Pro kompilaci zdrojového kódu pak slouží příkaz `build`. Současně se taktéž stahují verze jednotlivých softwarových komponent.
Kompilace různých verzí zdrojového kódu byla mnohokrát doprovázena chybovou hláškou `FAILURE: Build failure with and expection`. Po každém výskytu tohoto problému následovala detailní analýza všech možných problémů, kdy se ve většině případů nepodařilo spolehlivě detekovat zdroj této chyby. Po mnoha testech různých nastavení parametrů kompilace, bylo nakonec úspěšně zkompilováno několik verzí kódu. Úspěšně zkompilované verze pak byly nahrány do zařízení.

[1] Jedná se o SW komponenty SSP platformy.
[2] Gradle je open-source nástroj pro sestavování kódu.Gradle je taktéž oficiální „build tool“ pro Android a podporuje několik programovacích jazyků (Java, C++, Groovy atp.)[20].

3.1.1 Nahrání zdrojového kódu do zařízení

Pro nahrání zdrojového kódu do zařízení je používán tzv. Lauterbach debugger [21], který je připojen k zařízení pomocí JTAGu a umožňuje tak zapisování na jednotlivé piny mikroprocesoru, přístup k paměti flash a debuggování programu.
Použité uspořádání je znázorněno na obrázku 1 níže.



Obrázek 1: Uspořádání použitých zařízení pro připojení k debuggeru

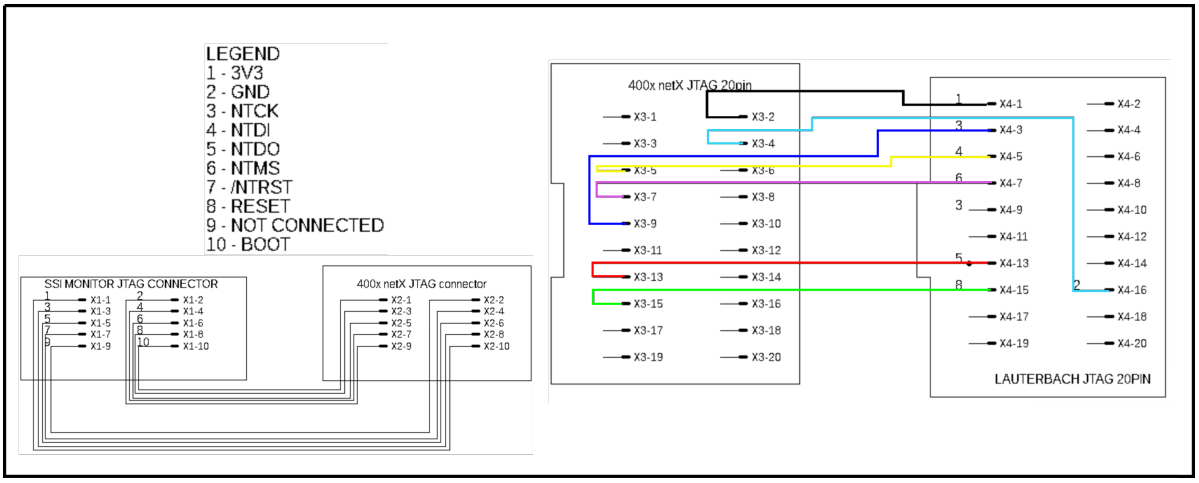
Zdrojový kód se pak nahrává do zařízení pomocí softwaru TRACE32, což je taktéž produkt firmy Lauterbach Development Tools. V aplikaci TRACE32 je nejprve nutné nakonfigurovat cesty ke zdrojovým kódům a udělat několik dalších konfiguračních nastavení. Jako první byla nastavena cesta ke složce platform, která byla zmíněna výše, jako hlavní cesta pro k přístup k jednotlivým souborům. Toto nastavení se v aplikaci TRACE32 nastavuje v konfiguračním stromě ve složce Config Tree » Example configuration » Podbus Device Chain » Power Debug USB » Core » Advanced Settings » Paths » WorkingPath. Ostatní cesty, jako například SystemPath, HelpPath a podobně, jsou již nastaveny defaultně a není třeba je měnit.
Dále probíhalo nastavení GDB portu v záložce ... » Advanced Settings » Interfaces » GDB port. Konfigurace je shrnuta v následující tabulce 1.

GDB port	
Use Port	yes
Use Auto Increment	yes
Port Start Value	30000
Port Value	30000
Protocol	TCP
Max UDP Packet Size	1024

Tabulka 1: Tabulka s hodnotami pro konfiguraci GDB portu v aplikaci TRACE32

Jako poslední krok je třeba nastavit cestu k souboru `netX.cmm`, který se nachází ve složce platform » T32. Cesta se pak v konfiguračním stromu nastavuje ... » Advanced Settings » StartupScript » File. Další konfigurace není nutná a postačí nechat vše ostatní ve výchozím nastavení. Poté stačí stisknout tlačítko Start v pravém horním rohu aplikace.
Po startu se uživateli zobrazí základní okno programu TRACE32. Před začátkem práce se zařízením je vhodné otestovat správnost připojení. Toho lze docílit pomocí detekce hodinového signálu

JTAGu. Detekci lze spustit přes horní lištu pomocí CPU > System > Detect > Continue. Pokud systém nedetekuje přítomnost zařízení, je uživateli zobrazena chybová hláška. S tímto problémem jsem se při vypracovávání této práce taktéž setkal. Hlavní problém byl, že TRACE32 pomocí lauterbach debuggeru nebyl schopen detekovat SSI monitor. Po několika analýzách jsem dospěl k závěru, že chyba byla na straně dodaného hardware, konkrétně u JTAG konektoru 400x netX, na kterém byly špatně propojeny jednotlivé piny. Dále byly na desce JTAG konektoru nalezeny studené spoje, které byly ihned, po jejich nalezení, připájeny. Správné propojení bylo realizováno pomocí propojovacích kabelů. Pro zabránění budoucích problémů bylo vytvořeno schéma s funkčním propojením, které je zobrazeno na obrázku 2.



Obrázek 2: Funkční propojení 20ti-pinového 400x netX JTAG konektoru a 20ti-pinového Lauterbach JTAG konektoru pomocí propojovacích kabelů

Pokud je zapojení v pořádku, TRACE32 detekuje SSI monitor jako jediné zařízení v daisy chainu. Následuje připojení se k SSI monitoru. Připojení probíhá opět v hlavním okně aplikace TRACE32 v horní liště netX > Connect > SsiMonitor. Po připojení Stačí kliknout na ssimonitor > start a poté v horní liště na start. V dalším kroku se do SSI monitoru nahraje sestavená verze a monitor se spustí.

Po úspěšném nahrání zdrojového kódu do zařízení, následovala analýza chování SSI monitoru s právě nahranou verzí kódu. Jak již bylo zmíněné, nejpodobnější funkcionalitou základní verze, disponuje monitor s trunkovou verzí softwaru *U.0.8.0*.

Z hlediska elementárního chování SSI monitoru, poskytuje verze *U.0.8.0* základní funkcionalitu v podobě monitorování dvou SSI kanálů. Po několika testech bylo zjištěno, že tato verze bohužel neumožňuje monitorovat pouze jeden samostatní kanál. To by v konečném důsledku ale nemuselo vadit, jelikož hlavním cílem je sledovat a monitorovat právě dva SSI kanály.

Další nepříjemností, která se při průběhu prací vyskytla, byly občasné chyby zobrazení HMI[3]. Ukázka jedné z chyb je zobrazena na obrázku 11 v příloze B. Tyto chyby by taktéž neměly mít vliv na primární funkci zařízení.

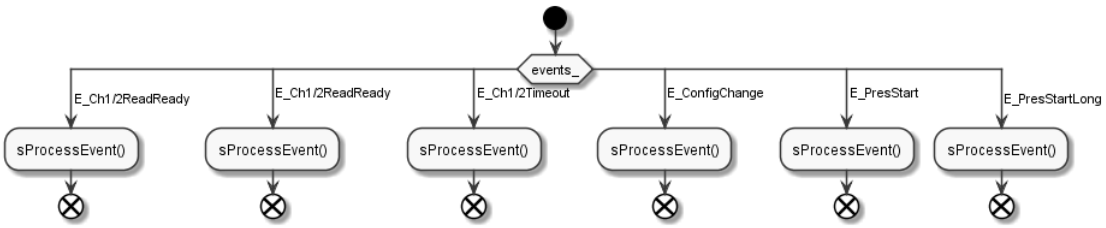
[3]Humane Machine Interface

3.1.2 Analýza zdrojového kódu verze *U.0.8.0*

V této části se práce zabývám detailnějším popisem dodaného zdrojového kódu. Nejsou zde samozřejmě popsány všechny části kódu, jelikož je software zkoumaného zařízení velice rozsáhlý. Při analýze byl kladen zřetel nejprve na základní funkcionalitu kódu a dále pak převážně na třídy a metody, které bude možné využít při následné implementaci jednotlivých rozšíření.

Základní funkcionalita

Základní funkční struktura dodaného zařízení je postavena na funkci stavového automatu, který se v průběhu běhu aplikace přepíná mezi jednotlivými stavy při reakci na předem specifikované události.
Základní inicializace je prováděna ve funkci `void SsiMonitorManager::taskInit()`, ve které jsou inicializovány používané časovače. Časovače jsou inicializovány pomocí třídy `OSAC::Timer`. Po inicializaci se přechází do funkce `void SsiMonitorManager::taskRun()`, kde se provádí nekonečná smyčka. V této funkci je pak implementován switch-case, který přepíná mezi stavy při reakci na jednotlivé události. Pro přehled byl vytvořen zjednodušený diagram 3, který tuto funkcionalitu popisuje.

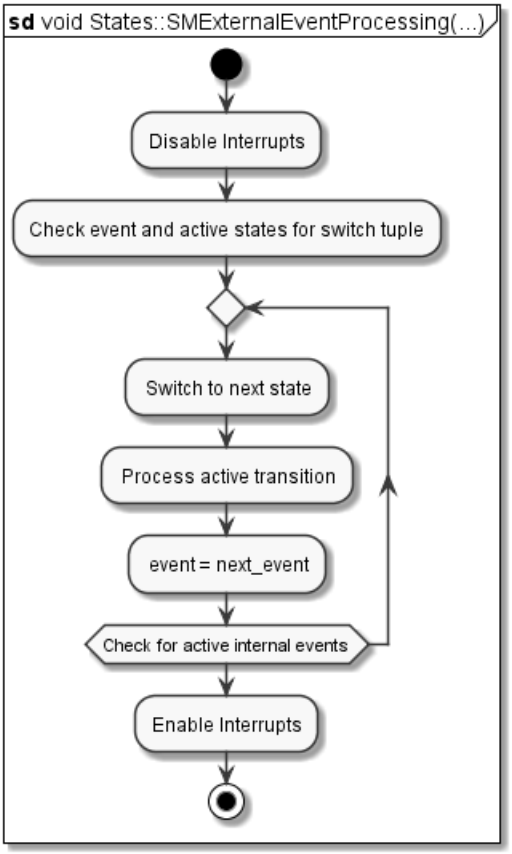


Obrázek 3: Zjednodušený diagram popisující přepínání stavů v metodě `taskRun`

Co se týče jednotlivých událostí, ve zdrojovém kódu byla vytvořena výčtová tabulka[4], ve které je 21 proměnných typu `enum`, se kterými pracuje switch-case v metodě `void taskRun()`. Taktéž byly vytvořeny tabulky obsahující 45 předběžných podmínek[5], 45 post-condition a celkem 17 stavů.
Jakmile je daná podmínka splněna, je volána metoda
`void ProcessingBase::stateProcessEvent(const S32 event)`. Tato metoda prohledává tabulku stavů, a následně volá další metodu
`void States::SM_ExternalEventProcessing(...) const`. Zde v této metodě jsou pak volány další metody. První volaná metoda `SM_CheckExternalEvents (...)`[6], kontroluje události a aktivní stavy (porovnává event pointer s tabulkou událostí). Následuje cyklus do-while, ve kterém je nejprve volána metoda která přepíná na jiný stav, následuje metoda která provádí dané instrukce během přechodu na další stav. V části while je pak kontrolováno, zda-li jsou aktivní nějaké interní události.

[4] enumeration table
[5] pre-condition
[6] Zápis (...) zde neznačí libovolný počet vstupních parametrů, jak tomu je například v prototypu funkce `printf()` (`int printf ( const char * format, ... );`), ale větší počet parametrů, kdy výpis by zabíral příliš mnoho místa.

Pro výše popisovanou funkcionalitu byl opět vytvořen jednoduchý diagram, který je zobrazen na obrázku 4 .



Obrázek 4: Zjednodušený diagram popisující funkcionalitu metody `SM_ExternalEventProcessing`, která je volána metodou `stateProcessEvent`

Pro funkci programu hraje klíčovou roli metoda `void States::SM_ProcessTransition(...)`, která je v diagramu 4 reprezentována blokem Process active transition. Přes tuto metodu se následně volá jedna z nejdůležitějších metod celého programu. Tato metoda nese název `void SsiMonitorManager::stateProcessTransition(S32 index, S32 & next_event)`.
V této metodě je opět vytvořen switch-case, který přepíná mezi mezi tzv. „action“ metodami. Celkem je přepínáno mezi 45 metodami, kdy každá metoda odpovídá přechodu mezi stavy. Nyní budou popsány pouze některé, pro budoucí rozšíření důležité, metody. První metody, odpovídající indexu 0 a 1 jsou metody:

`actionSotreAndVerifyChOne()` a `actionStoreAndVerifyChTwo()` — Asi nejvýznamnější vlastnost těchto dvou metod je, že při jejich vykonávání volají metodu `getTelegram(CHANNEL_X`[7]), díky které jsou čtena a ukládána jednotlivá data, která jsou na právě sběrnici. Pro ukázku metody `getTelegrams`, je níže naznačena její definice. Dále je v nadepsaných dvou metodách prováděna kontrola získaných telegramů. Pokud je telegram vyhodnocen jako chybný, je další událost nastavena do `E_I_TelegramError`, díky které program v následujícím kroku přejde do chybového stavu.

[7] Kde X je číslo kanálu.

```
1  void SsiMonitorManager::getTelegrams( SsiMonitorChannels_E channel_id ) const
2  {
3      if( internal_buffer_[channel_id] != NULL )
4      {
5        FrwkArgument_SsiMonitorRead_Data monitor_read_data;
6        monitor_read_data = internal_buffer_[channel_id]->prepareWrite(); // ...
             prepare for beeing written, passing the right write-array-index
7        // and the Pointer to the buffer
8        icallSync( ssi_read_triplebuffer_link_[channel_id], monitor_read_data ...
             ); // call the ressource-interface
9        internal_buffer_[channel_id]->correctIndex( ...
             monitor_read_data.array_index ); // pass the new array-index to the ...
             internal buffer
10     }
11   }
```

Další důležitá metoda, je metoda `actionRegulateBuffer()`. Tato metoda je pro tuto práci důležitá tím, že ukládá hodnoty časových razítek[8]. V této metodě, stejně jako ve velké části ostatních metod, je využívána třída `SsiMonitorTrippleBufferEntry`, pro kterou je v této metodě vytvořen ukazatel na objekt. Hodnoty časových razítek se ukládají do proměnných typu `CPP::TimeStampNano`. Taktéž je v této metodě počítán rozdíl mezi časy na obou kanálech. Tato funkcionalita bude pravděpodobně taktéž využita při budoucí implementaci potřebných rozšíření.

Jedna z důležitých metod je metoda `actionReset()`, jež při jejím zavolání vyvolá restart SSI monitoru a smazání uložených dat.

Další velmi důležitou metodou, která bude pravděpodobně v budoucnu využívána, je metoda `actionSaveData()`. Pomocí této metody jsou, při výskytu error bitu na sběrnici, pořizovány záznamy jednotlivých datových rámců. Tyto rámce jsou pak ukládány do externí paměti ve formě .csv souboru. Ukázka uložených dat v .csv souboru je zobrazena v tabulce níže.

Channel	id_	firstClkNegedge	lastClkPosedge	detectedDatabits	PositionData
0	9	4621 : 237425420	4621 : 237476440	25	13532
0	10	4621 : 237499540	4621 : 237550560	25	13532
0	11	4621 : 237573650	4621 : 237624670	25	13532
0	12	4621 : 237647760	4621 : 237698780	25	13532
0	13	4621 : 237721880	4621 : 237772900	25	13532
0	14	4621 : 237795990	4621 : 237847010	25	13532
0	15	4621 : 237870100	4621 : 237921120	25	13532
0	16	4621 : 237944220	4621 : 237995240	25	13532
0	17	4621 : 238018330	4621 : 238069350	25	13532
0	18	4621 : 238092440	4621 : 238143460	25	13532
0	19	4621 : 238166560	4621 : 238217570	25	13532

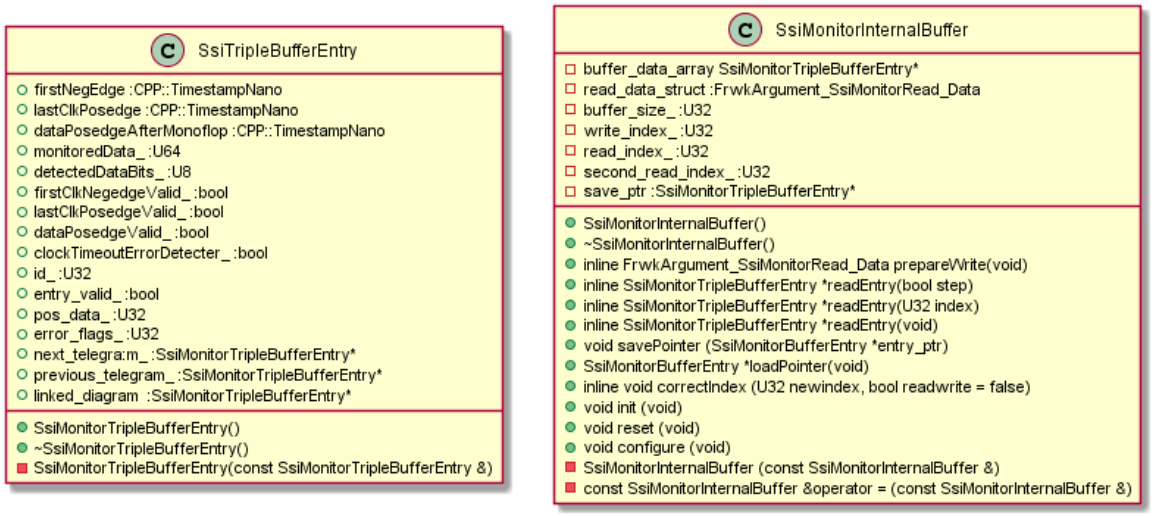
Tabulka 2: Výstup .csv vytvořený SSI monitorem uložený do externí paměti

Položka Channel v tabulce značí příslušný kanál[9]. Dále jsou v tabulce zachycena časová razítka.

[8] Ověření přesnosti časových razítek se bude věnovat pozdější část této práce.

[9] 0 - SSI kanál 1, 1 - SSI kanál 2

Formát uložených hodnot časových razítek je následující `sekundy : nanosekundy`. Do .csv souboru je uloženo ještě několik dalších informací, které v ukázkové tabulce 2, z důvodu ušetření místa, nejsou. Jedná se například o aktuální čas, kdy byl soubor vytvořen, pravděpodobný typ detekované chyby, která vznikla a podobně.
Ostatní metody, jež jsou zde volány, slouží například pro automatickou konfiguraci SSI monitoru, kontrolu správnosti datových rámců a nebo k vyhodnocování chyb vzniklých na sběrnici.
Asi nejdůležitějším výstupem analýzy zdrojového kódu jsou dvě velice často používané třídy, které se používají téměř v každé z výše zmíněných metod. Jedná se o třídy `SsiTrippleBufferEntry` a `SsiMonitorInternalBuffer`. Tyto třídy totiž obsahují metody a proměnné, které pracují s časovými razítky a jednotlivými daty, které se na sběrnici vyskytují. Na obrázku 5 jsou zobrazeny tyto třídy s jednotlivými metodami a proměnnými, které již dle názvů naznačují svou funkcionalitu.



Obrázek 5: Diagram dvou důležitých tříd, které pracují s časovými razítky a daty na sběrnici

V následující části, bude ověřena správnost časových razítek a taktéž jejich přesnost.

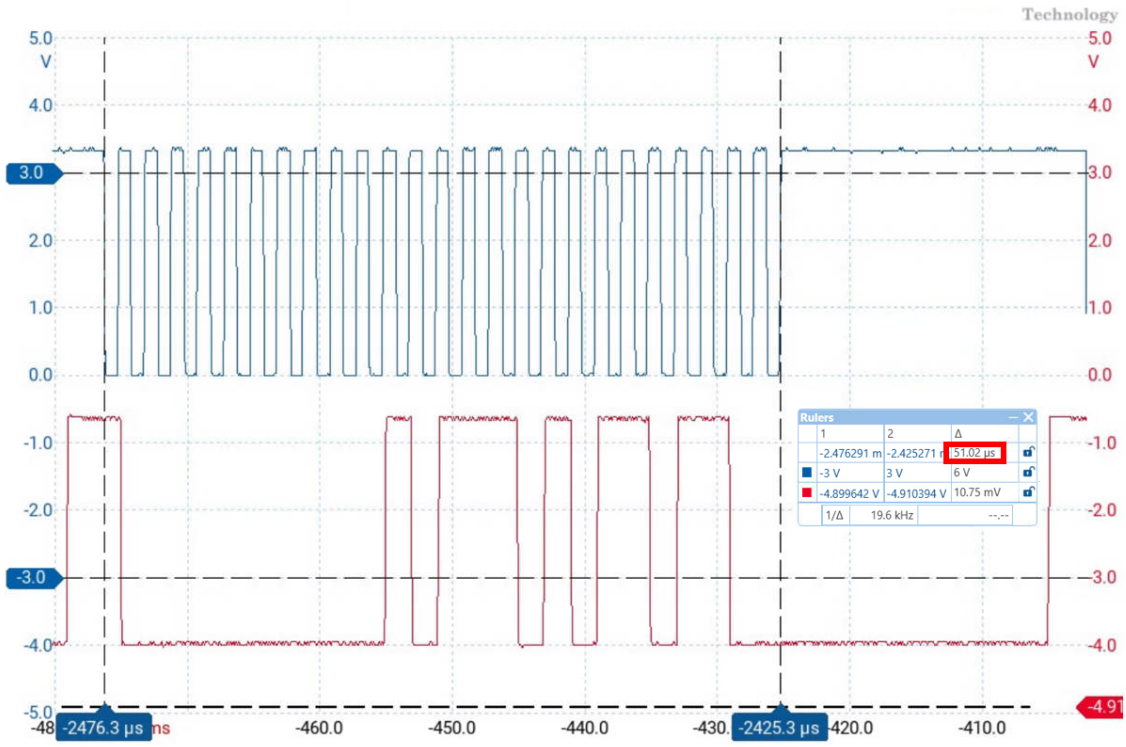
3.2 Časová razítka a synchronizace

Jak již bylo zmíněno v předchozí části, během detailní analýzy zdrojového kódu a funkcionality SSI monitoru bylo zjištěno, že tvorba časových razítek již byla implementována. Dle získaných informací, mají časová razítka přesnost v řádech nanosekund. Tuto skutečnost bylo třeba ověřit pomocí osciloskopu.
Nejprve byla převzata data z výstupního .csv souboru, která byla dále upravena. Jednou z úprav bylo rozklíčovat a odseparovat zápis `sekundy : nanosekundy` u jednotlivých časových razítek uložených ve výstupním souboru. Pro analýzu nebylo třeba použití celých sekund, ale pouze nanosekund. Dále byla vypočítána diference mezi první sestupnou hranou hodinového signálu (začátek přenosu rámce na sběrnici) a poslední náběžnou hranou hodinového signálu (ukončení přenosu rámce, za kterým následuje monoflop).

First clock negative edge (ns)	Last clock positive edge (ns)	Δ(ns)	Δ (μS)
237425420	237476440	51020	51,02
237499540	237550560	51020	51,02
237573650	237550560	51020	51,02
237647760	237698780	51020	51,02
⋮	⋮	⋮	⋮

Tabulka 3: Upravený výstup .csv souboru pro oveření přesnosti časových razítek

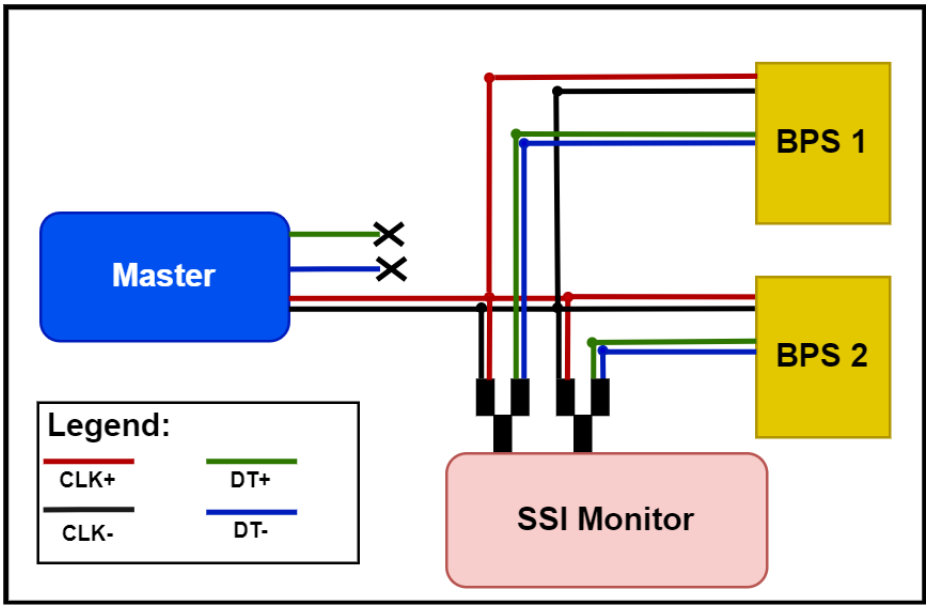
Z naměřených hodnot vidíme, že diference mezi zmíněnými časovými razítky je 51020 nanosekund. Po zisku těchto hodnot následovalo porovnání časových intervalů pomocí osciloskopu. Výstup z osciloskopu je zobrazen na obrázku 6.



Obrázek 6: Výstup z osciloskopu při měření časového intervalu mezi první sestupnou a poslední náběžnou hranou datového rámce

Modrý (horní) průběh na obrázku 6 je hodinový signál, průběh červený (dolní) je průběh dat. Měření bylo prováděno na hodinovém signálu, jelikož pro něj jsou v SSI monitoru vytvářena časová razítka. Po odečtení hodnot časů poslední náběžné hrany a první sestupné hrany pomocí kurzorů na osciloskopu, vychází diference 51,02 μS. Lze tedy prohlásit, že časová razítka vytvořená SSI monitorem jsou přesná přibližně v řádech desítek nanosekund. Po konzultaci s kolegy, pro než měla být tato funkcionalita vyvinuta, bylo dospěno k závěru, že přesnost již vytvářených časových razítek, je pro účely testování dostatečná.
Pro synchronizaci časových razítek, je nutné spouštět obě master zařízení ve stejnou dobu. Bohužel toho během této práce nebylo docíleno, jelikož software používaný pro ovládání master zařízení, dokáže komunikovat pouze s jedním z masterů. Testování základní funkčnosti probíhalo

za pomocí dvou PC, což samozřejmě znamenalo nezávislé spouštění obou master zařízení. Jedním z návrhů, jak synchronizaci jednoduše realizovat, je použití jednoho master zařízení, které bude sloužit pouze jako generátor hodinového signálu. Samotná data budou čtena pouze SSI monitorem a nebudou se dále vracet do master zařízení. Důvodem je nebezpečí „zkratování“ jednotlivých datových signálů při výskytu rozdílných potenciálů na sběrnici.
Návrh na realizaci jednoho společného hodinového signálu je zobrazen na obrázku 7.



Obrázek 7: Návrh zapojení jednoho master zařízení pro generování synchronního hodinového signálu na oba SSI kanály

Zde je nutno poznamenat, že zapojení uvedené na obrázku 7, nebylo možné realizovat z důvodu nedostatku propojovacích konektorů a kabelů s koncovkami typu M12. Zmíněné komponenty nebyly v době vypracovávání této práci k dispozici ani u samotných dodavatelů. Základní zapojení bude realizováno a otestováno ihned, jak budou komponenty k dispozici.

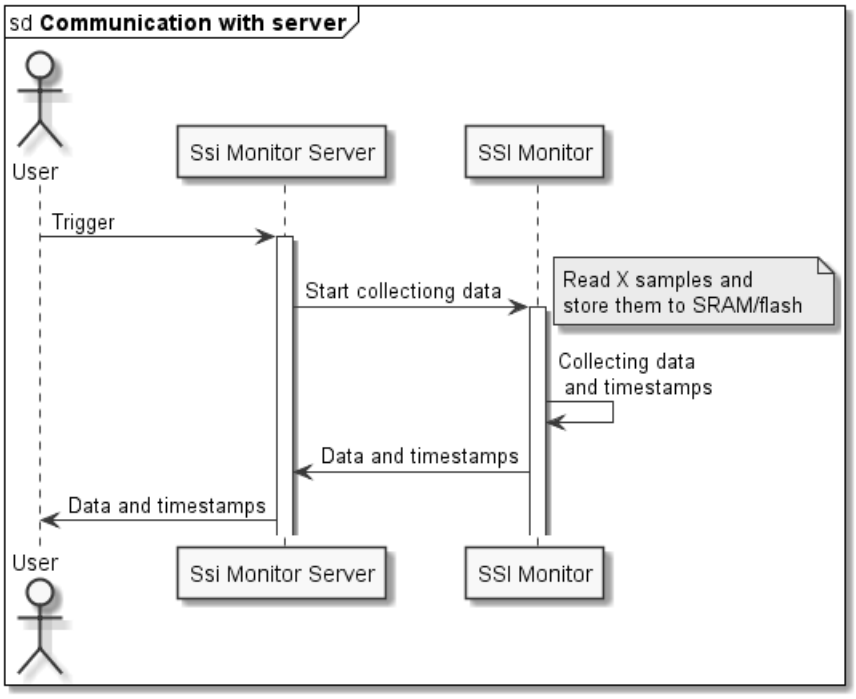
3.3 Návrh na rozšíření

Verzi softwaru *U.0.8.0* lze, po podrobné analýze, považovat za verzi, jež je použitelná pro budoucí rozšíření SSI monitoru. Pro přehled je níže uvedena tabulka, která porovnává verze SW.

-	***T.0.8.0* version**	***U.0.8.0* version**
One channel mode	Yes	No
Two channel mode	Yes	Yes
OPC UA	Yes	No
External memory data saving	partly functional	partly functional
Source code	No	Yes

Tabulka 4: Tabulka s porovnáním verze *T.0.8.0*, která je defaultně nahrána v zařízení, s verzí *U.0.8.0*, ke které je dostupný zdrojový kód

Jednou z nevýhod je, že aktuální verze použitých softwarových komponent nepodporuje OPC UA. Nelze tedy pro komunikaci s SSI monitorem použít například rozšiřující aplikaci napsanou v programovacím jazyce Python[10], nebo SCADA aplikaci, která by práci s monitorem umožňovala a zjednodušovala by tak i jeho konfiguraci atp.
Jak již ale bylo zmíněno, dostupný kód by měl být dostačující a použitelný pro možná rozšíření. Jedním z navrhovaných rozšíření je vytvořit základní komunikaci s monitorem, která nahradí chybějící OPC UA server. Komunikaci lze implementovat například za použití tzv. PT[11] příkazů, další možností je vytvořit vlastní jednoduchý komunikační protokol za použití telnetu.
Další rozšíření, které bude vytvořeno, je možnost uložení a odeslání většího množství zachycených dat s časovými razítky. Monitor zatím dokáže uložit jen několik datových rámců, které byly zachyceny před a po výskytu chyby na sběrnici. Ukládání samotných dat lze realizovat za použití externí paměti. Zde se opět nabízí otázka, jak budou data odesílána z monitoru k uživateli. Jedním z návrhů je vytvořit klient, který bude pracovat s UDP sokety, které budou odesílány monitorem.
Posledním návrhem na rozšíření, je použití již zmíněných PT příkazů, pomocí kterých by uživatel vybíral a nastavoval trigger, který by spouštěl sběr dat. Popis funkce budoucího rozšíření je zobrazen pomocí sekvenčního diagramu na obrázku 8.

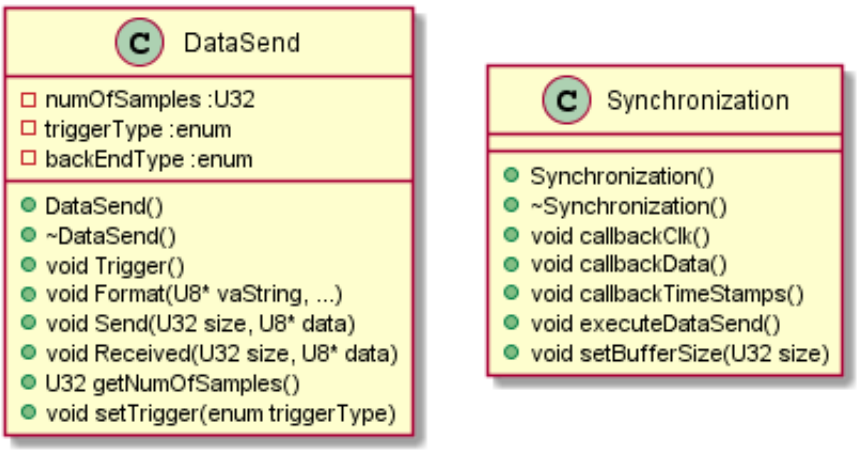


Obrázek 8: Diagram popisující návrh na rozšíření monitoru z hlediska komunikace s uživatelem

Pro tato rozšíření taktéž proběhl základní návrh tříd, které budou s největší pravděpodobností v budoucnu aplikovány. První ze tříd je třída `DataSend`, která se bude starat o ukládání a formátování získaných dat. Taktéž bude zpracovávat trigger a typ triggeru.
Druhou z těchto tříd bude třída `Synchronization`. Tato třída pak bude pracovat s jednotlivými časovými razítky a například s velikostí vyrovnávací paměti. Zatím se jedná pouze o návrh, ani

[10] Funkcionalita této aplikace je zobrazena na obrázcích 8 a 9 v předchozí kapitole.
[11] Parameter Transmit

jedna ze zmíněných tříd v tuto chvíli není přímo implementována v SSI monitoru. Ukázka návrhu těchto tříd, včetně navržených metod, je zobrazena na obrázku 9.



Obrázek 9: Návrh dvou tříd pro budoucí implementaci

Pro budoucí analýzu je vhodné mít k dispozici zařízení, které je schopno na sběrnici pracovat a poskytovat uživateli větší prostor pro možné testování ostatních zařízení, jež na SSI sběrnici pracují a samozřejmě pro testování SSI monitoru jako takového. Z tohoto důvodu vznikl návrh na zařízení, které bude na SSI sběrnici pracovat, a bude se chovat buď jako master této sběrnice, nebo jako slave zařízení (senzor). Výhodou tohoto zařízení bude například nízká cena v porovnání s používanými senzory. Návrh, vývoj a oživení tohoto zařízení, je detailně popsán v následující kapitole.

4 Master/Slave zařízení pracující na sběrnici SSI

Nadepsaná kapitola pojednává o zařízení, jehož výsledná implementace je určena pro účely testování SSI monitoru a ostatních systémů, jež se na sběrnici mohou objevit. Základní koncept tohoto zařízení je pracovat na sběrnici buď jako slave, nebo jako master zařízení. V této kapitole bude prodiskutován kompletní postup tvorby tohoto zařízení, od počátečního principiálního návrhu až po praktickou realizaci a testování.

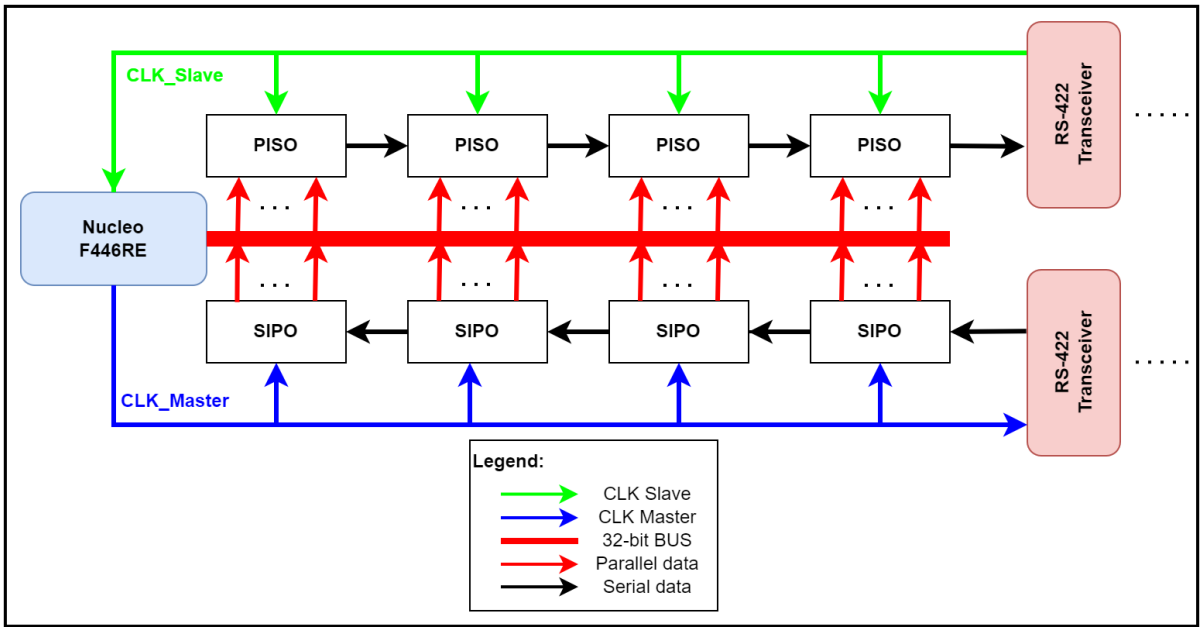
Definice požadavků na zařízení

Samotné zařízení bude fungovat jako rozšiřující deska pro vývojovou desku Nucleo F446RE od firmy STMicroelectronics. Nucleo bude rozšiřující desce poskytovat napájení 3,3 V a 5 V, rozšiřující deska tedy nebude, bez desky řídící, fungovat.
Zařízení také musí obsahovat reset-tlačítko, pomocí něž bude uživatel schopen zařízení kdykoliv restartovat.
Rozšiřující deska bude dále vybavena LE diodou pro signalizaci a informování uživatele o připojení desky k napájení. Signalizačními LED budou taktéž disponovat význačné řídící signály.
Na zařízení budou umístěny piny zmíněných řídících signálů, které budou sloužit pro kontrolu a analýzu pomocí osciloskopu či logického analyzátoru.
Zařízení musí generovat hodinový signál v režimu master a stejně tak musí provádět generování dat v režimu slave. Poněvadž jsou tyto signály určeny pro sběrnici SSI, zařízení je musí odesílat v diferenciální podobě, stejně tak musí být schopno diferenciální signály přijímat. Dále by zařízení mělo disponovat možností výběru frekvence hodinových signálů a počtem datových bitů.
Z hlediska dat, je nutností, aby zařízení dokázalo pracovat s binárním a Grayovým kódem pro správnou identifikaci přijímaných/odesílaných dat. Dalším důležitým požadavkem, je možnost interpretování dat o předem definovaném rozlišení[1].
Dále zařízení musí disponovat uživatelským rozhraním, ve kterém se budou jednotlivá data zobrazovat. Pomocí uživatelského rozhraní bude uživatel schopen nakonfigurovat zařízení dle vlastní potřeby (nastavení frekvence, počtu bitů, kódování, výběr módu, ve kterém zařízení bude pracovat atp.). Uživatelské rozhraní by také mělo uživatele informovat o všech důležitých parametrech, o aktuálním stavu zařízení (přijímaná/odesílaná data) a o tom, zda-li na sběrnici vznikla chyba atp. Uživatel by měl být, pomocí tohoto uživatelského rozhraní, schopen restartovat dané zařízení a znovu ho nakonfigurovat. Samotné zařízení by mělo být při zapnutí a konfiguraci ve stavu vysoké impedance.
Zařízení by mělo disponovat funkcí, kdy po každém měření budou získané hodnoty uloženy do .csv souboru pro možnost následné analýzy popř. převedení do grafu.

[1] SSI monitor a používané senzory umožňují pracovat s rozlišením 0,01 mm, 0,1mm nebo 1mm.

4.1 Základní princip a funkce zařízení

Základním úkolem zařízení, je přijímat/odesílat data a zároveň odesílat/přijímat hodinové signály ze sběrnice SSI. Pro snadnější pochopení principu funkce bylo vytvořeno principiální blokové schéma na obrázku 1.



Obrázek 1: Principiální blokové schéma univerzálního master/slave zařízení

Hlavní řídící deska Nucleo F446RE generuje či přijímá hodinové signály. Hodinové signály zařízení generuje v režimu master. Tyto hodinové signály jsou posílány do SIPO (Serial Input, Parallel Output) posuvných registrů, které jsou v tomto režimu aktivní. Stejně tak jsou hodinové signály posílány do diferenciálního převodníku na rozhraní RS-422, ze kterého jsou pak, ve formě diferenciálního signálu, odesílány do slave zařízení.

Slave zařízení, po obdržení hodinových signálů odesílá diferenciální sériová data. Data vstupují do převodníku, který je v tuto chvíli převádí z diferenciálních na „jednoduchý“ sériový signál. Získaný signál je poté odesílán do posuvných registrů, ze kterých je následně paralelně vyčtena zpráva na paralelní 32-bitové sběrnici. V režimu master musí být samozřejmě PISO (Parallel Input Serial Output) posuvné registry ve stavu vysoké impedance.

V režimu slave jsou naopak ve stavu vysoké impedance posuvné registry SIPO. Hlavní řídící deska Nucleo F446RE v režimu slave přijímá hodinové signály, které jsou přiváděny od externího master zařízení po SSI sběrnici. Hodinové signály přichází opět jako diferenciální. Po průchodu převodníkem na rozhraní RS-422, pokračuje „jednoduchý“ hodinový signál do PISO registrů.

Hlavní řídící deska v režimu slave generuje data, jež jsou následně z GPIO pinů procesoru, pomocí PISO registrů, vyčtena. Data jsou dále z PISO registrů, v reakci na externí hodinový signál, sériově posouvána až do převodníku, z něhož jsou, v podobě dvou komplementárních signálů, odeslána do master zařízení.

4.2 Návrh a realizace hardware

Tato část práce se zabývá hardwarovou implementací vytvářeného zařízení. Budou zde uvedeny použité komponenty, jejich funkce a zařazení do celkového schématu zapojení. Dále zde bude zobrazen výsledný návrh desky plošných spojů a její finální provedení.

Použité komponenty

Jak již bylo zmíněno, jako hlavní řídící deska je použita vývojová deska od firmy STMicroelectronics Nucleo F446RE. Jádro procesoru tvoří ARM 32-bit Cortex-M4. Jedním z hlavních důvodů použití tohoto procesoru byla rychlost procesoru, kterou lze nakonfigurovat až na 180 MHz. Dalším, ještě význačnějším důvodem volby, bylo množství programovatelných GPIO pinů, které jsou potřeba pro zapisování a čtení dat z jednotlivých posuvných registrů. Vývojová deska ještě navíc obsahuje ST-Link, který umožňuje programovat, flashovat a debugovat mikroprocesor bez nutnosti externího ladícího programu. Desku resp. ST-link lze připojit k PC pomocí USB mini. Pomocí USB mini je také možné realizovat komunikaci se zařízením za využití periférie UART, bez nutnosti použití dalších GPIO pinů pro vlastní konfiguraci UARTu. Z hlediska paměti, mikroprocesor disponuje pamětí flash, která má 512 kbytů a pamětí SRAM 128 kbytů. Více informací o této vývojové desce lze samozřejmě najít v datových listech [22]. Jedním z dalších aspektů, který hrál roli při výběru, byla zkušenost s mikroprocesory a vývojovými deskami od firmy STMicroelectronics.
Dalšími komponentami jsou posuvné registry. Nejprve se zaměříme na registry PISO. Použité PISO registry nesou označení 74HC165D [23]. Vstupem do těchto registrů jsou paralelní vstupy pro data a dále pak vstupy pro řídící signály. Jednotlivé vstupy a výstupy jsou uvedeny a popsány v tabulce 1 níže.

Název vstupního/výstupního pinu	Funkce
SH/nLD	Určuje, jestli jsou data posouvána, nebo čtena
SER	Sériový vstup z jiného sériově připojeného PISO registru
CLK INH	Povoluje příjem hodinových signálů
CLK	Vstup pro hodinový signál
Qh	Sériový výstup
nQh	Negovaný sériový výstup
A, B, C, D, E, F, G ,H	Paralelní vstupy

Tabulka 1: Popis pinů posuvného registru 74HC165D

PISO registry pracují v režimu slave. Jejich hlavním úkolem je, paralelně načítat data z výstupních pinů mikroprocesoru. Po paralelním výčtu, jsou data sériově posílána do master zařízení. Obecně je funkce těchto registrů zřejmá ze schématu vnitřního zapojení a z časového diagramu. Obojí je uvedeno v příloze C na obrázcích 12 a 13.
Posuvné registry SIPO, nesoucí označení 74HC595 [24], jsou aktivní v režimu master. Do těchto registrů přichází sériová data ze slave zařízení, která jsou následně paralelně vyčítána vstupními piny mikrokontroléru[2]. Do těchto posuvných registrů vstupují dva hodinové signály, jeden je pro

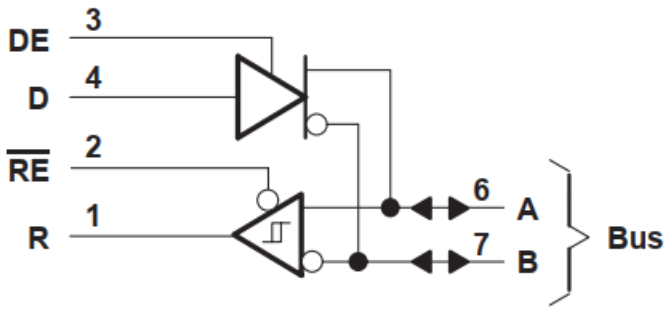
[2]V režimu master jsou GPIO piny procesoru nakonfigurovány jako piny vstupní a v režimu slave jako výstupní.

tzv. „storage register“ a druhý hodinový signál pro tzv. „shift register“. Dva zmíněné hodinové signály jsou odděleny, ale pro některé aplikace, kde je například k dispozici pouze jeden hodinový signál, mohou být tyto piny propojeny a pracovat pouze jeden vstup[3]. V tabulce 2 níže jsou opět uvedeny jednotlivé vstupy/výstupy registru 74HC595.

Název vstupního/výstupního pinu	Funkce
nOE	Stav vysoké impedance / povolen paralelní výstup
RCLK	Hodinový signál "shift"registru
nSRCLR	Smazání dat uložených v registru
SRCLK	Hodinový signál "storage"registru
SER	Sériový vstup z předchozího registru
Qa, Qb, Qc, Qd, Qe, Qf, Qg, Qh	Paralelní výstupy
nQh	Negovaný výstup

Tabulka 2: Popis pinů posuvného registru 74HC565

Pro lepší pochopení základní funkce tohoto registru je opět uveden časový diagram na obrázku 14 v příloze C.
Poslední komponentou je diferenciální převodník na rozhraní RS-422 nesoucí označení SN65176B [25]. Funkce tohoto převodníku je velmi jednoduchá. Převodník obsahuje šest pinů. Piny A, B jsou diferenciální vstupní/výstupní piny dat, které převodník přijímá/odesílá. Dále se na převodníku nachází piny D a nRE, které určují směr komunikace. Tyto dva piny lze jednoduše propojit a vytvořit tak jeden pin s označením DIR, který rozhoduje o tom, zda-li se převodník chová jako vysílač, nebo jako přijímač.



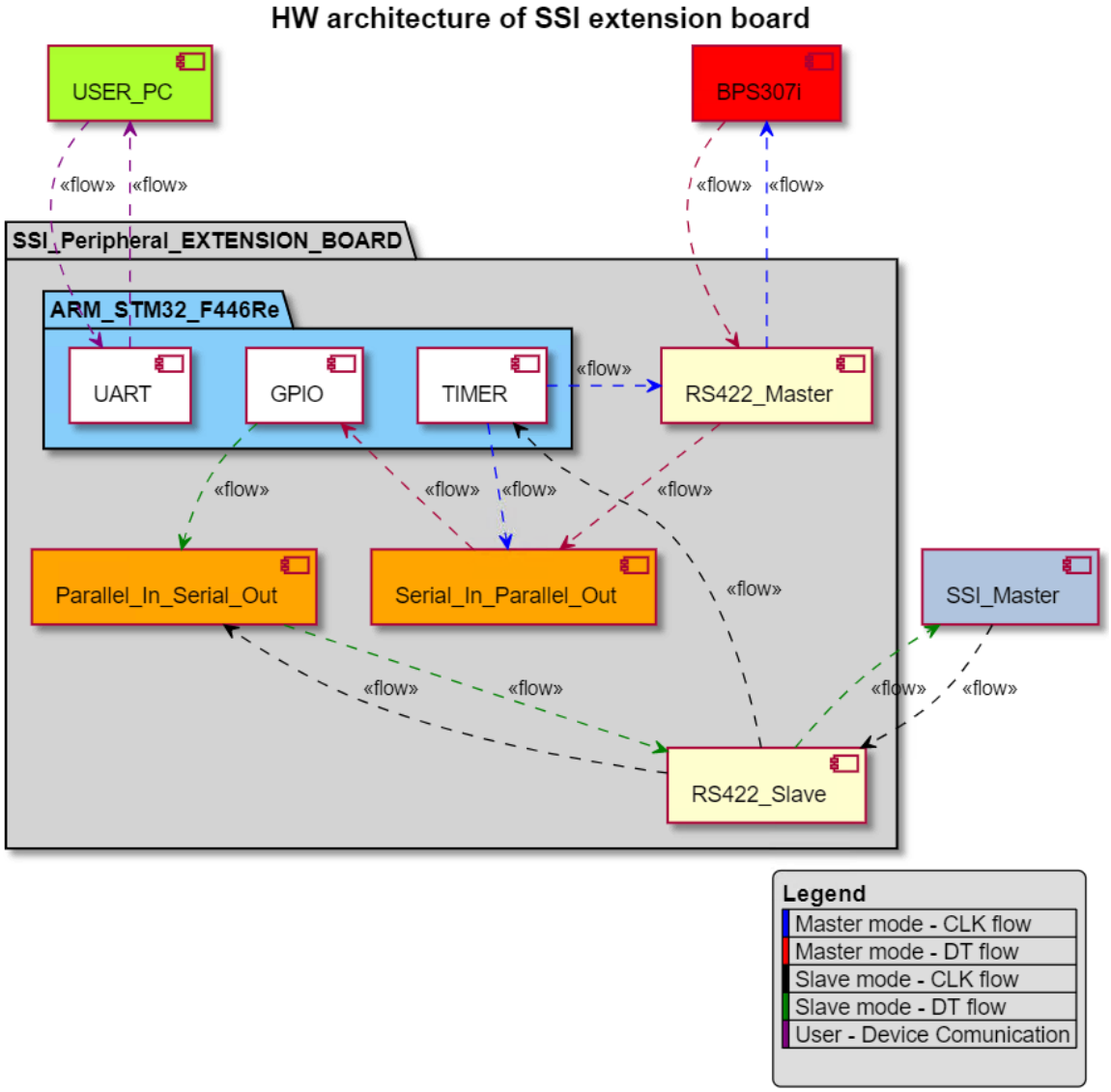
Obrázek 2: Vnitřní zapojení převodníku SN6517B [25]

Pokud je převodník v režimu přijímač, na piny A a B je přiváděn diferenciální signál, který následně pokračuje do třístavového bufferu se Schmittovým obvodem[4], dále je pak signál převeden na „jednoduchý“ signál na pinu R. Naopak, pokud je převodník v režimu vysílače, signál je přiváděn na pin DE a následně je pak v logickém hradle převeden na dva komplementární elektrické signály s opačnou polaritou.

[3] To ovšem není případ této aplikace.
[4] Vlivem kapacit kabelů by mohlo docházet k pomalejším změnám napětí (nabíjení a vybíjení kapacit), a tedy i k nesprávným překlopením klopného obvodu, proto je zde použito hradlo s hysterezí, tedy Schmittův klopný obvod, který zde taktéž plní funkci třístavového bufferu.

4.2.1 Výsledný hardware

Tato podkapitola se zabývá procesem vytvoření hardwarového celku s využitím výše uvedených součástek. Pro upřesnění a pochopení je níže uveden diagram s navrženou hardwarovou architekturou vyvíjeného zařízení.



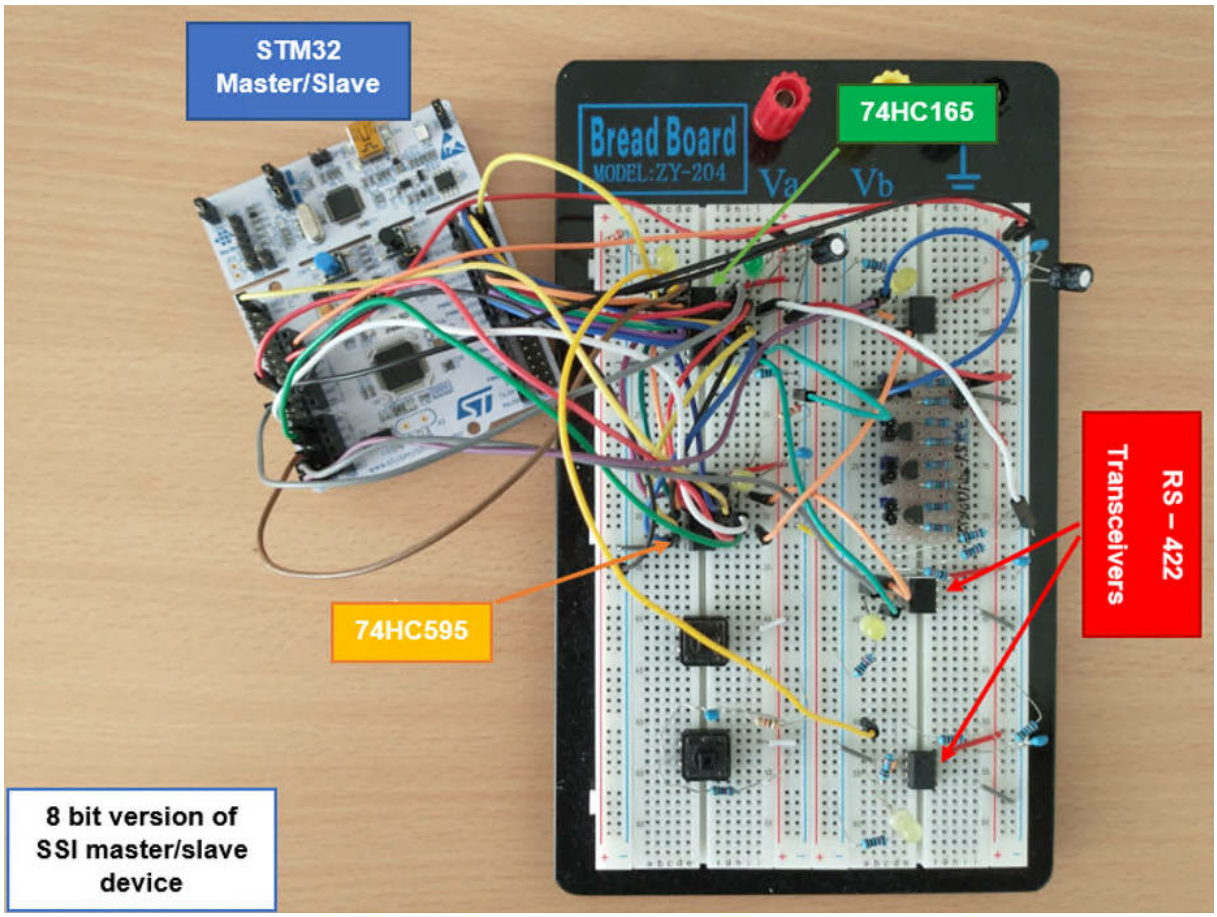
Obrázek 3: HW architektura vyvíjeného zařízení

Obrázek 3 popisuje, jak mezi sebou jednotlivé komponenty komunikují. Na obrázku jsou pro lepší pochopení zahrnuty i externí komponenty, se kterými vyvíjené zařízení bude ve finále pracovat. Samotné schéma zapojení jsem navrhoval v programu EAGLE Autodesk [26], ve kterém byla posléze vytvořena deska plošných spojů. Finální navržené schéma zapojení je uvedeno v příloze C na obrázku 15.

Jednotlivé moduly[5], jsou napájeny samostatně požadovaným napětí, dle katalogových listů jednotlivých součástek. U napájení jsou použity blokovací kondenzátory k zamezení rušení na vedení a zamezení rušení galvanickou, kapacitní a induktivní vazbou blízkého elektromagnetického pole.

[5]Části schématu zapojení, jako například převodníky RS-422, posuvné registry atd.

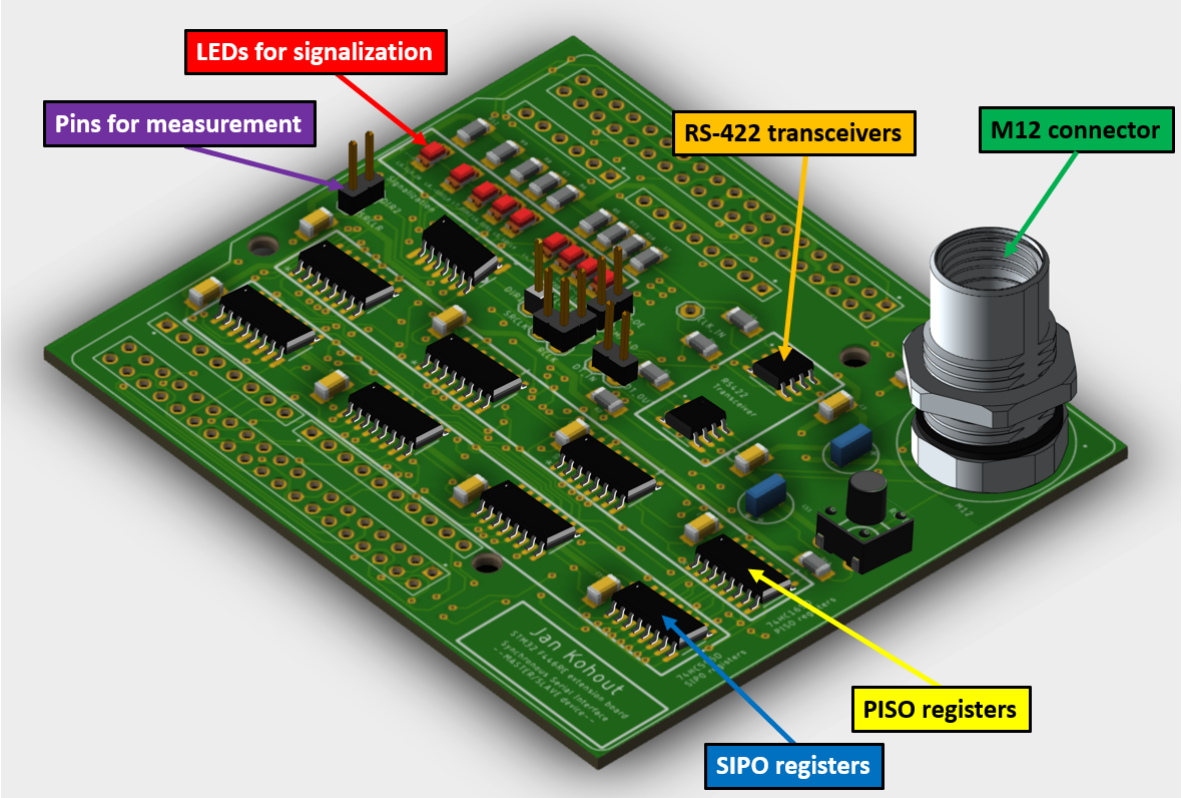
Napájení téměř všech modulů je 3,3 V, výjimkou jsou převodníky RS-422, které jsou napájeny napětím 5 V. Z toho důvodu, byl použit napěťový dělič, jenž převádí 5voltový signál na 3,3voltový. Samotné napájení je samozřejmě zajišťováno řídící deskou.
Proto, abychom se vyhnuly problémům s úbytky napětí vlivem připojení signalizačních diod, byl použit samostatně napájený invertor, díky čemuž se úbytky napětí neprojeví na zbytku obvodu. Po navržení schématu zapojení následovalo testování základního konceptu na nepájivém kontaktním poli. Pro jednoduchost bylo testování prováděno pouze jako zjednodušená osmibitová verze. Software, který byl použit pro základní ověření konceptu v této práci není uveden, jelikož později bude uvedena jeho finální implementace. Na obrázku 4 níže je uvedeno testovací zapojení na nepájivém kontaktním poli.



Obrázek 4: 8-bitové zapojení pro ověření základního konceptu

Pomocí osmibitového testovacího zapojení, uvedeného na obrázku 4, byl ověřen základní koncept funkce navrhovaného zařízení. Uvedená osmibitová verze je schopna generovat diferenciální hodinové signály při zvolené frekvenci a taktéž umí generovat data. Stejně tak dokáže přijímat data či hodinové signály. Nutno dodat, že v této osmibitové verzi nebylo nijak implementováno rozklíčování přijímaných dat ani monoflop. Uvedené zapojení 4 bylo samozřejmě realizováno dle finálního schématu 15. Zpracované grafy z jednotlivých měření pomocí osciloskopu a logického analyzátoru, jsou uvedeny v příloze C na obrázcích 16 (master mód) a 17 (slave mód). Výběr módu a následné spouštění bylo realizováno pomocí UARTu a vytvoření jednoduchého UI. Ukázka UI je opět uvedena v příloze C na obrázku 18. Více detailů o tvorbě finálního uživatel-

ského rozhraní, bude popsáno v další části této práce.
Po ověření základního konceptu a funkčnosti následoval návrh desky plošných spojů. Deska plošných spojů byla navržena v programu EAGLE Autodesk. Přesto, že se nepohybujeme na velmi vysokých frekvencích, byla snaha návrh desky co nejvíce optimalizovat z hlediska funkčnosti a odolnosti vůči rušení, které na desce může, vlivem interferenčních vazeb blízkého pole, vznikat[6]. Deska obsahuje pouze dvě vrstvy. Na horní vrstvě jsou umístěny všechny používané součástky. Spodní vrstva byla využita pouze pro vedení vodivých cest a pro přiletování pinheaderů pro následné připojení k řídící desce. Obrázky obou vrstev desky plošných spojů se nachází v příloze C (19 - horní vrstva, 20 - spodní vrstva). V příloze C je uveden i kompletní seznam součástek. Dále byl vytvořen 3D model desky, jenž je uveden na obrázku 5.



Obrázek 5: 3D model desky pošlých spojů pro master/slave zařízení

Na obrázku 5 jsou vyznačeny, z důvodu přehlednosti, pouze ty nejdůležitější komponenty. V pravé horní části se nachází konektor pro kabel M12. Modře je pak poukázáno na SIPO registry. Ve vedlejší řadě se pak nachází žlutě vyznačené PISO registry. Fialově jsou vyznačeny piny řídících, datových a hodinových signálů, které slouží pro připojení logického analyzátoru či osciloskopu. Červeně jsou vyznačeny signalizační LED, které jsou napájeny z invertoru. V pravé části desky se pak nachází reset tlačítko a převodníky na rozhraní RS-422. Dále je nutno poznamenat, že na delších krajích DPS jsou vytvořeny otvory, které slouží pro propojení s řídící deskou. Propojení je realizováno nasazením rozšiřující desky na desku řídící. Informativní obrázek 21 rozšiřující desky s připájenými součástkami a nasazením na řídící desku, je zobrazen v příloze C.

[6]Při návrhu a optimalizaci DPS jsem vycházel z obecných informací získaných ze zdrojů [27], [28] a [29].

4.3 Návrh a implementace software

Softwarová část zařízení je realizována v jazyce C++[7]. Pro psaní a úpravu kódu je využíváno prostředí Visual Studio Code. Velkou výhodou tohoto prostředí je možnost využívání různých druhů rozšíření, které mimo jiné pomáhají uživateli udržet kód čistý a přehledný.

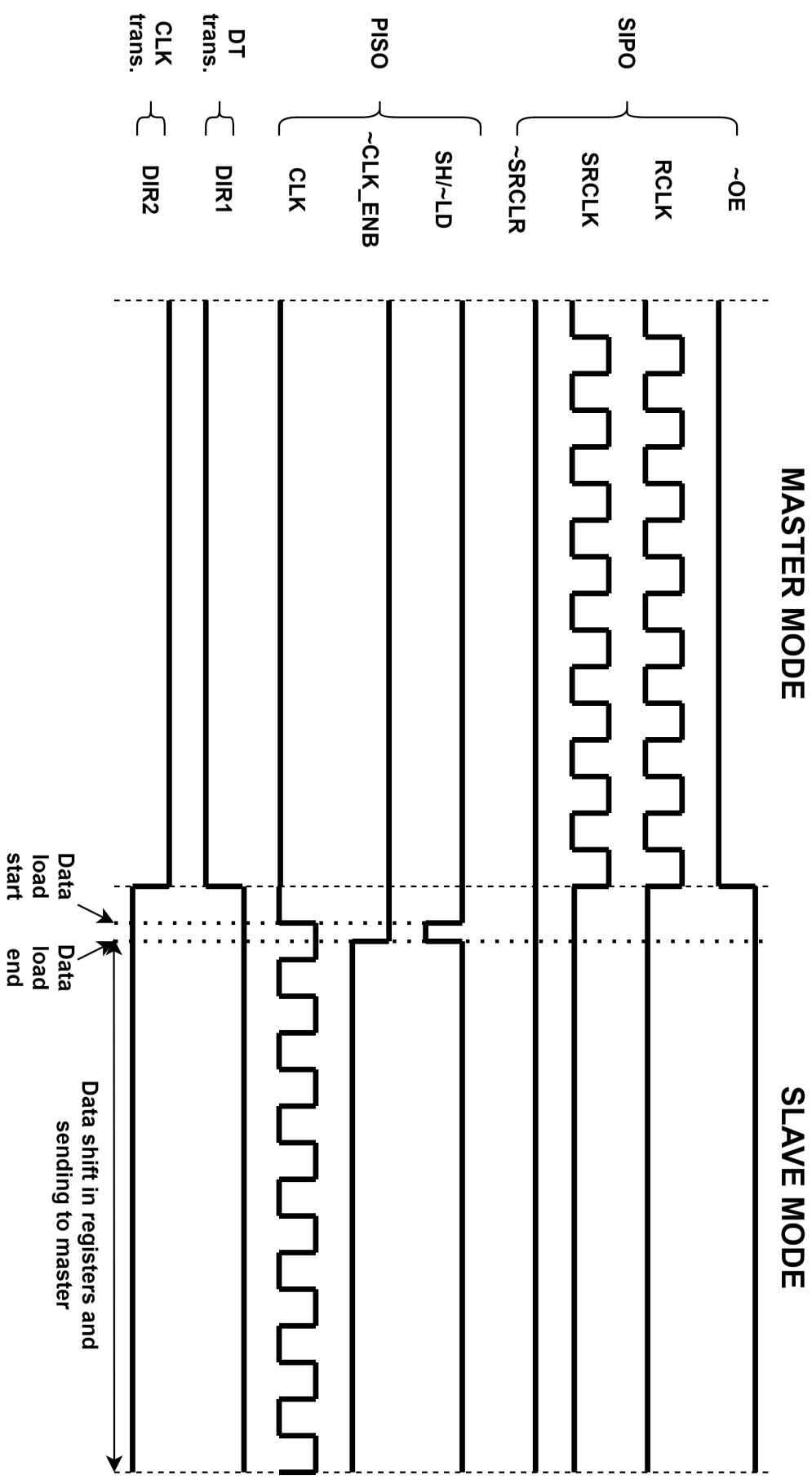
Základní nastavení programovacího prostředí

Proto, aby bylo možné kód přeložit, sestavit a nahrát do zařízení, je třeba využít sestavovací soubor[8]. Sestavovací soubor byl vytvořen pomocí aplikace STM32CubeMX, která také slouží pro konfiguraci jednotlivých vývodů mikroprocesoru[9].
Dále je v práci využíváno několik rozšíření. Jedním z nich je *STM-helper* [30], pomocí nějž jsou vytvořeny soubory s příponou .json[10] (JavaScript Object Notation), které jsou pro použití VSCodu nezbytné. Dalším rozšířením je pak *stm32-for-vscode* [31], které umožňuje uživateli kompilovat aplikaci a následně ji nahrát do řídící desky.
Ladění je umožněno díky ST-linku, který je součástí řídící desky, pouze připojením se k PC přes USB. Pro samotný debugging je opět použito rozšíření nesoucí název *Cortex-Debugg*[32].
Výchozí jazyk pro programování procesorů STM32 například pomocí STM32CubeIDE, je jazyk C. Jak již bylo zmíněno, soubory nezbytné pro práci ve VSCode byly vygenerovány aplikací STM32CubeMX. Mezi zmíněnými soubory je i soubor *main.c*. Tento soubor stačí jednoduše přepsat na *main.cpp* a znovu ho přeložit, díky čemuž je možné použít programovací jazyk C++.

4.3.1 Časový diagram zařízení

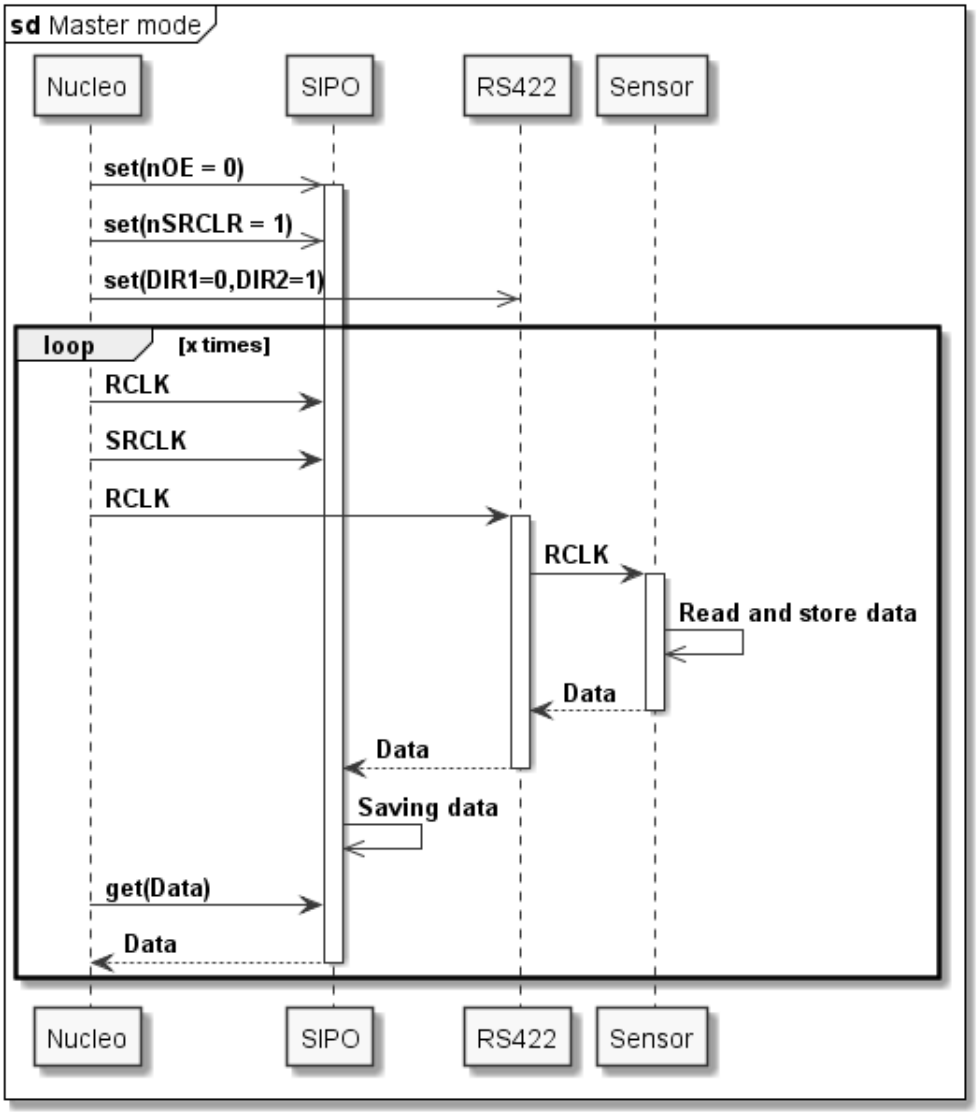
Po základním nastavení prostředí pro návrh a realizaci software, musí být opět zohledněna hardwarová část návrhu. Výsledný kód musí korespondovat s funkcí jednotlivých komponent a součástek, které jsou umístěny na desce plošných spojů. Z tohoto důvodu byl vytvořen časový diagram, který popisuje, jaké důležité řídící piny jsou v jakých časech v logické jedničce či logické nule. Vytvořený časový diagram je uveden na obrázku 6. Diagram zobrazuje oba módy, ve kterých se zařízení může nacházet. V diagramu jsou zohledněny pouze řídící signály na třech základních komponentech, kterými jsou PISO registry, SIPO registry a převodníky na rozhraní RS-422. Nejprve je na obrázku vyznačen režim master, ve kterém jsou aktivní SIPO registry. Zde je nutné poznamenat, že signály *SRCLK* a *RCLK* jsou posunuty o 180°, aby byl zajištěn správný posun dat posuvnými registry. Na desce se nachází dva převodníky na rozhraní RS-422, kdy jeden z převodníků je určen pro hodinové signály a druhý pro data. Režimy těchto dvou převodníků tedy samozřejmě závisí na režimu zařízení. Stavy těchto převodníků se mění v závislosti na signálu *DIR*, který je do převodníků odesílán z hlavní řídící desky.
V režimu slave dochází k paralelnímu načtení hodnot generovaných řídící deskou. Data jsou pak sériově odesílána do master zařízení.

[7]. Pro překlad programu byl použit kompilátor g++ verze 10.3.0.
[8]Častěji se setkáváme s anglickým pojmem „makefile“.
[9]V této práci však není tato možnost konfigurace mikroprocesoru využita. Důvodem je, že vygenerovaný kód sloužící ke konfiguraci je psaný v jazyce C, avšak vyvíjená aplikace je v C++.
[10]launch.json, settings.json, task.json



Obrázek 6: Časový diagram řídících a hodinových signálů master/slave zařízení

Pro lepší pochopení základní funkcionality uvedené v diagramu na obrázku 6, vznikly ještě dva sekvenční diagramy. Každý z diagramů popisuje základní nízkoúrovňovou interakci mezi jednotlivými komponentami.

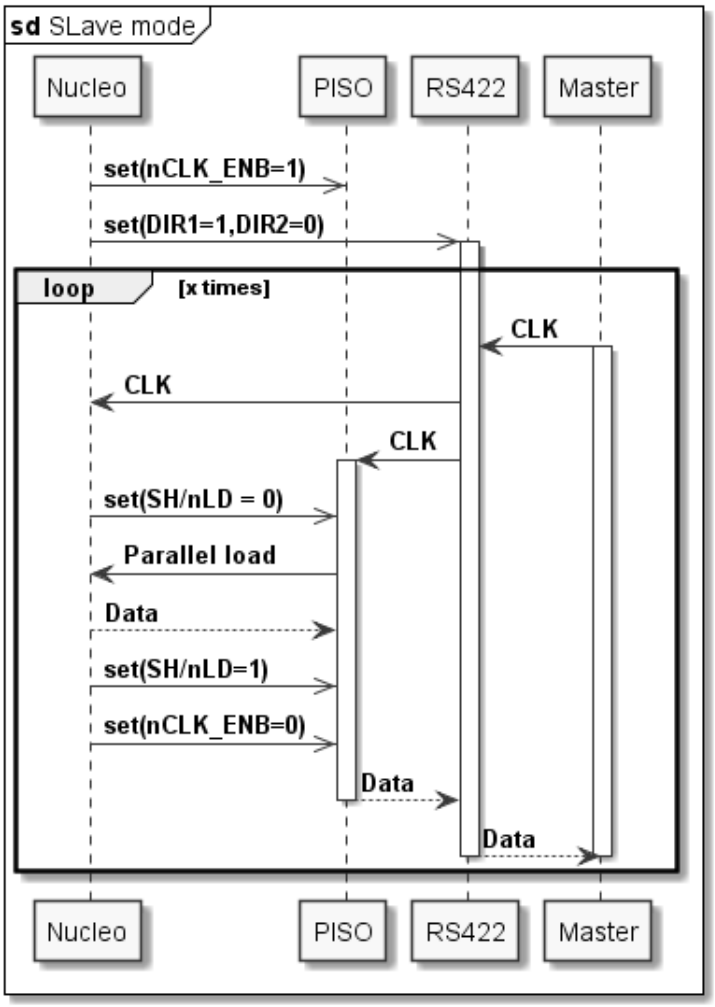


Obrázek 7: Sekvenční diagram základní komunikace mezi hardwarovými komponenty vyvíjeného zařízení v režimu master

Ze sekvenčního diagramu na obrázku 7 vidíme základní komunikaci mezi komponenty vyvíjeného zařízení. Obrázek 7 koresponduje s částí obrázku 6, konkretně s částí master.
V režimu master, jsou nejdříve odesílány řídicí signály do SIPO registrů a do převodníků na rozhraní RS-422. Tyto signály uvádí jednotlivé součástky do funkce. Následně jsou generovány hodinové signály, které jsou odesílány do SIPO registrů pro posouvání dat v registrech. Hodinové pulsy jsou zároveň odesílány přes převodník na rozhraní RS4-22[11], do senzoru, jenž v reakci na hodinový signál vrací právě čtená data. Data ze senzoru jsou v podobě diferenciálního signálu posílána do zařízení, kde se opět pomocí převodníku RS-422[12] převádí na „jednoduchý“ signál,

[11] Kde jsou převedeny na diferenciální signál.
[12] Na desce jsou umístěny dva tyto převodníky, jeden pro data a jeden pro hodinové signály. Z důvodu přehlednosti a ušetření místa, je v diagramu 7 použit jen jeden blok pro reprezentaci těchto dvou součástek.

který je sériově odesílán do posuvných registrů, z nichž pak vyvíjené zařízení, pracující v režimu master, vyčte data pro další zpracování.



Obrázek 8: Sekvenční diagram základní komunikace mezi hardwarovými komponenty vyvíjeného zařízení v režimu slave

Na obrázku 8 je zobrazen sekvenční diagram pro komunikaci mezi hardwarovými komponenty v režimu slave. Tento diagram opět odpovídá časovému diagramu zobrazenému na obrázku 6, konkrétně pravé části tohoto obrázku, tedy režimu slave.

V tomto režimu nejprve dochází odeslání řídících signálů jednotlivým komponentům na desce plošného spoje. Zařízení je v klidovém stavu, dokud externí master zařízení nezačne generovat hodinové signály. Diferenciální hodinový signál přichází na převodník RS-422, ze kterého pokračuje do PISO registrů a do řídícího mikroprocesoru. V reakci na příchod hodinového signálu nejprve mikroprocesor odešle *SH/nLD = 0* řídící signál do PISO registrů, čímž je přepne do režimu paralelního načtení. PISO registry paralelně vyčtou výstupní piny mikroprocesoru, reprezentující data, která jsou následně odeslána do master zařízení. Po načtení mikroprocesor opět nastaví PISO registry do režimu *shift* a současně povolí vyčítání dat pomocí hodinového signálu, generovaného master zařízením. Data jsou pak sériově vyčítána z posuvných registrů a přes převodník na rozhraní RS-422 odesílána do master zařízení.

4.3.2 Hardwarová abstraktní vrstva

Pro jednodušší manipulaci s používanými perifériemi, jako jsou GPIO, UART a jednotlivé časovače, byla pro tuto práci vytvořena hardwarová abstraktní vrstva, která umožňuje uživateli jednoduše nakonfigurovat jednotlivé periférie. Tato abstraktní vrstva je určena přímo pro tento projekt a je napsána v jazyce C++. Nejprve bude tato podkapitola pojednávat o tvorbě hardwarové abstraktní vrstvy pro univerzální vstupní a výstupní obvody, následovat budou časovače a periférie UART.

Univerzální vstupní/výstupní obvody

O abstraktní vrstvu pro univerzální vstupní/výstupní piny se stará třída `SsiGpio`. Zde je z hlediska hardwaru nutno podotknout, že přiřazení hardwarových vstupních a výstupních pinů mikroprocesoru k jednotlivým datovým a kontrolním pinům součástek, bylo realizováno tak, aby usnadnilo pozdější práci při návrhu desky plošných spojů. Tabulka přiřazení jednotlivých bran a pinů k datovým a kontrolním signálům je uvedena v příloze C na obrázku 22.
Po přiřazení jednotlivých GPIO pinů k pinům datovým a kontrolním, následovala tvorba jednotlivých metod pro práci s GPIO piny. Základní metody pro samotnou inicializaci jsou volány pomocí konstruktoru typu singleton[13]. Následuje základní výčet jednotlivých metod[14]. Nejprve se zaměříme na metody, které jsou volány při inicializaci tj. při vytvoření objektu třídy.

`void ssiconfigPin(GPIO_TypeDef *port,uint32_t pinNumber, uint32_t modeType)` - Pomocí této metody se nastavují jednotlivé piny do požadovaného módu (vstupní, výstupní, analogový mód atp.)

`void ssiOutputType(GPIO_TypeDef *port,uint32_t pinNumber, uint32_t outputType)` - Pokud se GPIO pin nachází ve výstupním módu, tato funkce přesněji určuje, jestli je výstup v režimu *push-pull* nebo v režimu *open-drain*.

`void ssiPullUpPullDown(GPIO_TypeDef *port,uint32_t pinNumber, uint32_t pullType)` - Tato metoda pracuje s v pull-up/pull-down registrem.

`void ssiOutputSpeed(GPIO_TypeDef *port,uint32_t pinNumber,uint32_t speed)` - Tato metoda pracuje s rychlostí GPIO pinů.

Následují metody, které již pracují s již zmíněnými metodami třídy `SsiGpio`, které jsou zmíněné výše.

`void SsiGPIOout(GPIO_TypeDef *port, uint32_t pinNumber)` - Vstupem této metody je pin a jeho příslušná brána. Použitím této metody můžeme libovolný GPIO pin nastavit do výstupního režimu, pouze zadáním jeho brány a čísla pinu.

`void SsiGPIOin(GPIO_TypeDef *port, uint32_t pinNumber)` - Tato metoda funguje obdobně, jako metoda předchozí, s tím rozdílem, že požadovaný pin nastaví do režimu vstupu.

[13] Tj. po vytvoření objektu třídy popř. ukazatele na objekt třídy.
[14] Uvedené metody jsou pro ušetření místa uváděny bez kontextu (např. `SsiGpio::`, `SsiUart::`,...) k příslušné třídě.

Dále jsou pak v této třídě dvě metody `void DTout(void)` a `void DTin(void)`, které jsou volány při inicializaci příslušného master/slave módu. Tyto metody nastaví všechny datové piny do režimu vstupu, nebo výstupu, dle daného módu rozšiřujícího zařízení.

Dále byly v této třídě vytvořeny dvě uživatelské funkce, sloužící pro přepínání jednotlivých pinů do logické jedničky, nebo nuly.

`void WRITE(GPIO_TypeDef *port, uint32_t pinNumber, uint32_t state)` - vstupem této metody je opět brána, pin a požadovaný stav.

`void TOGGLE(GPIO_TypeDef *port, uint32_t pinNumber)` - tato metoda, již dle svého názvu, přepíná logickou hodnotu zadaného pinu.

Nejdůležitější metodou je samozřejmě konstruktor třídy (v tomto případě je použit konstruktor typu singleton). Při vytvoření instance této třídy, se zavolá konstruktor `SsiGpio()`, který provede prvotní inicializaci, povolí přístup hodin na jednotlivé brány a následně pomocí metod třídy nastaví všechny řídící signály do výstupu. Informaci o dokončení ukládá do proměnné třídy `bool infoSsiGpioCtrl` jako hodnotu *true* při úspěšné inicializaci. Tato třída samozřejmě obsahuje i destruktor, který provádí de-inicializaci periférie.

Časovače

Před samotným popisem abstraktní vrstvy periférie časovače, je třeba zmínit samotnou konfiguraci hodin. Tato konfigurace je vytvářena funkcí `void SystemClock_Config(void)`, která je automaticky generována při základní tvorbě projektu v prostředí STM32CubeMX. V této práci je tato funkce využívána, jelikož v ní lze jednoduše upravovat parametry a tím si přizpůsobit konfiguraci dle vlastních požadavků.
Nejprve bylo do této funkce přidáno povolení jednotlivých časovačů, které jsou v práci využívány. Dále jsou v této funkci nakonfigurovány systémové hodiny. Hlavní výstupní frekvence mikroprocesoru dosahuje 180 MHz, vlivem před-děliček, ale některé periférie nedosahují frekvencí 180 MHz, ale například 90 MHz pro GPIO. Při nastavování systémových hodin je vhodné využívat diagram uvedený v příloze C na obrázku 23.
O hardwarovou abstraktní vrstvu se starají tři třídy. Třída `SsiTimer` je tzv. „base class“, jež obsahuje proměnné, důležité pro inicializaci a konfiguraci jednotlivých časovačů. Následují dvě zděděné třídy, kterými jsou třídy `Timer2` a `Timer3`. Třída `Timer2` přísluší módu master. Obsahuje dva přetížené konstruktory. První konstruktor je tzv. „deleted constructor“ `Timer2() = default`, který zabraňuje možnosti inicializovat konstruktor bez parametrů. Druhý konstruktor `Timer2(uint32_t prsc, uint32_t per)`, obsahuje dva vstupní parametry a při vytvoření instance třídy provede základní inicializaci[15] časovače, povolení přerušení od časovače atp. Pokud se inicializace provede, ukládá se tato informace do proměnné `bool infoTimer2`.
Vstupní proměnné konstruktoru slouží pro nastavení frekvence časovače. Vše je odvozeno z následujícího vzorce, který je dostupný v katalogových listech řídící desky [22]. Samotný vzorec vypadá takto,

$$\frac{Peripheral\ clock}{Desired\ timer\ frequency} = Prescaler \cdot ARR. \tag{1}$$

[15]Pro inicializaci jsou využívány některé funkce vytvořené výrobcem mikroprocesoru [22].

Pokud bude uživatel chtít nastavit výstupní frekvenci například na 500 kHz, při frekvenci periférie 180 MHz, po dosazení dostává,

$$\frac{180\ 000\ 000}{500\ 000} = 360. \tag{2}$$

Číslo 360 je pak nutno rozdělit do před-děličky (tu reprezentuje proměnná `uint32_t prsc`) a period registru (reprezentace proměnnou `uint32_t per`).
Stejně tak třída obsahuje i destruktor `~Timer2()`, který se volá vždy, když objekt zaniká. Dále byla vytvořena metoda `Timer2End()`, která způsobí okamžitou de-inicializaci časovače, při jejím zavolání.
Třída `Timer3` je pak aktivní v režimu slave a má obdobnou funkcionalitou jako třída `Timer2` .

UART

Hardwarová abstraktní vrstva pro periferii UART je opět reprezentována třídou. Tato třída se nazývá `SsiUart`[16]. Třída obsahuje několik metod určených pro konfiguraci a inicializaci UART periférie. Jednotlivé metody budou nyní popsány[17].
`void ssiUARTWordLen(uint32_t& word_len)` - Tato metoda nastavuje délku slova, tedy počet bitů zprávy, která může obsahovat devět, nebo osm bitů.
`void ssiUARTStopBits(uint32_t& stop_bits)` - V této metodě je nastavován stop-bit, tj. poslední bit přenosu, který je vždy v logické jedničce.
`void ssiUARTParity( uint32_t& parity_bits)` - Tato metoda, jak název napovídá, slouží k nastavení parity bitu, což je bit pro detekci chyby, který se vkládá mezi datové bity a stop bit. Zde je možné nastavit lichou či sudou paritu, nebo také žádnou.
`void ssiUARTOverSampling(uint32_t& over_sampling)` - Tato metoda řeší převzorkování (častěji „oversampling“), což je kontrola jednotlivých bitů zprávy proti tzv. noise erroru. Noise error vzniká, právě tehdy, když se jednotlivé vzorky daného bitu neshodují. Zde je možné počet vzorků nastavit na 8 nebo 16.
`void ssiUARTtransreceive(uint32_t& trans_or_rec)` - Následující metoda je využívána pro nastavení periférie UART do režimu vysílače, přijímače anebo do režimu, kdy periférie funguje souběžně jako vysílač a přijímač.
Tato třída obsahuje dva přetížené privátní konstruktory[18]. Konstruktor `SsiUart()`, bez vstupních parametrů inicializuje periferii UART dle výchozího nastavení[19]. V této práci se s tímto konstruktorem dále pracuje a výchozí nastavení je uvedeno v tabulce 3.

[16] Během budoucích úprav této práce by bylo jednoznačně vhodné, místo názvů Ssi..., použít například namspace SsiHAL, který by obsahoval všechny třídy zmíněných periférií.
[17] Opět bez kontextu `SsiUart::`.
[18] U tříd, které obsahují privátní konstruktory, se také lze setkat s označením „singletony“.
[19] Výchozí nastavení je samozřejmě možné změnit v definici konstruktoru.

Parametr	Výchozí hodnota
Přenosová rychlost	115200
Délka slova	8 bitů
Stop bit	1 bit
Paritní bit	Žádní parita
Mód zařízení[20]	Přijímač i vysílač
Převzorkování	16 vzorků

Tabulka 3: Výchozí nastavení UART periferie

Samozřejmě je možné si UART periferii nakonfigurovat pomocí přetíženého konstruktoru, jehož vstupními parametry jsou všechny parametry uvedené v tabulce 3. Oba výše uvedené konstruktory taktéž povolí přerušení od periférie UART. Informace o úspěšné inicializaci se, podobně jako u předchozích periferií, ukládají do proměnné `bool infoUart`.
Tato periférie bude dále využívána pro zobrazování jednotlivých dat získávaných z externích zařízení, jež budou tato data odesílat. Dále bude pomocí této periférie vytvořena jednoduchá aplikace/uživatelské rozhraní, které poslouží uživateli k základnímu výběru parametrů, módu zařízení a dále také pro informování uživatele o aktuálním stavu zařízení a podobně.

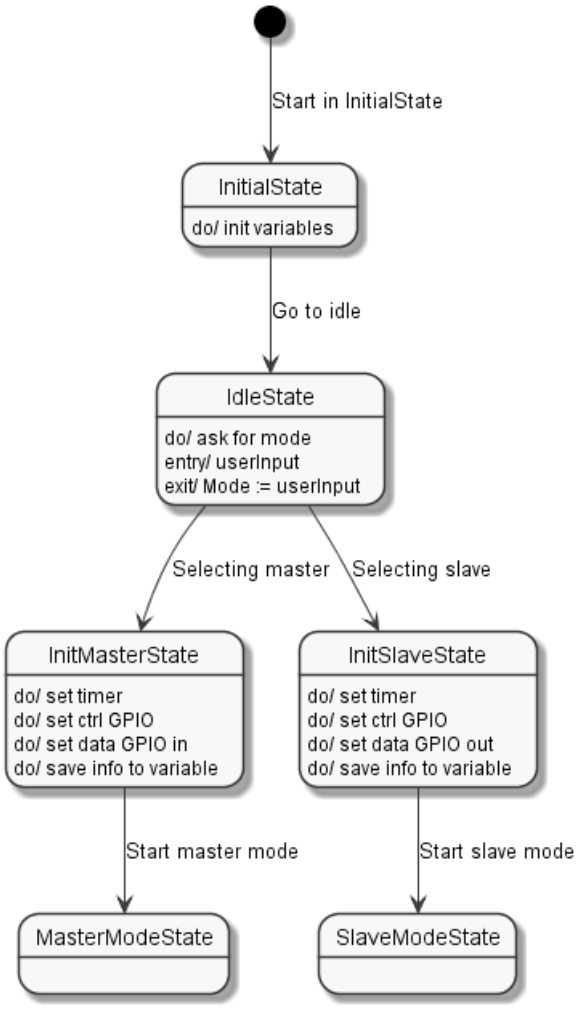
4.3.3 Realizace algoritmu pro základní funkci zařízení

Tato část se zabývá návrhem a realizací algoritmu pro funkci master/slave zařízení. Tvorba algoritmu pro vyvíjené zařízení vychází ze základního principiálního schématu, jež je uvedeno na obrázku 1 a z časového diagramu 6. Zohledněn je i kompletní hardwarový návrh. Zde je nutno poznamenat, že návrh softwaru a hardwaru probíhal paralelně.

Inicializace

Deklarace jednotlivých proměnných probíhá před vstupem do hlavní funkce main. Stejně tak se zde nachází prototyp funkce, které slouží pro převod kódu z binárního do Grayova, o této funkci ale bude řeč později.
Následuje vstup do funkce main, ve které se provede inicializace systméhových hodin a základních periférií, jako je UART a kontrolní GPIO. Dále jsou volány metody, které se dotáží uživatele na jednotlivé parametry, jako například frekvence, počet bitů zprávy a podobně. Podrobnější informace o těchto metodách se nachází v pozdější části této práce.
Dále je volána funkce `void SelectMode(void)`, která čeká na vstup uživatele, jenž si musí vybrat patřičný mód (slave nebo master). Funkce uživateli také umožňuje restartovat mikroprocesor a provést tak celou inicializaci znovu. Po výběru módu se o následující stará třída `Mode`, která slouží jako abstraktní třída pro třídy `Slave` a `Master`. Tato třída obsahuje virtuální funkci `virtual void ModeSet(SsiGpio& gpio) = 0`. Tuto funkci samozřejmě dědí zmíněné dvě třídy. Pokud uživatel vybere režim slave, je vytvořen objekt třídy `Slave` a volána metoda `void Slave::ModeSet(SsiGpio& gpio)`, ve které probíhá nastavení jednotlivých kontrolních pinů dle časového diagramu 6 pro režim slave. Dále proběhne inicializace a nastavení časovače do režimu *input capture* a datových pinů, které jsou v tomto režimu nastaveny jako piny výstupní. Jakmile toto

proběhne, následuje nastavení proměnných, podle kterých se řídí hlavní smyčka programu (`bool master_flag = false` a `bool slave_flag = true`).
Pokud si uživatel ve funkci `void SelectMode(void)` zvolí režim master, je vytvořen objekt třídy `Master` a je volána stejná metoda jako v předchozím případě `void Master::ModeSet(SsiGpio& gpio)`. V této metodě jsou opět nastaveny řídící piny dle časového diagramu pro režim master, viz obrázek 6, datové piny jsou nastaveny do režimu vstupu a samozřejmě nechybí inicializace a nastavení časovače, který je v tomto režimu nastaven jako *Output compare*. Po inicializaci se, jako v předchozím případě, zapíší informační hodnoty do pomocných proměnných (`bool master_flag = true` a `bool slave_flag = false`).
Výše popsaná inicializace je pro úplnost zobrazena ve stavovém diagramu[21] na obrázku 9.



Obrázek 9: Stavový diagram popisující inicializaci

Na obrázku 9 vidíme několik základních stavů ve kterých se při inicializaci nacházíme. V dolní části diagramu jsou zobrazeny dva hlavní stavy/módy tohoto zařízení. Oba stavy jsou popsány dále a jsou pro ně vytvořeny jejich sub-state-diagramy, které navazují na diagram 9.

[21] Nutno dodat, že výsledný kód není implementován jako state-machine. Diagramy tohoto typu slouží pouze pro lepší znázornění daného problému.

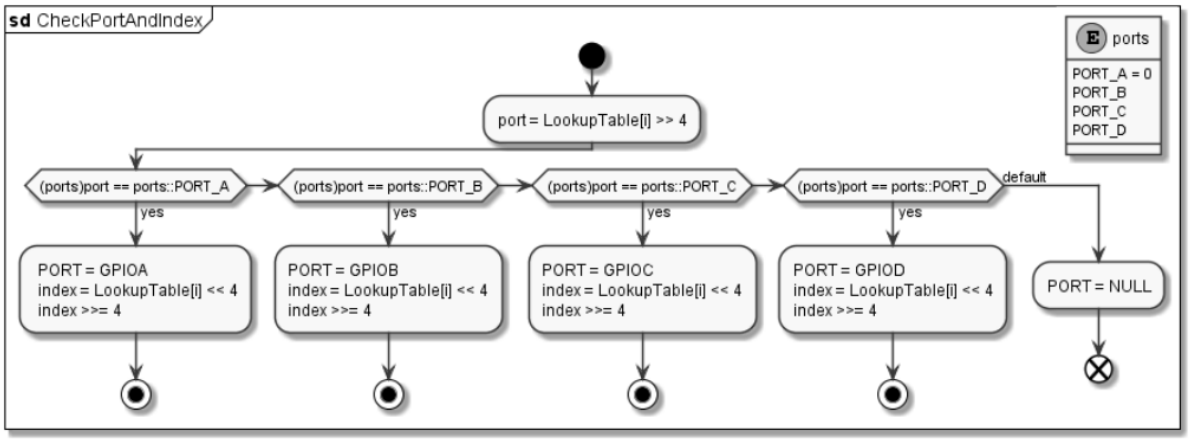
4.3.3.1 Režim master

Jak již bylo zmíněno v popisu základní funkce sběrnice SSI, master zařízení na této sběrnici se stará o generování hodinového signálu, na čež mu slave zařízení vrací data.
Funkcionalita generování hodin je zajištěna inicializací časovače v metodě `void Master::ModeSet (SsiGpio& gpio)` a následným přepínáním hodnoty výstupu časovače ve funkci obsluhy přerušení `void TIM2_IRQHandler(void)`. V této funkci jsou přepínány hodnoty na výstupech SRCLK a RCLK, čímž je generován požadovaný hodinový signál. Jako pomocná proměnná zde slouží proměnná `uint16_t cnt_value`, která pracuje s aktuální hodnotou čítače.
Během generování hodinového signálu jsou sériově přijímána data, která jsou při jednotlivých hodinových pulsech posouvána v posuvných registrech. Jakmile je odpočítán požadovaný počet tiků hodin, který odpovídá délce bitů zprávy, jsou data z posuvných registrů vyčtena a uložena pro další zpracování.
O čtení a parsování dat se stará třída `Data`. Parsování dat je prováděno pomocí „vyhledávací tabulky“[22]. V této tabulce jsou vytvořeny osmibitové, celočíselné, bezznaménkové proměnné, kdy každá obsahuje informaci o čísle datového signálu, bráně a příslušném pinu. Ukázka je uvedena níže.

```
1   const uint8_t dt0 = 0x29;  // DT0, GPIOC, PIN9
```

Je vidět, že prvnímu (nultému) datovému signálu přísluší brána C, která je reprezentována hexadecimálním číslem 0x2 a pin devět, které je reprezentován hexadecimálním číslem 0x09. První čtyři bity tedy reprezentují bránu a další čtyři bity reprezentují pin.
Proměnné pro ostatní data jsou vytvořeny obdobným způsobem. Nakonec jsou všechny tyto proměnné uloženy do pole `uint8_t LookUpTable[SIZE]`, kdy velikost `SIZE` koresponduje s počtem bitů zprávy.
V přerušovací rutině je vytvořena instance třídy `Data`. Pomocí instance této třídy je volána metoda `uint32_t getNum()`, která vrací a ukládá výslednou hodnotu čtených dat. V metodě `uint_32 getNum()`, je volána další metoda třídy `Data`, a to metoda `uint32_t ProcessData Master()`.
V této metodě probíhá for cyklus, ve kterém jsou volány další dvě metody. První metoda `void CheckPortAndIndex(int i)` slouží k rozklíčování portu a pinu aktuálního datového bitu. Funkcionalita metody `void CheckPortAndIndex(int i)` je zobrazena na obrázku 10.

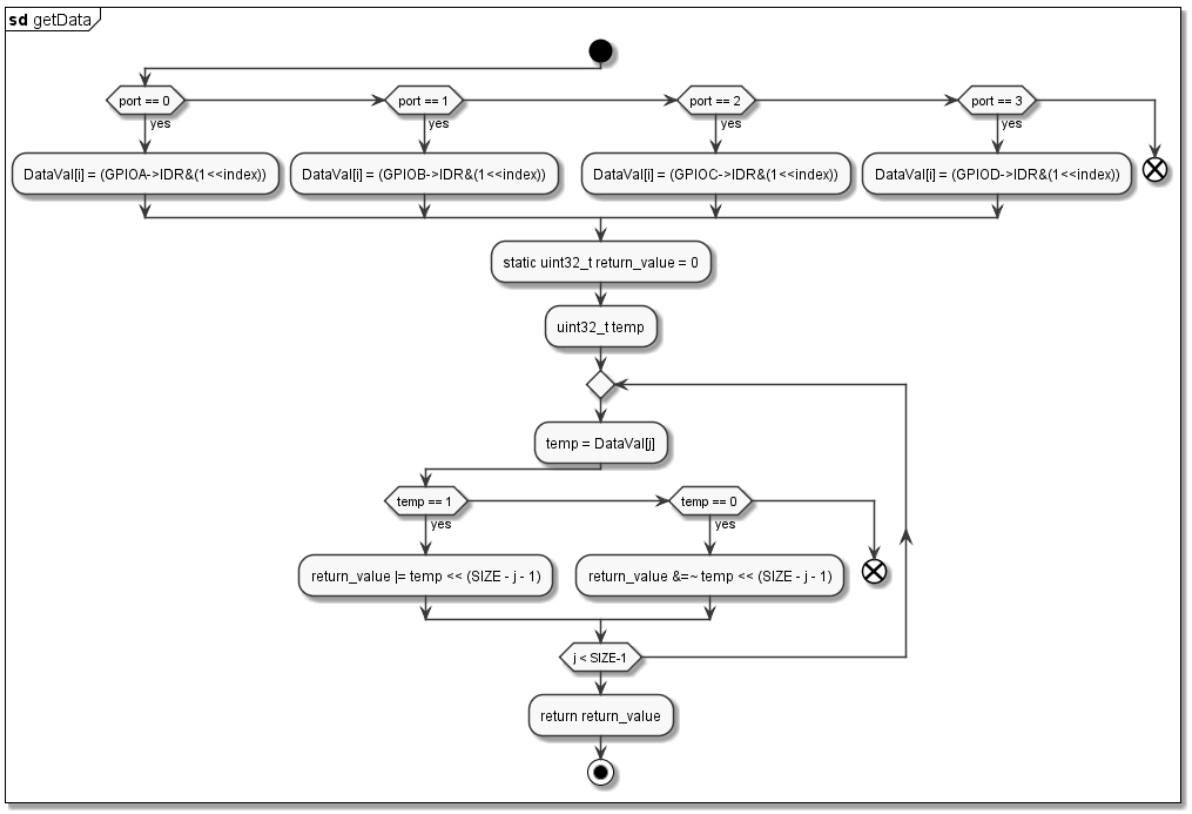
[22] Častěji pojem „Lookup table.“



Obrázek 10: Diagram popisující funkcionalitu metody `CheckPortAndIndex`

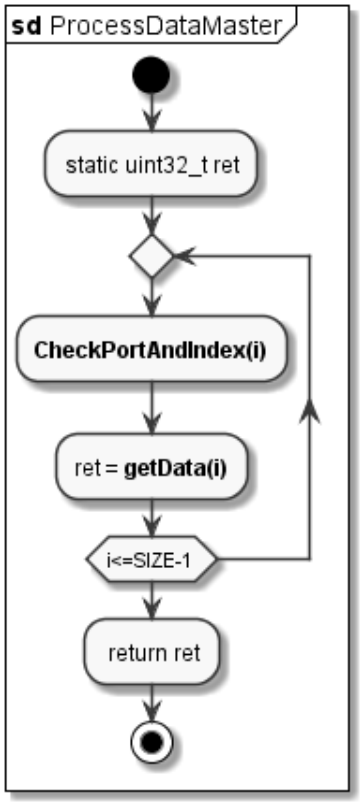
Na obrázku 10 nejprve dochází k instrukci bitového posunu a přiřazení dané hodnoty do proměnné `port`, tato proměnná pak prochází příkazem switch, kde se ověří na jakém portu a indexu se daný datový signál nachází.

Po vykonání této metody pokračuje for cyklus metody `uint32_t ProcessDataMaster()` do další metody třídy `Data`, nesoucí název `uint32_t getData(int i)`.
Tělo této metody pracuje s polem `bool DataVal[SIZE]`, do kterého ukládá aktuálně čtené hodnoty ze SIPO registrů. Diagram metody je zobrazen na obrázku 11.



Obrázek 11: Diagram popisující funkcionalitu metody `getData`

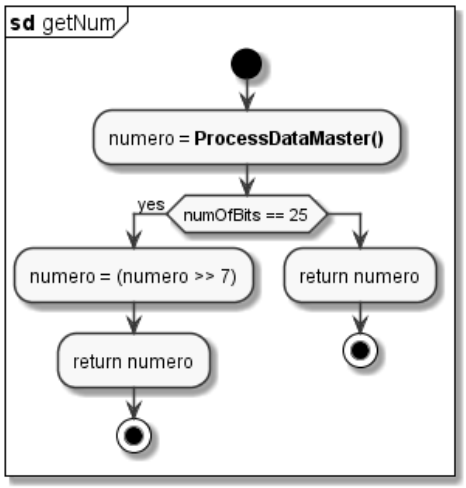
V metodě `uint32_t getData(int i)`, je nejprve porovnávána brána, získaná v předchozí metodě. Dále je proveden výčet z daného pinu a uložení do pole `bool DataVal[SIZE]`. Po uložení následuje for cyklus, ve kterém probíhá maskování bitů statické proměnné `static uint32_t return_value`, do které se v průběhu čtení ze všech pinů ukládá výsledná hodnota dat.
Po dokončení for cyklu v metodě `uint32_t ProcesDataMaster()`, metoda vrací výslednou hodnotu dat, která jsou získávána ze senzoru. Pro názornost byl vytvořen diagram 12, jenž metodu popisuje.



Obrázek 12: Diagram popisující funkcionalitu metody `ProcessDataMaster`

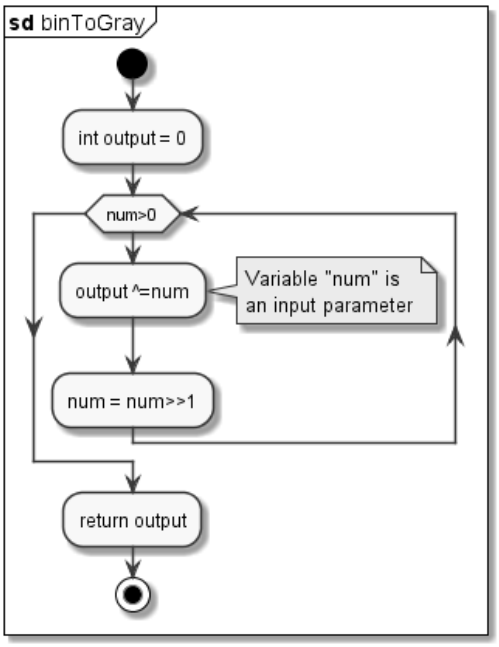
Diagram 12 popisuje funkcionalitu metody `uint32_t ProcessDataMaster()`, která je volána z přerušovací rutiny. Tato metoda pak pracuje s metodami uvedenými na obrázcích 10 a 11. V tomto diagramu jsou dvě zmíněné metody zobrazeny pouze jako bloky, které vykonávají přesně to, co je uvedeno na příslušných diagramech těchto dvou metod. Metoda `ProcessDataMaster()` pak vrací 32 bitové číslo.
Získané číslo je uložené do proměnné třídy `Data` s názvem `uint32_t numero`. Dále je v metodě `getNum()` porovnáváno, jaký počet bitů má zpráva obsahovat[23]. Pro případ 32-bitové zprávy, metoda vrací 32-bitové číslo získané metodu `ProcessDataMaster()`, pokud ale zařízení pracuje se zprávou, která má 25 bitů, je získané číslo `numero` binárně posunuto a zbývající počet bitů[24]. Tato logika je opět ukázána na diagramu níže.

[23] Tuto informaci zadává uživatel.
[24] Pro zprávu o délce 25 bitů se musí posouvat o 7 bitů.



Obrázek 13: Diagram popisující funkcionalitu metody `getNum`

Po uložení vyčtených dat je možné je zobrazovat. Samotný princip zobrazování dat pomocí uživatelského rozhraní bude popsán později, v kapitole jež je na toto téma zaměřena. Nyní bude popsáno, jak je k těmto datům přistupováno v hlavní smyčce ve funkci `main.cpp`. K uloženým datům přistupujeme pomocí metody třídy `Data`, `uint32_t getNumero() const`. Jelikož je proměnná `numero` privátní proměnnou, musíme k ní přistupovat pomocí tzv. getteru, kterým je právě zmíněná metoda.
K datům je přistupováno v hlavní funkci main. Ve funkci `main.cpp` se s daty dále pracuje, například probíhá jejich dekódování z binárního do Grayova kódu[25]. O převedení čísla z binárního do Grayova kódu se stará funkce `uint64_t binToGray(uint64_t num)`. Tato funkce pracuje s bitovým operátorem *XOR*. Princip funkce je zobrazen v diagramu na obrázku 14.



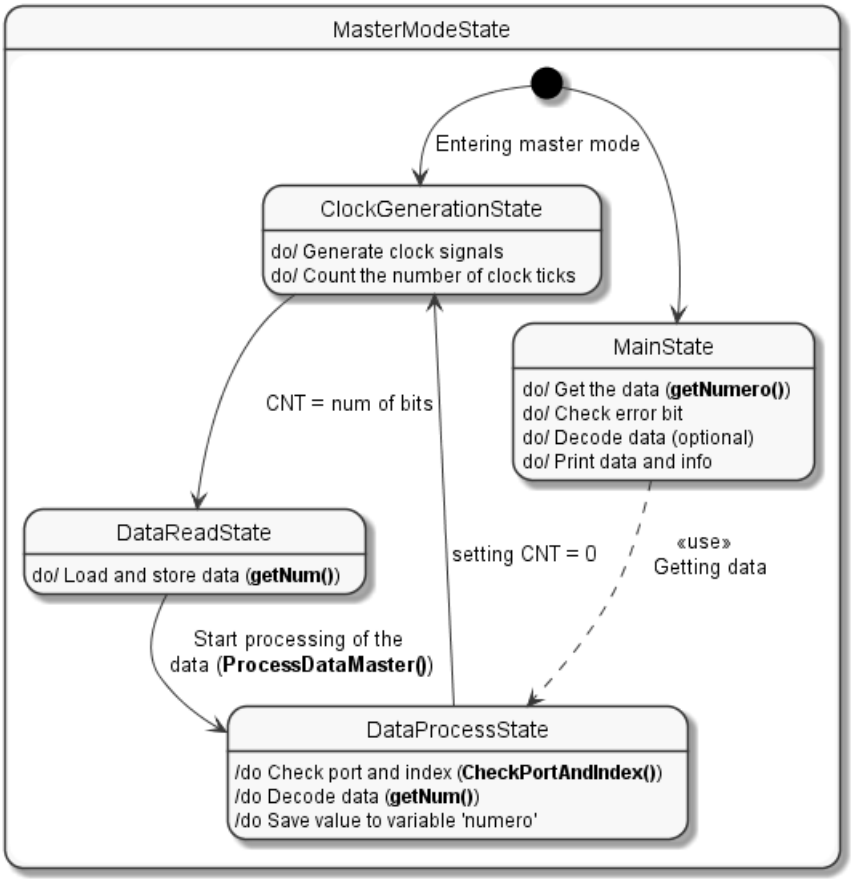
Obrázek 14: Diagram popisující funkcionalitu funkce převodu z binárního na Grayův kód

[25]Pokud je tak zadáno uživatelem.

V hlavní funkci `main.cpp` taktéž probíhá kontrola tzv. error bitu. Pokud na sběrnici dochází k chybě, například když senzor nevidí čárový kód, vygeneruje chybovou zprávu, která je reprezentována již zmíněným error bitem. Error bit je poslední bit datového rámce přenášeného po sběrnici SSI. Pokud se senzor nachází v normálním stavu, kdy nedochází k žádné chybě, senzor odesílá data a poslední error bit je v logické nule. Naopak, pokud chyba nastane, datové rámce, jež jsou odesílány senzorem, jsou reprezentovány pouze error bitem, který je v případě poruchy v logické jedničce.
Detekce tohoto error bitu je provedena použitím bitového operátoru *AND*. Obecně lze princip popsat takto.

$$Data\ ze\ senzoru\ \&\ 0x01 = \begin{cases} 1, & \text{Error} \\ 0, & \text{OK} \end{cases} \tag{3}$$

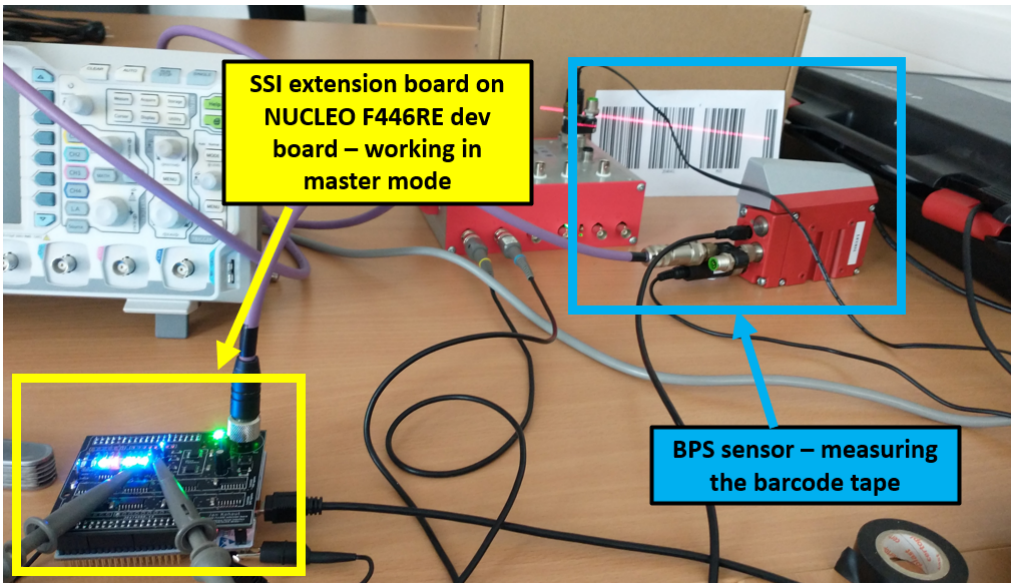
Pokud se error bit ve zprávě vyskytuje, je o tom uživatel informován pomocí uživatelského rozhraní.
Následuje stavový diagram, který se zobrazuje již finální funkcionalitu softwaru pro režim master. Tento diagram přímo navazuje na diagram 9, kde se po inicializaci režimu master vstupuje do stavu *MasterModeState*. Taktéž jsou v tomto diagramu zobrazeny metody, které jsou detailněji vysvětleny na obrázcích 10, 12 a 11. Tyto metody jsou v diagramu vyznačeny tučně.



Obrázek 15: Stavový diagram popisující stav master a jeho pod-stavy

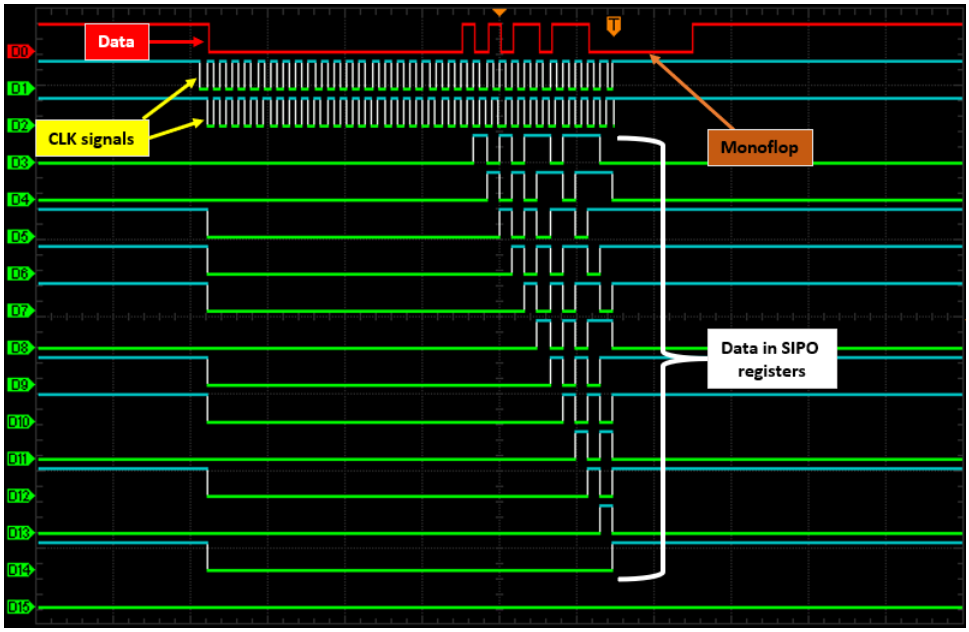
Algoritmus byl testován pomocí osciloskopu, kdy byla naměřená data porovnána s daty uloženými v registrech. Další porovnání probíhalo pomocí uživatelského rozhraní. Na následujícím obrázku

16 je zobrazeno propojení vyvíjeného zařízení se senzorem BPS, který čte data z pásky s čárovým kódem.



Obrázek 16: Zapojení a testování funkčnosti vyvíjeného zařízení

Následuje obrázek 17, který zobrazuje výstup z osciloskopu při práci zařízení v režimu master. Měření bylo prováděno pomocí logického analyzátoru, kdy byla měřena data získaná ze senzoru, hodinové signály generované vyvíjeným master zařízením a piny SIPO registrů, díky jejichž hodnotám, bylo následně možné jednotlivá data analyzovat a porovnat jejich správnost s daty, uvedenými na pásce s čárovým kódem.

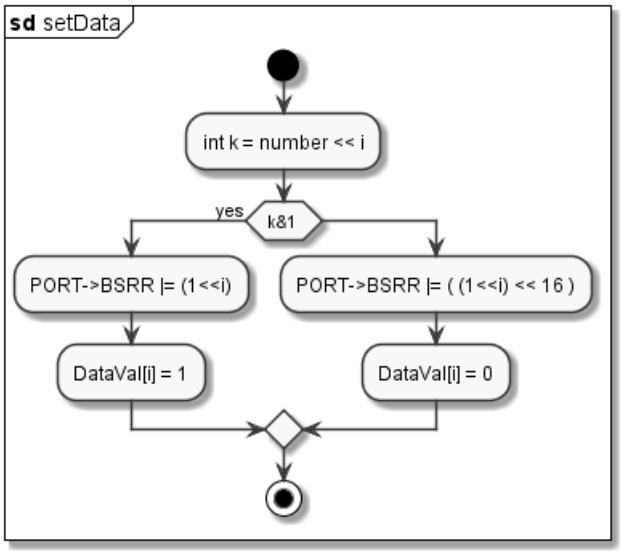


Obrázek 17: Výstup osciloskopu při práci zařízení v režimu master

Na obrázku 17 je zachycen pouze jeden datový rámec. Výstup s více datovými rámci je pro zajímavost uveden na obrázku 24 v příloze C.

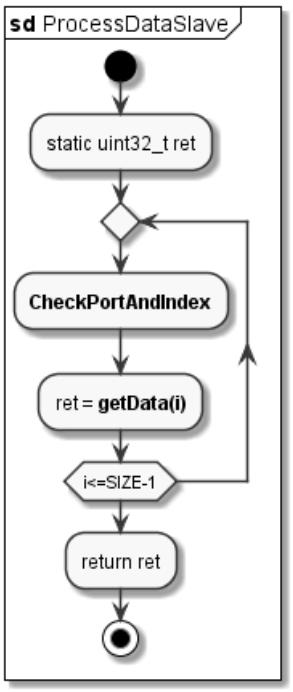
4.3.3.2 Režim slave

Dle popisu v předchozích částech této práce, slave zařízení reaguje na příchozí hodinové signály od master zařízení tím, že vrací data.
Pro inicializaci, je stejně jako v případě master zařízení využívána stejná virtuální funkce, tentokrát ale pro třídu `Slave`. Základní inicializaci a nastavení tedy provádí metoda `void Slave ::ModeSet(SsiGpio& gpio)`, ve které se provede základní konfigurace řídících GPIO pinů. Dále je zde nakonfigurován časovač do režimu *input capture*, aby byl schopen přijímat hodinové signály generované master zařízením. Jako poslední krok, jsou inicializovány datové GPIO piny do režimu výstupu.
Významnou roli zde opět hraje funkce pro obsluhu přerušení, pro režim slave je to funkce `void TIM3_IRQHandler(void)`.
Během přerušení se v této funkci paralelně načítají data z GPIO, která jsou generována řídícím mikroprocesorem. Tato data je třeba načíst do příslušných posuvných registrů, toho je docíleno přepnutím řídícího signálu `SH_nLD` do logické nuly, čímž se PISO registry přepnou do stavu „load“ a jednotlivá data si načtou. Následuje přepnutí zpět do režimu „shift“ tím, že je na řídící pin `SH_nLD` odeslána zpět logická jednička. Data jsou tedy načtena v posuvných registrech. Dále jsou data sériově vysouvána v reakci na příchozí hodinové signály. Pro správnou funkcionalitu sériového vysouvání dat z posuvných registrů je nutné, nastavit řídící signál `nCLK_ENB` do logické nuly.
Pro zapisování dat do PISO registrů je opět využívána třída `Data`, ve které je využívána metoda `void ProcessDataSlave(uint32_t number)`. V této metodě se pak volají další metody třídy `Data`. Jednou z metod je již zmíněná metoda `void CheckPortAndIndex`, která pracuje s „lookup“ tabulkou bran a indexů pinů pro jednotlivá data. Více je ukázáno na obrázku 10.
Následuje metoda `setData(uint32_t number, int i)`. Jedním ze vstupů této metody je číslo, které chceme zapsat na datové piny. Požadované číslo, které vstupuje do této metody, převáděno z dekadického na binární a binární hodnoty požadovaného čísla jsou následně zapsány na příslušné datové piny. Ukázka funkcionality metody `setData()` je zobrazena na obrázku 18.



Obrázek 18: Diagram zobrazující funkcionalitu metody `setData`

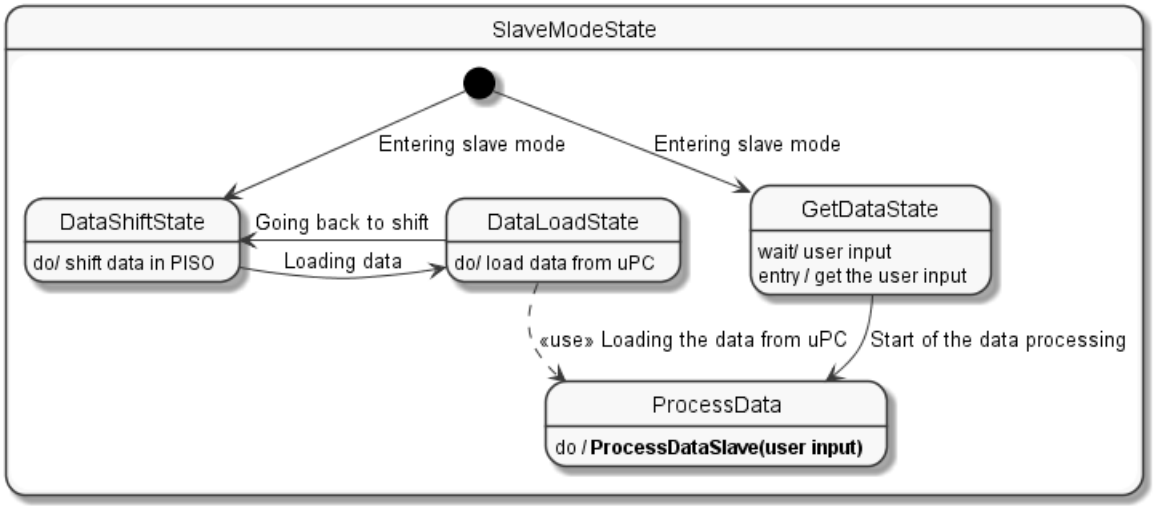
Další diagram, který je zobrazen na obrázku 19 níže, popisuje celkovou funkcionalitu metody `void ProcessSlaveData(uint32_t number)`. Tato metoda pracuje ve for cyklu[26], ve kterém jsou postupně vybírány dané datové piny a jejich příslušné brány, pomocí metody na obrázku 10. Dále pak následuje metoda `setData()`, pomocí které jsou datové piny nastavovány do příslušných logických úrovní. V diagramu na obrázku 19 jsou dané metody vyznačeny tučně.



Obrázek 19: Diagram zobrazující funkcionalitu metody `ProcessDataSlave`

Pokud je tedy uživatelem vybrán režim provozu slave, v hlavní funkci `main.cpp` dochází k tomu, že uživatel zadává požadované číslo (pomocí uživatelského rozhraní), které chce po sběrnici odeslat. Dále je třeba, aby uživatel zadal i počet bitů zprávy, což samozřejmě musí korespondovat s počtem tiků hodinového signálu.
Následuje volání metody `void Data::ProcessDataSlave(uint32_t number)`, ve které se data uloží na výstupní piny mikroprocesoru.
Při příchodu externího hodinového signálu, jsou nejprve uvedeny PISO registry do režimu „load“, ve kterém jsou hodnoty z výstupních pinů mikroprocesoru načteny do PISO registrů. Ihned poté se režim PISO registrů opět mění do režimu „shift“ a data jsou pak sériově z PISO registrů odváděna v reakci na příchozí hodinový signál.

[26] Délka for cyklu odpovídá počtu bitů zprávy.

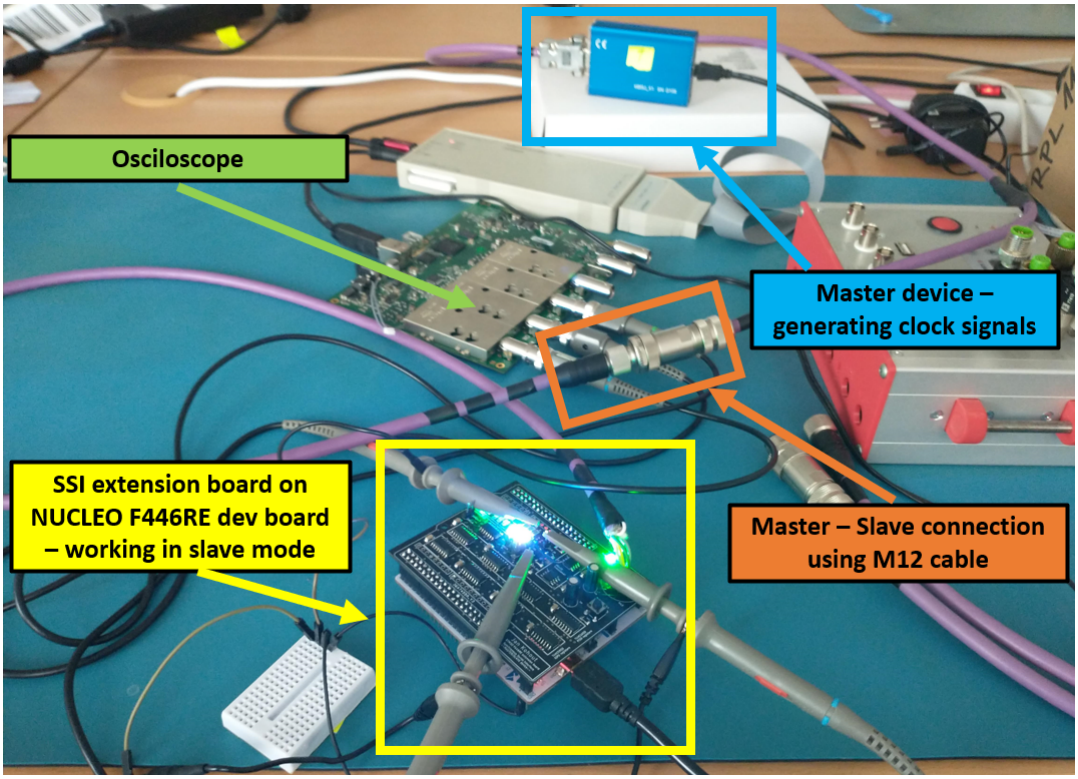


Obrázek 20: Stavový diagram zobrazující stav slave a jeho pod-stavy

Na obrázku 20 je zobrazen stav zařízení v režimu slave. Obrázek opět navazuje na stavový diagram 9.

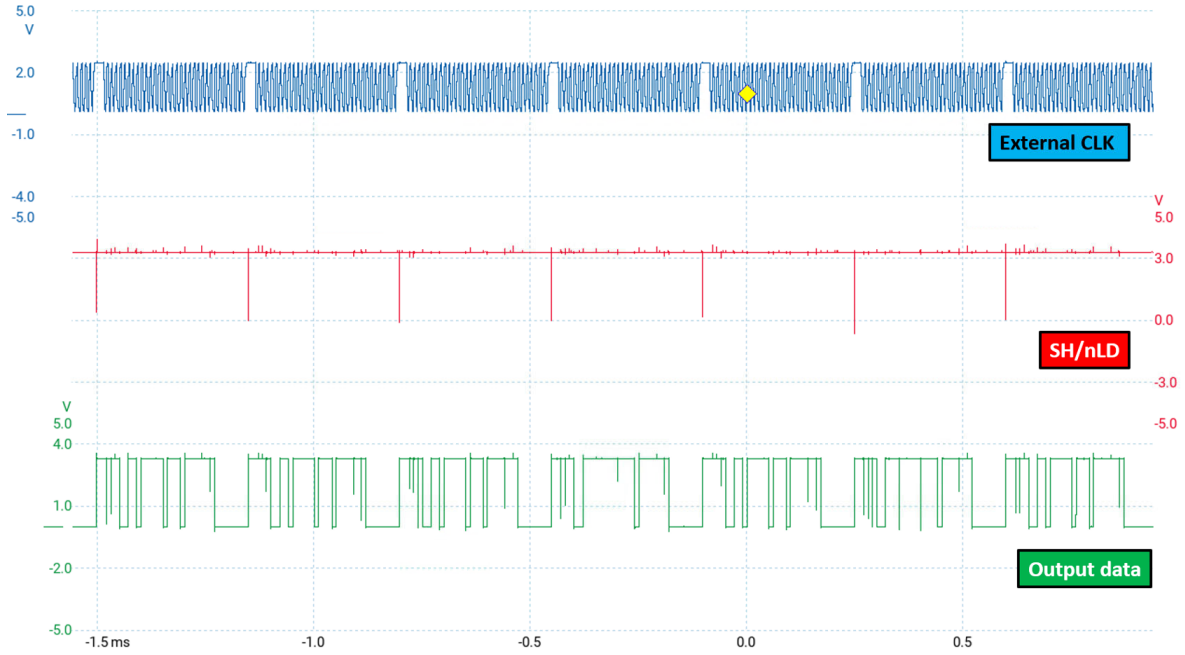
Zde je nutno poznamenat, že během vypracovávání této práce nebyla, z důvodu časové náročnosti, zcela realizována funkcionalita výběru odesílané hodnoty. Byla vytvořena pouze verze, která odesílá stále stejnou sekvenci dat. Tato schopnost zařízení, je však pro testovací účely dostačující, ale bylo by vhodné ji v budoucnu rozšířit.

Na následujícím obrázku 21 je zobrazena fotografie pracoviště s jednotlivými komponentami v průběhu testování vyvíjeného zařízení.



Obrázek 21: Zapojení a testování funkčnosti vyvíjeného zařízení v režimu slave

Dále je na obrázku 22 je zobrazen výstup z osciloskopu při testování základní zařízení v režimu slave, kdy byla mikprocesorem generována data v reakci na externí hodinový signál.



Obrázek 22: Výstup osciloskopu při práci zařízení v režimu slave

Je vidět, že zařízení v režimu slave reaguje na externí hodinový signál tím, že načte požadovaná data do posuvných registrů, ihned poté je do posuvných registrů vyslán signál pro návrat do režimu *shift*, ve kterém jsou data sériově vysouvána ven na sběrnici SSI a doručována do master zařízení. Více informací ohledně zpracování dat je uvedeno v další kapitole.

4.3.4 Realizace uživatelského rozhraní

Uživatelské rozhraní je realizováno pomocí periférie UART. Vytvořené uživatelské rozhraní musí uživateli umožnit komunikovat se zařízením. Tatkéž musí být interaktivní a umožnit uživateli zadávat požadované parametry. Mezi ty nejzákladnější parametry patří frekvence, počet bitů, rozlišení polohy a samozřejmě výběr, v jakém režimu má zařízení pracovat.
Pro umožnění komunikace mezi uživatelem a zařízením je využívána aplikace PuTTY. Nyní následuje nastavení a konfigurace aplikace PuTTY pro správnou funkčnost vytvářeného uživatelského rozhraní.
Prvotní nastavení probíhá v sekci Session, kde se nastavují nejzákladnější parametry komunikace. Pro komunikaci je používána modulační rychlost UARTu 115200 Baudů, typ připojení je sériový. Nastavení portu je závislé na použitém PC, tudíž zde není příslušný port uveden. Nastavení ukládání průběhu celé komunikace probíhá v podsekci Session » Logging, kde si uživatel může vybrat způsob zápisu a uložení souboru s informacemi o komunikaci. Přesnější nastavení komunikace je pak v sekci Connection » Serial, kde je základní nastavení UARTu upřesněno, viz tabulka 4 níže.

Serial data connect to	COM X
Speed (baud)	115200
Data bits	8
Parity bit	None
Flow control	None

Tabulka 4: Tabulka s hodnotami pro konfiguraci sériové komunikace v aplikaci PuTTY

Dalším důležitým bodem je nastavení možnosti řízení pomocí příkazů z klávesnice, což je nastaveno v Terminal » keyboard » The Function keys and keypad na ESC[.
Uživatelské rozhraní se zobrazuje v okně o šířce 80 na 24. Toto je nutné nastavit v sekci Window. Další nastavení aplikaci PuTTY, může být ponecháno ve výchozím stavu.

Základní koncept tvorby uživatelského rozhraní

Koncept uživatelského rozhraní je založen na komunikaci se zařízením pomocí „terminálu“, který je realizován na principu kontrolních kódů VT100[27]. Tyto kódy jsou implementovány přímo ve zdrojovém kódu zařízení. Ukázka některých příkazů, jež jsou v práci využívány, je níže.

```
1 /*Basics VT100 commands*/
2 PRINT("\x1B[2J");  //erase entire screen (cursor doesn't move)
3 PRINT("\x1B[24A"); //set cursor to y=0
4 PRINT("\x1B[80D"); //set cursor to x=0
5 PRINT("\x1B[?5h"); //white background
6 PRINT("x1B[?5l");  //black backgroud
7 PRINT("\x1B[41m"); //text color - red
8 PRINT("\x1B[40m"); //text color - black
```

[27] Tento koncept byl představen již v roce 1978 firmou *Digital Equipment Corporation (DEC)* [33]. Tato technologie byla hojně využívána pro ovládání terminálu. Jelikož se jedná o celkem jednoduchý koncept, byl implementován i do této práce.

O základní funkci uživatelského rozhraní se stará třída `Print`, která obsahuje několik metod, které zajišťují správnou funkci uživatelského rozhraní.
Po vytvoření instance třídy ve funkci `main`, je volán konstruktor této třídy, který neobsahuje žádné parametry. Konstruktor vyčistí obrazovku od případných předchozích textů. Dále je volána metoda `void PrintStart()`, které čeká na reakci uživatele, který musí pro pokračování stisknout klávesu 1. Samozřejmě je zde ošetřena výjimka, pokud uživatel stiskne jinou klávesu, uživatelské rozhraní ho upozorní. Dále jsou ve funkci `main` volány metody sloužící k nastavení jednotlivých parametrů komunikace, jako je počet bitů, frekvence, rozlišení a kódování zprávy (metody `void SelectBit(), void SelectDec(), void SelectFreq()`, atp.[28]). Uživatel vždy dostane na výběr z několika možností, kdy výběr potvrdí stiskem příslušné klávesy.
Před samotným spuštěním komunikace na sběrnici SSI, je uživatel dotázán na výběr příslušného módu. Ten opět potvrzuje stiskem příslušné klávesy. Nutno podotknout, že v průběhu práce v uživatelském rozhraní může uživatel vždy svoji volbu změnit restartem mikrokontroléru, který je vyvolán stisknutím klávesy 3. Stiskem tohoto tlačítka je vyvoláno přerušení, ve kterém je zavolána funkce `HAL_NVIC_SystemReset()`.
Po přechodu do módů master nebo slave, jsou uživateli zobrazeny potřebné informace[29]. Tyto informace se během běhu programu mění a je nutné je tedy přepisovat. Pro co nejrychlejší přepisování je taktéž nutné zabránit zbytečnému přepisování informací, které se nemění. O toto se starají následující metody třídy `Print`[30].

`void Print_xy_zero()`, tato metoda pracuje s kurzorem, který vypisuje dané informace. Pomocí této metody je přesunut kurzor terminálu do levého horního rohu obrazovky, čehož je samozřejmě opět dosaženo pomocí VT100 příkazů.

`void Print_xy(const char x [], const char y [])`, tato metoda přesouvá kurzor přímo na pozici, která je určena vstupními parametry této metody. Metoda tatkéž využívá VT100 příkazy, které jsou však v této metodě rozdělené tak, aby do nich byly zahrnuty parametry této metody.

Dále jsou v této třídě vytvořeny metody, které jsou určeny pro práci s celými čísly. Metody jsou využívány v režimu master, kdy zobrazují hodnotu příchozího čísla po sběrnici. Tyto metody jsou tři, každá pracuje s jinou přesností. Důležité je poznamenat, že senzor odesílá hodnoty v pevné řádové čárce, následující metody jsou podle toho napsány[31].

`void PrintInt(unsigned int Integer)`, pracuje s rozlišením na celá čísla.

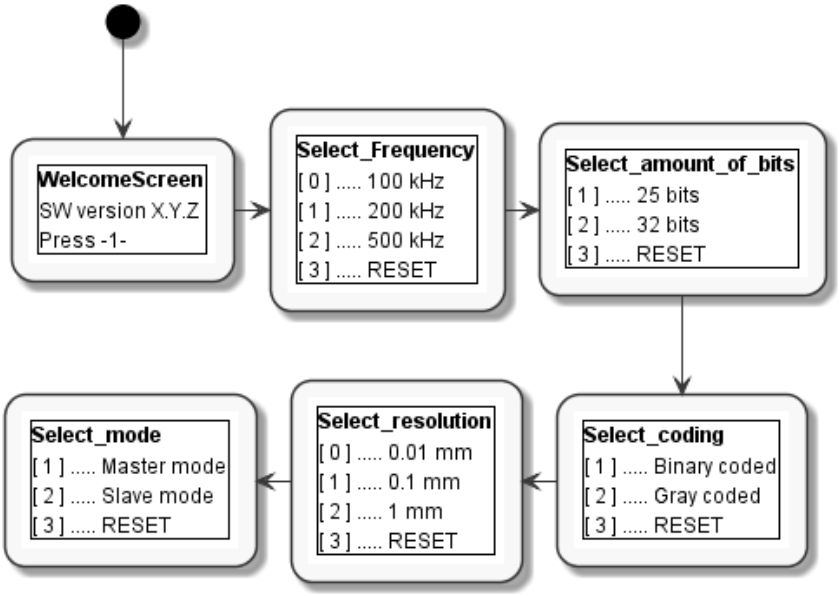
`void PrintIntOne(unsigned int Integer)`, pracuje s rozlišením na jedno desetinné místo.

`void PrintIntTwo(unsigned int Integer)`, pracuje s rozlišením na dvě desetinná místa.

Dále je v této třídě několik metod zapouzdření[32], jež pracují s privátními proměnnými třídy. Tyto

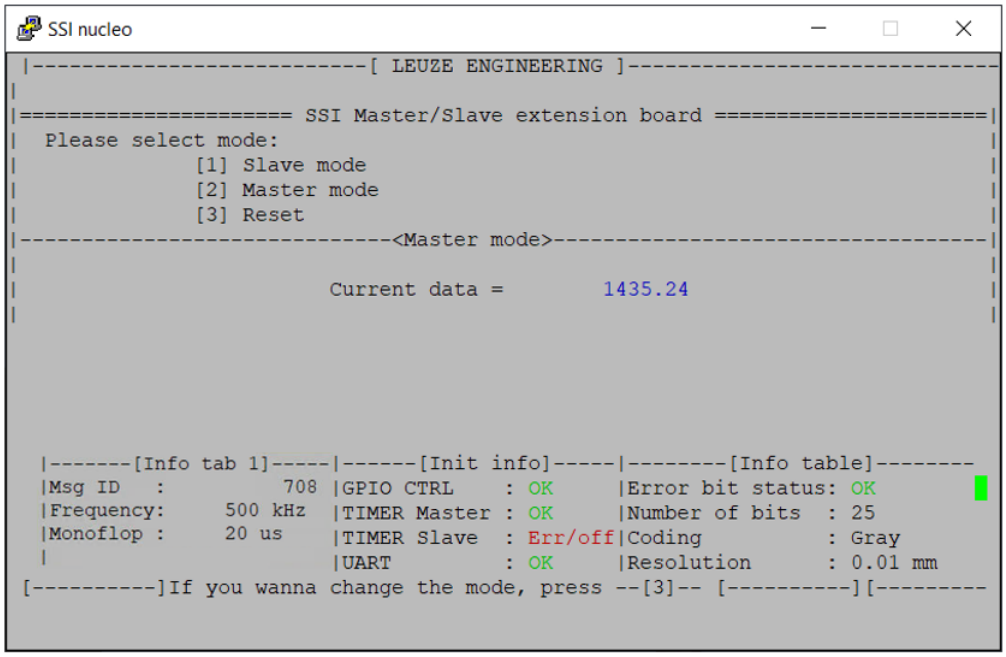
[28] Tato třída obsahuje velké množství metod, ale pro základní přehled není potřeba, aby zde byly všechny uvedeny.
[29] O tom, o jaké informace se jedná, bude pojednávat pozdější část této podkapitoly.
[30] Metody jsou opět uváděny bez kontextu.
[31] Může se zdát jako zrůdnost, pracovat s proměnnými typu `int` v desetinných místech. Jelikož ale senzor posílá hodnotu v pevné řádové čárce, stačí pouze přidat desetinnou tečku/čárku do výpisu funkce. Vypisované hodnoty jsou tedy opravdu typu `int` nikoli typu `float` nebo `double`.
[32] Častěji se lze setkat s anglickým pojmem „encapsulation“, jedná se tedy o tzv. „gettery“ a „settery“.

proměnné jsou převážně informačního charakteru, nebo se jedná o proměnné spojené s volbou uživatele, například proměnné nesoucí informaci o vybrané frekvenci, počtu bitů, kódování atp. Tyto proměnné pak samozřejmě pracují s výše zmíněnými metodami.
Jednou z velmi důležitých metod v této třídě je metoda `void PrintWrongInput()`, která vždy uživatele upozorní na špatně zadanou vstupní hodnotu.
Na obrázku 23 je zobrazen diagram postupu uživatele v uživatelském rozhraní.



Obrázek 23: Diagram zobrazující postup uživatele v uživatelském rozhraní

Po výběru módu se uživateli zobrazí příslušná obrazovka. To, co se uživateli reálně v uživatelském rozhraní zobrazuje je uvedeno na obrázcích v příloze. Na obrázku 24 je ukázka obrazovky v režimu master.



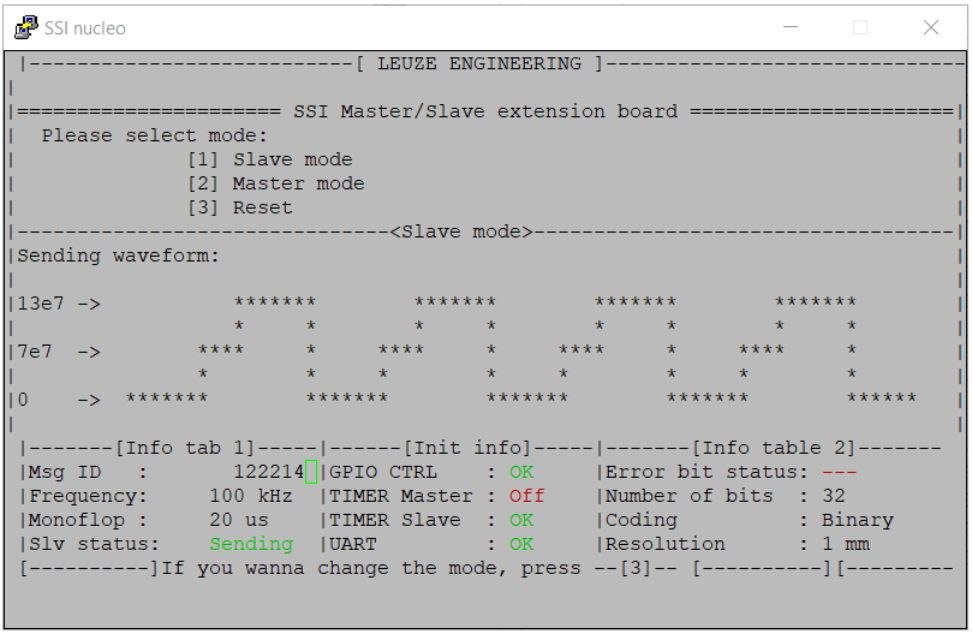
Obrázek 24: Uživatelské rozhraní v režimu master

V prostřední části obrazovky jsou zobrazena právě snímaná data senzorem BPS. S tím koresponduje i informace `Error bit status` v tabulce v pravé části obrazovky. Pokud jsou data bez problémů čtena senzorem, senzor je odesílá do master zařízení. Pokud ale senzor žádná data nevidí, například kvůli překrytí senzoru, nebo poničené pásce s čárovými kódy, senzor odesílá error bit, který je následně zařízením pracujícím v režimu master rozpoznán a uživatelské rozhraní pak uživatele na chybový stav upozorní.
V informačních tabulkách jsou pak dále uvedeny všechny důležité informace týkající se konfigurace zařízení. V levé části je navíc zobrazena informace o ID zpráv. Tato informace je dále využívána pro ukládání dat do .csv souboru, který lze využít pro pozdější zpracování naměřených dat. Data jsou ve výstupním .csv souboru uložena způsobem, naznačeným v tabulce 5.

Error bit	Msg ID	Data
OK	116	1435
OK	117	1435
Empty bus	118	0
Empty bus	119	0
Empty bus	120	0
Empty bus	121	0
OK	122	1435
OK	123	1435
⋮	⋮	⋮

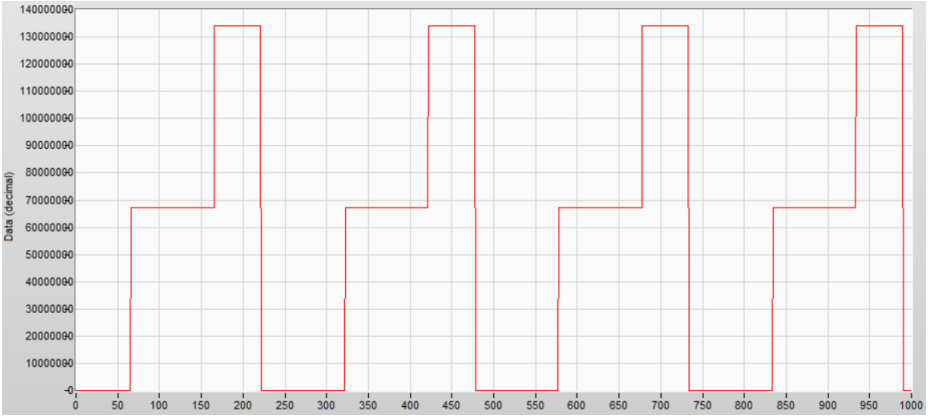
Tabulka 5: Tabulka dat ve výstupním .csv souboru

Uvedená tabulka obsahuje pouze krátký úsek z výsledného .csv souboru. Pro ukázku byl vybrán úsek, kdy byla senzoru, na velice krátkou chvíli, zakryta páska s čárovým kódem, je tedy zachycena i chybová hláška, která se uživateli zobrazí na obrazovce uživatelského rozhraní.
Následuje ukázka obrazovky v režimu slave. Ta je zobrazena na obrázku 25.



Obrázek 25: Uživatelské rozhraní v režimu slave

Ve středu obrazovky v režimu slave, je naznačen průběh, který je generován a odesílán do master zařízení. Pro kontrolu funkčnosti byla tato data vyčtena master zařízením[33], pomocí aplikace BiSS GUI[34]. Data, která přichází do master zařízení jsou zobrazena na obrázku níže 26.



Obrázek 26: Přijatá data generovaná vyvíjeným zařízením v režimu slave

Je vidět, že přijatá data master zařízením, korespondují s průběhem generovaným emulátorem, který pracuje v režimu slave.
Dále byla do výchozího zobrazení režimu slave přidána řádka upozorňující na to, zda-li do zařízení přichází hodinové signály[35]. Jak již v této práci bylo zmíněno, více obrázků z uživatelského rozhraní, i s různými stavy, je zobrazeno v příloze C na obrázcích 25, 26, 27, 28, 29 a 30.
V příloze se taktéž nachází kompletní dokumentace kódu.

[33] Externí master zařízení MB5U.
[34] Více informací o používaném master zařízení a taktéž o aplikaci pro tato zařízení, předchozí části této práce, která se zabývá analýzou dodaného zařízení.
[35] Levý dolní roh na obrázku 25.

5 Závěr

Předkládaná diplomová práce se zabývá synchronní sériovou komunikační sběrnicí. V první části této práce je popsána základní funkcionalita této sběrnice a informace o jejím použitím v průmyslu pro lineární senzory polohy, nebo jako komunikace s absolutními rotačními čidly ARC, při získávání informace o natočení hřídele motorů. Taktéž byly vytvořeny jednoduché simulace pomocí programovacího jazyka Python, které funkci této sběrnice popisují. Zdrojové kódy k těmto simulacím jsou dostupné v přílohách této práce.

Ve druhé části této práce byla pozornost věnována návaznosti na předchozí práci [1]. Dále je v této části popsána základní hardwarová výbava zařízení a stručný úvod do softwaru, se kterým zařízení pracuje. Následovala analýza chování dodaného zařízení, prvotní konfigurace, spuštění a testování. V další části této kapitoly je popsána konfigurace jednotlivých zařízení, která na sběrnici pracují, tedy konfigurace senzorů a zařízení typu master. Testování v této části práce bylo prováděno pro jeden i dva aktivní kanály sběrnice. Pro testování byly použity dva typy senzorů, prvním byl senzor BPS307i, a druhým senzor FBPS 607i. Základní informace o těchto senzorech jsou stručně uvedeny v první části práce. Po prostudování této kapitoly, by měl být uživatel schopen zprovoznit a nakonfigurovat dodané zařízení s dodanou verzí softwaru *T.0.8.0*, provést základní konfiguraci zařízení BiSS master MB5U a zmíněných senzorů, a také pracovat s dodaným zařízením v provozních režimech pro jeden nebo dva aktivní kanály synchronní sériové sběrnice.

Třetí část práce pojednává o dostupných verzích softwaru a jejich funkcionalitě. Zde je nutno poznamenat, že nebyl k dispozici zdrojový kód softwaru, který je ve výchozím stavu nahrán do analyzovaného zařízení. Během práce je proto využívána verze *U.0.8.0*, která poskytuje alespoň částečnou funkcionalitu, jako verze *T.0.8.0*.

Následuje popis kompilace dané verze a její nahrání do zařízení pomocí Lauterbach debuggeru, který je připojen k zařízení pomocí JTAGu. Je zde popsána základní konfigurace aplikace TRACE32, která je Lauterbach debuggerem používána a funkční zapojení JTAGu pro jeho správnou funkci. Dále je v této práci popsána analýza zdrojového kódu verze *U.0.8.0*. Tato práce popisuje pouze malou část z celkového kódu, kdy pozornost byla kladena převážně na metody a funkcionality, které bude možné použít v budoucím rozšíření. Pro nejdůležitější funkce, metody a třídy, byly vytvořeny diagramy, které zmíněné části kódu vizuálně popisují.

Důležitým výstupem z analýzy je skutečnost, že zohlednění synchronizace pomocí časových razítek bylo již v zařízení implementováno. Vytvářená časová razítka pracují s přesností desítek nanosekund, kdy přesnost byla ověřena měřením pomocí osciloskopu. Tato skutečnost umožňuje měřit několik vzorků na sběrnici s relativně velkou přesností. Zde je vhodné zmínit, že přesnost časových razítek byla konzultována s kolegy, kteří v budoucnu budou toto zařízení využívat, podle nichž je pro zařazení zařízení do testovacího procesu přesnost časových razítek dostatečná.

Následně byly v této části vytvořeny návrhy na budoucí rozšíření. Prvním návrhem je vytvoření synchronizace hodinových signálů pomocí jednoho master zařízení. Tento návrh bohužel nebyl prakticky realizován, důvodem byl nedostatek hardwarových komponent a taktéž jejich nedostupnost u externího dodavatele. Další návrhy na rozšíření se týkají převážně ukládání získaných dat, které je v tuto chvíli omezeno jen na několik málo vzorků a výsledná komunikace s uživatelem, který by měl být schopen vzdáleně komunikovat se zařízením například za pomoci UDP soketů a analyzovat dodaná data i s příslušnými časovými razítky. Detailnější informace jsou uvedeny ve třetí části této práce.

Poslední, čtvrtá část práce se zabývá návrhem, realizací a zprovozněním univerzálního zařízení, které pracuje na synchronní sériové sběrnici buď jako master zařízení, nebo jako zařízení slave. Nejprve jsou zde definovány požadavky na dané zařízení, následuje návrh základního principu, dle kterého byly vybrány vhodné součástky, jejichž detailnější popis je v této části taktéž uveden. Z hlediska ověření konceptu byla vytvořena zjednodušená osmibitová verze na nepájivém kontaktním poli. V návaznosti na tento prototyp bylo vytvořeno schéma zapojení a návrh desky plošných spojů, která byla posléze vyrobena externím dodavatelem. Po dodání desky plošných spojů a součástek bylo provedeno ruční osazení desky jednotlivými součástkami, prvotní oživení desky a základní testování. Výsledné schéma desky je uvedeno v příloze na obrázku 15. V příloze je taktéž uveden soubor s použitými součástkami a obrázky obou vrstev desky plošných spojů 19, 20.

Realizace softwaru probíhala v jazyce C++, kdy nejprve byla vytvořena hardwarová abstraktní vrstva, pro snazší manipulaci s periferiemi mikroprocesoru (GPIO, UART a časovače). Dále byl vytvořen algoritmus pro práci zařízení v režimu master, kdy zařízení generuje hodinové signály a přijímá data generovaná senzorem. Obdobně byl vytvořen algoritmus pro práci zařízení v režimu slave, kdy zařízení generuje data a odesílá je v reakci na externí hodinový signál generovaný master zařízením.

Pro komunikaci uživatele se zařízením bylo vytvořeno interaktivní uživatelské rozhraní, které umožňuje základní nastavení a konfiguraci vytvořeného zařízení pro práci na synchronní sériové sběrnici. Pomocí tohoto uživatelského rozhraní jsou pak uživateli zobrazována jednotlivá data, informace o právě používaném režimu a přehled uživatelem nastavených parametrů, se kterými zařízení na sběrnici pracují.

Realizované zařízení je možné použít pro testování elementární funkčnosti senzorů BPS a FPBS, kdy se zařízení chová jako master synchronní sériové sběrnice. Taktéž bude použito pro ověřování základní funkce jiných master zařízení. Samozřejmostí je použití tohoto zařízení pro testování funkčnosti a chování samotného SSI monitoru.

Kompletní dokumentace softwarové části projektu byla vygenerována pomocí nástroje Doxygen z dokumentačních komentářů. Tato dokumentace je dostupná jako příloha této práce. Úplný zdrojový kód je pak dostupný u autora, popřípadě vedoucího této práce.

Seznam použité literatury

[1] F. Heil, „Konzipierung und Realisierung eines Analysewerkzeugs für das Monitoring von Telegrammen der synchron-seriellen Schnittstelle SSI im Besonderen unter Aspekten der Funktionssicherheit für Positioniersysteme der Automatisierungstechnik - im Studiengang Technische Informatik der Fakultät Informationstechnik," 2017.

[2] *Synchron-Serialle Schnittstelle*. URL: https://de.wikipedia.org/wiki/Synchron-Serielle_Schnittstelle (cit. 02. 03. 2022).

[3] D. Rabijns, W. Van Moer a G. Vandersteen, „A full grown differential signal source," *IEEE - Microwave Measurements*, 2003.

[4] B. Ševčík, „High-Speed Serial Differential Signaling Links with Commercial Equalizer," *IEEE - 20th International Conference Radioelektronika 2010*, 2010.

[5] *Introduction to RS.422 serial data standard*. URL: https://www.electronics-notes.com/articles/connectivity/serial-data-communications/rs422-basics-tutorial.php1978 (cit. 21. 11. 2021).

[6] *Sensor/actuator cable SAC-5P M12*. URL: https://www.farnell.com/datasheets/2309206.pdf (cit. 23. 11. 2021).

[7] Z. Peroutka, *Přednášky z předmětu mikroprocesorové řízení pohonů 1*, 2020.

[8] *Implementation of SSI Master Interface aplication note*. URL: https://www.posital.com/media/posital_media/documents/AbsoluteEncoders_Context_Technology_SSI_AppNote.pdf (cit. 23. 03. 2022).

[9] *BPS307i katalogové listy*. URL: https://www.leuze.com/en/deutschland/produkte/messende_sensoren/bps_8_34_3/bps_3_i_4/selector.php?supplier_aid=50125677&grp_id=1392989065861&lang=eng (cit. 23. 03. 2022).

[10] *BPS 300i Barcode Positioning System*. URL: https://www.youtube.com/watch?v=pTH_ok8WgfM&t=47s&ab_channel=LeuzeUSA (cit. 23. 03. 2022).

[11] *FBPS607i katalogové listy*. URL: https://www.leuze.com/en/deutschland/produkte/selector.php?supplier_aid=50140954&grp_id=9137990127363821&lang=eng (cit. 24. 03. 2022).

[12] M. Mráz, *Funkční bezpečnost (Functional Safety), zvaná přednáška pro předmět „Principy návrhu aplikací pro elektrotechniku"*, 2021.

[13] *NetX500/100 katalogové listy*. URL: https://www.hilscher.com/fileadmin/cms_upload/en-US/Resources/pdf/netX_100_500_Technical_Data_Reference_Guide_TRG_18_EN.pdf (cit. 24. 03. 2022).

[14] „PicoScope® 3000 Series," URL: https://www.picotech.com/oscilloscope/3000/picoscope-3000-oscilloscope-specifications (cit. 04. 05. 2022).

[15] R. Singh, „OPC UA Lesson 1- What is OPC UA?,“ URL: https://www.youtube.com/watch?v=3EREV8Q5PNU&ab_channel=RajvirSingh (cit. 04. 04. 2022).

[16] „OPC UA - Simulation server,“ URL: https://www.prosysopc.com/products/opc-ua-simulation-server/ (cit. 04. 04. 2022).

[17] „UaExpert—A Full-Featured OPC UA Client,“ URL: https://www.unified-automation.com/products/development-tools/uaexpert.html (cit. 04. 04. 2022).

[18] J. A. S. Michael Jesse Chonoles, „UML 2 For Dummies,“ 2003.

[19] „icHaus - BiSS Interface,“ URL: https://www.ichaus.de (cit. 04. 04. 2022).

[20] *Gradle User Manual*. URL: https://docs.gradle.org/current/userguide/userguide.html (cit. 19. 03. 2022).

[21] *Lauterbach debbuger*. URL: https://www.lauterbach.com/frames.html?home.html (cit. 20. 03. 2022).

[22] *STM446RE - katalogové listy*.

[23] *Posuvný registr 74HC165D - katalogové listy*.

[24] *Posuvný registr 74HC595N - katalogové listy*.

[25] *Převodník SN65176B - katalogové listy*.

[26] „Eagle Autodesk,“ URL: https://www.autodesk.com/products/eagle/overview?term=1-YEAR&tab=subscription (cit. 10. 04. 2022).

[27] V. Kůs, Z. Kubík, J. Skála a E. Müllerová, „Přednášky z předmětu elektromagnetická kompatibilita.,“ 2021.

[28] G. Deb, „Low EMI Desing of Microprocessor Based PCB - A Case Study,“ *998 IEEE EMC Symposium. International Symposium on Electromagnetic Compatibility*, 1998.

[29] A. Hamáček a J. Čengery, „Přednášky z předmětu navrhování elektronických zařízení,“ 2019.

[30] *STM-Helper*. URL: https://marketplace.visualstudio.com/items?itemName=BuzzyElectronics.stm-helper (cit. 27. 03. 2022).

[31] *stm32-for-vscode*. URL: https://marketplace.visualstudio.com/items?itemName=bmd.stm32-for-vscode (cit. 27. 03. 2022).

[32] *Cortex-Debug*. URL: https://marketplace.visualstudio.com/items?itemName=marus25.cortex-debug (cit. 27. 03. 2022).

[33] *VT100 - video terminal*. URL: https://en.wikipedia.org/wiki/VT100 (cit. 30. 04. 2022).

Příloha A - zdrojové kódy k simulacím

SSI simulation 1

```
from matplotlib import pyplot as plt
import matplotlib.patches as mpatches
import numpy as np

#Frame 1
bits = [0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0]
data = np.repeat(bits, 2)
clock = 1 - np.arange(0, len(data)) % 2
t = 0.5 * np.arange(0, len(data))
plt.step(t, clock + 4, 'r', linewidth = 2, where='post')
plt.step(t, data + 2, 'b', linewidth = 2, where='post')
plt.ylim([1,6])

#Monoflop
bits_mon = [0,0,0,0,0,0,0,1,1,1]
data_mon = np.repeat(bits_mon, 2)
clock_mon = np.array([0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1])
monoflop = 0.5 * np.arange(len(data)-1, (len(data)-1)+len(data_mon))
plt.step(monoflop, clock_mon + 4, 'r', linewidth = 2, where='post')
plt.step(monoflop, data_mon + 2, 'b', linewidth = 2, where='post')

#Frame 2
bits2 = [1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0]
data2 = np.repeat(bits2, 2)
clock2 =1 - np.arange( (len(data)-1)+(len(data_mon)-1), ...
    (len(data)-1)+(len(data_mon)-1)+len(data2)) % 2
t2 = 0.5 * np.arange((len(data)-1)+(len(data_mon)-1), ...
    (len(data)-1)+(len(data_mon)-1)+len(data2))
plt.step(t2, clock2 + 4, 'r', linewidth = 2, where='post')
plt.step(t2, data2 + 2, 'b', linewidth = 2, where='post')

#Monoflop2
t1 = (len(data)-1)+(len(data_mon)-1)
t2 = (len(data)-1)+(len(data_mon)-1)+(len(data2)-1)

bits_mon2 = [0,0,0,0,0,0,0,0,1,1,1]
data_mon2 = np.repeat(bits_mon2, 2)
clock_mon2 = np.array([0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1])
monoflop2 = 0.5 * np.arange(t2, t2+len(data_mon2))
plt.step(monoflop2, clock_mon2 + 4, 'r', linewidth = 2, where='post')
plt.step(monoflop2, data_mon2 + 2, 'b', linewidth = 2, where='post')


#Print graph
plt.grid(axis = 'x', color = '0.80')
plt.grid(axis = 'y', color = '0.80')
plt.xlabel('Time')
```

```
plt.title('Simulation of SSI communication - monoflop + idle')

plt.annotate('Data', xy=(1,2.5), xytext=(-3,2.2),
             arrowprops={'arrowstyle': '->', 'lw': 4, 'color': 'blue'},
             va='center')

plt.annotate('CLK', xy=(0,4.5), xytext=(-3,4.2),
             arrowprops={'arrowstyle': '->', 'lw': 4, 'color': 'red'},
             va='center')


plt.annotate('Monoflop time', xy=(27,2), xytext=(2,3.5),
             arrowprops={'arrowstyle': '-|>'}, va='center')

plt.annotate('Idle state', xy=(33,3), xytext=(33,3.5),
             arrowprops={'arrowstyle': '-|>'}, va='center')

plt.gca().axes.yaxis.set_ticklabels([])
plt.gca().axes.xaxis.set_ticklabels([])
plt.show()
```

SSI simulation 2

```
from matplotlib import pyplot as plt
import matplotlib.patches as mpatches
import numpy as np

#Frame 1
bits = [0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0]
data = np.repeat(bits, 2)
clock = 1 - np.arange(0, len(data)) % 2
t = 0.5 * np.arange(0, len(data))
plt.step(t, clock + 4, 'r', linewidth = 2, where='post')
plt.step(t, data + 2, 'b', linewidth = 2, where='post')
plt.ylim([1,6])

#Monoflop
bits_mon = [0,0,0,0,0,0,0,0]
data_mon = np.repeat(bits_mon, 2)
clock_mon = np.array([0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1])
monoflop = 0.5 * np.arange(len(data)-1, (len(data)-1)+len(data_mon))
plt.step(monoflop, clock_mon + 4, 'r', linewidth = 2, where='post')
plt.step(monoflop, data_mon + 2, 'b', linewidth = 2, where='post')

#Frame 2
bits2 = [0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0]
data2 = np.repeat(bits2, 2)
clock2 =1 - np.arange( (len(data)-1)+(len(data_mon)-1), ...
    (len(data)-1)+(len(data_mon)-1)+len(data2)) % 2
t2 = 0.5 * np.arange((len(data)-1)+(len(data_mon)-1), ...
    (len(data)-1)+(len(data_mon)-1)+len(data2))
plt.step(t2, clock2 + 4, 'r', linewidth = 2, where='post')
```

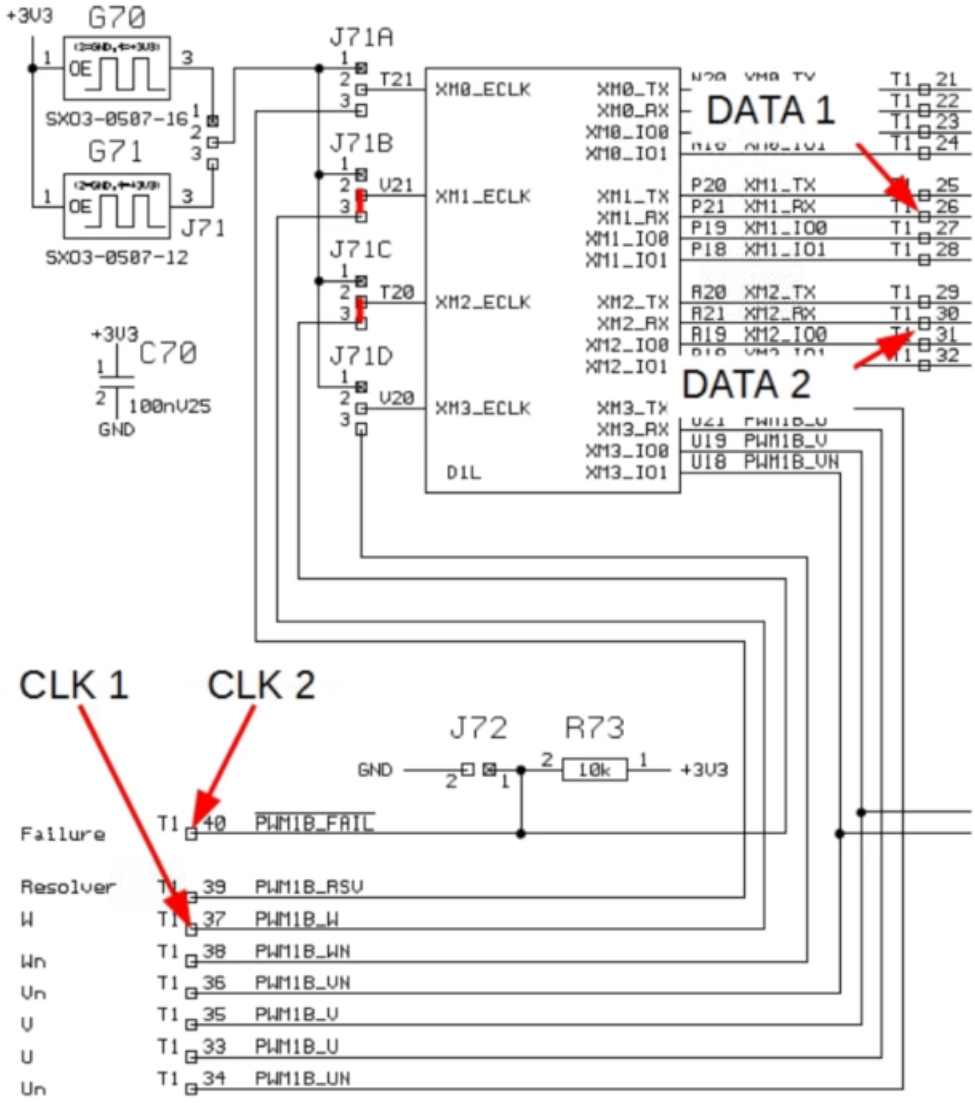
```
plt.step(t2, data2 + 2, 'b', linewidth = 2, where='post')

#Monoflop2
t1 = (len(data)-1)+(len(data_mon)-1)
t2 = (len(data)-1)+(len(data_mon)-1)+(len(data2)-1)

bits_mon2 = [0,0,0,0,0,0,0,0]
data_mon2 = np.repeat(bits_mon2, 2)
clock_mon2 = np.array([0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1])
monoflop2 = 0.5 * np.arange(t2, t2+len(data_mon2))
plt.step(monoflop2, clock_mon2 + 4, 'r', linewidth = 2, where='post')
plt.step(monoflop2, data_mon2 + 2, 'b', linewidth = 2, where='post')


#Print graph
plt.grid(axis = 'x', color = '0.80')
plt.grid(axis = 'y', color = '0.80')
plt.xlabel('Time')
#plt.ylabel('Data                                                                CLK')
plt.title('Simulation of SSI communication - only with monoflop')

#red_patch = mpatches.Patch(color='red', label='CLK')
#plt.legend(handles=[red_patch])
#blue_patch = mpatches.Patch(color='blue', label='DATA')
#plt.legend(handles=[blue_patch])

plt.annotate('Data', xy=(1,2.5), xytext=(-3,2.2),
             arrowprops={'arrowstyle': '->', 'lw': 4, 'color': 'blue'},
             va='center')

plt.annotate('CLK', xy=(0,4.5), xytext=(-3,4.2),
             arrowprops={'arrowstyle': '->', 'lw': 4, 'color': 'red'},
             va='center')


plt.annotate('Monoflop time', xy=(30,2), xytext=(15,3.5),
             arrowprops={'arrowstyle': '-|>'}, va='center')


plt.gca().axes.yaxis.set_ticklabels([])
plt.gca().axes.xaxis.set_ticklabels([])
plt.show()
```

Simulation of OPC UA client

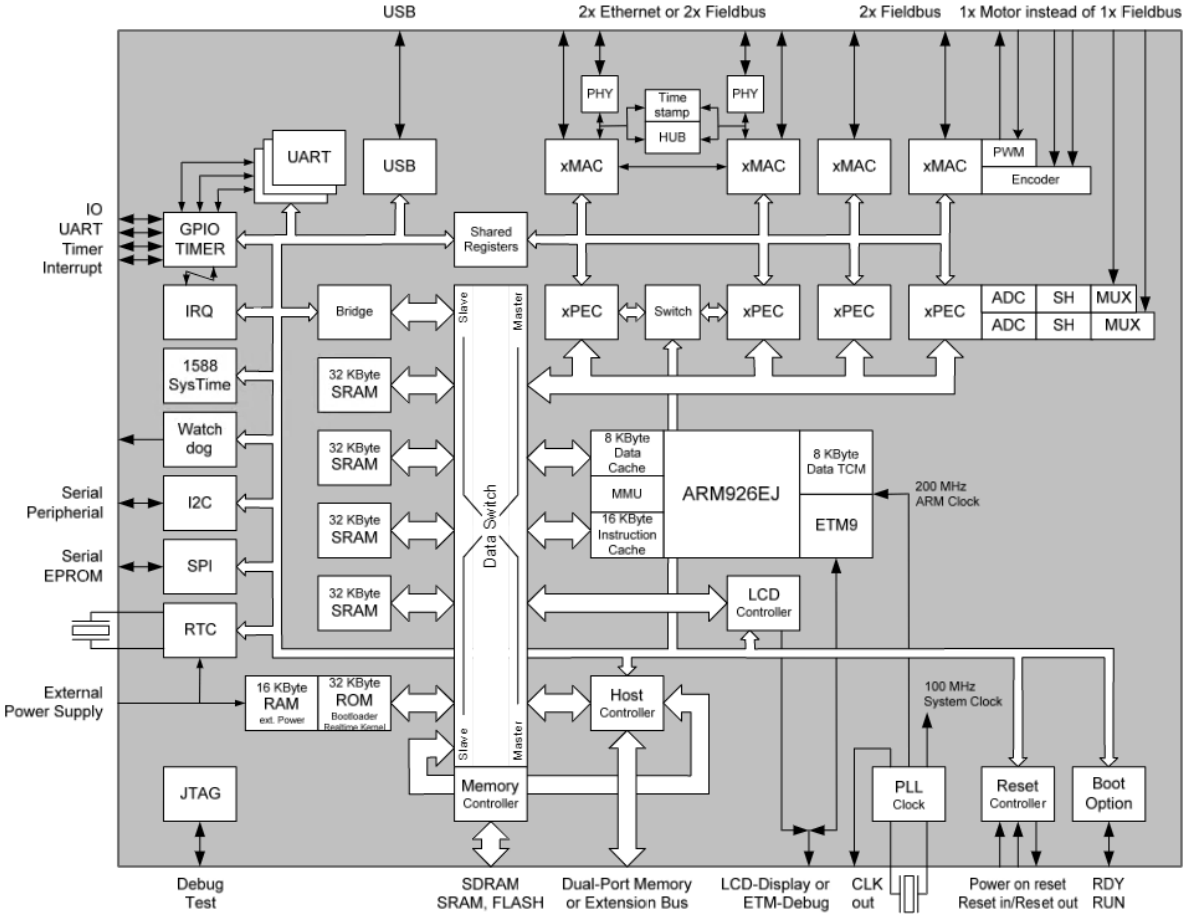
```
# Simulation of UPC UA server of SSI monitor
# ----------------------------------------

from opcua import Client
import time
import datetime
# Set url adrress generated by ProSys server
```

```
8  url = "opc.tcp://WIN-L0HG94HBIL2:53530/OPCUA/SimulationServer"
9  client = Client(url)
10 # Info, that client is connected
11 print("Client connected")
12
13 #Ask user, how many samples he/she wants read
14 print("Set how many samples:")
15 samples = int(input())
16
17 #Ask user about the "resoliton" of timestamp
18 print("Set how time stamp resolution: ")
19 time_stamp_resolution = float(input())
20
21 # Connect client
22 client.connect()
23
24 # for loop
25 for i in range(samples):
26
27     # Set the nodes and indexes, from which we want to read
28     CH1_VAL_node = client.get_node("ns=3;i=1002")
29     CH1_ID_node = client.get_node("ns=3;i=1001")
30
31     # Read the values from nodes
32     CH1_VAL_val = CH1_VAL_node.get_value()
33     CH1_ID_val = CH1_ID_node.get_value()
34
35     # Print everything to terminal
36     print("Channel 1 values: ", int(CH1_VAL_val), "Channel 1 msg ID: ", ...
           CH1_ID_val, "Timestamp", datetime.datetime.now())
37
38     # wait
39     time.sleep(time_stamp_resolution)
40
41 # After for loop, disconnect from the server
42 client.disconnect()
43
44 #Info, that client is disconected
45 print("Client disconnected")
```

Příloha B - Dodatečné informace, schémata a obrázky k SSI monitoru



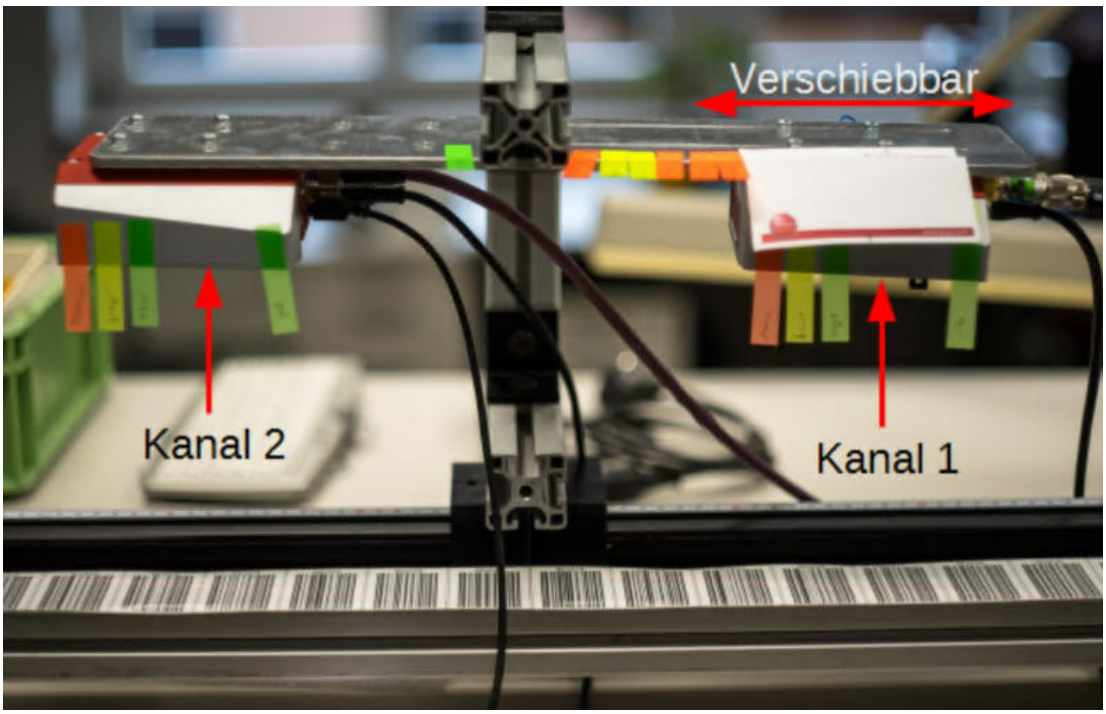
Obrázek 1: Připojení signálových vodičů k procesoru [1]



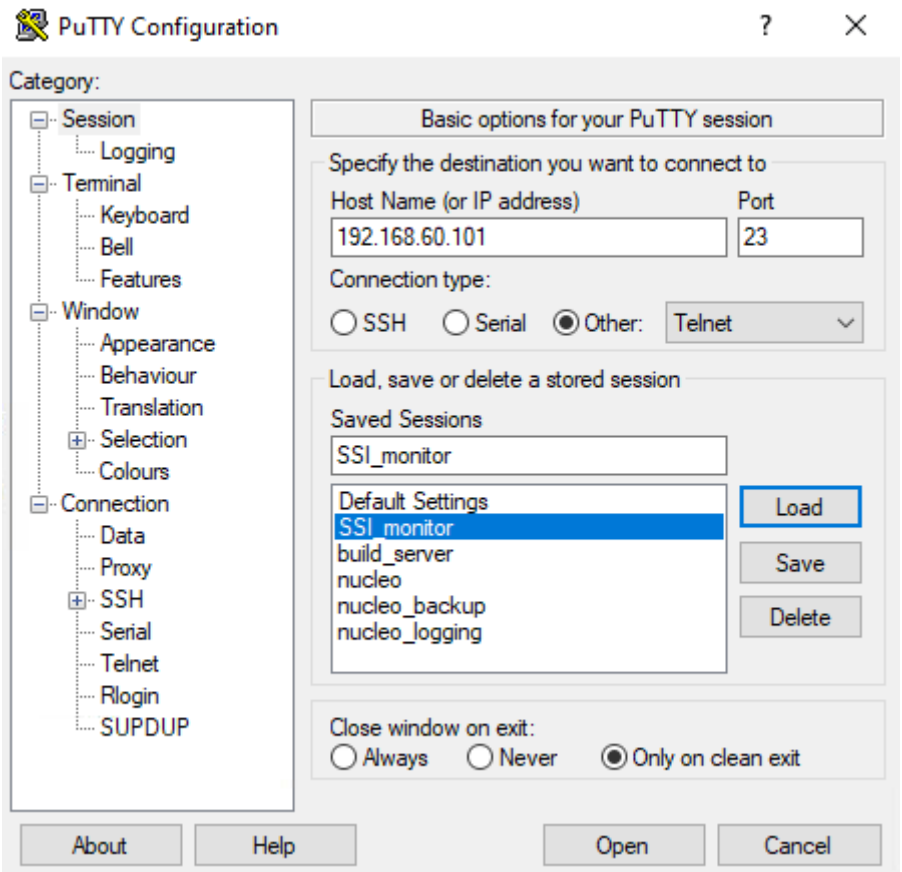
Obrázek 2: Blokové schéma procesoru netX500 [1]

- `const U32 SSI_PACKET_OK = 0x00000000`: Telegramm ok.
- `const U32 SSI_FIRST_CLOCK_NEGATIVE_EDGE_INVALID = 0x00000001`: nicht abschaltbar
- `const U32 SSI_LAST_CLOCK_POSITIVE_EDGE_INVALID = 0x00000002`: nicht abschaltbar
- `const U32 SSI_DATA_POSITIVE_EDGE_INVALID = 0x00000004`: nicht abschaltbar
- `const U32 SSI_CLOCK_TIMEOUT_ERROR = 0x00000008`: nicht abschaltbar
- `const U32 SSI_INVALID_CLOCK_PAUSE_LENGTH = 0x00000010`: abschaltbar
- `const U32 SSI_INVALID_CLOCK_PERIOD_LENGTH = 0x00000020`: abschaltbar
- `const U32 SSI_INVALID_MONOFLOP_TIME = 0x00000040`: abschaltbar
- `const U32 SSI_INVALID_POSITION_JUMP = 0x00000080`: abschaltbar
- `const U32 SSI_INVALID_DATABIT_COUNT = 0x00000200`: abschaltbar
- `const U32 SSI_TWO_CHANNEL_CHANNEL_ONE_DOWN = 0x00000400`: abschaltbar, gibt an, dass Kanal 1 keine Telegramme mehr sendet
- `const U32 SSI_TWO_CHANNEL_CHANNEL_TWO_DOWN = 0x00000800`: abschaltbar, gibt an, dass Kanal 2 keine Telegramme mehr sendet
- `const U32 SSI_TWO_CHANNEL_POSITION_CHECK_ERROR = 0x00001000`: abschaltbar, gibt eine Verletzung des Positions-Offset-Toleranzbandes an
- `const U32 SSI_INTERNAL_BUFFER_ERROR = 0x10000000`: nicht abschaltbar, interner Fehler. Wenn dieser auftritt, muss der Entwickler informiert werden.

Obrázek 3: Přehled jednotlivých Error ID [1]

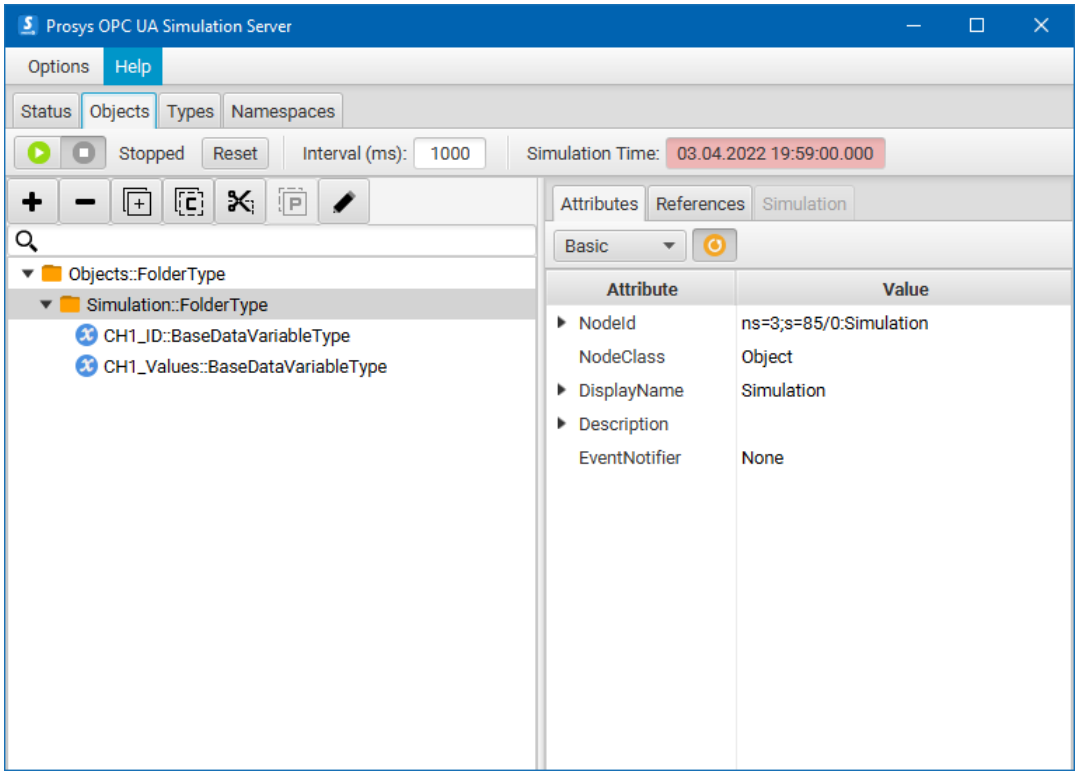


Obrázek 4: Dva kanály se vzdálenostním offsetem [1]

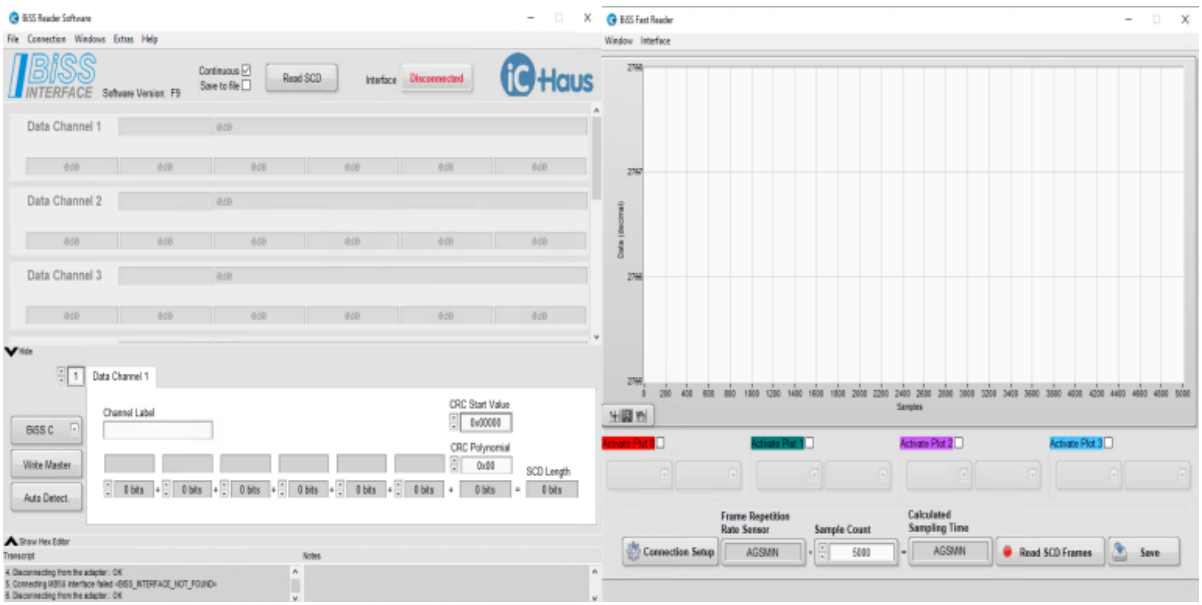


Obrázek 5: Nastavení v aplikaci PuTTY pro připojení k SSI monitoru pomocí telnetu

Obrázek 6: Uvodni stranka s automaticky vygenerovanou adresou serveru



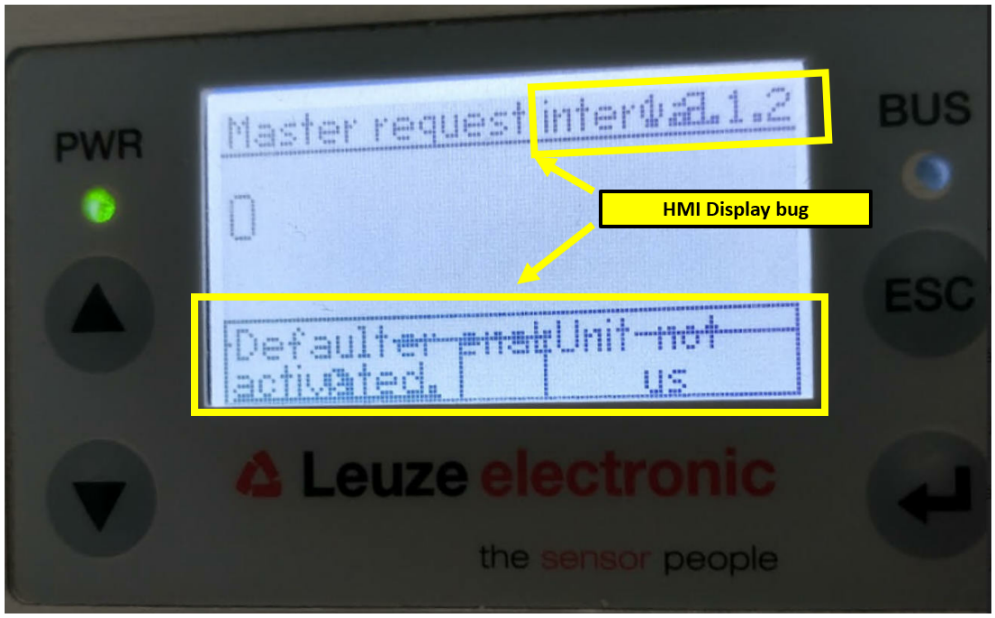
Obrázek 7: Nastaveni serveru a uzlů



Obrázek 8: Ukázka SW pro konfiguraci master zařízení MB5U

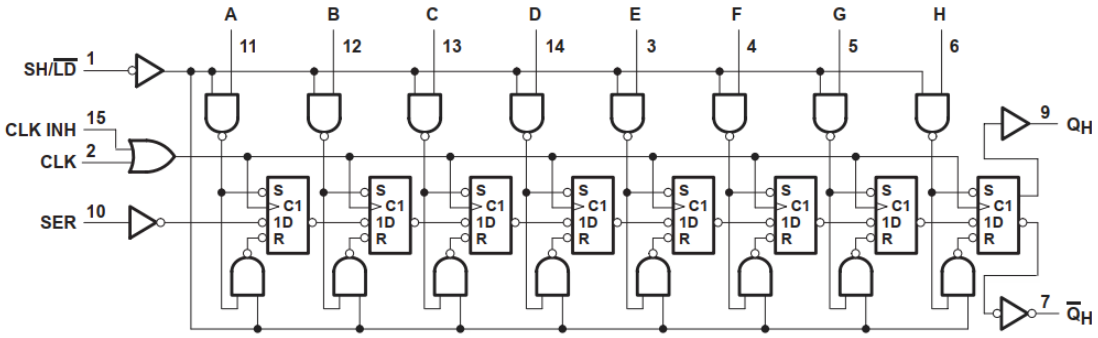
Obrázek 9: Páska s čárovým kódem 30 mm [9]

Obrázek 10: Páska s čárovým kódem 40 mm [9]

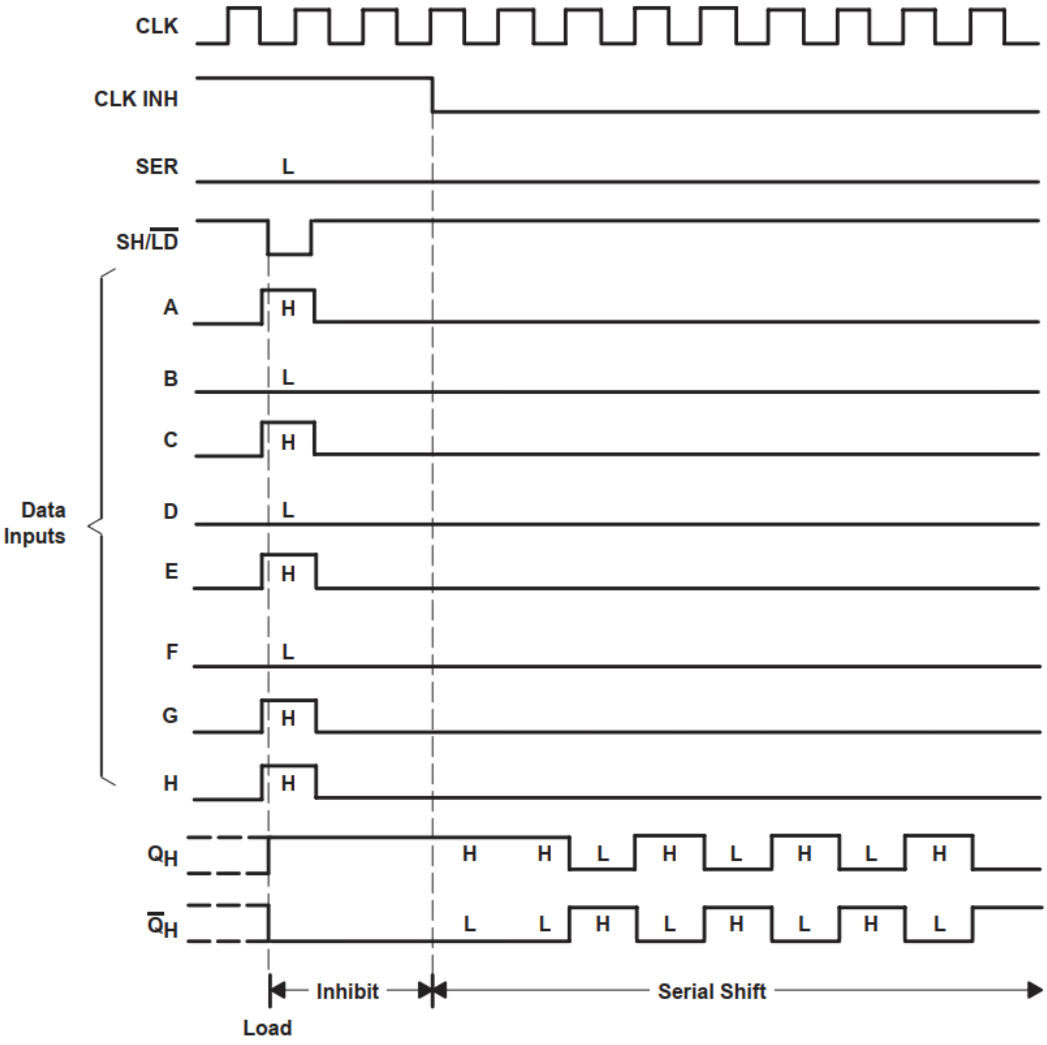


Obrázek 11: Chyba zobrazení na obrazovce SSI monitoru

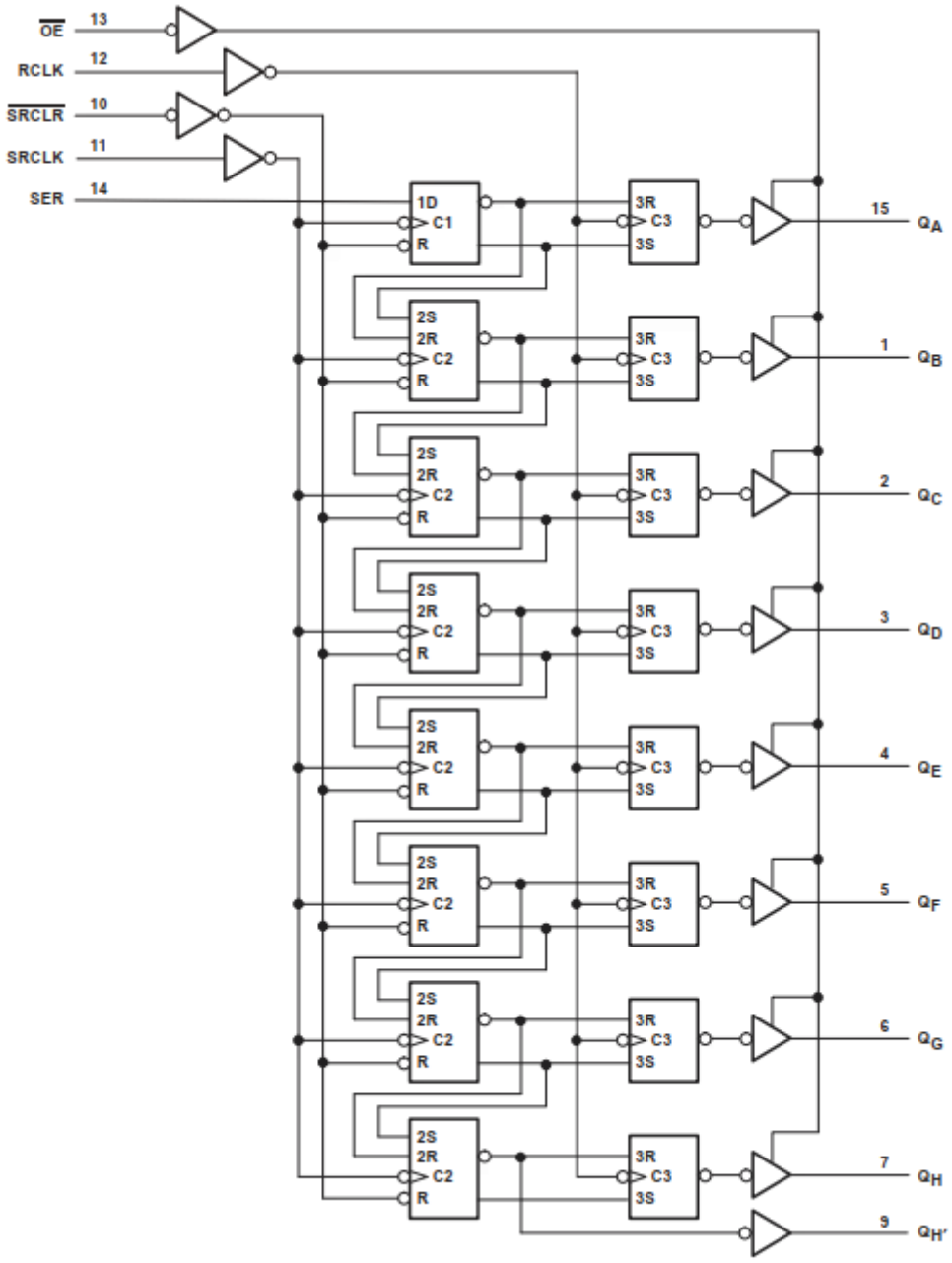
Příloha C - Dodatky k master/slave zařízení



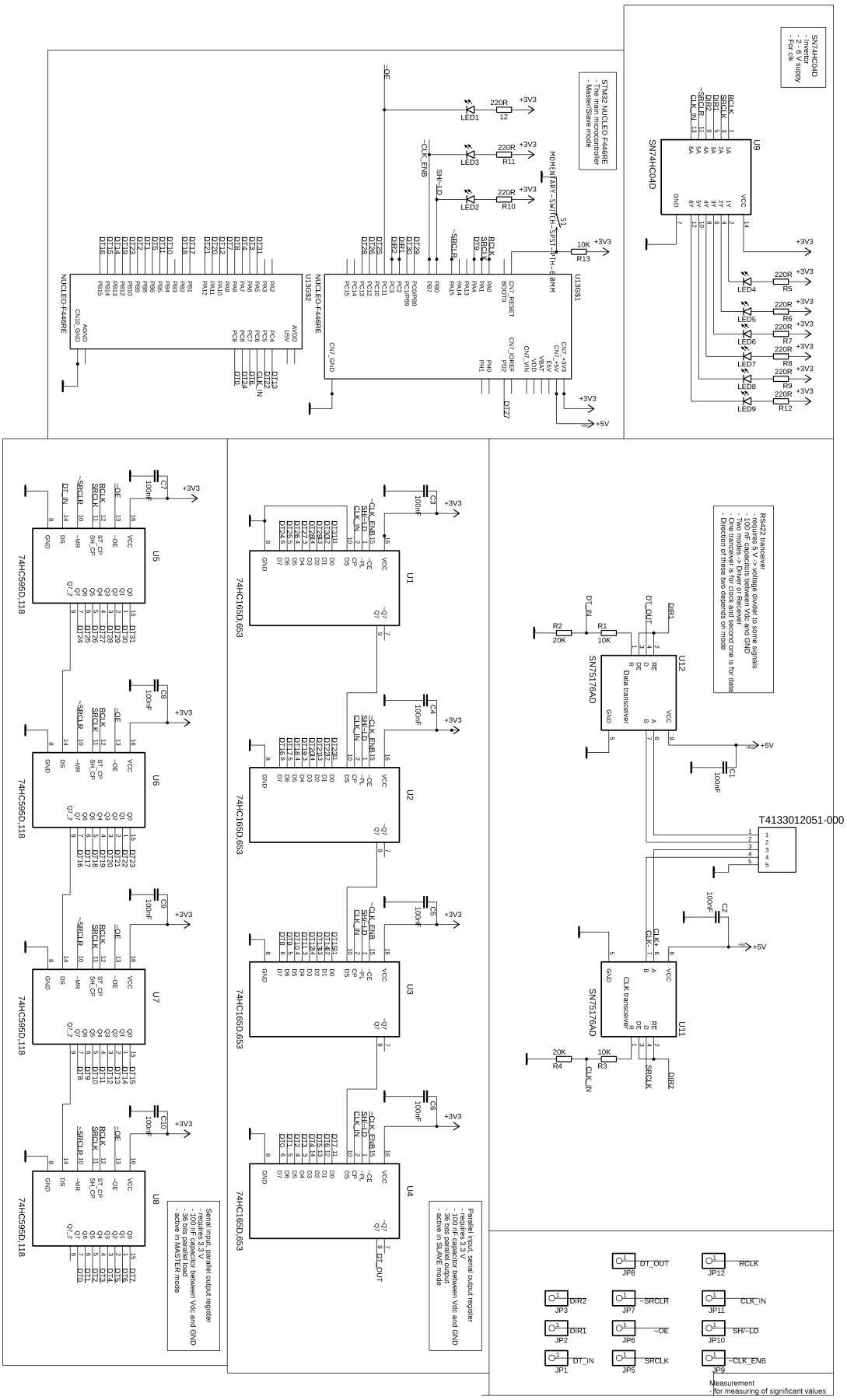
Obrázek 12: Vnitřní zapojení posuvného registru 74HC165D [23]



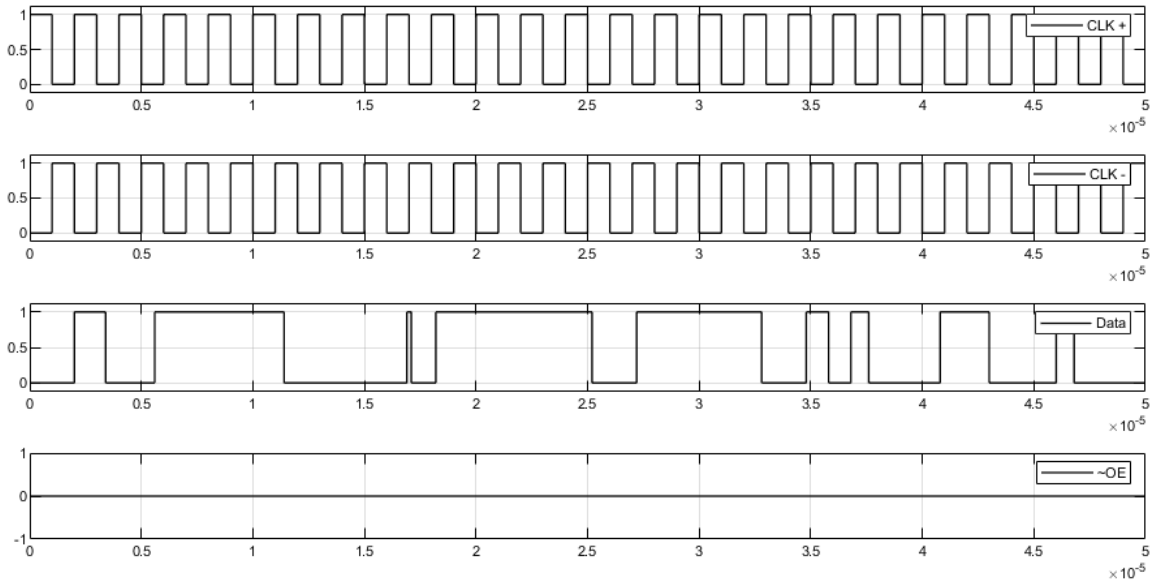
Obrázek 13: Časový diagram posuvného registru 74HC165D [23]



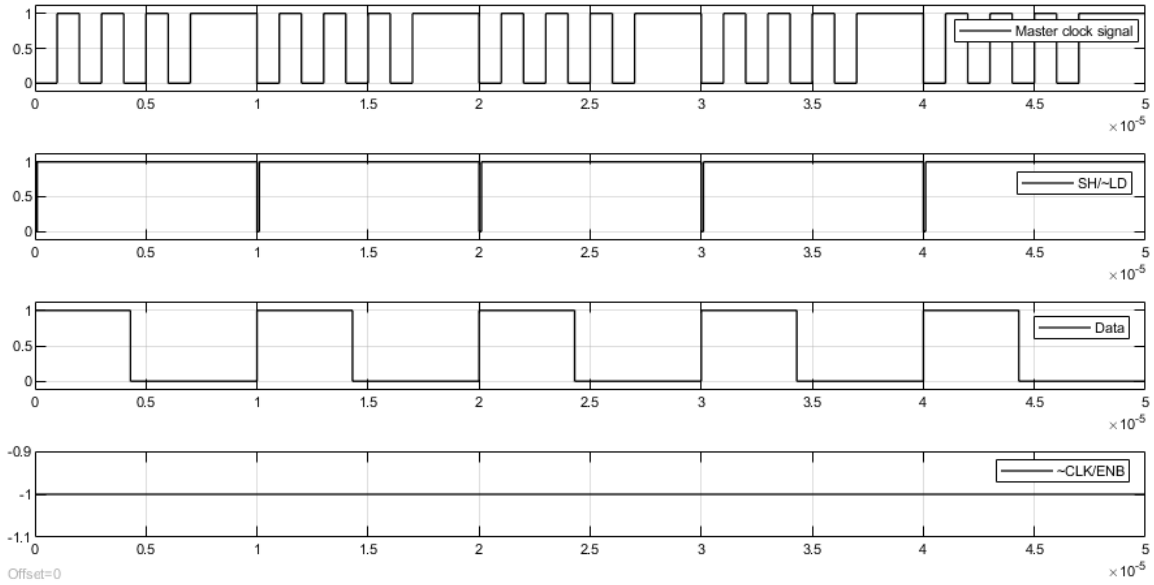
Obrázek 14: Časový diagram posuvného registru 74HC565 [24]



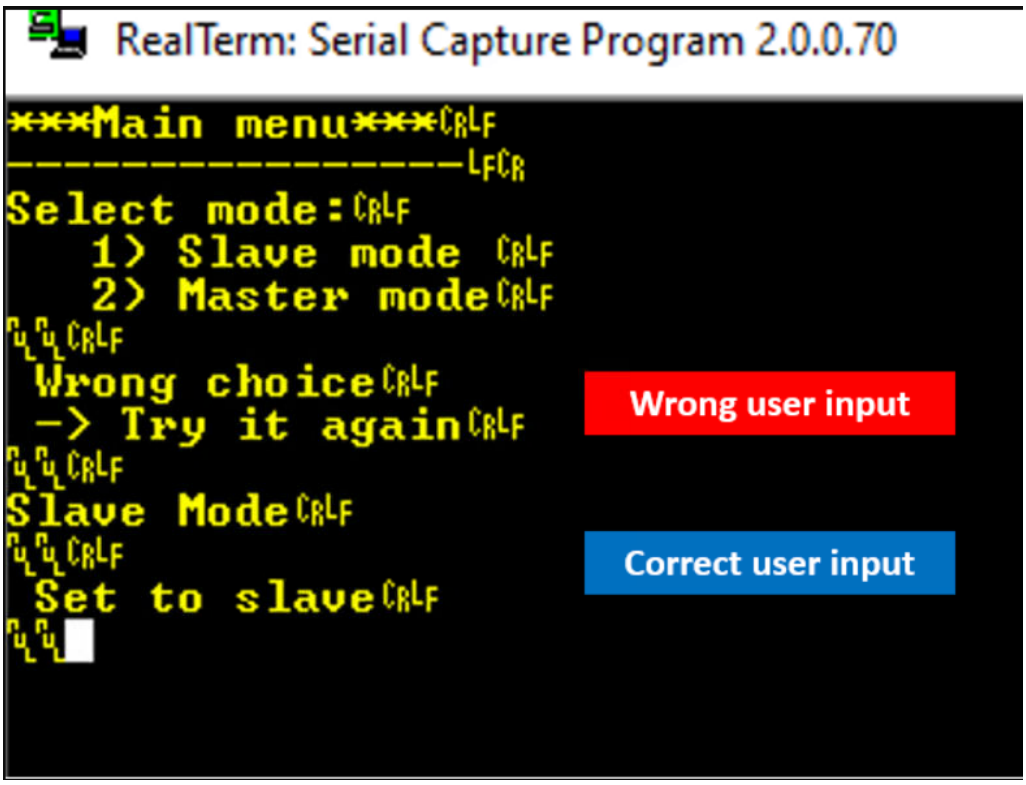
Obrázek 15: Schéma zapojení univerzálného master/slave zařízení



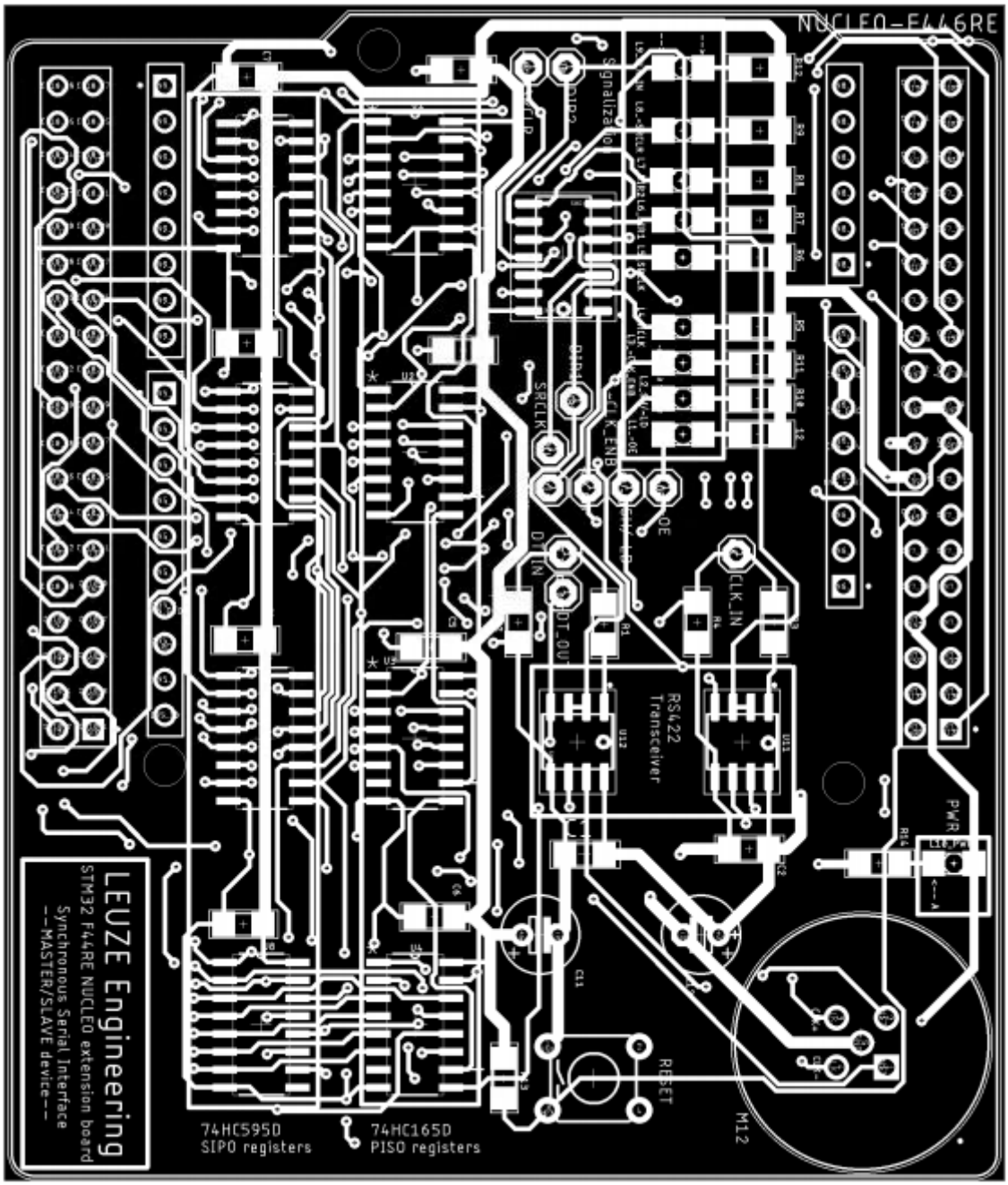
Obrázek 16: Zpracovaný výstup logického analyzátoru - 8-bit master mód



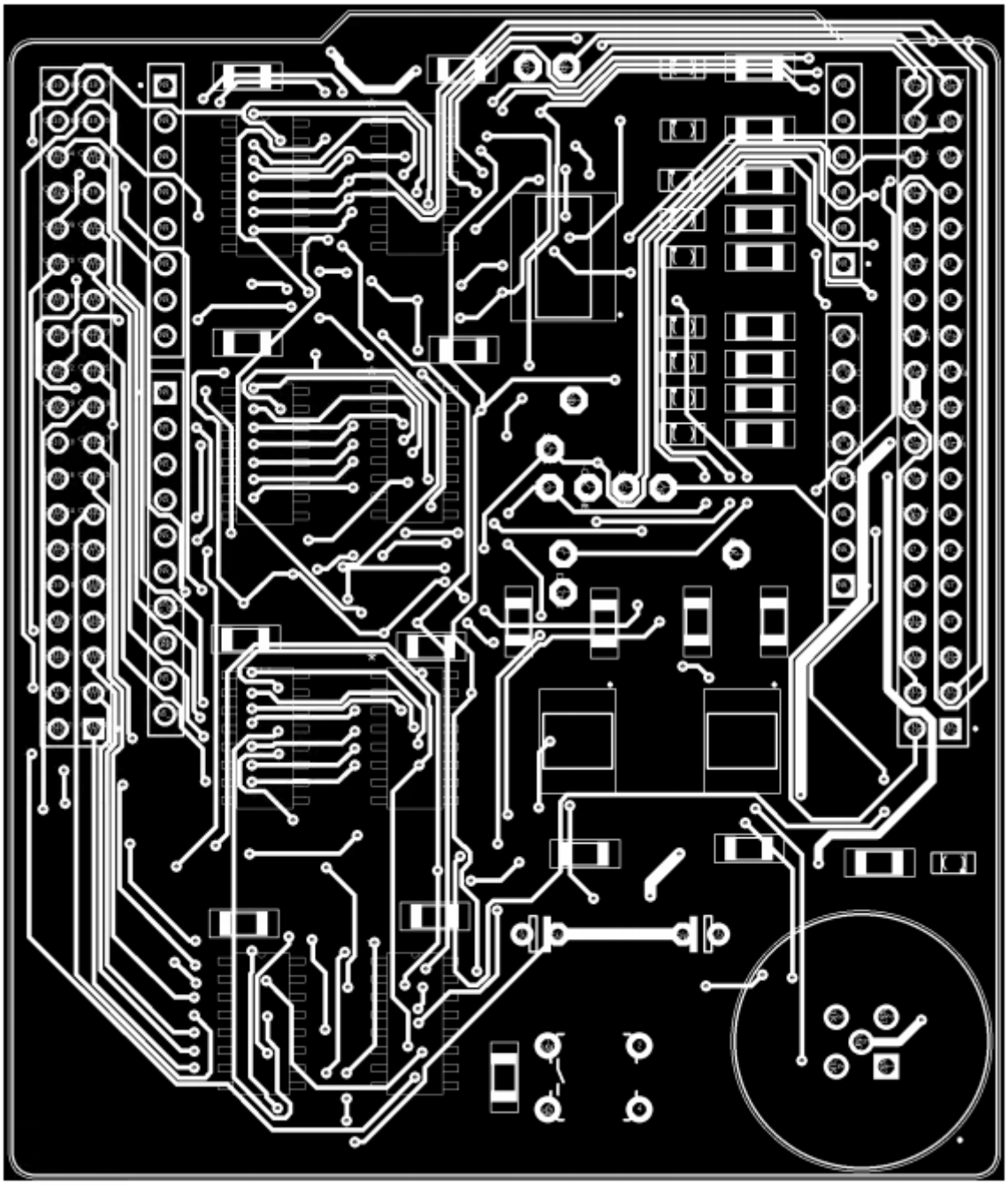
Obrázek 17: Zpracovaný výstup logického analyzátoru - 8-bit slave mód



Obrázek 18: Ukázka UI pro volbu příslušného módu



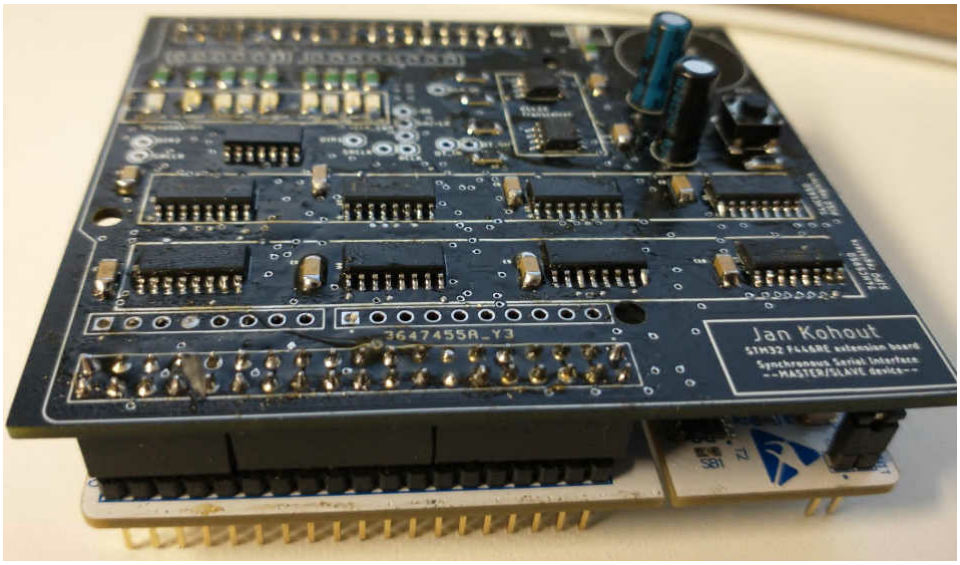
Obrázek 19: Horní vrstva desky plošných spojů



Obrázek 20: Spodní vrstva desky plošných spojů

Part	Value	Device	Package
12	220R	R-EU_R1206	R1206
C1	100nF	C-EUC1206	C1206
C2	100nF	C-EUC1206	C1206
C3	100nF	C-EUC1206	C1206
C4	100nF	C-EUC1206	C1206
C5	100nF	C-EUC1206	C1206
C6	100nF	C-EUC1206	C1206
C7	100nF	C-EUC1206	C1206
C8	100nF	C-EUC1206	C1206
C9	100nF	C-EUC1206	C1206
C10	100nF	C-EUC1206	C1206
C11	1u	CPOL-EUE2,5-6E	E2,5-6E
C12	1u	CPOL-EUE2,5-6E	E2,5-6E
JP1	PINHD-1X1	1X01	
JP2	PINHD-1X1	1X01	
JP3	PINHD-1X1	1X01	
JP5	PINHD-1X1	1X01	
JP6	PINHD-1X1	1X01	
JP7	PINHD-1X1	1X01	
JP8	PINHD-1X1	1X01	
JP9	PINHD-1X1	1X01	
JP10	PINHD-1X1	1X01	
JP11	PINHD-1X1	1X01	
JP12	PINHD-1X1	1X01	
LED1	LEDSMT1206	1206	
LED2	LEDSMT1206	1206	
LED3	LEDSMT1206	1206	
LED4	LEDSMT1206	1206	
LED5	LEDSMT1206	1206	
LED6	LEDSMT1206	1206	
LED7	LEDSMT1206	1206	
LED8	LEDSMT1206	1206	
LED9	LEDSMT1206	1206	
LED10	LEDSMT1206	1206	
R1	10K	R-EU_R1206	R1206
R2	20K	R-EU_R1206	R1206
R3	10K	R-EU_R1206	R1206
R4	20K	R-EU_R1206	R1206
R5	220R	R-EU_R1206	R1206
R6	220R	R-EU_R1206	R1206
R7	220R	R-EU_R1206	R1206
R8	220R	R-EU_R1206	R1206
R9	220R	R-EU_R1206	R1206
R10	220R	R-EU_R1206	R1206
R11	220R	R-EU_R1206	R1206
R12	220R	R-EU_R1206	R1206
R13	10K	R-EU_R1206	R1206
R14	220R	R-EU_R1206	R1206
S1	TACTILE_SWITCH_PTH_6.0MM		
T4133012051-000	T4141012041-000	T4141012041-000	TE_T4141012041-000
U1	74HC165D,653	74HC165D,653	SOIC127P600X175-16N
U2	74HC165D,653	74HC165D,653	SOIC127P600X175-16N
U3	74HC165D,653	74HC165D,653	SOIC127P600X175-16N
U4	74HC165D,653	74HC165D,653	SOIC127P600X175-16N
U5	74HC595D,118	74HC595D,118	SOIC127P600X175-16N
U6	74HC595D,118	74HC595D,118	SOIC127P600X175-16N
U7	74HC595D,118	74HC595D,118	SOIC127P600X175-16N
U8	74HC595D,118	74HC595D,118	SOIC127P600X175-16N
U9	SN74HC04D	SN74HC04D	SOIC127P600X175-14N
U11	SN75176AD	SN75176AD	SOIC127P599X175-8N
U12	SN75176AD	SN75176AD	SOIC127P599X175-8N
U13	NUCLEO-F446RE	NUCLEO-F446RE	MODULE_NUCLEO-F446RE

Partlist - SSI ext board

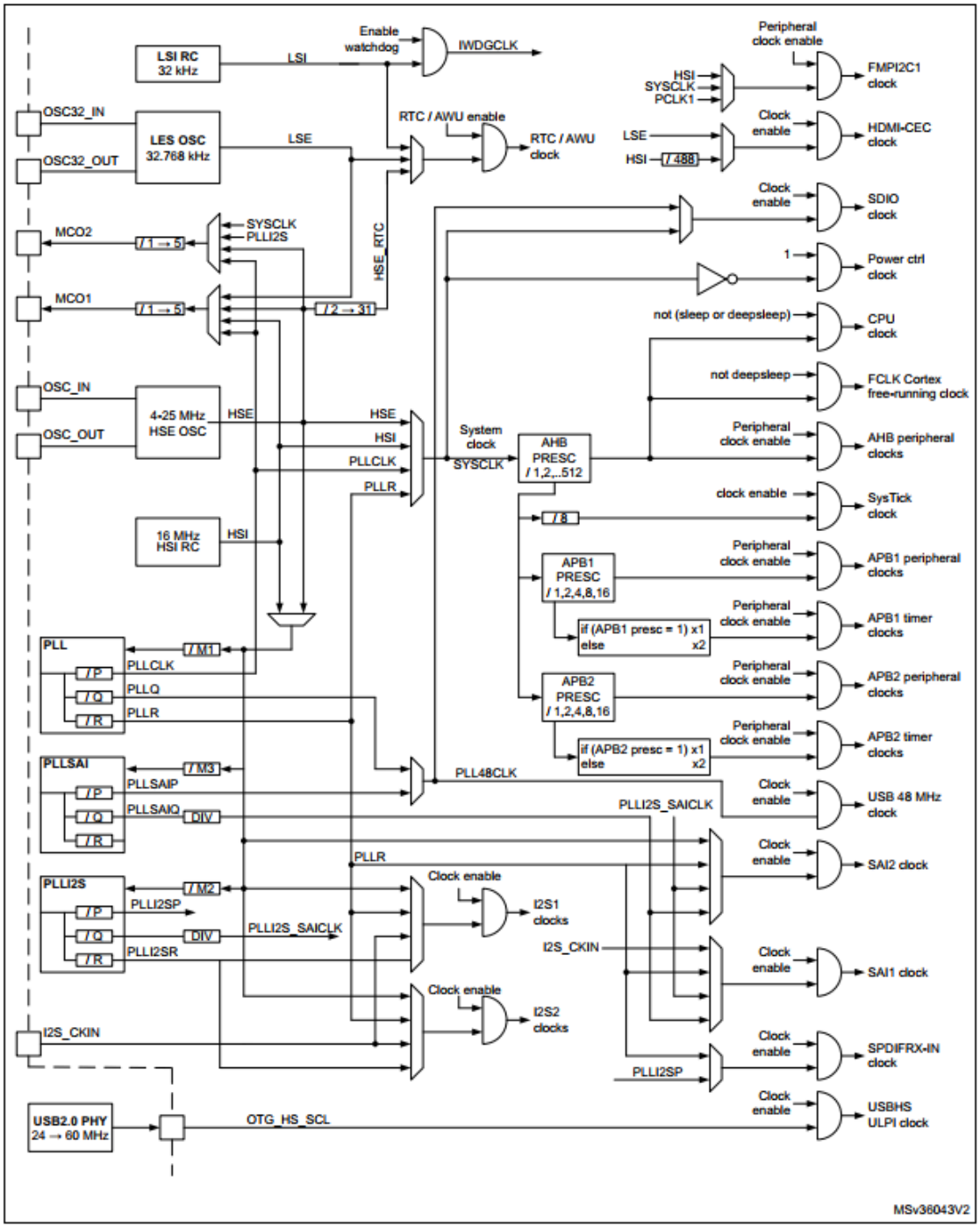


Obrázek 21: Výsledné zapojení DPS s řídící deskou

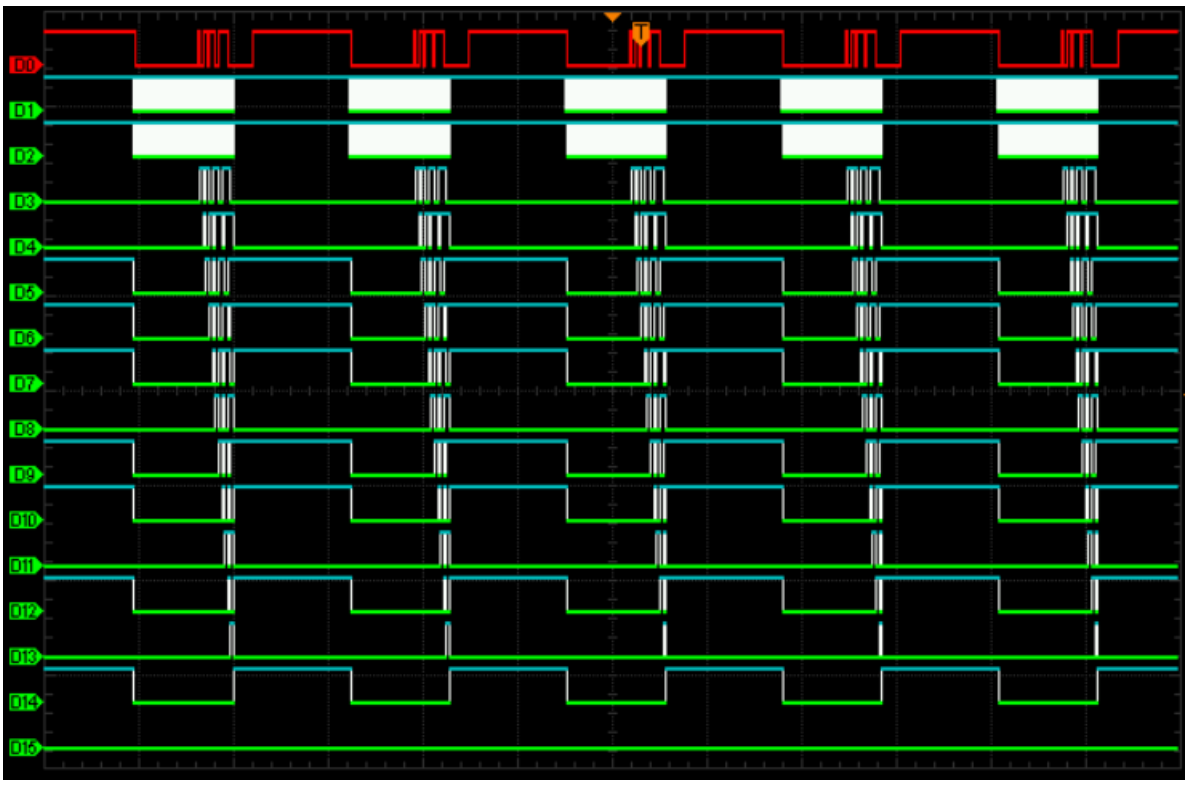
Data	PORT	PIN	Data	PORT	PIN
DT0	C	9	DT16	B	15
DT1	B	8	DT17	B	1
DT2	B	9	DT18	B	2
DT3	A	6	DT19	B	12
DT4	A	7	DT20	A	11
DT5	B	6	DT21	A	12
DT6	C	7	DT22	C	5
DT7	A	9	DT23	C	1
DT8	A	8	DT24	C	8
DT9	A	4	DT25	C	10
DT10	B	4	DT26	C	12
DT11	B	5	DT27	D	2
DT12	A	10	DT28	C	13
DT13	C	4	DT29	C	0
DT14	B	13	DT30	C	1
DT15	B	14	DT31	A	5

Control signal	Port	PIN
nSRCLR	A	15
SH_nLD	B	0
nCLK_ENB	B	7
DIR1	C	2
DIR2	C	3
nOE	C	11

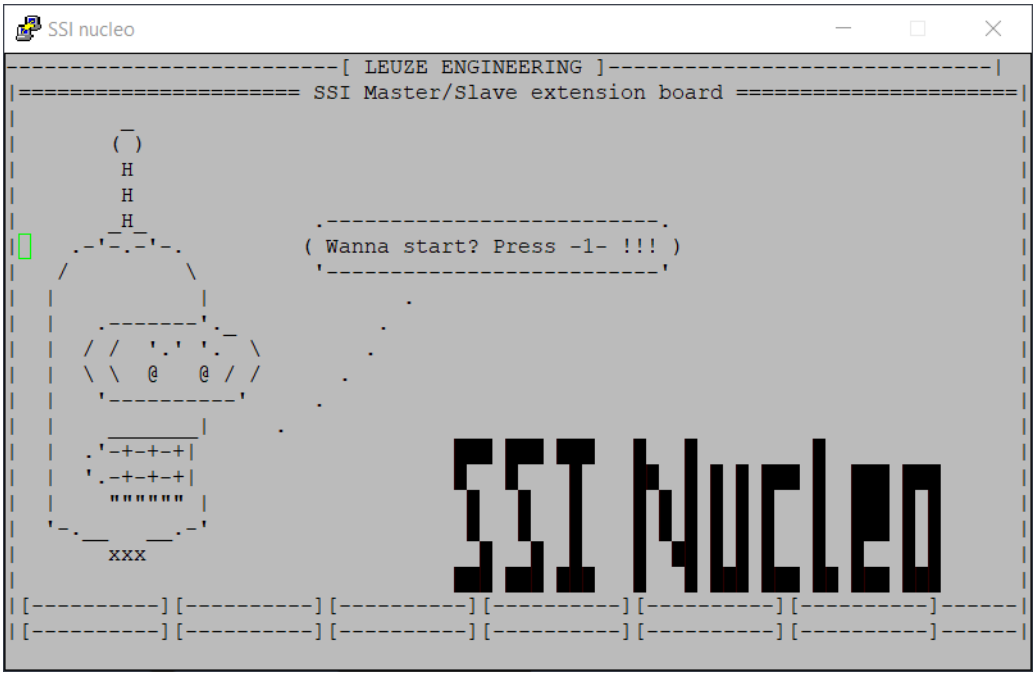
Obrázek 22: Tabulka jednotlivých GPIO pinů



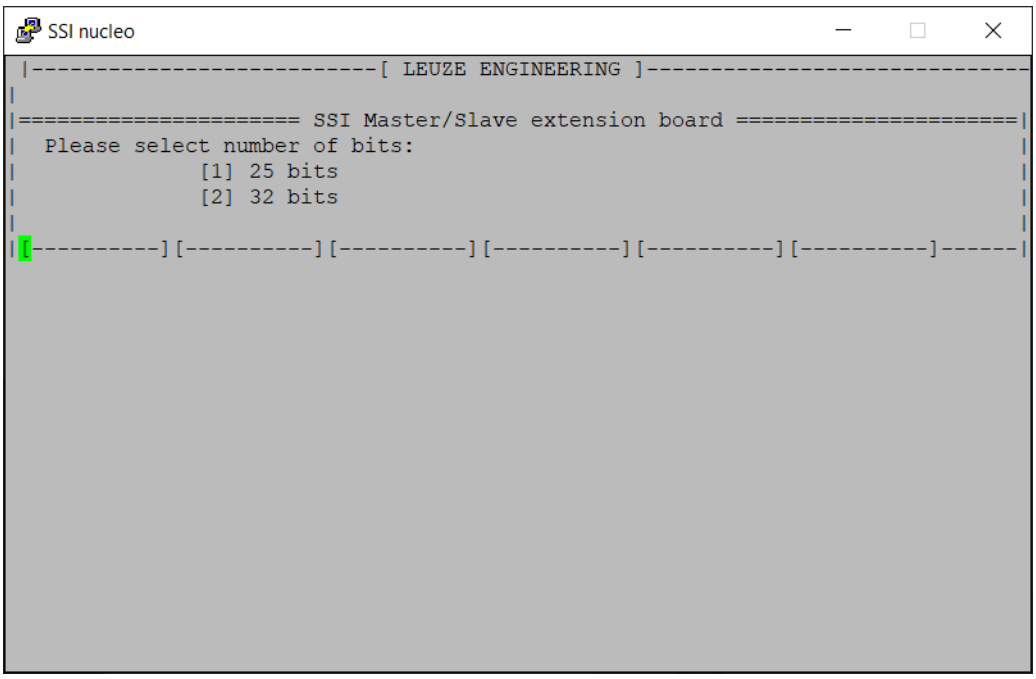
Obrázek 23: Diagram pro konfiguraci systémových hodin [22]



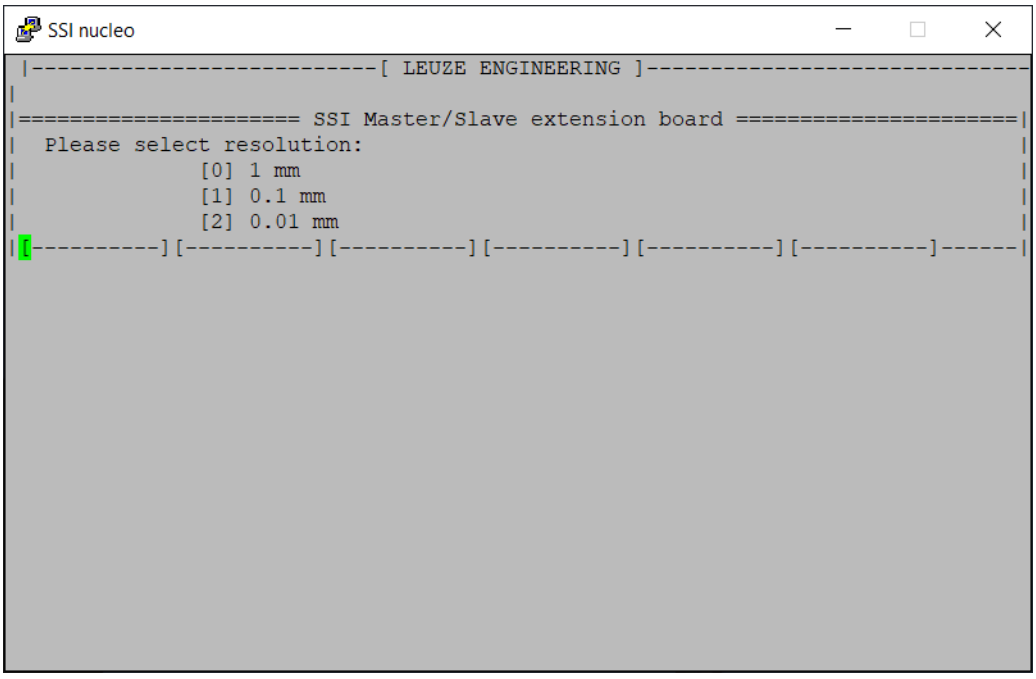
Obrázek 24: Výstup osciloskopu při práci zařízení v režimu master - více datových rámců



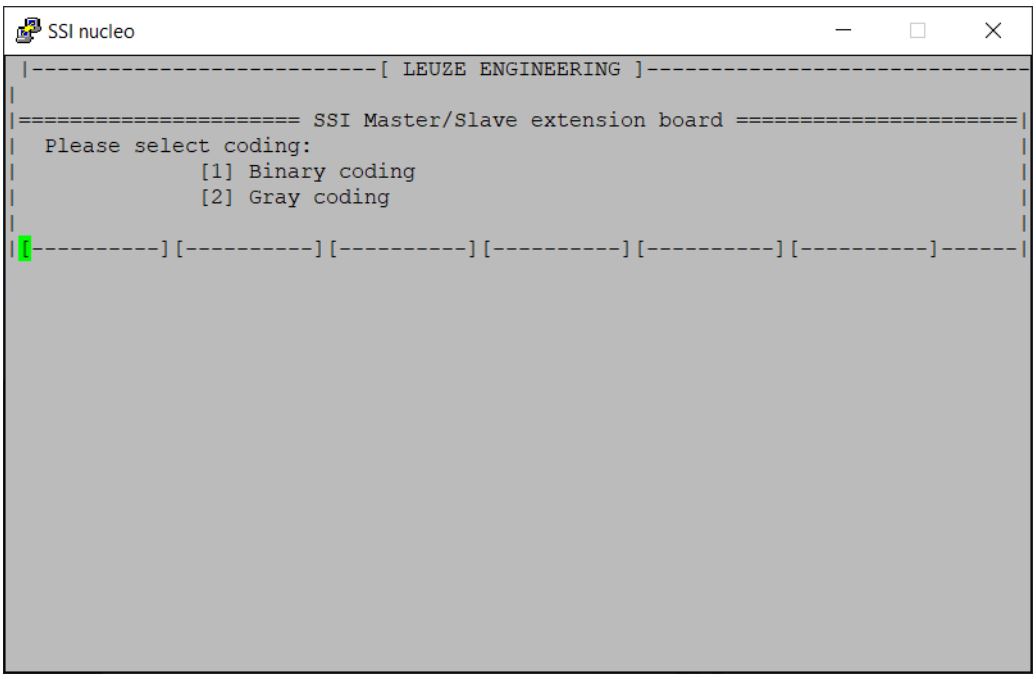
Obrázek 25: Úvodní obrazovka



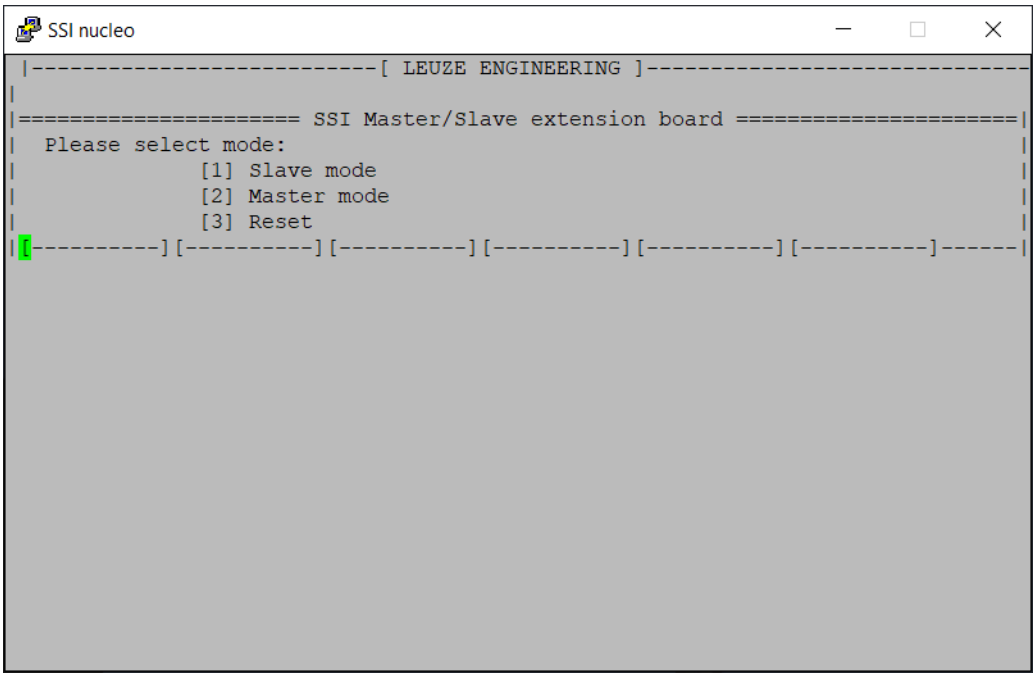
Obrázek 26: Výběr počtu bitů



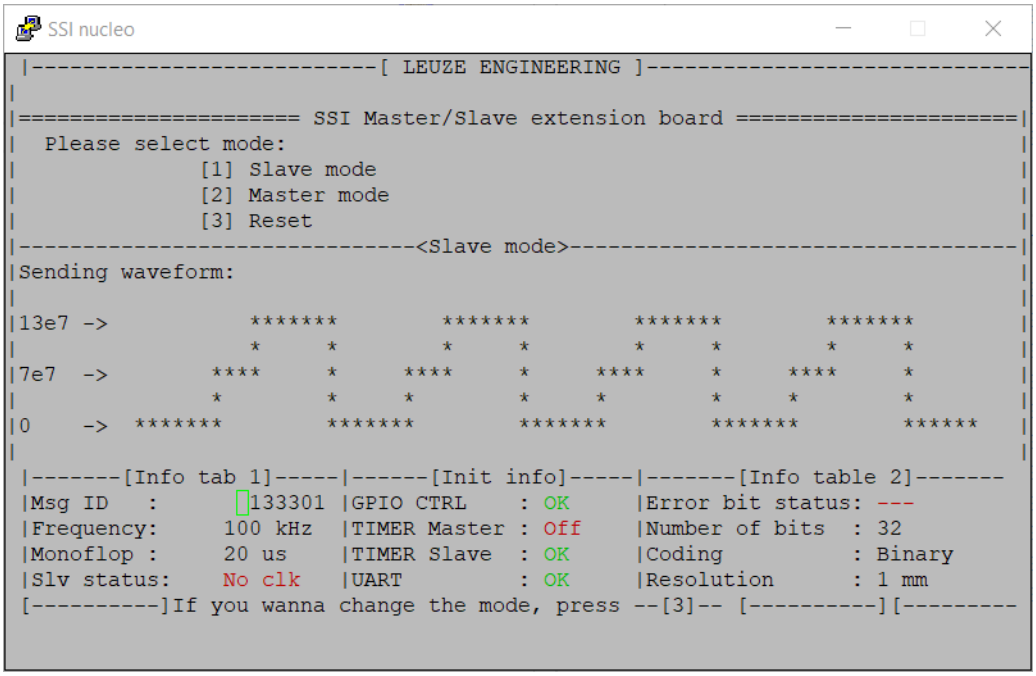
Obrázek 27: Výběr rozlišení



Obrázek 28: Výběr kódování



Obrázek 29: Výběr příslušného módu



Obrázek 30: Menu salve módu při stavu iddle, kdy nejsou generovány hodinové signály master zařízením

Příloha D - Doxygen dokumentace k master/slave zařízení

SSI_extension

Generated by Doxygen 1.8.14

Contents

1 Todo List

Member Timer3::Timer3 (uint32_t prsc, uint32_t per)

check the types of clock sources!!!

2 Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

3 Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

4 File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

5 Class Documentation

5.1 Buffer Class Reference

Public Member Functions

- void put (uint32_t value)
- uint32_t **get** ()
- int **count** ()
- void **setWritePtr** (int wp)
- void **setReadPtr** (int rp)
- int **getWritePtr** ()
- int **getReadPtr** ()
- void **resetBuff** ()
- void **bitCorrection** (int bits)

5.1.1 Member Function Documentation

5.1.1.1 put()

```
void Buffer::put (
            uint32_t value )
```

writePtr =0; SIZE = 32; 1%32 = 1 2%32 = 2 3%32 = 3 4%32 = 4 ... 17%32 = 17

The documentation for this class was generated from the following files:

- CircBuff.h
- CircBuff.cpp

5.2 Data2 Class Reference

Public Types

- enum **ports** { **PORT_A** = 0, **PORT_B**, **PORT_C**, **PORT_D** }

Public Member Functions

- void CheckPortAndIndex (uint32_t number, int i)

  Checking the the port and index of particular pin using lookup table.
- uint32_t getData (int i)

  Getter which get the data from particular pin.
- uint32_t ProcessDataMaster ()

  Works only in master mode Reads and processes data from SIPO.
- void **readDataReg** ()
- uint32_t getNum ()

  Method for getting particcular num, which is stored in SIPO regisetrs.
- uint32_t getNumero ()

  Getter for returning variable numero.
- uint32_t **setDataValaue** ()
- void setData (uint32_t number, int i)

  Setter for setting data to particular pin.
- void ProcessDataSlave (uint32_t number)

  Works only in slave mode Writes and processes data to SIPO.

Public Attributes

- uint32_t **bitnumber** =25

Static Public Attributes

- static const int **SIZE** = 32

Protected Attributes

- const uint8_t **dt0** = 0x29
- const uint8_t **dt1** = 0x18
- const uint8_t **dt2** = 0x19
- const uint8_t **dt3** = 0x06
- const uint8_t **dt4** = 0x07
- const uint8_t **dt5** = 0x16
- const uint8_t **dt6** = 0x27
- const uint8_t **dt7** = 0x09
- const uint8_t **dt8** = 0x08
- const uint8_t **dt9** = 0x04
- const uint8_t **dt10** = 0x14
- const uint8_t **dt11** = 0x15
- const uint8_t **dt12** = 0x0A
- const uint8_t **dt13** = 0x24
- const uint8_t **dt14** = 0x1D
- const uint8_t **dt15** = 0x1E
- const uint8_t **dt16** = 0x1F
- const uint8_t **dt17** = 0x11
- const uint8_t **dt18** = 0x12
- const uint8_t **dt19** = 0x1C
- const uint8_t **dt20** = 0x0B
- const uint8_t **dt21** = 0x0C
- const uint8_t **dt22** = 0x25
- const uint8_t **dt23** = 0x21
- const uint8_t **dt24** = 0x28
- const uint8_t **dt25** = 0x2A
- const uint8_t **dt26** = 0x2C
- const uint8_t **dt27** = 0x32
- const uint8_t **dt28** = 0x2D
- const uint8_t **dt29** = 0x20
- const uint8_t **dt30** = 0x21
- const uint8_t **dt31** = 0x05
- uint8_t **LookUpTable** [SIZE]
- bool **DataVal** [SIZE]
- GPIO_TypeDef * **PORT**
- uint8_t **port**
- uint8_t **index**

5.2.1 Member Function Documentation

5.2.1.1 CheckPortAndIndex()

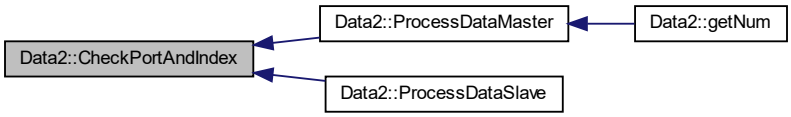
```
void Data2::CheckPortAndIndex (
            uint32_t number,
            int i )
```

Checking the the port and index of particular pin using lookup table.

Parameters

number	
i	

Here is the caller graph for this function:



5.2.1.2 getData()

```
uint32_t Data2::getData (
            int i )
```

Getter which get the data from particular pin.

I should implement following:

- buffer element end read data during interrupt
- parse the data in main function !!!

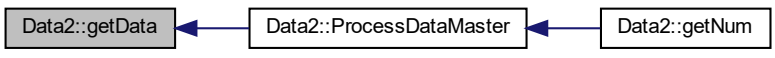
Parameters

i	

Returns

uint32_t

Here is the caller graph for this function:



5.2.1.3 getNum()

```
uint32_t Data2::getNum ( )
```

Method for getting particcular num, which is stored in SIPO regisetrs.

Returns

uint32_t

Here is the call graph for this function:

5.2.1.4 getNumero()

```
uint32_t Data2::getNumero ( )
```

Getter for returning variable numero.

Returns

uint32_t

5.2.1.5 ProcessDataMaster()

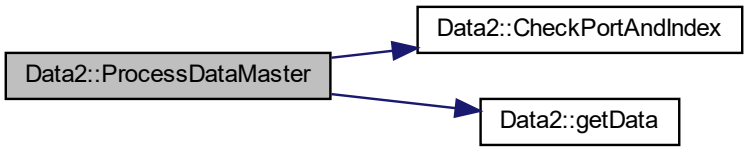
```
uint32_t Data2::ProcessDataMaster ( )
```

Works only in master mode Reads and processes data from SIPO.

Returns

uint32_t

Here is the call graph for this function:



Here is the caller graph for this function:

5.2.1.6 ProcessDataSlave()

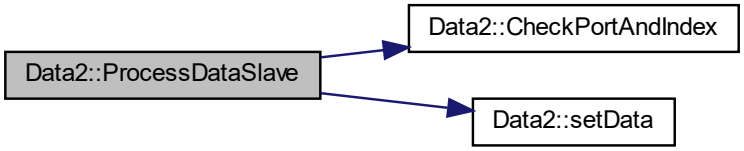
```
void Data2::ProcessDataSlave (
            uint32_t number )
```

Works only in slave mode Writes and processes data to SIPO.

Parameters

number	

Here is the call graph for this function:



5.2.1.7 setData()

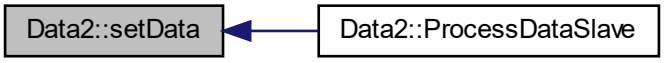
```
void Data2::setData (
            uint32_t number,
            int i )
```

Setter for setting data to particular pin.

Parameters

number	
i	

Here is the caller graph for this function:



5.2.2 Member Data Documentation

5.2.2.1 LookUpTable

```
uint8_t Data2::LookUpTable[SIZE]  [protected]
```

Initial value:

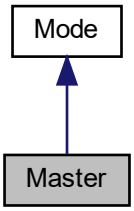
```
=
    {
     dt0, dt1, dt2, dt3, dt4, dt5 , dt6, dt7,
     dt8, dt9, dt10, dt11, dt12, dt13 , dt14,
     dt15, dt16, dt17, dt18, dt19, dt20, dt21,
     dt22, dt23, dt24, dt25, dt26, dt27, dt28,
     dt29, dt30, dt31
    }
```

The documentation for this class was generated from the following files:

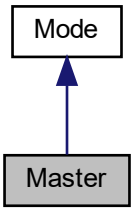
- data2.h
- data2.cpp

5.3 Master Class Reference

Inheritance diagram for Master:



Collaboration diagram for Master:



Public Member Functions

- void ModeSet (SsiGpio &gpio)

  Method which sets and starts master mode.

5.3.1 Member Function Documentation

5.3.1.1 ModeSet()

```
void Master::ModeSet (
            SsiGpio & gpio )  [virtual]
```

Method which sets and starts master mode.

Parameters

gpio	

Implements Mode.

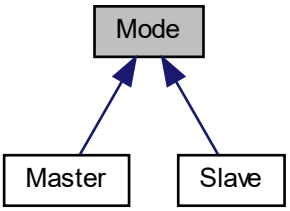
The documentation for this class was generated from the following files:

- modes.h
- modes.cpp

5.4 Mode Class Reference

```
#include <modes.h>
```

Inheritance diagram for Mode:



Public Member Functions

- virtual void **ModeSet** (SsiGpio &gpio)=0

5.4.1 Detailed Description

Modes of SSI .h

Author

Jan Kohout

The documentation for this class was generated from the following file:

- modes.h

5.5 Print Class Reference

Public Types

- enum **Frequency** { **F100kHz** = '0', **F300kHz** = '1', **F500kHz** = '2' }
- enum **Decimal** { **ZERO_DEC** = 0, **ONE_DEC**, **TWO_DEC** }
- enum **Coding** { **BINARY** = false, **GRAY** = true }
- enum **Bits** { **B25** =true, **B32** =false }
- enum **SendSlave** { **Send0** = '0', **Send1** = '1', **SendError** = '2' }

Public Member Functions

- Print ()
  Construct a new Print:: Print object.
- void ShowMenu ()
  Method for showing main menu.
- void ShowMaster ()
  Method for showing the master mode in UI.
- void Print_xy (const char x[ ], const char y[ ])
  Select the position of the cursor.
- void PrintStart ()
  Method which shows the start window in UI.
- void Print_xy_zero ()
  Set cursor to zero.
- void PrintWrongInput ()
  Method which points out user to wrong input.
- void ShowSlave ()
  Method for showing the slave mode in UI.
- void PrintInt (unsigned int Integer)
  This method prints integer number.
- void PrintIntOne (unsigned int Integer)
  This method prints integer numer in fix point and adds decimal point.
- void PrintIntTwo (unsigned int Integer)
  This method also prints integer number in fix point and adds decimal point.
- void SelectDec ()
  Method which show window for selection of resolutin in UI.
- void SelectCoding ()

Prints window for coding selection.

- bool getFlagZero ()

  Is resolition 1 mm is selected?
- bool getFlagOne ()

  Is resolutin 0.1 mm selected?
- bool getFlagTwo ()

  Is resolutin 0.01 mm selected?
- void **setFlag** (Decimal select_dec)
- bool getFlagCoding ()

  Which coding is selected.
- void **setFlagCoding** (Coding select_code)
- void SelectBit ()

  Prints window for num of bits selection in UI.
- void **setNumOfBits** ()
- uint32_t getNumOfBits ()

  Getter for getting number of bits.
- void SelectFreq (void)

  Prints window for frequecy selection in UI.
- uint32_t getTimerPeriod (void)

  Getter for getting the timer period.
- uint32_t getTimerPrescaler (void)

  getter for getting prescaler
- void getInfoCoding (void)

  Info about coding.
- void getInfoBits (void)

  Info about bits.
- void getInfoResolution (void)

  Info about resolutin.
- void PrintInfoTable (void)

  This method prints info tables.
- void getInfoErrorBitPlace ()

  set place in UI window for info about error bit
- void getInfoErrorBit (uint32_t x)

  Info about error bit.
- void getInfoGpio (bool x)

  Show info about gpio settings.
- void getInfoTimer (bool x)

  Show info about master timer.
- void getInfoTimerSlave (bool x)

  Show info about slave timer.
- void getInfoUart (bool x)

  Info about UART initialization.
- void getInfoFreq (void)

  get info about selected frequency
- void PrintMsgNum (uint32_t cnt)

  Number of msg id.
- void setSendNum ()

  method which sends the number in slave mode
- uint32_t getSlaveData (void) const

  Getting data in salve mode.
- bool getDataSetSlaveFlag (void) const

Info about getting data in slave mode.

- void **setDataSetSlaveFlag** (bool set)
- void **DefaultSendDataSelection** ()
- void **SlvStatus** ()

Public Attributes

- bool **coding** = false

5.5.1 Member Function Documentation

5.5.1.1 getDataSetSlaveFlag()

```
bool Print::getDataSetSlaveFlag (
            void  ) const
```

Info about getting data in slave mode.

Returns

true
false

5.5.1.2 getFlagCoding()

```
bool Print::getFlagCoding ( )
```

Which coding is selected.

Returns

true - gray
false - bin

5.5.1.3 getFlagOne()

```
bool Print::getFlagOne ( )
```

Is resolutin 0.1 mm selected?

Returns

true
false

5.5.1.4 getFlagTwo()

```
bool Print::getFlagTwo ( )
```

Is resolutin 0.01 mm selected?

Returns

true
false

5.5.1.5 getFlagZero()

```
bool Print::getFlagZero ( )
```

Is resolition 1 mm is selected?

Returns

true
false

5.5.1.6 getInfoErrorBit()

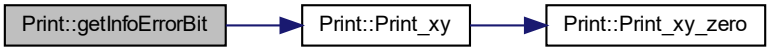
```
void Print::getInfoErrorBit (
            uint32_t x )
```

Info about error bit.

Parameters

x	

Here is the call graph for this function:



5.5.1.7 getInfoGpio()

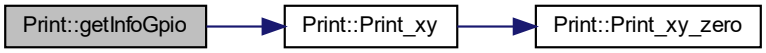
```
void Print::getInfoGpio (
            bool x )
```

Show info about gpio settings.

Parameters

x	

Here is the call graph for this function:



5.5.1.8 getInfoTimer()

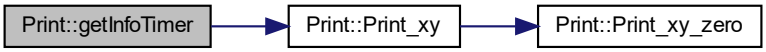
```
void Print::getInfoTimer (
            bool x )
```

Show info about master timer.

Parameters

x	

Here is the call graph for this function:



5.5.1.9 getInfoTimerSlave()

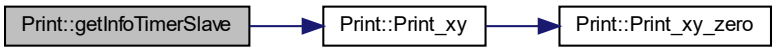
```
void Print::getInfoTimerSlave (
            bool x )
```

Show info about slave timer.

Parameters

x	

Here is the call graph for this function:



5.5.1.10 getInfoUart()

```
void Print::getInfoUart (
            bool x )
```

Info about UART initialization.

Parameters

x	

Here is the call graph for this function:

5.5.1.11 getNumOfBits()

```
uint32_t Print::getNumOfBits ( )
```

Getter for getting number of bits.

Returns

uint32_t

5.5.1.12 getSlaveData()

```
uint32_t Print::getSlaveData (
            void  ) const
```

Getting data in salve mode.

Returns

uint32_t

5.5.1.13 getTimerPeriod()

```
uint32_t Print::getTimerPeriod (
            void  )
```

Getter for getting the timer period.

Returns

uint32_t

5.5.1.14 getTimerPrescaler()

```
uint32_t Print::getTimerPrescaler (
            void  )
```

getter for getting prescaler

Returns

uint32_t

5.5.1.15 Print_xy()

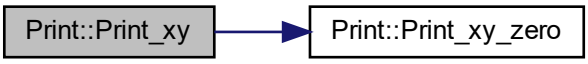
```
void Print::Print_xy (
            const char x[],
            const char y[] )
```

Select the position of the cursor.

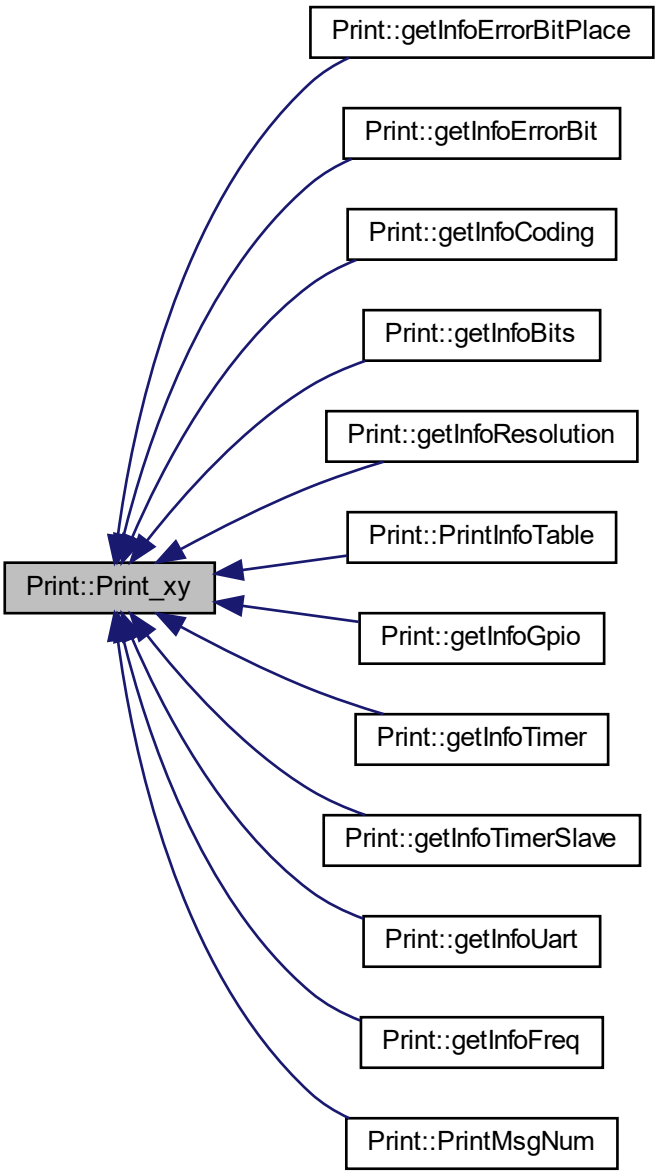
Parameters

x	
y	

Here is the call graph for this function:



Here is the caller graph for this function:



5.5.1.16 PrintInt()

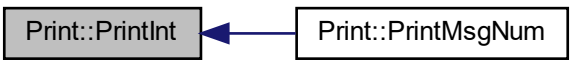
```
void Print::PrintInt (
            unsigned int Integer )
```

This method prints integer number.

Parameters

Integer	

Here is the caller graph for this function:



5.5.1.17 PrintIntOne()

```
void Print::PrintIntOne (
            unsigned int Integer )
```

This method prints integer numer in fix point and adds decimal point.

Parameters

Integer	

5.5.1.18 PrintIntTwo()

```
void Print::PrintIntTwo (
            unsigned int Integer )
```

This method also prints integer number in fix point and adds decimal point.

Parameters

Integer	

5.5.1.19 PrintMsgNum()

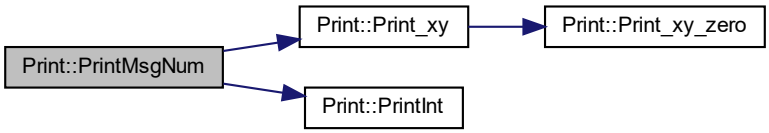
```
void Print::PrintMsgNum (
            uint32_t cnt )
```

Number of msg id.

Parameters

cnt	

Here is the call graph for this function:

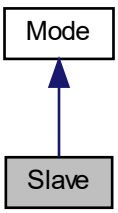


The documentation for this class was generated from the following files:

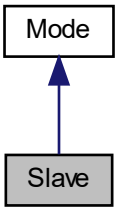
- print.h
- print.cpp

5.6 Slave Class Reference

Inheritance diagram for Slave:



Collaboration diagram for Slave:



Public Member Functions

- void ModeSet (SsiGpio &gpio)
  Method which sets and start the Slave mode.

5.6.1 Member Function Documentation

5.6.1.1 ModeSet()

```
void Slave::ModeSet (
            SsiGpio & gpio )  [virtual]
```

Method which sets and start the Slave mode.

Parameters

gpio	

Implements Mode.

The documentation for this class was generated from the following files:

- modes.h
- modes.cpp

5.7 SsiGpio Class Reference

Class for setting of the gpio pins.

```
#include <ssigpio.h>
```

Friends

- class **Slave**
- class **Master**

5.7.1 Detailed Description

Class for setting of the gpio pins.

The documentation for this class was generated from the following files:

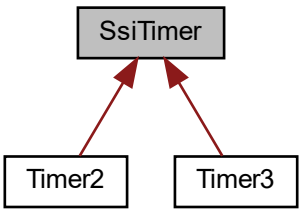
- ssigpio.h
- ssigpio.cpp

5.8 SsiTimer Class Reference

Class for setting of the timers.

`#include <ssitimer.h>`

Inheritance diagram for SsiTimer:



Static Public Member Functions

- static void **ssiTimer2init** (TIM_TypeDef *timer_port, uint32_t arp, uint32_t clk_div, uint32_t cnt_mode, uint32_t per, uint32_t prsc)
- static void **ssiTimer2startIT** ()
- static void **ssiOCStart** ()
- static void **ssiTimer3init** (TIM_TypeDef *time_port, uint32_t prsc, uint32_t cnt_mode, uint32_t per, uint32_t clk_div, uint32_t arp)
- static void **ssiICStartIT** (void)

Friends

- class **Timer2**

5.8.1 Detailed Description

Class for setting of the timers.

The documentation for this class was generated from the following file:

- ssitimer.h

5.9 SsiUart Class Reference

```
#include <ssiuart.h>
```

Public Member Functions

- **SsiUart** (const SsiUart &)=delete
- void ssiUARTWordLen (uint32_t &word_len)

  This method configs word length Two options:
- void ssiUARTStopBits (uint32_t &stop_bits)

  This function set up the stop bits for UART.
- void ssiUARTParity (uint32_t &parity_bits)

  This method sets UART parity.
- void ssiUARTOverSampling (uint32_t &over_sampling)

  This method sets the oversampling of UART.
- void ssiUARTtransreceive (uint32_t &trans_or_rec)

  This method select in which mode UART works.

Static Public Member Functions

- static SsiUart & GetInstance ()

  Singleton.

Public Attributes

- bool **infoUart** = true

5.9.1 Detailed Description

Tiner driver .h file

Author

Jan Kohout

5.9.2 Member Function Documentation

5.9.2.1 GetInstance()

```
SsiUart & SsiUart::GetInstance ( )  [static]
```

Singleton.

Returns

SsiUart&

5.9.2.2 ssiUARTOverSampling()

```
void SsiUart::ssiUARTOverSampling (
            uint32_t & over_sampling )
```

This method sets the oversampling of UART.

This function sets the oversampling of uart

- oversampling by 16
- oversamplink by 8
- oversampling by 16
- oversamplink by 8

Parameters

over_sampling	

5.9.2.3 ssiUARTParity()

```
void SsiUart::ssiUARTParity (
            uint32_t & parity_bits )
```

This method sets UART parity.

This function sets UART Parity

- NONE_PARITY
- EVEN_PARITY
- ODD_PARITY
- NONE_PARITY
- EVEN_PARITY
- ODD_PARITY

Parameters

parity_bits	

5.9.2.4 ssiUARTStopBits()

```
void SsiUart::ssiUARTStopBits (
            uint32_t & stop_bits )
```

This function set up the stop bits for UART.

- 1 stop bit
- 0.5 stop bit
- 2 stop bits
- 1.5 stop bits

Parameters

stop_bits	

5.9.2.5 ssiUARTtransreceive()

```
void SsiUart::ssiUARTtransreceive (
            uint32_t & trans_or_rec )
```

This method select in which mode UART works.

- Transmitter mode
- Receiver mode
- Transmiter and receiver mode

Parameters

trans_or_rec	

5.9.2.6 ssiUARTWordLen()

```
void SsiUart::ssiUARTWordLen (
            uint32_t & word_len )
```

This method configs word length Two options:

- |WORD_LEN_8B| - 8 bits
- |WORD_LEN_9B| - 9 bits

Parameters

word_len	

More information in datasheet page –893–

The documentation for this class was generated from the following files:

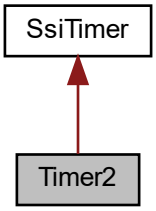
- ssiuart.h
- ssiuart.cpp

5.10 Timer2 Class Reference

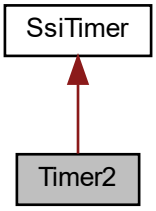
Class for setting of timer 2.

`#include <ssitimer.h>`

Inheritance diagram for Timer2:



Collaboration diagram for Timer2:



Public Member Functions

- Timer2 (uint32_t per, uint32_t prsc)

  Construct a new Timer 2:: Timer 2 object.
- ~Timer2 ()

  Destroy the Timer 2:: Timer 2 object.
- void Timer2End ()

  timer 2 end

Public Attributes

- bool **infoTimer2** = false

5.10.1 Detailed Description

Class for setting of timer 2.

5.10.2 Constructor & Destructor Documentation

5.10.2.1 Timer2()

```
Timer2::Timer2 (
            uint32_t per,
            uint32_t prsc )
```

Construct a new Timer 2:: Timer 2 object.

Periph. clk (180 MHz) Timer freq = ----------——— Prescaler ∗ auto reload register

```
Periph. clk
```

ARR ∗ PRSC = ----------——— Desired freq Periph. clk (180 MHz) Timer freq = ----------—— Prescaler ∗ auto reload register

```
Periph. clk
```

ARR ∗ PRSC = ----------— Desired freq

Parameters

per	
prsc	

The documentation for this class was generated from the following files:

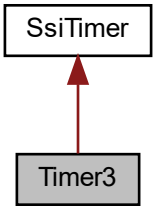
- ssitimer.h
- ssitimer.cpp

5.11 Timer3 Class Reference

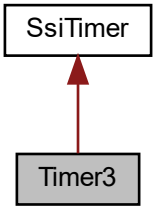
Class for setting of timer 3.

`#include <ssitimer.h>`

Inheritance diagram for Timer3:



Collaboration diagram for Timer3:



Public Member Functions

- Timer3 (uint32_t prsc, uint32_t per)

  Construct a new Timer 3:: Timer 3 object Constructor for timer 3.
- ~Timer3 ()

  Destroy the Timer 3:: Timer 3 object.

Public Attributes

- bool **infoTimer3** = false

5.11.1 Detailed Description

Class for setting of timer 3.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 Timer3()

```
Timer3::Timer3 (
            uint32_t per,
            uint32_t prsc )
```

Construct a new Timer 3:: Timer 3 object Constructor for timer 3.

Parameters

per	
prsc	

Todo check the types of clock sources!!!

The documentation for this class was generated from the following files:

- ssitimer.h
- ssitimer.cpp

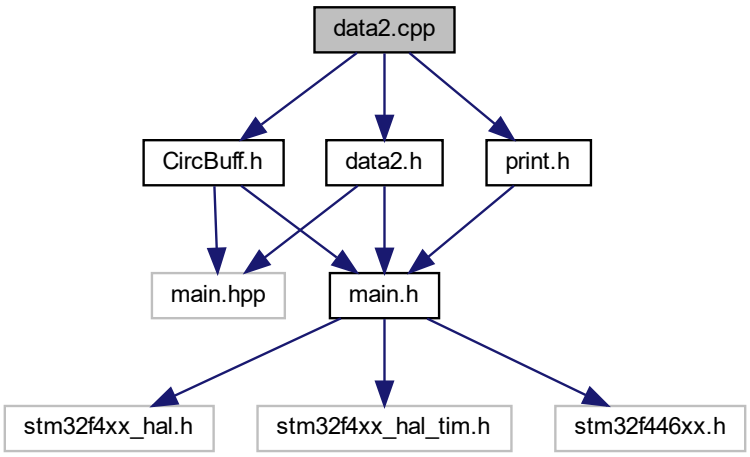
6 File Documentation

6.1 data2.cpp File Reference

.cpp file for class Data2

```
#include "data2.h"
#include "CircBuff.h"
#include "print.h"
```

Include dependency graph for data2.cpp:



6.1.1 Detailed Description

.cpp file for class Data2

Author

Jan Kohout (jkohout@leuze.com)

Version

0.1

Date

2022-04-30

Copyright

Copyright (c) 2022

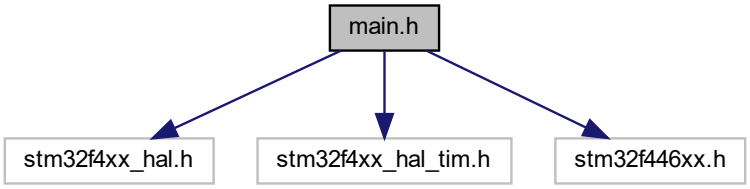
6.2 main.h File Reference

: Header for main.c file. This file contains the common defines of the application.

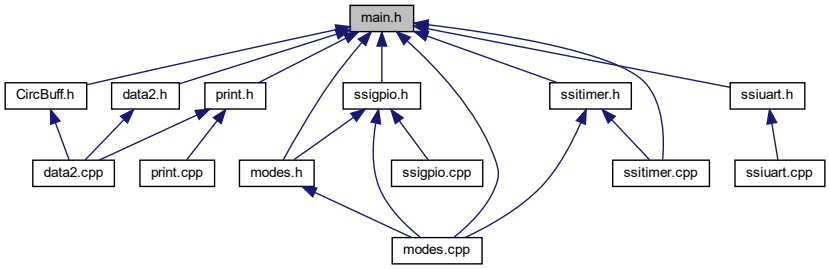
```
#include "stm32f4xx_hal.h"
#include "stm32f4xx_hal_tim.h"
```

```
#include "stm32f446xx.h"
```

Include dependency graph for main.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define **PRINT**(x) HAL_UART_Transmit(&huart2,(uint8_t∗)x, sizeof(x), 15)
- #define **B1_Pin** GPIO_PIN_13
- #define **B1_GPIO_Port** GPIOC
- #define **USART_TX_Pin** GPIO_PIN_2
- #define **USART_TX_GPIO_Port** GPIOA
- #define **USART_RX_Pin** GPIO_PIN_3
- #define **USART_RX_GPIO_Port** GPIOA
- #define **LD2_Pin** GPIO_PIN_5
- #define **LD2_GPIO_Port** GPIOA
- #define **TMS_Pin** GPIO_PIN_13
- #define **TMS_GPIO_Port** GPIOA
- #define **TCK_Pin** GPIO_PIN_14
- #define **TCK_GPIO_Port** GPIOA
- #define **SWO_Pin** GPIO_PIN_3
- #define **SWO_GPIO_Port** GPIOB
- #define **SIZE_OF_DATA** 32

Functions

- void **HAL_TIM_MspPostInit** (TIM_HandleTypeDef ∗htim)
- void Error_Handler (void)

  This function is executed in case of error occurrence.

Variables

- unsigned char **end**
- unsigned char **reset**
- uint32_t **tim2Period**
- uint32_t **tim2Prescaler**
- uint32_t **bitnum**

6.2.1 Detailed Description

: Header for main.c file. This file contains the common defines of the application.

Attention

6.2.2 Function Documentation

6.2.2.1 Error_Handler()

```
void Error_Handler (
             void  )
```

This function is executed in case of error occurrence.

Return values

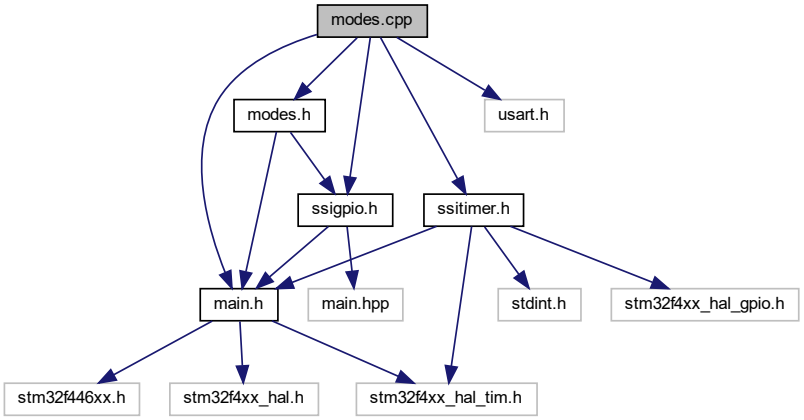
None	

6.3 modes.cpp File Reference

```
#include "main.h"
#include "modes.h"
#include "ssigpio.h"
#include "ssitimer.h"
```

```
#include "usart.h"
```

Include dependency graph for modes.cpp:



6.3.1 Detailed Description

Author

Jan Kohout (jkohout@leuze.com)

Version

0.1

Date

2022-04-30

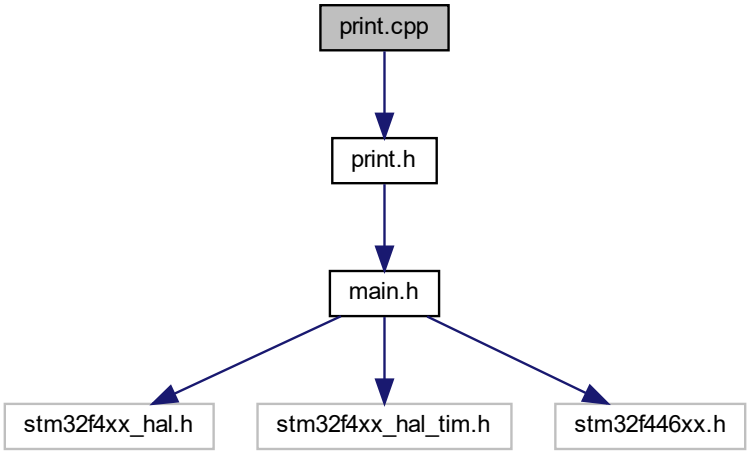
Copyright

Copyright (c) 2022

6.4 print.cpp File Reference

```
#include "print.h"
```

Include dependency graph for print.cpp:



Variables

- UART_HandleTypeDef **huart2**

6.4.1 Detailed Description

Author

Jan Kohout (jkohout@leuze.com)

Version

0.1

Date

2022-04-30

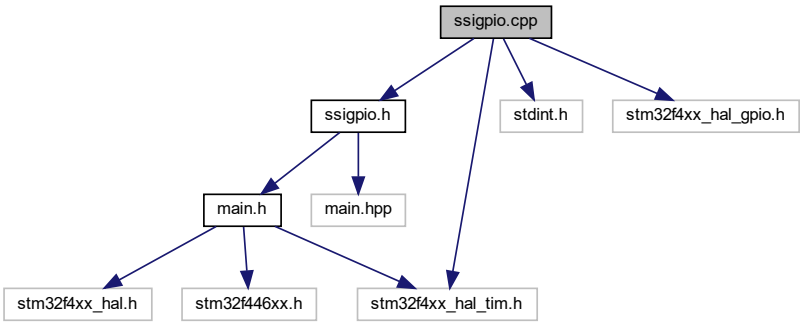
Copyright

Copyright (c) 2022

6.5 ssigpio.cpp File Reference

```
#include "ssigpio.h"
#include <stdint.h>
#include "stm32f4xx_hal_gpio.h"
#include "stm32f4xx_hal_tim.h"
```

Include dependency graph for ssigpio.cpp:



Variables

- uint32_t **pin_2_position** [16]
- uint32_t **pin_1_position** [16]

6.5.1 Detailed Description

Author

Jan Kohout (jkohout@leuze.com)

Version

0.1

Date

2022-04-30

Copyright

Copyright (c) 2022

6.5.2 Variable Documentation

6.5.2.1 pin_1_position

```
uint32_t pin_1_position[16]
```

Initial value:

```
= {
    (0x00),
    (0x01),
    (0x02),
    (0x03),
    (0x04),
    (0x05),
    (0x06),
    (0x07),
    (0x08),
    (0x09),
    (0x0A),
    (0x0B),
    (0x0C),
    (0x0D),
    (0x0E),
    (0x0F)

}
```

6.5.2.2 pin_2_position

```
uint32_t pin_2_position[16]
```

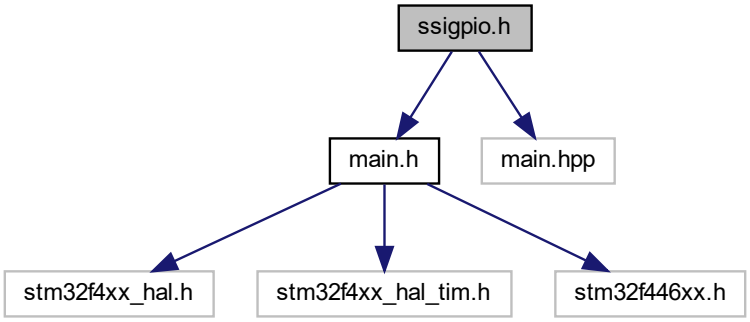
Initial value:

```
= {
    (0x00),
    (0x02),
    (0x04),
    (0x06),
    (0x08),
    (0x0A),
    (0x0C),
    (0x0E),
    (0x10),
    (0x12),
    (0x14),
    (0x16),
    (0x18),
    (0x1A),
    (0x1C),
    (0x1E),
}
```

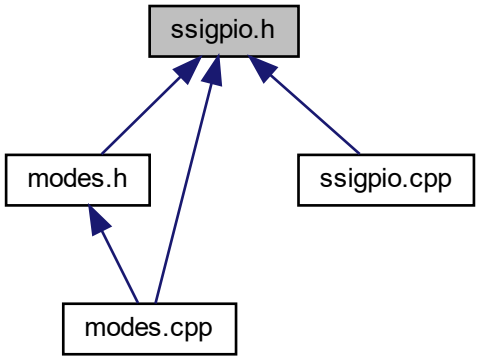
6.6 ssigpio.h File Reference

```
#include "main.h"
#include "main.hpp"
```

Include dependency graph for ssigpio.h:



This graph shows which files directly or indirectly include this file:



Classes

- class SsiGpio

  Class for setting of the gpio pins.

Macros

- #define **GPIO_CLK_ENB_A** (RCC->AHB1ENR |= (1<<0))
- #define **GPIO_CLK_ENB_B** (RCC->AHB1ENR |= (1<<1))
- #define **GPIO_CLK_ENB_C** (RCC->AHB1ENR |= (1<<2))
- #define **GPIO_CLK_ENB_D** (RCC->AHB1ENR |= (1<<3))
- #define **GPIO_CLK_ENB_E** (RCC->AHB1ENR |= (1<<4))
- #define **GPIO_CLK_ENB_F** (RCC->AHB1ENR |= (1<<5))
- #define **GPIO_CLK_ENB_G** (RCC->AHB1ENR |= (1<<6))
- #define **GPIO_CLK_ENB_H** (RCC->AHB1ENR |= (1<<7))
- #define **POS_BIT1** (pin_2_position[pinNumber])
- #define **POS_BIT2** (pin_2_position[pinNumber]+1)

6.6.1 Detailed Description

Author

Jan Kohout (jkohout@leuze.com)

Version

0.1

Date

2022-04-30

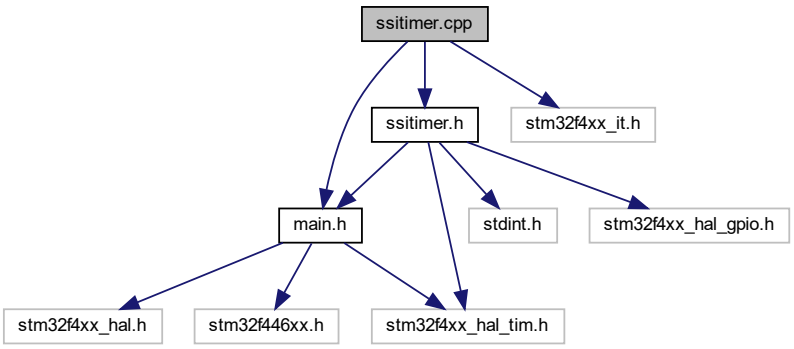
Copyright

6.7 ssitimer.cpp File Reference

HAL for timers.

```
#include "main.h"
#include "ssitimer.h"
#include "stm32f4xx_it.h"
```

Include dependency graph for ssitimer.cpp:



6.7.1 Detailed Description

HAL for timers.

Author

Jan Kohout (jkohout@leuze.com)

Version

0.1

Date

2022-04-30

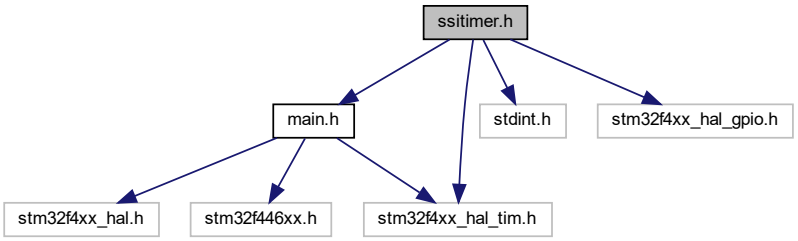
Copyright

6.8 ssitimer.h File Reference

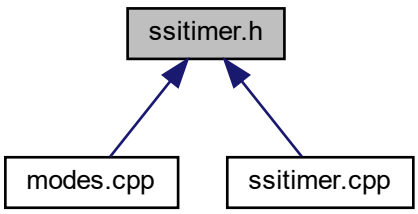
HAL for timers.

```
#include "main.h"
#include <stdint.h>
#include "stm32f4xx_hal_gpio.h"
#include "stm32f4xx_hal_tim.h"
```

Include dependency graph for ssitimer.h:



This graph shows which files directly or indirectly include this file:



Classes

- class SsiTimer

  Class for setting of the timers.
- class Timer2

  Class for setting of timer 2.
- class Timer3

  Class for setting of timer 3.

Variables

- TIM_HandleTypeDef **timer2**
- TIM_HandleTypeDef **timer3**

6.8.1 Detailed Description

HAL for timers.

Author

Jan Kohout (jkohout@leuze.com)

Version

0.1

Date

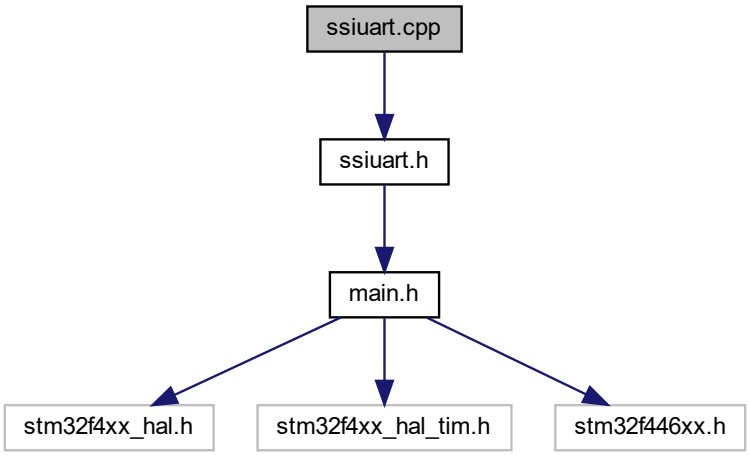
2022-04-30

Copyright

6.9 ssiuart.cpp File Reference

HAL for UART peripheral.

```
#include "ssiuart.h"
```
Include dependency graph for ssiuart.cpp:



6.9.1 Detailed Description

HAL for UART peripheral.

Author

Jan Kohout (jkohout@leuze.com)

Version

0.1

Date

2022-04-30

Copyright

Index