BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING

FAKULTA STROJNÍHO INŽENÝRSTVÍ

INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS

ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

CONFIGURABLE SPI DEVICE

KONFIGUROVATELNÉ SPI ZAŘÍZENÍ

MASTER'S THESIS

DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. Pavol Ženčár
SUPERVISOR VEDOUCÍ PRÁCE	doc. Ing. Jiří Krejsa, Ph.D.

BRNO 2021

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF MECHANICAL ENGINEERING

Assignment Master's Thesis

Institut:	Institute of Solid Mechanics, Mechatronics and Biomechanics
Student:	**Bc. Pavol Ženčár**
Degree programm:	Applied Sciences in Engineering
Branch:	Mechatronics
Supervisor:	**doc. Ing. Jiří Krejsa, Ph.D.**
Academic year:	2020/21

As provided for by the Act No. 111/98 Coll. on higher education institutions and the BUT Study and Examination Regulations, the director of the Institute hereby assigns the following topic of Master's Thesis:

Configurable SPI device

Brief Description:

Serial peripheral interface (SPI) is a synchronous serial communication interface specification used for short–distance communication, primarily in embedded systems. For the purposes of development and testing of system containing SPI slave elements it is useful to have the possibility to simulate the behavior of real devices – for example the memory – and therefore invoke the behaviour of the whole system, which would be difficult to reach otherwise. The connection with PC and data presentation in appropriate software are inseparable parts of the thesis goal.

Master's Thesis goals:

1. Get acquainted with the properties of SPI and its variants.
2. Design the device that can simulate the behavior of SPI slave device that can be controlled and configured via USB.
3. Design proper method of real device behavior simulation.
4. Select existing device that supports SPI communication and prepare its virtual replacement.
5. Evaluate usability of proposed solution for different types of tasks, describe the device in question with respect to achievable data transfer speed.

Recommended bibliography:

AXELSON J., USB Complete: Everything You Need to Develop Custom USB Peripherals, Lakeview Research; 3rd Edition, 2005.

Deadline for submission Master's Thesis is given by the Schedule of the Academic year 2020/21

In Brno,

L. S.

prof. Ing. Jindřich Petruška, CSc. Director of the Institute	doc. Ing. Jaroslav Katolický, Ph.D. FME dean

Abstrakt

Táto práca sa zaoberá vývojom a testovaním konfigurovateľného SPI slave zariadenia, ktorého hlavným účelom je testovanie reálnych SPI master zariadení. Zariadenie je implementované pomocou STM32 mikrokontroléra na vývojovej doske NUCLEO. Medzi základné funkcionality zariadenia patrí meranie frekvencie SPI prenosu, stream mód, ktorý umožní umožní presne sledovať, čo pripojené master zariadenie posiela po SPI zbernici a LUT mód, ktorý umožňuje nakonfigurovať zariadenie tak, aby odpovedalo prednastavenými odpoveďami na aktuálnu príchodziu správu. Rozšírená funkcionalita sa skladá z EEPROM emulátor módu. V tomto móde sa zariadenia správa ako virtuálna náhrada reálneho EEPROM zariadenia. Zariadenie je pripojené k počítaču pomocou sériového portu a je možné toto zariadenie konfigurovať pomocou python programáterského rozhrania. Zariadenie taktiež hlási každú aktivitu na SPI zbernici tomuto python programátorskému rozhraniu.

Summary

This thesis deals with the development and testing of a configurable SPI slave device, the main purpose of which is to test real SPI master devices. The device is implemented using STM32 microcontroller on NUCLEO development board. The basic functionalities of the device include measuring frequency of the SPI transmission, stream mode, which allows the user to accurately monitor what the connected master device is transmitting over the SPI interface, and LUT mode, which allows the user to configure the device to respond with preset responses according to the current incoming message. More advanced functionality consists of an EEPROM emulator mode. In this mode, the device behaves as a virtual replacement of a real EEPROM device. The device is connected to a personal computer using a serial port and it is possible to configure the device using a python programming interface. The device also reports every activity on the SPI interface to this python programming interface.

Klíčová slova

SPI, mikrokontrolér, EEPROM pamäť, DMA

Keywords

SPI, microcontroler, EEPROM memory, DMA

Bibliographic citation

ŽENČÁR, P. *Configurable SPI device.* Brno: Brno University of Technology, Faculty of Mechanical Engineering, 2021. 122 pages, Master's thesis supervisor: doc. Ing. Jiří Krejsa, PhD..

Rozšírený abstrakt

Motivácia

SPI slave zariadenia sú vo väčšine prípadov rôzne typy senzorov. Preto počas vývinu ľubovoľného produktu, ktorý používa senzory, je dosť pravdepodobné, že vývojár bude musieť vyvinúť SPI master zariadenie, ktoré je spojené s týmito SPI slave senzormi. Jediný reálny spôsob ako otestovať tieto SPI master zariadenia je použiť logický analyzér, pretože vývojár nemá prístup do SPI slave zariadení a nemôže sa "pozrieť dovnútra" pomocou debugging softvéru alebo niečoho podobného. Testovanie logickým analyzérom má svoje limitácie. Pomocou logického analyzéra je možné presne sledovať, čo sa deje na SPI signálových zberniciach, ale hociaká logika zariadenia nemôže byť otestovaná. Aby bolo možné otestovať logiku SPI master zariadenia, je potrebné použiť iné SPI slave zariadenie. Cieľom tejto práce je vyvinúť toto SPI slave zariadenie, s ktorým je možné testovať SPI master zariadenia. Bolo by užitočné, aby toto zariadenie bolo možné jednoducho nakonfigurovať tak, aby základná logika a funkcionalita SPI master zariadení mohla byť otestovaná.

Na detailné testovanie SPI master zariadenia, ktoré slúži na ovládanie jedného konkrétneho SPI slave zariadenia je potrebné simulovať chovanie reálneho SPI slave zariadenia podrobnejšie a ideálne je potreba vytvoriť jeho model, ktorý ma podobné až identické chovanie ako reálne zariadenie. Tento model nie je možné vytvoriť univerzálnym spôsobom aby zahŕňal všetky SPI slave zariadenia, preto bolo vybrané jedno reálne SPI slave zariadenie a bol vytvorený model tohto zariadenia. Účel tohto modelu je taký, aby tento model mohol byť považovaný za virtuálnu náhradu reálneho SPI slave zariadenia, ktorá má rovnaké správanie ako reálne zariadenie, môže byť konfigurovateľná a je možné sledovať, čo sa aktuálne deje vo vnútri tohto zariadenia. Tento model môže byť použitý na vývoj SPI master zariadenia, keď reálne SPI slave zariadenie je ťažko dostupné.

Ciele

Cieľom tejto práce je vytvoriť zariadenie, ktoré sa bude chovať ako generické SPI slave zariadenie. Je potrebné vyvinúť metódy, s ktorými bude možné nasimulovať najzákladnejšie SPI slave funkcionality s jednoduchou konfiguráciou. Ďalší cieľ je vybrať jedno reálne SPI slave zariadenie a rozšíriť vytvorené SPI slave zariadenie o funkcionalitu, ktorá bude simulovať vybrané reálne SPI slave zariadenie. Vytvorené SPI slave zariadenie by malo byť konfigurovatelné pomocou programovacieho rozhrania v preferovanom programovacom jazyku.

Zhrnutie a popis riešenia

Hlavný cieľ tejto práce je vyvinúť testovacie zariadenie pre SPI master zariadenia. Toto testovacie zariadenie by malo mať univerzálne testovacie schopnosti a taktiež schopnosti emulovať jedno konkrétne reálne SPI slave zariadenie. STM32F303K8 mikrokontrolér bol vybraný ako platforma zariadenia. Bola použitá NUCLEO vývojová doska s vybraným mikrokontrolérom. Hlavná charakteristika tohto zariadenia je, že toto zariadenie musí byť konfigurovateľné pomocou počítača. Tým pádom je toto zariadenie pripojené k počítačovému sériovému portu pomocou UART rozhrania. ST-LINK, ktorý obsahuje NUCLEO vývojová doska je použitý ako USB-UART adaptér. Bolo vytvorené python API a toto API vie komunikovať s embedded zariadením. Hlavná úloha API je aplikovať konfiguráciu na zariadenie a taktiež príjmať informačné a potvrdzovacie správy z embedded zariadenia. Potvrdzovacie správy musia byť príjmuté za každým príkazom a musia obsahovať informácie o danom príkaze, inak príkaz nie je validný. Prenosy medzi embedded zariadením a počítačom obsahujú CRC číslo, ktoré je použité na zistenie chýb prenosu. YAML súbor je podporovaný z dôvodu jednoduchého formátu konfigurácie. Tento YAML súbor je vytvorený uživateľom a presne špecifikuje zvolenú konfiguráciu. Konfigurácia z YAML súboru je aplikovaná na zariadenie v momente úspešného pripojenia zariadenia k python API.

Predtým ako boli vyvinuté metódy, ktoré testujú logiku SPI master zariadenia, bola vyvinutá funkcionalita, ktorá dokáže merať frekvenciu SPI hodinového signálu. Frekvencia sa meria pomocou časovača, ktorý zachytáva momenty stúpajúcich hrán hodinového signálu. Ak sú známe momenty stúpajúcich hrán signálu, je možné spočítať periódu tohto signálu a tým pádom aj frekvenciu. Frekvencia sa vypočíta viackrát a výsledná frekvencia je určená ako módus týchto meraní. Frekvenciu SPI hodinového signálu je možné merať s dostačujúcim rozlíšením.

Prvým módom, ktorý testuje logiku SPI master zariadenia je tzv. stream mód. V stream móde sa žiadne dáta neposielajú zo slave zariadenia, ale dáta sú len príjmané. Hlavným účelom stream módu je preposielať každý prijatý byte SPI rozhraním priamo do python API. Nejaké problémy vznikli z dôvodu rozdielu rýchlostí SPI a UART rozhraní, ale väčšina problémov bola vyriešená pomocou kruhového buffru s "head" a "tail" pointermi. Použitím tohto buffru sa zaručilo, že preposielanie prijatých bytov sa uskutočnuje asynchrónne od momentu prijatia daného bytu SPI rozhraním a taktiež, že všetky byty sú poslané a žiadny byte nie je stratený.

Ďalšia metóda testovania základnej logiky SPI mastra je LUT (Look Up Table) mód. Tento mód dovoluje uživatelovi definovat LUT páry. Jeden LUT pár obsahuje žiadosť, ktorá bude prijatá SPI rozhraním od SPI mastra a odpoveď, ktorá bude poslaná z SPI slave zariadenia podľa toho, aká žiadosť bola prijatá. Tento mód vie fungovať buď v obojsmernom alebo jednosmernom móde.

Na detailnejšie testovanie špecifického SPI mastra bolo vybrané jedno reálne SPI slave zariadenie a bol vyvinutý jeho emulátor. SPI EEPROM pamäť bola vybraná ako preferované reálne zariadenie. Emulátor bol vyvinutý a ten presne emuluje logické správanie daného reálneho SPI slave zariadenia. Jedna významná limitácia tohto emulátor zariadenia je, že maximálna frekvencia SPI rozhrania emulátora je výrazne menšia ako maximálna frekvencia SPI rozhrania reálneho zariadenia. Táto limitácia je spôsobená tým, že všetka vnútorná logika emulátora je riešená vo vnútri interruptov. Táto limitácia nie je braná ako závažný problém, pretože vyvinuté zariadenie je považované za prototyp. V budúcom vývoji bude použitý oveľa rýchlejší mikrokontrolér, čo výrazne zlepší problémovú situáciu. Taktiež pri meraní tejto limitácie bol predpokladaný nepretržitý "bit stream", čo znamená rovnaké časy medzi bytami ako medzi bitmi. Nepretržitý bit stream je v realite nepravdepodobný a preto skutočná maximálna frekvencia SPI rozhrania je výrazne vyššia.

Následne boli vyvinuté verifikačné testy a každá funkcionalita bola overená týmito testami. Výsledky testov boli úspešné a všetky funkcionality fungujú tak ako majú.

Zhodnotenie výsledkov

Všetky dané ciele boli dosiahnuté a v niektorých prípadoch prekročené. Výsledok tejto práce je zariadenie, ktoré môže byť použité na testovanie existujúcich SPI master zariadení. Je možné otestovať univerzálne SPI master zariadenia pomocou stream a LUT módu a taktiež je možné otestovať špecifické SPI EEPROM master zariadenie EEPROM emulátor módom. Zariadenie je možné konfigurovať pomocou python API.

V budúcom vývoji je možné uplatniť pár vylepšení. Jedno výrazné vylepšenie pre stream mód by bolo modifikovať proces posielania informačných správ. Momentálne jedna správa obsahuje len jeden prijatý byte. Efektívnejšie riešenie by bolo, keby jedna správa obsahovala všetky byty, ktoré neboli doposiaľ poslané. Plán pre EEPROM emulátor mód je pridať podporu pre viac EEPROM zariadení, ideálne vyvinúť nejakú metódu ako jednoducho definovať univerzálne EEPROM zariadenie.

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and that this master's thesis is my own work and contains nothing which is the outcome of work done in collaboration with others.

Pavol Ženčár

Brno . . . .

First of all, I would like to thank my supervisor doc. Ing. Jiří Krejsa, PhD. for the great support, motivation and insightful comments. I also would like to thank Jakub Lešinský for all his help, motivation, knowledge and patience while answering all my questions. Last but not the least, I would like to thank my family for support.

Pavol Ženčár

Contents

1. Introduction

1.1. Motivation

SPI slave devices are in most cases various types of sensors. While designing any product that uses sensors, which use SPI interface to communicate, designer will most likely have to design SPI master device that interfaces with these SPI slave sensors. In order to test the SPI master devices that are being developed, the only real possibility is to use a logic analyzer as designer usually does not have access to the SPI slave sensors and cannot "look" inside them through debugging software or something similar. Testing with a logic analyzer does have its limitations. It is only possible to see what is happening on data buses, therefore the designer sees what exactly is being sent and received, but more in-depth logic of the device cannot be tested. In order to test more in-depth logic, separate SPI slave device must be used. It would be useful if this separate SPI slave device could be easily configured in such ways, that basic logic and basic functionality of an SPI master device could be tested.

For more detailed SPI master device testing, the SPI slave device would have to be simulated in a more detailed way and possibly a model of this device should be created with same or very similar behaviour. Second part of the thesis deals with this problem. This model cannot be created in a universal way to include all SPI slave devices, therefore one particular real SPI slave device is chosen and simulated. The motivation of this assignment is to have a virtual replacement of a real SPI slave device that ideally has the same behaviour as a real SPI slave device and can be configured and "looked" into by API running on a PC. This virtual replacement can be used for development of a specific SPI master, when it is not possible to acquire a real SPI slave device.

1.2. Goals

One of the goals of this thesis is to get acquainted with the characteristics of SPI protocol and microcontrollers in general. One goal is to create a device, that will act as a general SPI slave. It is important to design methods that will be able to simulate basic SPI slave functionalities with simple configuration. The second goal is to choose a real SPI slave device and expand the already created device with functionality to simulate chosen real SPI slave device. Created device should be configurable via API running on a PC in a preferred programming language.

2. Research

In the main part of the thesis, a device will be designed that has certain functionality and uses communication protocols such as SPI, UART and methods such as CRC error checking. This chapter contains descriptions of all the major building blocks that were needed for device development as well as a few already existing solutions.

2.1. Existing solutions

There are not many solutions for this problem on the market today. Two companies that stand out in creating devices with similar functionality are *Total phase* and *Diolan*.

Total phase

Total phase designed a product with similar functionality with the name of *Aardvark I2C/SPI Host Adapter*. The manufacturer claims according to [1] that this product does not only have SPI functionality, but also can be used as I2C device and can control GPIO (general purpose input-output) pins. *Aardvark I2C/SPI Host Adapter* also interfaces with PC via USB port and can be programmed via API. The main specifications relating SPI interface are claimed as:

- Master mode up to 8Mbit/s signalling rate.
- Slave mode up to 4Mbit/s signalling rate.
- Full duplex master transmit/receive.
- Asynchronous slave transmit/receive.

More information about this product and more detailed description of SPI slave mode functionality can be found at the manufacturers official website [1].

Figure 2.1: Aardvark I2C/SPI Host Adapter [1]

Diolan

Diolan does have an SPI-USB adapter in its product range. One of these types of devices is a device called *DLN-4S*. This device can also be used with I2C interface and can control GPIO pins. The device interfaces with PC through USB and can be controlled and programmed via API. The main specifications relating SPI interface are claimed according to [2] as:

- Master mode up to 48Mbit/s
- Slave mode up to 48Mbit/s
- Configurable SPI modes.

More information about this product and more detailed description of SPI slave mode functionality can be found at the manufacturers official website [2].

Figure 2.2: Diolan DLN-4S device [2]

2.2. Microcontrollers

2.2.1. Overview

Microcontrollers are small computers that are present on a single integrated circuit chip. Figure 2.3 displays an example of a microcontroller component. The difference between commercial microprocessors used in personal computers and microcontrollers is that microcontrollers already contain other components that are needed for successful operation of processing unit on chip. Microcontrollers usually contain one or more CPU (central processing unit) cores, some sort of RAM memory and FLASH memory. Also I/O pins are present and these pins can be used along with ADC/DAC converters, PWM generators etc. . Figure 2.4 illustrates a block diagram example of a microcontroller die with all the components. [3]

2.2.2. Main usage

Microcontrollers are mainly used in embedded systems. An embedded system is a system with some processing unit (microcontroller) that is capable of controlling peripherals that

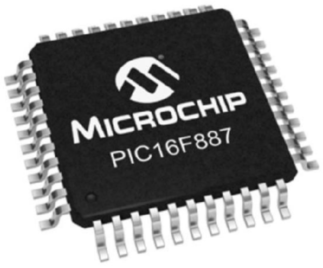


Figure 2.3: PIC16F887 microcontroller in 44TQFP package [6]

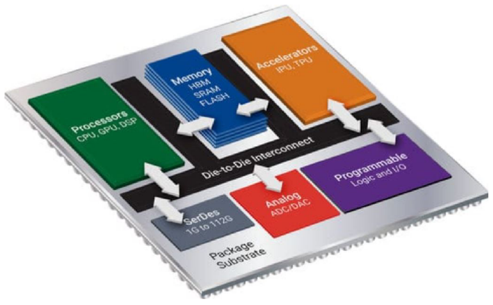


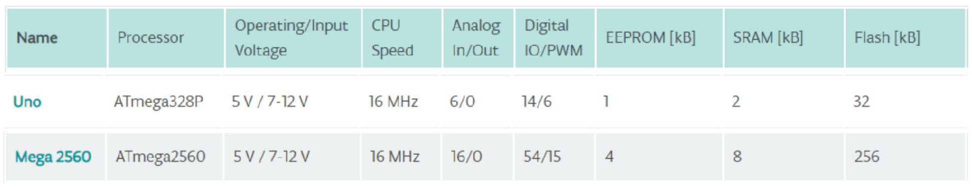
Figure 2.4: Block diagram of microcontroller components [5]

are also included in this system. The main uses of these peripherals are to monitor and control some larger mechanical or electrical system. This implies that embedded systems are part of some larger system. Embedded systems usually control a physical mechanism, therefore microcontrollers need to be optimized to work in real time. According to estimation in 2009 [4], 98% of all microcontrollers were being used for embedded systems. [4]

2.2.3. Common types

The two types of microcontrollers that were taken into consideration while choosing a working platform are microcontrollers manufactured by *Atmel Corporation* and microcontrollers manufactured by *STMicroelectrics*.

Atmel produces *ATmega* family of microcontrollers that are used on *Arduino* development boards. These boards are used in many learning applications for their very user friendly integrated development environment (IDE) and also many examples and tutorials exist for these development boards. Figure 2.5 contains a comparison table between the two most common ATmega microcontrollers in Arduino platform.



Name	Processor	Operating/Input Voltage	CPU Speed	Analog In/Out	Digital IO/PWM	EEPROM [kB]	SRAM [kB]	Flash [kB]
Uno	ATmega328P	5 V / 7-12 V	16 MHz	6/0	14/6	1	2	32
Mega 2560	ATmega2560	5 V / 7-12 V	16 MHz	16/0	54/15	4	8	256

Figure 2.5: Comparison of the most used ATmega microcontrollers in Arduino platform [19]

STMicroelectrics produces a lot of different products, but the main competitor of *ATmega* microcontrollers is *STM32* family of microcontrollers. STM32 microcontrollers also use standalone IDE for programming which has got more advanced functions such as real time debugging. In general, STM32 microcontrollers are more capable than ATmega microcontrollers based on various reasons. Table 2.1 displays the main feature difference between these two types of microcontrollers.

	ATmega	STM32
Word length	16bit	32bit
Clock frequency	Lower clock frequency (16MHz)	Higher clock frequency (Up to 550 MHz)
Memory	Smaller RAM and FLASH memory	Larger RAM and FLASH memory
Debugging	No debugging	Real time debugging

Table 2.1: Comparison of the main features of ATmega and STM32 microcontrollers

2.2.4. Interrupts

MCU devices have functionality called interrupts. Interrupts are event driven and happen upon some important event. MCU's main code happens inside a main while loop that is called periodically by the MCU. The main purpose of an interrupt is to ”jump” away from this code at a certain time and to execute different code that is specified for every interrupt. Interrupts are usually triggered by peripherals. Once interrupt is triggered, MCU saves current place in the code along with all the necessities and enters an interrupt handler that contains code that needs to be executed. Interrupts can be nested, meaning interrupt can happen inside another interrupt. Priority can be set for every interrupt separately in order for MCU to know which interrupt to execute first if multiple are triggered at the same time. If multiple interrupts of the same priority are triggered, interrupts will execute in chronological order.

2.2.5. Timer

MCU devices usually contain timers. A timer can do a lot of different things like PWM generation, input capture on hardware pin or a timer can even be used as a literal timer for event timing which will be called normal mode. The main function of a timer depends on its counter. Counter counts or ”ticks” periodically with the same time interval and the value of the counter is incremented on every counter tick. The time interval of ticks can be changed as well as the maximum value of the counter. When the counter reaches its maximum value, the timer can either stop or roll back and start from 0 again. In this project, timer was used in two modes.

Input capture mode

In input capture mode, a timers responsibility is to detect rising or falling edge event of a signal on specified pin and capture the counter value (time) of this event. A timer can be set up to trigger an interrupt on this event.

Normal mode

In normal mode timer is used for timing. The timer's counter frequency and maximum value are set in such ways, that the counter reaches its maximum value in a certain specified time. Once the counter reaches maximum value, an interrupt is usually triggered to precisely time events.

2.2.6. DMA

DMA (Direct Memory Access) controller is used for directly connecting peripherals to memory. The main purpose of this controller is to write data from and to peripherals instead of the processor. Processing time would otherwise have to be used to operate basic features of peripherals. Once the address of peripheral and the address of memory is specified along with the direction of write operation, it is possible to specify the number of writes to be executed by DMA controller. Once this number of writes is executed, an interrupt can be set to trigger. Peripherals usually provide a trigger to DMA controller telling it when to write data. It is possible to set DMA for memory to memory connection, but that setting is not used in this project.

2.3. SPI

Overview

SPI (Serial Peripheral Interface) is a communication protocol, which is used by microcontrollers to communicate with peripheral devices or between each other. Communication is implemented using synchronous serial bus. The architecture of this protocol relies on the master-slave relation. For successful SPI transmission one master device must exist and this master device can be connected to multiple slave devices. It is a masters responsibility to provide a clock signal for transmission and to enable/disable slave devices using slave select pins which are sometimes referred to as chip select pins. The SPI protocol uses four data lines.

- **SCK** (Serial Clock)
  This data line is used as a transmission clock. When a master device wants an SPI transmission to happen, he must generate clock pulses on this line. On rising or falling edges of this clock (based on mode), MISO and MOSI lines are sampled. The clock signal is used to synchronize master and slave devices.
- **MOSI** (Master out, Slave in)
  On this line master sends data to a slave device. It is sampled by the slave device on clock rising or falling edge (based on mode).

- **MISO** (Master in, Slave out)
  On this line slave sends data to a master device. It is sampled by the master device on clock rising or falling edge (based on mode).

- **SS** (Slave select)
  This line describes which slave is currently active. Only one slave can be active at a certain time. This line is active low.

SPI modes

SPI can be setup in four modes (Mode 0, 1, 2, 3). These modes describe how and when a clock signal is sampled, meaning moment when data are actually being read from MISO and MOSI lines. These modes are described with two boolean values:

- **CPOL** (Clock polarity)
  This value describes the polarity of a clock signal while in idle. If this value equals to zero, the idle value of a clock signal is logic LOW. If this value equals to one, the idle value of a clock signal is logic HIGH.

- **CPHA** (Clock phase)
  This value describes the phase of a clock signal, meaning when is data sampled. If this value equals to zero, a clock pulse is sampled on the first edge of a clock pulse. If this value equals to one, a clock pulse is sampled on the second edge of a clock pulse.

Table 2.2 describes which SPI mode belongs to which CPOL and CPHA values. Figure 2.6 illustrates data sampling in each SPI mode and also displays a complete time diagram of an SPI one byte transmission.

Mode	**CPOL**	**CPHA**
0	0	0
1	0	1
2	1	0
3	1	1

Table 2.2: SPI modes

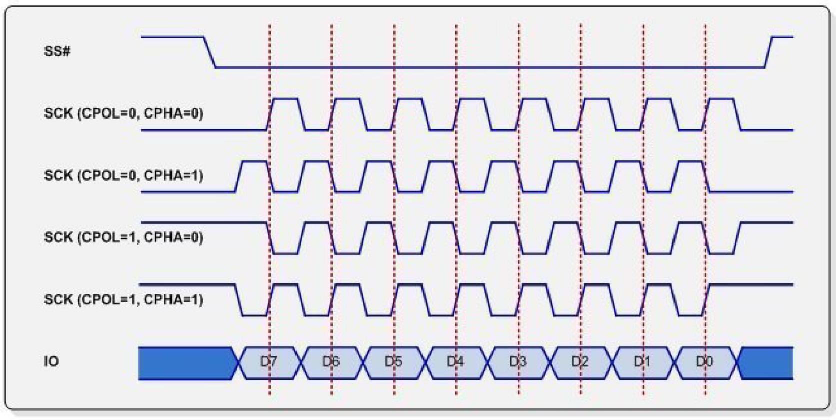


Figure 2.6: SPI timing diagram [8]

Multiple slave connection

In case when multiple slave devices need to be used, a connection must be made as is illustrated in figure 2.7. All devices share same CLK, MISO and MOSI line, but every slave device must have a separate slave select line. By controlling these slave select lines, a master can choose which slave is currently active. Only one slave can be active at one time, otherwise slave devices would interfere on MISO line.

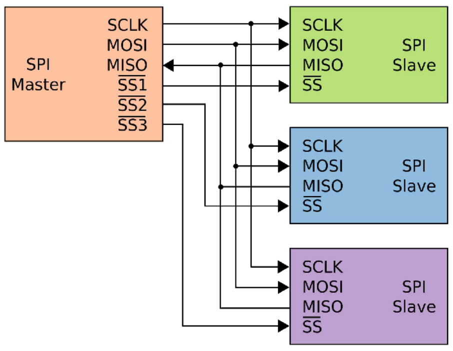


Figure 2.7: SPI multiple slaves connection [7]

2.4. UART

Overview

UART (universal asynchronous receiver-transmitter) is a hardware device for asynchronous serial communication. This device is mostly used in computer serial ports. Contrary to an SPI transmission protocol, UART uses asynchronous transmission, meaning there is no clock signal to synchronize transmitting and receiving devices. This means that data lines cannot be simply sampled on a clock pulse and need some other form of synchronization. Due to this reason, for successful transmission a certain driver needs to be used on both sides to synchronize and decode data.

Data format

Figure 2.8 displays a typical UART one byte packet. Packet consists of a start bit to synchronize the transmitting and receiving devices, five to nine bits of the actual data, an optional parity bit and one or two stop bits. A parity bit is used for error checking and can be set for odd or even parity, meaning that this parity bit is generated so that the total number of logical HIGH bits have to be odd or even in a packet. Stop bits are used to mark the end of transmission.

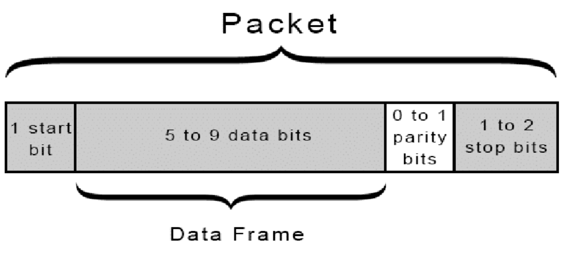


Figure 2.8: UART packet [9]

Data timing

Every UART device has 2 data lines:

- **TX** used for transmitting data.
- **RX** used for receiving data.

Both these lines are in logical high state at idle. UART devices can be set to different data rates. For successful transmission, both the transmitting and the receiving devices must be set to the same data rate. This guarantees that data will be sampled at the correct time. UART device contains an internal clock source which runs at a multiple of data rate. UART usually also contains two FIFO shift registers. One for the TX line and one for the RX line.

Transmitting

Transmitting is fairly simple. There is no need for any synchronization based on the line state, so device loads data to TX shift register, generates start bit, outputs all data from the TX shift register, generates an optional parity bit and lastly generates stop bits. All this is outputted on the TX line at correct times based on set data rate. After a valid start bit is generated, an internal clock is used to time next data change so it corresponds to set data rate.

Receiving

Receiving is a little bit more complicated. To receive data, the device must know when to sample data, therefore it needs to synchronize with the RX line. For this synchronization, a start bit is used. In order for a start bit to be declared valid, it must last at least half of the bit time. Once a valid start bit are detected, an internal clock is used to time data sampling. Data is sampled in the middle of each bit time and shifted into the RX shift register. For this reason it is important that both transmitting and receiving devices run at the same data rate so that the internal clocks get correctly synchronized and data is sampled at the right times. After set number of bits and a valid stop bit is detected, data from the RX shift register is made available to the receiving system. Usually a flag is generated indicating successful data receive. Figure 2.9 illustrates a timing diagram of data transfer and how data is sampled by the receiving device. [10]

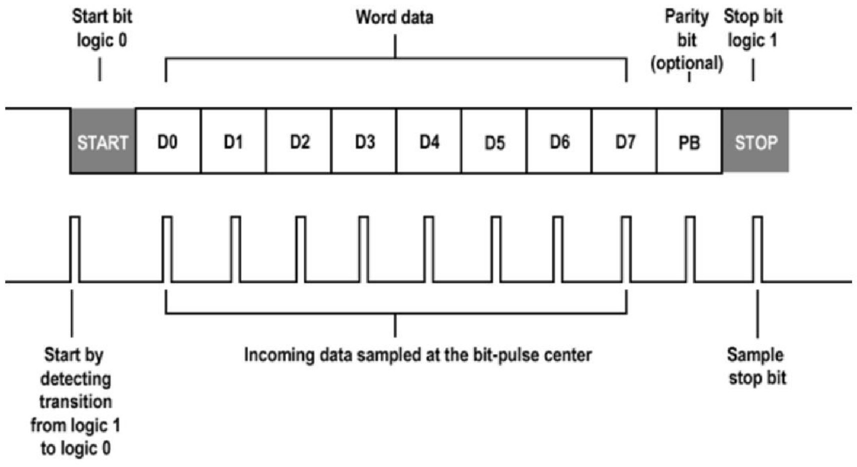


Figure 2.9: UART timing diagram [10]

2.5. CRC

Overview

In many data transfers, errors can occur. There can be a lot of different types of errors, for example errors due to environment noise, bad connections or faulty devices. It is common that data transmissions include redundant data, which can be used to error check the whole data frame. One of the simplest error check is a parity bit. A parity bit simply completes data packet, so that the total logical HIGH bits in this packet are either odd or even based on the configuration. This method is the simplest kind of error check and cannot reliably detect errors. For more advanced error checking CRC (Cyclic Redundancy Check) can be used. This method is used in a lot of transmission protocols (Bluetooth, Ethernet etc.). CRC can differ vastly in its configuration (even parity bit error check can be interpreted as CRC) and can be used for many different use cases. The principle is that a transmitting device calculates CRC value from data that will be sent and then appends it to the end of this data. A receiving device calculates the CRC value of received data and then compares it with the CRC value that was appended to the data by a transmitting device.

Main parameters

The main difference between CRCs is a bit length. CRC can have a bit length of any value, but a length of 8, 16, 32, 64 bits is most common. Bit length and the actual total characteristic of CRC is defined by a polynomial (divider). The highest power of this polynomial equals to the CRC bit length, while the total length of the polynomial is exactly one larger than the CRC bit length. For example, if CRC-8 (CRC with 8 bit length) is used, the polynomial has length equal to 9 as polynomial equals to $x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x^1 + x^0$. To fully define a polynomial, all members of a polynomial can also equate to 0. To specify and define polynomial for $n - bit$ CRC, a binary number with $n + 1$ number of bits is used, where every bit represents value of the corresponding polynomial member. For example:

- CRC-3 with polynomial $0b1010$:
  $1 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x^1 + 0 \cdot x^0$
- CRC-8 with polynomial $0b111000111$:
  $1 \cdot x^8 + 1 \cdot x^7 + 1 \cdot x^6 + 0 \cdot x^5 + 0 \cdot x^4 + 0 \cdot x^3 + 1 \cdot x^2 + 1 \cdot x^1 + 1 \cdot x^0$

Calculation

To calculate the actual CRC value of data, polynomial division is used. First a single binary number is created from all bits of every byte in data in the correct order, then this number is divided by a CRC polynomial also defined by a binary number. The remainder of this division represents the CRC value of data. CRC value has the same bit length as the CRC bit length. Specific algorithm that describes how exactly this calculation executes can be found at [12].

2.6. EEPROM memory

Overview

EEPROM (electrically erasable programmable read-only memory) is a type of a non-volatile (data retention after reboot) memory used for storing data. EEPROM devices are usually used with computers and microcontrollers. EEPROMs are constructed using floating gate transistor arrays. The main characteristics of EEPROMs are small data erase and write blocks and long lifetime. Usually EEPROMs can erase and write to blocks as small as one byte, this means that EEPROMs are ideal for storing small chunks of data. Lifetime of EEPROMs can be 1000000 cycles or even more. EEPROMs can be divided into two groups. One group is using a parallel interface for programming and the other group is using a serial interface. [13]

Programming interface - parallel

Parallel interface needs a lot more connections as every bit of data and address needs to have its own data line as well as data lines for a chip select and operation code need to be present on this device. For example, if a word length was 8 bits and address was also 8 bits, minimum of 16 lines would need to be connected, which means a minimum of 16 hardware pins would need to be used on a microcontroller. The advantages of this interface compared to a serial interface are that it's faster as all data is transferred at once. The disadvantages are that the devices need to be a lot bigger because a lot more hardware pins need to be used. In current times the popularity of parallel devices decreases rapidly.

Programming interface - serial

On the other hand, serial devices use a very small number of data lines. The necessary data lines for synchronous serial communication are clock signal, chip select signal, one two-directional or 2 one-directional data lines. Data is not transferred all at once as in parallel devices, but it is transferred one bit at a time, a clock signal is used for synchronization. For this type of operation, many communication interfaces can be used. The most common ones are *SPI, I2C, Microwire*. A general command consists of an op-code (operation code describing the type of command) followed by an address (if command uses address) and lastly followed by the actual data either to be written or read.

Instruction set

Six instructions need to be defined for basic EEPROM functionality.

- **READ**
  (Usually 0x3) Read instruction to read data from a specific address. Usually this command can also read multiple subsequent bytes.
- **WRITE**
  (Usually 0x2) Write that writes data to a specific address. Usually this command can also write multiple subsequent bytes that are in the same block of memory. For this command first a write enable command needs to be executed to set a write enable latch in a status register. In most cases an EEPROM device upon receiving this command first erases data on the specified address and then after that data is written.
- **Write status register [WRSR]**
  (Usually 0x1) This command writes data to a status register. The status register is used for functionality such as block protect, write protect etc. The status register consists of read only bits and readable/writable bits. This command alters only bits that are writable.
- **Read status register [RDSR]**
  (Usually 0x5) This command reads data from a status register. This command can be used to check the current configuration or state of an EEPROM. It is used very frequently after any **WRITE** or **WRSR** command to check the value of write in progress bit.
- **Write enable [WREN]**
  (Usually 0x6) Write enable command sets the write enable bit in a status register to logical HIGH value. This action enables **WRITE** and **WRSR** commands.
- **Write disable [WRDI]**
  (Usually 0x4) Write disable command sets the write enable bit in a status register to logical LOW value. This action disables **WRITE** and **WRSR** commands.

3. Detailed goals description

In the Introduction chapter, general goals and motivation were mentioned. In this chapter specific goals and the way of implementation will be described. This chapter should mention every main feature that the created device will have. Detailed description of goals could be interpreted as follows.

Device hardware selection

The first goal is to choose the appropriate hardware, so it does have all the desired functionality that will be used later in development. The main features of this hardware should be the ability to connect to a PC via some kind of interface and also to be able to communicate through SPI interface between other hardware devices. SPI interface on this hardware should be configurable and should be able to function in all possible SPI configurations. The maximum baud rate of SPI interface should also be taken into consideration.

PC connection and programming API

The main feature of this device is that it should interface with a PC. The device should be configurable via some kind of API running on PC, meaning API should be able to send commands and receive messages from an embedded device. In order to create this API, first a connection of embedded device and PC must be made using a communication interface. After a successful connection is made, the API should be used for loading configuration onto embedded device as well as know the current state of the embedded device and receive report messages about SPI transmissions that happened. It is important that every transmission from PC to embedded device or from embedded device to PC is error checked so that only the correct configuration is loaded into the device and it is known exactly what is happening on SPI data buses.

Stream mode

Now that the device is connected to a PC that is running API that can program this device, the actual methods for emulating an SPI slave device and to debug SPI master device need to be created. The first of these methods is stream mode. In this mode device will act as receive only SPI slave, meaning no data will be transmitted from this device through SPI interface. All received data will be directly transmitted (streamed) to an API running on a PC. The API should decode these messages and display them to the user. This mode is useful for checking exactly what an SPI master is sending on MOSI bus and this can resolve a lot of issues in SPI master development. The main problem in this mode is that SPI interface is faster than chosen interface for PC to embedded device communication, therefore some method must be developed to stream every byte received via SPI, otherwise some data would be lost due to interface speed difference.

LUT mode

Stream mode is great for the lowest level debugging of an SPI master, but cannot test the actual logic of said SPI master. For this LUT mode is designed. Contrary to stream mode, this mode can also transmit data through the SPI interface. The main idea of this mode is that the designed SPI slave device responds to masters requests according to LUT (look up table). In LUT request-response pairs are stored, meaning user can set what response embedded device will return for what request that was received from an SPI master. This mode is useful to test basic SPI master logic. Every SPI transmission should be reported to API running on a PC. The same problem with an interface speed difference as in stream mode exists.

Real device emulation mode

LUT mode can test the basic logic of an SPI master device, but cannot emulate existing SPI slave device completely as usually SPI slave devices are more complex and contain some sort of state machine and more advanced functionalities. No universal SPI slave device that could be easily configured to cover every real SPI slave device can be created. Because of this reason, one specific existing SPI slave device will be chosen and its emulator created on our embedded device. The main idea behind this mode is that when SPI master is connected, our device should behave exactly the same as real SPI slave device, so there is no difference from the masters point of view. Some limitations will naturally occur, so it is important that these limitations are known and possibly optimized for better result. This mode will be used when a real SPI slave device cannot be easily acquired and developer cannot use real device for developing, instead developer can develop an SPI master device by connecting to the embedded device in emulator mode. Every SPI transmission or SPI command needs to be reported to an API running on a PC. Same problem with an interface speed difference as in stream mode exists.

Frequency measurment

All modes described in last sections test logic of an SPI master device. There may be situations where the hardware characteristics of SPI master need to be tested. One of the main SPI hardware characteristics is clock frequency. The clock is provided by master so it could be useful to measure this clock frequency with created SPI slave device. Ideally frequency should be measured for every SPI transmission and reported to API running on a PC.

4. Hardware selection and setup

Figure 4.1 illustrates a simple block diagram of the main device connections. The device will need to be connected to a personal computer via UART interface. The device will also be connected to another SPI device. This chapter contains a description on how the hardware was chosen and how it was made operational.

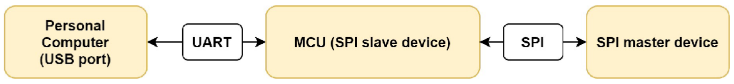


Figure 4.1: Block diagram of the main device connections

4.1. Microcontroller selection

Microcontroller was selected as a preferred type of hardware for various reasons. The main reasons why microcontroller was selected are:

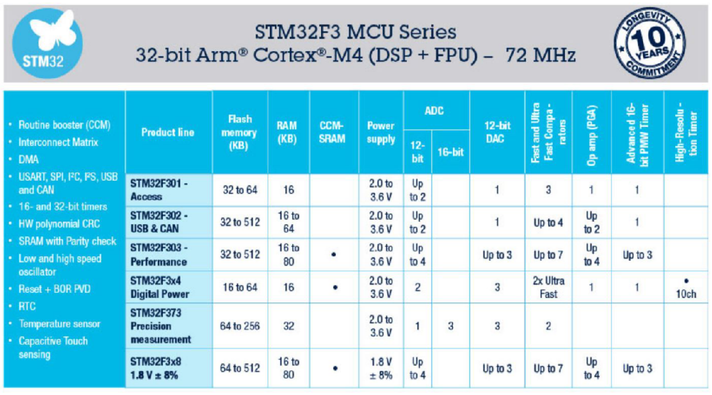
- All needed functionalities are available
- Relatively easy to program
- Availability of development boards
- Affordable for prototyping

Platform selection

Microcontrollers are available in a lot of different platforms with different architecture, IDEs and abilities. Few of them are described in the Research chapter. For this project, the STM32 platform was chosen. The STM32 family of MCUs is made by *STMicroelectrics* company. This 32 bit family of microcontrollers uses ARM CORTEX-M core (processor) and can be used for a very wide variety of applications. The IDE for MCU programming is called *STM32CubeIde* along with *STM32CubeMx*. These programs include features such as real time debugging, where user can ”go through” the code step by step and more. For programming and debugging, MCU has to be connected to a PC through ST-LINK which is a programmer/probe made by *STMicroelectrics*. More about this family of MCUs can be found at [14].

Model selection

As was mentioned before, STM32 family of MCUs has got a wide range of MCUs that are each optimized for a different use case. It is important to choose a specific MCU model that will suit the desired application. For this specific application, the STM32F3 family of microcontrollers was chosen, more specifically STM32F303 MCU. Figure 4.2 displays different models of MCUs in STM32F3 family and more information about this MCU family can be found at [15]. The STM32F303 MCU was chosen. This MCU comes in different variants and packages. In the process of choosing a specific MCU, multiple attributes were taken into consideration such as clock frequency, memory capacity, SPI interface configurability and ability, development board availability. Figure 4.3 displays all different variants of STM32F303 MCU and more information about this MCU can be found at [16].


STM32F3 MCU Series
32-bit Arm® Cortex®-M4 (DSP + FPU) – 72 MHz

- Routine booster (CCM)
- Interconnect Matrix
- DMA
- USART, SPI, I²C, I²S, USB and CAN
- 16- and 32-bit timers
- HW polynomial CRC
- SRAM with Parity check
- Low and high speed oscillator
- Reset + BOR PVD
- RTC
- Temperature sensor
- Capacitive Touch sensing

Product line	Flash memory (KB)	RAM (KB)	CCM-SRAM	Power supply	ADC		12-bit DAC	Fast and Ultra Fast Compa - rators	Op amp (PGA)	Advanced 16-bit PMW Timer	High-Resolu - tion Timer
					12-bit	16-bit					
STM32F301 - Access	32 to 64	16		2.0 to 3.6 V	Up to 2		1	3	1	1	
STM32F302 - USB & CAN	32 to 512	16 to 64		2.0 to 3.6 V	Up to 2		1	Up to 4	Up to 2	1	
STM32F303 - Performance	32 to 512	16 to 80	•	2.0 to 3.6 V	Up to 4		Up to 3	Up to 7	Up to 4	Up to 3	
STM32F3x4 Digital Power	16 to 64	16	•	2.0 to 3.6 V	2		3	2x Ultra Fast	1	1	• 10ch
STM32F373 Precision measurement	64 to 256	32		2.0 to 3.6 V	1	3	3	2			
STM32F3x8 1.8 V ± 8%	64 to 512	16 to 80	•	1.8 V ± 8%	Up to 4		Up to 3	Up to 7	Up to 4	Up to 3	

Figure 4.2: STM32F3 family of microcontrollers [15]

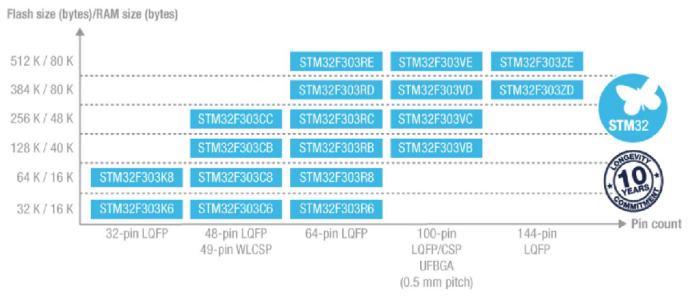


Figure 4.3: STM32F303 variants of microcontrollers [16]

In the end STM32F303K8 variant was chosen. This variant comes in a 32 pin package, which is enough for this application and is on the lower side regarding memory size, which proved to be just enough for this project. For future development and adding more features, the STM32F303RE MCU variant will be used.

Development board

Microcontrollers are constructed as a single chip and cannot be used by themselves, therefore some kind of PCB (Printed Circuit Board) needs to be used that contains desired microcontroller. For complex projects, a custom PCB has to be designed, but for educational purposes and prototyping, *STMicroelectrics* makes development boards called NUCLEO. These boards contain all the necessary components for proper MCU operation as well as an integrated ST-LINK programmer/probe and this board is also a breakout board for all the functional pins. More information about NUCLEO boards and information about which specific MCUs do have their own NUCLEO board can be found at [17].

NUCLEO-F303K8 board was used as a development board for this project. This board contains the STM32F303K8 MCU and is very affordable with all the functionalities needed. Figure 4.4 displays the development board and more info can be found at [18].

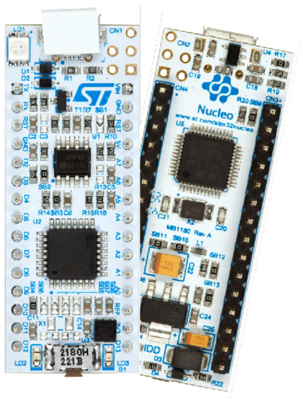


Figure 4.4: STM32F303K8 nucleo development board [18]

4.2. PC connection

First step is to establish connection to a PC. One of the main features of this device is that every functionality must be controlled from a PC, therefore connecting device to a PC is one of the main tasks. The device should be configurable from a PC and certain messages should be sent from the MCU to a PC.

For MCU to PC connection, UART is used. The device will be connected to a USB serial port located on a PC. UART data lines cannot be simply connected to a USB serial interface, therefore some kind of UART-USB convertor must be used. One of the advantages of the NUCLEO development board is that it contains ST-LINK programmer/probe. One of the features that ST-LINK has, is that it can create a virtual serial port on a PC, therefore ST-LINK also acts as a UART-USB converter.

4.3. MCU setup

The first step to initialize MCU is to create a STM32CubeMx file with .io extension and configure it for the desired purpose in STM32CubeMx software. The configuration is done in these steps.

Initailizing hardware pins

Figure 4.5 illustrates the pin setup. The debug interface is connected via **PA14** and **PA13** pins. UART is connected to pins **PA2** and **PA15**. SPI interface is connected on pins **PB4, PB3, PA4, PA7**. Timer input capture pins are **PB5, PB0**. Pin **PB5** is externally connected to pin **PB3**. Pin **PB0** is externally connected to the pin **PA4**. For general purpose debugging, pin **PA1** is used as GPIO pin.

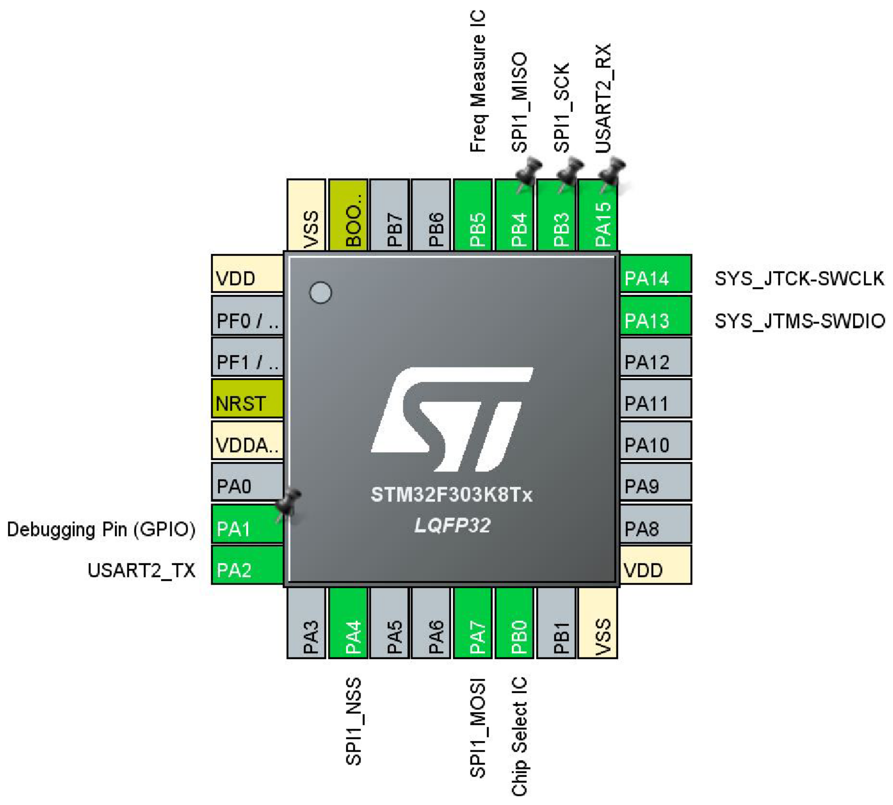


Figure 4.5: Pin initialization

Clock configuration

This NUCLEO board does not have an external oscillator, therefore the internal oscillators must be used. The MCU has high speed and low speed oscillators. High speed is 8MHz and low speed is 40kHz. The internal high speed oscillator will be used along with prescalers and clock multipliers to create the main CPU clock (HCLK). HCLK is set to the maximum value possible without an external oscillator and this value is $f_{HCLK} = 64MHz$. Figure 4.6 displays the complete clock configuration. In this figure also other clocks such as an advanced peripheral bus clocks can be seen.

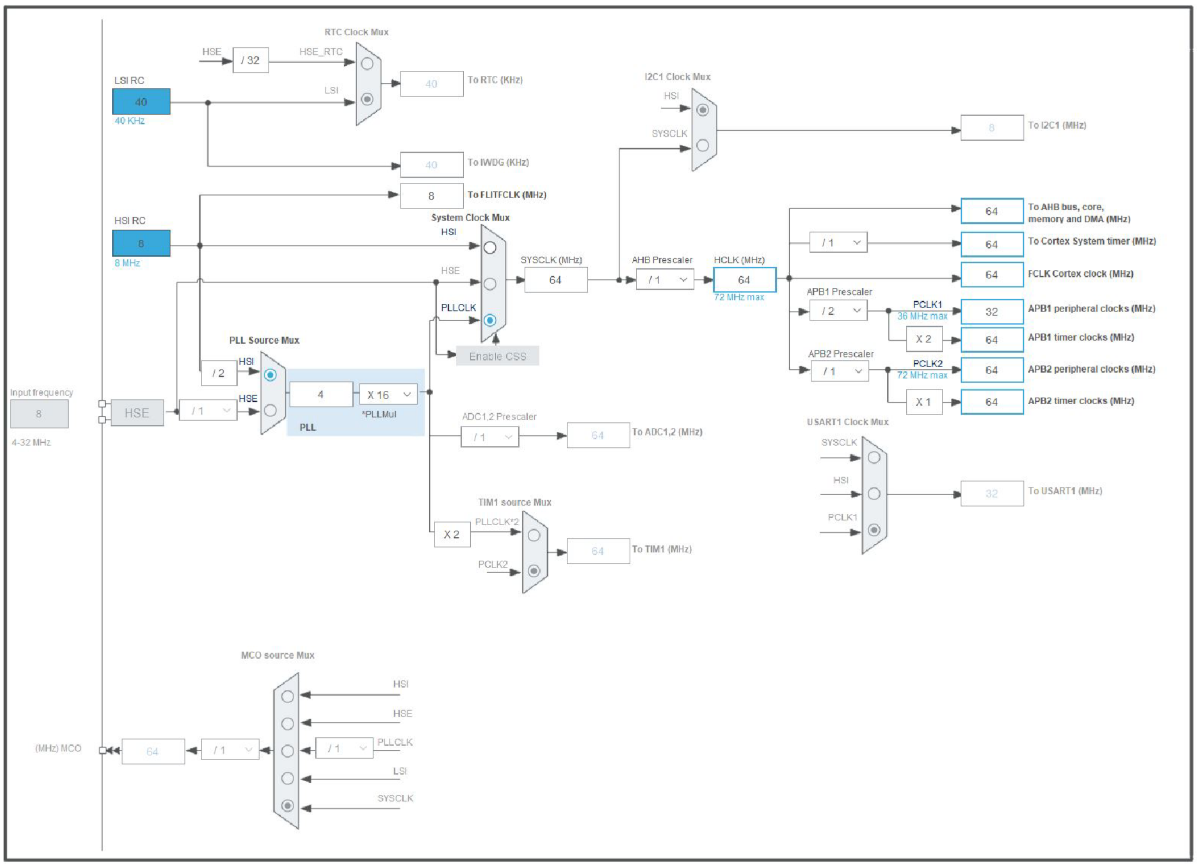


Figure 4.6: MCU clock diagram

Code generation and drivers

Next step is to generate code. STM32CubeMx generates code with all the needed registers initializations based on the .ioc file. In this generated project, the user code can be written. For most of the functionalities, the HAL (Hardware Abstraction Layer) drivers [20] will be used apart from SPI, where after optimization, the LL (Low Layer) drivers [20] will be used. The user can either modify the registers in own code or use these drivers to control peripherals and MCU in general.

5. Basic device functionality

The device will have multiple operational modes. All of these modes need to use certain features such as communicating through SPI interface, using timer input capture on hardware pin and more. In this chapter, first the important features are explained and in the end of this chapter, two specific operational modes are described. The overall internal structure of the device along with what operational modes exist and how the device can be configured to enter these modes will be described in chapter 7.1.1.

5.1. Frequency measurement

One of the tasks of this device is to measure the frequency of an SPI clock pin, on which clock is generated by SPI master. For this task, a timer integrated on the MCU is used in Input Capture mode.

Theory

The clock signal that is being measured is a square wave signal, which means that it is a periodic waveform with a fixed frequency. Fixed frequency means that the times between each pulses are the same. The easiest way to measure frequency of this type of a signal is to know the exact time between two pulses. A pulse start is detected by the rising edge of the signal. Assuming the time of the first pulse is known as t_{fp} and the time of the second pulse is known as t_{sp}, the frequency f can be calculated as $f = \frac{1}{t_{sp}-t_{fp}}$. Figure 5.1 illustrates the shape of square signal and moments of t_{fp} and t_{sp}.

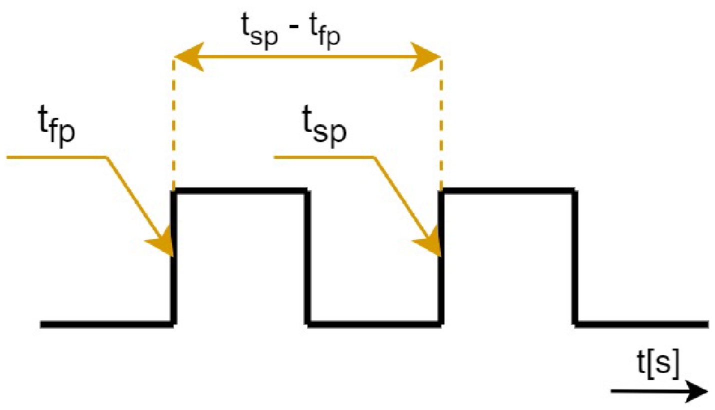


Figure 5.1: Square signal with pulse times

Timer setup

To enable a timer and to set it up for input capture mode, STM32CubeMx settings need to be changed. Figure 5.2 displays these settings. Only important settings to note are:

- **Prescaler**
  This is a general timer setting. This setting dictates frequency of the timer's counter. To determine frequency of timer f_{tim}, first frequency of the advanced peripheral bus ,to which this specific timer is connected, must be known. This value can be read

from figure 4.6 as **APB2 Timer Clocks** and its value f_{APB} is equal to $64MHz$. Lastly the timer frequency can be calculated as $f_{tim} = \frac{f_{APB}}{Prescaler+1}$. In this specific case, the $Prescaler$ value is equal to 0, therefore $f_{tim} = \frac{64000000}{1} = 64MHz$. This means that the time between subsequent counter ticks is $t_{cnt} = \frac{1}{64000000} = 15.625ns$.

- **Counter Period**
  This is also a general timer setting. This setting dictates the maximum value of the counter. Type of this timer's counter is *uint16_t*, which means that this setting needs to be set in the appropriate range : $< 0, 65535 >$. In this mode, the timer behaves in such way, that if the counter reaches this value, the counter naturally rolls back to 0.

- **Polarity selection**
  This is a specific input capture setting. This setting dictates which edge of the signal does timer capture. There are 3 options for this setting:

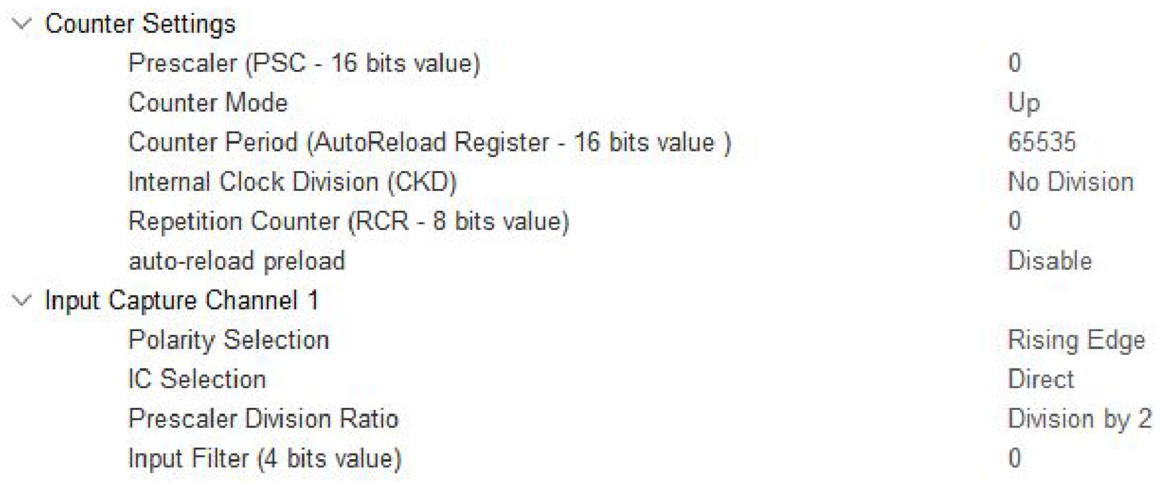
  - **Rising edge**
    Capture happens on the rising edge event.
  - **Falling edge**
    Capture happens on the falling edge event.
  - **Both edges**
    Capture happens on the rising and on the falling edge events.

  In this specific case, the capture will happen on the rising edge of the signal.

- **Prescaler Divisio Ratio**
  This is a specific input capture setting. This setting dictates how many events need to happen for a capture to trigger. There are 4 options for this setting:

  - **No division**
    Capture happens on every signal event.
  - **Division by 2**
    Capture happens once every 2 signal events.
  - **Division by 4**
    Capture happens once every 4 signal events.
  - **Division by 8**
    Capture happens once every 8 signal events.

  In this specific case, a capture will happen once every 2 signal events, meaning that the time between 2 subsequent clock pulses will not be known, instead the time between every third pulse will be known and this time corresponds to the time of 2 pulses.



Counter Settings	
Prescaler (PSC - 16 bits value)	0
Counter Mode	Up
Counter Period (AutoReload Register - 16 bits value)	65535
Internal Clock Division (CKD)	No Division
Repetition Counter (RCR - 8 bits value)	0
auto-reload preload	Disable
Input Capture Channel 1	
Polarity Selection	Rising Edge
IC Selection	Direct
Prescaler Division Ratio	Division by 2
Input Filter (4 bits value)	0

Figure 5.2: STM32CubeMx input capture timer settings for frequency measurement

After these settings are applied and the code is generated, only things left to do are to enable the timer's capture compare channel and enable the timer's counter. This is done by modifying the timer register by commands:

```
TIM17->CCER |= ( 0x01 << 0 ); // capture compare 1 enable
TIM17->CR1 |= ( 0x01 << 0 ); // counter enable
```

More information about MCU registers can be found in reference manual [reference manual source].

Frequency measurement

Now that the timer is setup in input capture mode, the time between clock pulses will be measured and the frequency can be calculated. In order to calculate the frequency, the first two or ideally more of the input capture times need to be stored in the MCU memory. Timer does not store the capture event times into the memory by itself, but instead on every capture event, current counter value is stored in the timer register. Upon the next capture event, previous event counter value is erased.

One of the solution for measuring the frequency would be to use interrupts. The timer could be set up to trigger an interrupt on every input capture event. For successful frequency measurement, two interrupts would need to happen. On both interrupts, the counter value from the timer register would be stored in the MCU memory and on the second interrupt, frequency would be calculated from the counter values of each interrupt. This solution was implemented but was quickly proved very inefficient and uneffective. To enter interrupt, MCU must use valuable processing time and this process is slow compared to possible SPI clock frequency as the SPI clock frequency on this type of MCU can reach up to 18 MHz. This solution is good for signals with very low frequency.

The second type of solution, which is used in final design, is to use DMA to store the input capture data to memory. DMA (Direct Memory Access) controller has direct access to memory as well as to peripheral registers. DMA and Timer can be setup such that upon every input capture event, the timer triggers DMA to write the value of the timer counter register to memory. Upon every trigger, DMA can increment memory address to which it is writing. This means that DMA can store multiple input capture events into subsequent memory locations (array). To implement this solution, first DMA requests on correct input capture channel need to be enabled in timer registers. This is done by code:

```
TIM17->DIER |= ( 0x01 << 9 ); // enable capture compare 1 DMA requests
```

Next to initialize DMA, 3 parameters must be set.

- **Memory address**
  The initial memory address to which DMA will write data. In this mode, this address will be incremented with every DMA event.
- **Peripheral address**
  The peripheral address from which data will be read and written to memory address. In this mode this peripheral address will not increment. For different kind of application, DMA could be set to increment this peripheral address.
- **Number of DMA events**
  This dictates how many DMA events need to be triggered in order to indicate the end of DMA action. Upon the end of DMA action, DMA could be set to trigger interrupt. DMA can be set to circular mode, which means that when this number of DMA events happen, DMA will not stop, but DMA will continue to work while the memory address is set to the initial address again. DMA could also be set to normal mode, which means that after this number of DMA events happen, DMA will stop and in order to work again, it must be initialized again. In this specific case, DMA is set to normal mode and interrupt is not set to trigger.

These parameters are set by modifying DMA register as following code illustrates:

```
hdma->Instance->CCR &= ~(0x01 << 0); // disable DMA

hdma->Instance->CMAR = &hfreq->raw_ccr_data[0]; // set memory address
hdma->Instance->CPAR = &TIM17->CCR1; // set peripheral register adress
hdma->Instance->CNDTR = SPI_FREQ_BUFFER_LEN; // set number of DMA events

hdma->Instance->CCR |= (0x01 << 0); // enable DMA
```

This solution is very efficient and effective as DMA does not use any MCU processing time and multiple input capture events can be stored between every frequency calculation, meaning results can be statistically evaluated.

The frequency measurement needs to be manually restarted and calculated. The optimal times to restart measurement and calculate need to be determined individually based on the device mode and should be chosen in such ways, that the frequency is measured and calculated maximum number of times, ideally for every SPI transmission report individually.

Frequency calculation

Once the input capture data is stored in the memory array, clock frequency needs to be calculated from this data. Data in the array represents current counter tick at which input capture event happened, therefore the unit of measurement is timer ticks and not seconds. In order to convert units to seconds, values could be multiplied by the timer's time between two ticks, but this would mean more complex calculation. The values are kept as *uint16_t* timer counter values and the timer's frequency will be used in the frequency calculation to compensate. Calculation can be split into 3 parts:

- **Finding time differences**
  Data represents the time of each input capture event. In order to calculate the frequency, the time between pulses must be known. In this specific case, the time of exactly two pulses will be known when calculating the differences between each subsequent values. Assuming the input capture data are labeled as raw_ccr, the differences ($diff$) for the whole array are calculated using this equation: $diff(n) = raw\_ccr_(n+1) - raw\_ccr(n)$ for $n \in \{0, 1, ..., array\_length - 2\}$. This will give an array with length one less than the input capture data array. In this array, the differences of each two subsequent input capture events will be stored.

  For example, assuming the input capture data array equals to:
  $[\mathbf{1000, 2716, 5382, 10714, 13380}]$,
  the difference array would be :
  $[\mathbf{1716, 2666, 5332, 2666}]$

- **Calculating frequencies**

  While calculating the frequency, two facts need to be taken into consideration. The first is that the difference array represents the time of two clock pulses and not the time of one clock pulse. The second fact is that the difference array units are not seconds, but timer ticks. If these facts are taken into consideration, the equation for calculating the frequency for each value of the difference array is as follows: $f(n) = \frac{f_{tim}}{diff(n)} * 2$ for $n \in \{0, 1, ..., array\_length - 1\}$, where f are the values of frequency in Hz, f_{tim} is the frequency of timer in Hz, $diff$ are the difference values of input capture events in timer ticks.

  In previous example, the difference array is equal to:
  $[\mathbf{1716, 2666, 5332, 2666}]$,
  if this equation is applied on this array, the result will be :
  $[\mathbf{74592, 48012, 24006, 48012}]$
  This array contains calculated frequencies for each input capture event pair in Hz.

- **Calculating final frequency**
  In order to calculate the final one number that represents the value of measured frequency, frequency array needs to be statistically evaluated. For this a simple solution was chosen. The final frequency value will be chosen as a mode of the frequency array. If multiple modes exist, one which frequency appears first in the array will be chosen.

  In previous example, the frequency array is equal to:
  $[\mathbf{74592, 48012, 24006, 48012}]$

By applying this method, the mode and the final calculated frequency value is equal to $\boldsymbol{48012Hz}$.

This method was chosen because the input capture data does not have to be from the same byte or even the same data frame and the times between bytes and the times between data frames will most likely be vastly different than the time between bits (time between clock pulses). The timer's counter can also overflow in the process of acquiring input capture data. By using a large enough sample set, these unwanted measurements will be filtered.

Testing

Table 5.1 displays the measured frequencies for different values of real SPI clock frequencies. Measurment was done in stream mode and the SPI clock frequency was verified using an osciloscope. In current implementation, input capture array with length 20 is used, meaning 20 input capture events are triggered between each calculation and 19 different frequencies are calculated and evaluated.

Real	Measured		Real	Measured
15.6 kHz	15 kHz		3750 kHz	3764 kHz
31.2 kHz	31 kHz		4000 kHz	4000 kHz
62.5 kHz	62 kHz		4500 kHz	4571 kHz
125 kHz	124 kHz		5000 kHz	4923 kHz
250 kHz	249 kHz		5500 kHz	5565 kHz
312.5 kHz	312 kHz		6000 kHz	6095 kHz
500 kHz	500 kHz		6500 kHz	6400 kHz
750 kHz	749 kHz		7000 kHz	7111 kHz
1000 kHz	1000 kHz		7500 kHz	7529 kHz
1250 kHz	1242 kHz		8000 kHz	8000 kHz
1500 kHz	1505 kHz		9000 kHz	9142 kHz
1750 kHz	1753 kHz		10000 kHz	9846 kHz
2000 kHz	2000 kHz		11000 kHz	10666 kHz
2250 kHz	2245 kHz		12000 kHz	11636 kHz
2500 kHz	2509 kHz		13000 kHz	12800 kHz
2750 kHz	2723 kHz		14000 kHz	14222 kHz
3000 kHz	2976 kHz		15000 kHz	14222 kHz
3250 kHz	3282 kHz		16000 kHz	16000 kHz
3500 kHz	3459 kHz		18000 kHz	18285 k Hz

Table 5.1: Frequency measurement test

*Note: At higher frequencies, if the real value and the measured value are equal, it does not mean that the frequency measurement is accurate within one Hz or kHz, but it means that the real frequency happens to be exactly matching to one value that can be calculated. For example, when measuring the real frequency of 16000 kHz, measured frequency equals to 16000 kHz as well. This would suggest that the measurement is absolutely accurate, but in reality, the closest smaller value that could be measured is 14222 kHz and the closest larger value that could be measured is 18285 kHz. This means that measurement 16000 kHz would be returned for a wider range (approx. 15111 kHz to 17126 kHz).

Discussion and limitations

As can be seen from table 5.1, the frequency measurement works very well for lower frequencies, but is a little limited for higher frequencies. This is due to poor measurement resolution at higher frequencies, which is caused by the timer's frequency. For better resolution at higher frequencies, the timer's frequency would need to be much higher. Overall, the frequency measurement can be classified as successful and its function is more than adequate for this purpose.

One characteristic of this calculation needs to be taken into consideration while calculating the frequency. The calculation itself takes relatively long time. For current implementation with input capture array of size 20, calculation takes approximately 450µs to execute. This means that calculation needs to happen outside time-critical events such as interrupts.

One other important characteristic is the measurement range. The absolute minimum frequency that can be measured in this configuration is $f_{min} = 1953Hz$, but in reality measuring frequencies near this value are not recommended as the difference between input capture events would need to be same or close to the whole counter range and the counter would most probably overflow and the measurement would be flawed. The theoretical absolute maximum frequency that could be measured is $f_{max} = 128MHz$, but this limit is highly unrealistic as the closest lower frequency that could be measured is $f = 64MHz$. SPI interface on this MCU can operate with maximum clock frequency $f_{clock} = 18MHz$, so this means that no larger frequency shall realistically be measured. Measuring frequency is still not ideal on the higher side of the measurement range, but it is considered adequate for this task.

5.2. Methods of SPI transmission using MCU

In order to successfully receive or transmit data via SPI interface, the slave must wait until clock pulses appear on the clock pin, read data from the MOSI line or transmit data on the MISO line on these clock pulses and write every byte received into some place in the MCU memory. The SPI peripheral works one byte at a time, meaning data is read and stored one byte at a time. After successful byte is received, this byte appears in the SPI data register (SPI->DR). This register must therefore be read and stored in-between each byte. There are 3 methods how to accomplish this.

Receive using blocking mode

SPI can function in blocking mode. In this mode, after the receive (transmit or transmitreceive) function is called, the CPU remains in this function and polls the SPI peripheral registers until a byte is received. When a byte is received, this byte from the SPI->DR is stored into the MCU memory and SPI registers are polled again for another byte. This function terminates after a number of bytes specified by the user were received. Functions for this method are:

```
HAL_SPI_Receive(hspi, pData, Size, Timeout); // Only receive
HAL_SPI_Transmit(hspi, pData, Size, Timeout); // Only transmit
HAL_SPI_TransmitReceive(hspi, pTxData, pRxData, Size, Timeout); // Transmit AND Rec
```

Where hspi is a SPI handle structure, pData is a pointer to the data buffer, pTxData is a pointer to the transmit buffer from which data will be transmitted, pRxData is a pointer to the receive buffer where receive data will be stored, Size is the number of bytes to transmit/receive, Timeout is the time in ms after which this function will terminate even if specified number of bytes were not transmitted/received.

This method could be used on an SPI master device as it is known exactly when data will be transferred on the SPI bus, but using it on an SPI slave device is extremely ineffective and inappropriate for almost any application. The reason being that a slave device does not control when data is transmitted over SPI bus, therefore times of data transmissions are not known and by using these functions, CPU would "freeze" in this function until SPI master decides to send or receive SPI data. This action would be an extreme waste of processing time.

Receive using interrupts

If SPI is controlled using the interrupt method, the main principles are the same as in blocking mode, but the CPU does not wait inside a function and poll the SPI registers, instead interrupts are used to trigger an action after a byte transmit/receive. This means that the function just sets up SPI peripheral for transmission and enables interrupt, therefore CPU is not "stuck" inside this function. If a successful byte is clocked on the clock pin, an interrupt is triggered and inside this interrupt, SPI data register is read and written into the memory receive buffer and also SPI is prepared for next byte receive/transmit and interrupt flag is cleared. SPI can be set to trigger a different interrupt after a specified number of bytes were received. Functions for this method are:

```
HAL_SPI_Receive_IT(hspi, pData, Size); // Receive only
HAL_SPI_Transmit_IT(hspi, pData, Size); // Transmit only
HAL_SPI_TransmitReceive_IT(hspi, pTxData, pRxData, Size); // Transmit AND receive
```

Where attributes of functions are same as in blocking mode, but Timeout is not needed.

This mode is superior to blocking mode as the CPU is not bound to oversee and wait for the whole process, instead the process is event driven and in the meantime, when master is not providing a clock signal, the CPU can do other things. But this method is not flawless. One thing that can cause problems is the interrupt time. It takes time for the CPU to enter and exit interrupts as well as to manage the SPI data register inside interrupt. This means that the minimum time between bytes can be limited. Another possible problem could be that in the time of SPI interrupt, the CPU can be doing some time-critical action and interrupting this action would cause problems.

Receive using DMA

Using a DMA (Direct Memory Access) controller means that the CPU does not have to manage the SPI data register and therefore the actual CPU time is not wasted on SPI transmissions. All needed action is done by the peripheral and the DMA controller. Upon successful byte clocked on the SPI clock pin, the SPI peripheral triggers a DMA request. On this request, the SPI data register is accessed by DMA controller and data is read from the SPI data register and written to the memory receive buffer. DMA controller can be set to trigger an interrupt when the specified number of bytes are received to indicate end of the SPI data transfer of known size. Functions for this method are:

```
HAL_SPI_Receive_DMA(hspi, pData, Size); // Receive only
HAL_SPI_Transmit_DMA(hspi, pData, Size);// Transmit only
HAL_SPI_TransmitReceive_DMA(hspi, pTxData, pRxData, Size); // Transmit AND receive
```

Where attributes of functions are the same as when using interrupts.

This mode is the most sophisticated as no actual CPU time is used to operate the SPI peripheral other than to setup DMA for next packet transmisson/receival.

Transmitting

If data has to be transmitted instead of received, the same 3 methods as described in the previous sections are used, but instead of reading from the SPI data register in-between bytes, data is written into the SPI data register and SPI internally stores this data into the Tx fifo buffer. SPI can work in full duplex mode, meaning data can be received and transmitted at the same time.

5.3. SPI peripheral drivers

At first, the unmodified HAL drivers were used for controlling the SPI peripheral. While optimizing the EEPROM emulator mode, SPI drivers were modified completely. The LL (Low Layer) drivers were used and all functions that control the SPI peripheral were written from scratch using the LL drivers. The created SPI peripheral driver functions were written to accommodate only the absolute minimum functionality needed for a better performance. Only the DMA method of SPI transmission is supported. SPI peripheral driver functions were developed in multiple iterations and the final iteration will be described.

5.3.1. Main idea

The developed SPI peripheral drivers are not universal as the user code needs to be added inside the driver functions if they were to be used for a different application. Every SPI transmission will trigger a receive interrupt and no transmit interrupts will be triggered by the DMA, but the transmit interrupt handler will be called from within the receive interrupt handler. This means that the slave can be set for receive only transmission, but every transmit will actually be transmit-receive transmission. If only transmit is desired, data will be received to dummy address and will be ignored.

5.3.2. Receive function

Receive function is defined as follows.

```
void SPI_receive_DMA(uint8_t *pData, uint16_t Size)
{
    /* Disable all SPI DMA channels and SPI peripheral */
    LL_DMA_DisableChannel(DMA1, LL_DMA_CHANNEL_2); // DMA Rx channel
    LL_DMA_DisableChannel(DMA1, LL_DMA_CHANNEL_3); // DMA Tx channel
    LL_SPI_Disable(SPI1);

    /* Set the transaction information */
    LL_DMA_SetPeriphAddress(DMA1, LL_DMA_CHANNEL_2, &SPI1->DR);
    LL_DMA_SetMemoryAddress(DMA1, LL_DMA_CHANNEL_2, (uint8_t *)pData);
    LL_DMA_SetDataLength(DMA1, LL_DMA_CHANNEL_2, Size);

    /* Enable the Rx DMA Stream/Channel  */
    LL_DMA_EnableChannel(DMA1, LL_DMA_CHANNEL_2);

    /* Set Rx FIFO buffer treshold for correct RXNE event generation */
    LL_SPI_SetRxFIFOThreshold(SPI1, SPI_CR2_FRXTH);

    /* Enable transfer complete interrupt for DMA Rx channel */
    LL_DMA_EnableIT_TC(DMA1, LL_DMA_CHANNEL_2);

    /* Enable Rx DMA Request */
    LL_SPI_EnableDMAReq_RX(SPI1);

    /* Enable SPI peripheral */
    LL_SPI_Enable(SPI1);
}
```

First, all the SPI DMA channels are disabled as well as SPI is disabled. This is to correctly reset any ongoing transmissions. The next transaction information is set for the SPI DMA receive (Rx) channel. The pointer pData indicates memory location, where received data will be stored. The DMA Rx channel is enabled. RxFIFOTreshold is set for the correct RXNE event generation which indicates that the SPI internal receive buffer is not empty. The transfer complete interrupt is enabled for the Rx DMA channel, which allows an interrupt to be triggered after the specified number of bytes were received. DMA request for the Rx channel is enabled inside SPI peripheral, which enables SPI to interface with the DMA Rx channel. Lastly, the SPI peripheral is enabled.

5.3.3. Transmit function

The transmit function does in reality always function as a transmit-receive function. This function is defined as follows.

```
void SPI_Transmit_DMA(uint8_t *pData, uint16_t Size)
{
    /* Start SPI receival based on current device mode */
    if ( spi_slave.curr_mode == SPI_EEPROM ) {
        SPI_receive_DMA(&dummy, 1);
    }
    else if ( spi_slave.curr_mode == SPI_receivE_RESPOND_RX ) {
        SPI_receive_DMA(&dummy, spi_slave.cnfg.data_size);
    }
    else if ( spi_slave.curr_mode == SPI_receivE_RESPOND_RXTX ) {
        SPI_receive_DMA(&spi_slave.rx_buff, spi_slave.cnfg.frame_size);
    }

    /* Set the transaction information */
    LL_DMA_SetPeriphAddress(DMA1, LL_DMA_CHANNEL_3, &SPI1->DR);
    LL_DMA_SetMemoryAddress(DMA1, LL_DMA_CHANNEL_3, (uint8_t *)pData);
    LL_DMA_SetDataLength(DMA1, LL_DMA_CHANNEL_3, Size);

    /* Enable the Rx DMA Stream/Channel  */
    LL_DMA_EnableChannel(DMA1, LL_DMA_CHANNEL_3);

    /* Enable Tx DMA Request */
    LL_SPI_EnableDMAReq_TX(SPI1);
}
```

The transmit function first calls a receive function, so that a proper interrupt is triggered after the specified number of bytes were transmitted. Based on the current device mode, a receival address is chosen either as a dummy address or as an address specified by the user in the code. Next, the DMA transmit (Tx) channel data are specified. pData is a pointer to a data array from where SPI will transmit bytes. DMA Tx channel is enabled. Tx DMA requests are enabled inside the SPI peripheral, which enables SPI to interface with the DMA Tx channel.

5.3.4. Receive interrupt handler

Handler for an interrupt triggered by a DMA flag is defined as follows.

```
void SPI_DMA_RX_IrqHandler(void)
{
    /* Clear the transfer complete flag */
    LL_DMA_ClearFlag_TC2(DMA1);

    /* Tickstart for BSY bit timeout */
    uint32_t tickstart = uwTick;

    /* DMA Normal Mode */
    if ( LL_DMA_GetMode(DMA1, LL_DMA_CHANNEL_2) == LL_DMA_MODE_NORMAL ) {
        /* Disable DMA Rx channel aswell as DMA Rx requests */
        LL_DMA_DisableChannel(DMA1, LL_DMA_CHANNEL_2);
        LL_SPI_DisableDMAReq_RX(SPI1);

        /* Check the end of the transaction */
        if ( SPI1->SR & SPI_SR_BSY ) {
            while ( SPI1->SR & SPI_SR_BSY ) {
                if ( (uwTick - tickstart) > SPI_DEFAULT_TIMEOUT ) {
                    /* something for breakpoint debugging, should go to error handl
                    tickstart = uwTick;
                }
            }
        }
    }

    /* Call Tx handler */
    SPI_DMA_TX_IrqHandler();

    /* Call user callbacks */
    SPISLAVE_SPI_IrqCallback();
}
```

First, the transfer complete flag is cleared. Next, a variable is initialized with the current system tick to note the current time. If DMA is set to normal mode (contrary to circular), DMA Rx channel is disabled as well as the DMA Rx requests inside the SPI peripheral. Also the device waits for the BSY (busy) bit in the SPI registers to have value zero. If DMA mode is set to circular, DMA channel is not disabled and the device does not wait for the BSY bit. Next, the SPI Tx interrupt handler is called. This interrupt handler does not have to be called every time, but calling it while it is not needed does not cause any errors and no real time loss. Lastly, a function is called which decides which user defined interrupt handler must be called based on the current device state and mode. All user interrupt code happens inside this function.

5.3.5. Transmit interrupt handler

This handler is not called inside an interrupt that was triggered by the DMA Tx channel flag, but instead it is called inside a receive interrupt. Handler is defined as follows.

```
void SPI_DMA_TX_IrqHandler(void)
{
    /* Clear the transfer complete flag */
    LL_DMA_ClearFlag_TC3(DMA1);
    /* Disable Tx DMA channel for SPI */
    LL_DMA_DisableChannel(DMA1, LL_DMA_CHANNEL_3);


    /* Normal case */
    LL_SPI_DisableDMAReq_TX(SPI1);
}
```

First, the transfer complete flag is cleared. DMA Tx requests are disabled inside the SPI peripheral. DMA SPI Tx channel is disabled. In all the modes of this device, transmit happens only in normal mode, therefore circular mode is not considered.

5.4. Reseting SPI

Resetting the SPI peripheral is needed mainly after a new configuration is loaded into the device. Not only does resetting the SPI calls all functions that are needed for initializing the device and the SPI peripheral, which in turn applies new configuration, but the internal SPI transmit buffer is cleared. This buffer cannot be cleared any other way instead of transmitting data on the SPI lines, but that is controlled by the SPI master. Clearing this buffer is needed after a transmit is initiated after another transmit was already initiated before but never completely executed. SPI reset procedure is as follows:

- Set the APB2 (advanced peripheral bus 2) reset register inside the RCC registry to 1. This will start resetting APB2.
- Wait the minimum necessary time
- Set the APB2 (advanced peripheral bus 2) reset register inside RCC registry to 0.
- Call a function that initiates SPI peripheral based on STM32CubeMx settings.
- Call a user function that initiates device in all modes.

5.5. Stream mode

Stream mode is the first operational mode of the device. This mode will be automatically chosen on device startup. Stream mode can be used to directly display incoming SPI data through the python API.

5.5.1. Main features

The main idea of the stream mode is to capture all incoming data on the SPI interface and transmit (stream) it directly to a PC where it can be read. In this mode, the device is acting receive-only, therefore no data will be sent out on the SPI data bus. SPI is master controlled, meaning the master controls when data transmission (master transmit or master receive) will happen by controlling the SPI clock pin. This means that the slave needs to wait with data ready until a clock signal appears. After the data was transmitted by the slave, the slave needs to prepare the next data before the next clock signal appears on the SPI clock pin.

5.5.2. Early implementation

The first implementation was done by SPI receival using interrupts. Figure 5.3 displays a block diagram of the implemented algorithm.

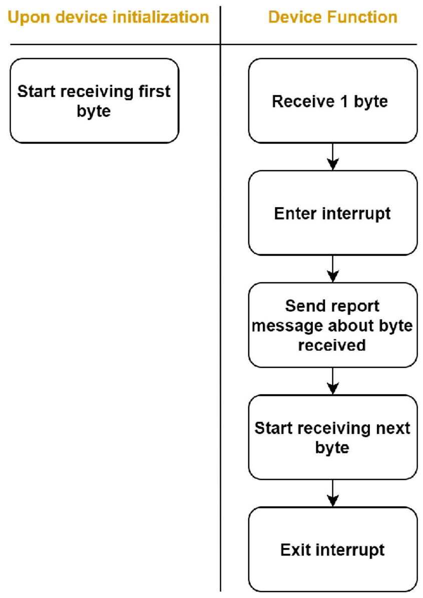

Upon device initialization
Device Function
Start receiving first byte
Receive 1 byte
Enter interrupt
Send report message about byte received
Start receiving next byte
Exit interrupt

Figure 5.3: Block diagram of an algorithm for the first implementation of stream mode

This method is the easiest way to implement this mode. Upon mode initialization, a single byte is set to be received. After this single byte is received, an interrupt is triggered. In this interrupt, a report message about this byte is formed and sent to the python API via UART. Lastly, the SPI is set to receive next byte and the CPU exits the interrupt.

Discussion

This is the easiest way to implement stream mode, but it has some major limitations. One limitation that renders this solution almost unusable is the interrupt time. The device receives bytes one by one and in-between each one is an interrupt. This means that the CPU needs to exit the interrupt before the next byte arrives or received data will not be processed correctly. Also the report message needs to be successfully sent before the next byte arrives, while UART interface is generally much slower than SPI interface. Data is usually sent in packets, which contain more than one byte, therefore a master would need to artificially add delay between bytes in order for this mode to work correctly. Worst case for SPI slave to receive data is a continuous data stream, meaning the time in-between bytes is the same as the time in-between bits (clock pulses). This means that in order for this mode to work, the clock frequency would have to be low enough for an interrupt to execute in-between clock pulses. If worst case is considered, clock frequency f_{clock} would be severely limited to less than 50 kHz, while the maximum possible SPI clock frequency for this MCU is 18MHz.

One way to improve this method would be to implement a FIFO buffer. In the interrupt, rather than sending a report message immediately, received bytes would be stored in this FIFO buffer and sent later asynchronically in the main while cycle. This would improve the maximal clock frequency, but still it would not reach even 500 kHz. Also, this means that the FIFO buffer would have to be large enough to accommodate a data packet or multiple data packets and there would have to be enough time where the SPI is in idle to send the report messages from this buffer.

5.5.3. Final implementation

Head-Tail FIFO buffer

For the final implementation, a different solution was found. The main component of this solution is a relatively large FIFO buffer, where the received bytes will be stored. As the SPI receival method, DMA is used in a circular mode. Circular mode means, that after the whole buffer is filled, received bytes will be stored in the beginning of the buffer again. An interrupt will be triggered when the buffer is filled, but only to increment a variable that describes the number of overflows. Therefore, in this SPI setup, after an SPI receival is initialized, no CPU time is used to actually receive any bytes as it is the responsibility of the DMA controller.

There are two pointers that point to the specific index of this buffer. One pointer points to the current last index, where data was written by the DMA controller. This pointer will be called ”head”. The second pointer points to the last index, which was sent to a PC in a report message. This pointer will be called ”tail”. The tail pointer always chases the head pointer. If the head and the tail pointer point to the same index, no action happens. If the head pointer is incremented, action is needed so that the tail pointer is being incremented also until the head and the tail pointers are equal again.

As no data is being deleted from the buffer, it is important to note how many overflows each of the pointers have. Overflow means reaching the end of the buffer and looping back to the beginning.

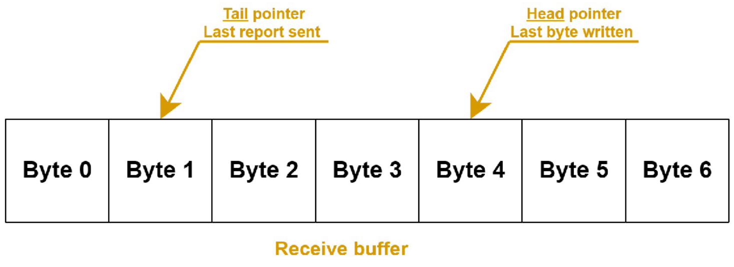


Figure 5.4: Stream buffer with head-tail pointers

Figure 5.4 represents a seven byte stream buffer with the head pointer pointing at the byte number 4 and the tail pointer pointing at the byte number 1. This would mean that action is required. In order to equalize the head and the tail pointers, the byte number 2, the byte number 3 and the byte number 4 would need to be reported to a PC API. In the final implementation, a stream receive buffer with the length of 1000 bytes was used.

Valid conditions for sending a report

The indices of the bytes that need to be reported are indicated by the state of the head and the tail pointers and by the number of overflows of each pointer. The conditions that indicate that a report messages needs to be sent are as follows:

- **The head pointer is larger than the tail pointer with the same number of overflows**
  This means that the bytes on indices starting from the tail pointer (not included) to the head pointer (included) need to be reported.
- **The head pointer is smaller than the tail pointer AND head the pointer number of overflows is exactly one greater than the tail pointer number of overflows**
  This means that the bytes need to reported starting from the tail pointer index (not included) to the end of the buffer and also the bytes starting from the index 0 to the head pointer index (included) need to be reported.

Head and tail pointer errors

There can be a few errors with the head and the tail pointers. These errors can either indicate that the SPI bytes are being received too fast and there is not enough time for sending the report messages or that the head and the tail pointers are in a state, which should not be possible and therefore an unknown internal error occured. The error states are as follows:

- **The head or the tail pointers are larger than the buffer size**
- **The tail pointer number of overflows is greater than the head pointer number of overflows**
  This indicates that the tail pointer overtook the head pointer.
- **The head pointer is greater than the tail pointer WITH the same number of overflows**
  This indicates that the tail pointer overtook the head pointer.
- **If the head pointer number of overflows is larger than the tail pointer number of overflows by a margin larger than 1**
  This means that SPI data are being received too fast and there is not enough time to send report messages.

Main algorithm

The CPU is not involved in receiving SPI data, therefore no trigger will signalize the receival of a new byte, therefore, if report needs to be sent is checked in the main while loop. The block diagram of the whole algorithm can be seen in figure 5.5.

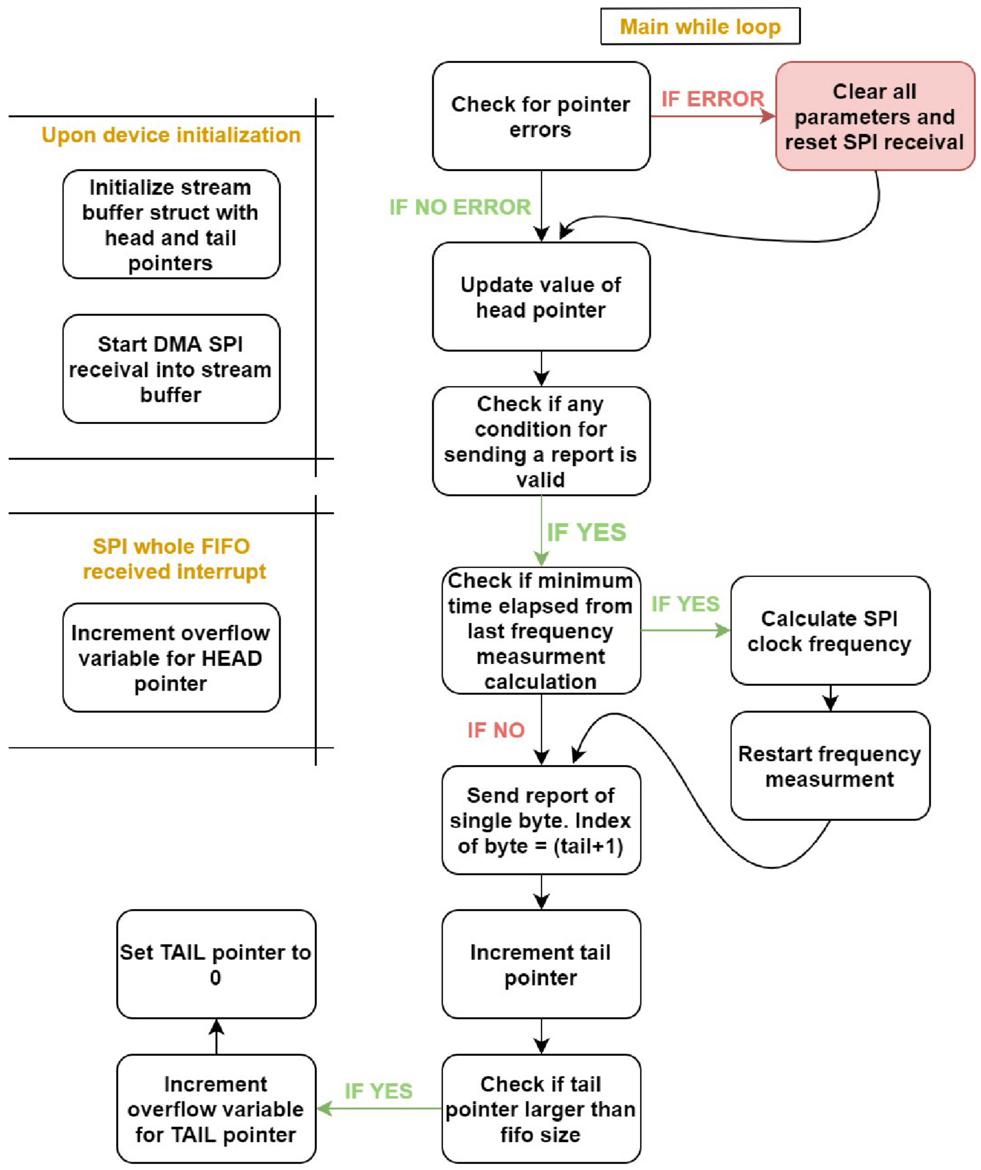

Main while loop
Upon device initialization
Initialize stream buffer struct with head and tail pointers
Start DMA SPI receival into stream buffer
SPI whole FIFO received interrupt
Increment overflow variable for HEAD pointer
Check for pointer errors
IF ERROR
Clear all parameters and reset SPI receival
IF NO ERROR
Update value of head pointer
Check if any condition for sending a report is valid
IF YES
Check if minimum time elapsed from last frequency measurment calculation
IF YES
Calculate SPI clock frequency
Restart frequency measurment
IF NO
Send report of single byte. Index of byte = (tail+1)
Increment tail pointer
Check if tail pointer larger than fifo size
IF YES
Increment overflow variable for TAIL pointer
Set TAIL pointer to 0

Figure 5.5: Block diagram of the final implementation of stream mode algorithm

Upon device initialization or when the device switches into this mode, two things need to be performed. The first is to initialize/clear all parameters associated with the stream mode. These parameters are the buffer itself, the overflow variables for the head and the tail pointers. The second thing is to initialize DMA to receive SPI data to a specified buffer in circular mode and with the length of SPI receival same as the length of the buffer.

When enough bytes are received to fill the whole buffer, an interrupt is triggered. In this interrupt, only one action is performed. The overflow variable that indicates how many times the head pointer reached the buffer end is incremented.

All the other main actions are performed in the main while loop of this MCU and can be described as follows.

- **Check for the pointer errors**
  First any head and tail pointer errors are checked. If any error occurs, all stream mode parameters are cleared and SPI DMA receival is reset and restarted.

- **Update the value of the head pointer**
  Next, the head pointer is updated. The information about what index will the DMA write next can be found in the DMA register. Let's assume that this register is called *data_length*. This register indicates how many more bytes need to be received until the end of the buffer. From this value,the current head index can be calculated as:

  ```
  head_pointer = SPI_STREAM_BUFFER_SIZE - dma_register_data_length;
  ```

- **Check if any condition for sending a report is valid**
  Once the head pointer value is updated, the conditions required for any report to be sent are checked.

- **Check if the minimum time elapsed from the last frequency measurement calculation**
  If any of the conditions are valid, first the time elapsed from the last frequency measurement calculation is checked. If the minimum time elapsed, the frequency is calculated and the frequency measurement is restarted. Having minimal time in-between the frequency calculation ensures that the lengthy calculation is not performed too frequently.

- **Send a report of a single byte**
  After the frequency minimum time in-between calculations is checked and possibly the frequency is calculated, a report message is initiated and sent to PC via UART. This message contains only a single byte and the index of this byte is one larger than the tail pointer. All the possible report messages will be listed in chapter 7.3.6.

- **Increment the tail pointer**
  Lastly, the tail pointer is incremented by the value of 1. Overflow of this tail pointer is checked. If the tail pointer after incrementation reaches the end of the buffer, the overflow variable for the tail pointer is incremented and the tail pointer is set to 0.

It is important that there is no other activity in the main while loop, that would slow down its iteration. This algorithm needs to be executed frequently enough, so that the report sending can keep up with the SPI data receival. In other words, the algorithm needs to be executed frequently enough so that the head pointer does not overtake the tail pointer by the size of the whole buffer.

5.5.4. Discussion and limitations

Stream mode is implemented by using a FIFO buffer with a head and a tail pointer and the DMA method of receiving SPI data works effectively, however one limitation exists. Received SPI data are being written to a buffer. Later, this buffer is asynchronically checked if any new data exist and if the new data exist, this data is sent in reports to a PC. It is possible that there is not enough time to send this new data in the buffer and the data that needs to be sent will be overwritten by new data that will be received by the SPI interface. This is caused by either too small of a buffer or a continuous stream of SPI data that contains more bytes than the buffer size. This means that if a continuous SPI data stream of infinite size is assumed, this method will not work correctly. In reality, such transmissions are very uncommon and usually there is enough time in-between packets to report all the received bytes. Also, this mode is aimed at testing and debugging masters, so transmitting a continuous data stream would not be beneficial for the user as the main purpose of this mode is to manually check what is being sent. In the final implementation, buffer with size of 1000 bytes is used, which should be plenty for any frame or packet as they will rarely be larger than 1000 bytes.

5.5.5. Possible improvements

For the future development, one obvious improvement could be made. Instead of including only one byte in each report message, all bytes that need to be sent could be included in the report message. This would vastly improve the total time that would be needed to equalize the head and the tail pointers as all needed bytes would be reported in one pass of the main while loop. But this would mean, that this one main while loop iteration would be longer as more bytes need to be processed.

5.6. LUT mode

LUT mode is the second operational mode of the device. This mode can be used to automatically respond to the received requests from an SPI master with preset responses according to the current received request.

5.6.1. Main features

In contrary to the stream mode, LUT (Look up table) mode will not only receive data, but also transmit data. The data that will be transmitted is dependent on the last data that was received. From every transmission, a report will be formed and sent to a PC via UART. This mode can work in full-duplex mode as well as in half-duplex mode. Half-duplex mode means that one single transmission is either a transmit or a receive, not both at the same time. This means that in half-duplex mode, one data bus (MISO or MOSI) will be inactive. In full-duplex mode, data is being transmitted and received at the same time, therefore both data lines are active.

5.6.2. Implementation

In this mode, SPI transmission will be performed in packets. The packets will contain a set number of bytes. The number of bytes in a packet can be configured and can only be changed by changing the configuration. This means, that for one configuration, only one length of a packet can be received and transmitted. Packets will not be separated into smaller sections, meaning, the whole packet will be considered as data (no heading or footer). For receiving and transmitting, the DMA method of SPI transmission will be used and after any successful packet transmission, an interrupt is set to trigger. The frequency will be measured and calculated for every packet separately.

Lookup table structure

The main feature of this mode is the lookup table itself. The lookup table contains multiple request-response pairs. If a packet equal to a request in LUT is received, in the next transmission, a packet with the value of the response belonging to the received request packet will be transmitted.

LUT row	Request packet	Response packet
0	[1, 2, 3, 4]	[200, 100, 5, 10]
1	[10, 20, 30, 40, 50, 60]	[11, 21, 31, 41, 51, 61]
2	[255, 255]	[1, 1]
3	[100, 100, 100]	[100, 100, 100]

Table 5.2: Lookup table structure

Example of a look up table instance can be seen in table 5.2. In this table, there are 4 rows with reuqest-response pairs. If any of the requests is received by SPI, in the next transmission, the response packet from the same LUT row will be transmitted.

LUT parameters in current implementation are as follows. The LUT number of rows equals to 10. The maximum request and response size equals to 10. These parameters are relatively low, but in order to increase them significantly, MCU with more memory would need to be used.

Lookup table conditions

A few conditions must be set in order for LUT to be valid:

- **Packet lengths**
  The request and the response in the same pair need to have the same length of bytes, but different pairs can have different lengths.

- **No duplicate requests**
  Every request inside the LUT must be unique. If duplicate requests are found in the LUT, the LUT is not valid. This is because if multiple identical requests were present in the LUT, only the one with the smallest index would be used and the other ones would be redundant. Duplicate responses are allowed.

Important LUT mode variables and parameters

There are important variables and parameters associated with this mode, that are needed to describe the current configuration and state. These variables are:

- **LUT**
  The first variable is the LUT itself. Structure of the LUT was described in previous section. In C implementation, requests and responses are split into separate variables. Each of this variable is a two dimensional array that describes all of the

requests or responses for the whole LUT. Every row of these variables is a different request/response and pairs are defined by the same row index. Both of these variables are part of the LUT handle struct.

- **sizes**
  This variable is an array with the same length as the number of LUT rows and with type 8 bit unsigned integer. Values of this array describe byte length of the current request-response pair. Value on the specific index of this array belongs to the same row index of LUT. This variable is a part of LUT handle struct.

- **is_used**
  This variable is an array with the same length as the number of LUT rows and with boolean type (0 or 1). Values of this array describe if the current LUT row on the same index is currently being used or if it is empty. This variable is important for LUT configuration as LUT rows can be configured only if they have been cleared before. This variable is a part of the LUT handle struct.

- **default response**
  An array of type 8 bit unsigned integer. A default response needs to be defined for situations when the received packet is not present in the LUT request array. If this happens, no LUT row is chosen and the default response is returned.

- **full-duplex mode enable**
  A single boolean value that describes if the device is in full-duplex or half-fuplex mode.

- **data length**
  A single 8 bit unsigned integer value describing the current packet length to be received or sent.

Choosing a response

Once a request was received, the according response needs to be chosen. First, the index of LUT row, where the LUT request and the received request are identical, is determined. LUT request must have the same byte length as is the current received packet length, which is the same as the current packet length set in the configuration. If the LUT does not contain the same request, the default response is chosen as next response. If LUT index is found with the same request, the response from this LUT index is chosen as next response. The response in LUT should have the same size as the request in the same LUT row, but this condition is not checked while configuring LUT in current implementation. This should not be a problem, if the action for different sizes is known. If the response is larger than the request, only first request size number of bytes are chosen as the next response. If the response is smaller than the request, the response is appended with zeros.

Main algorithm - common

LUT mode can function in 2 modes. Half-duplex and full-duplex. The main idea of these modes is the same, but in order for them to function properly, some differences in implementation need to exist. Despite these differences, some parts of the algorithm are the same. The parts of the algorithm that are the same are what happens on device or LUT mode initialization and what happens in the main while loop of the MCU. The common parts of the algorithm can be seen in the block diagram 5.6.

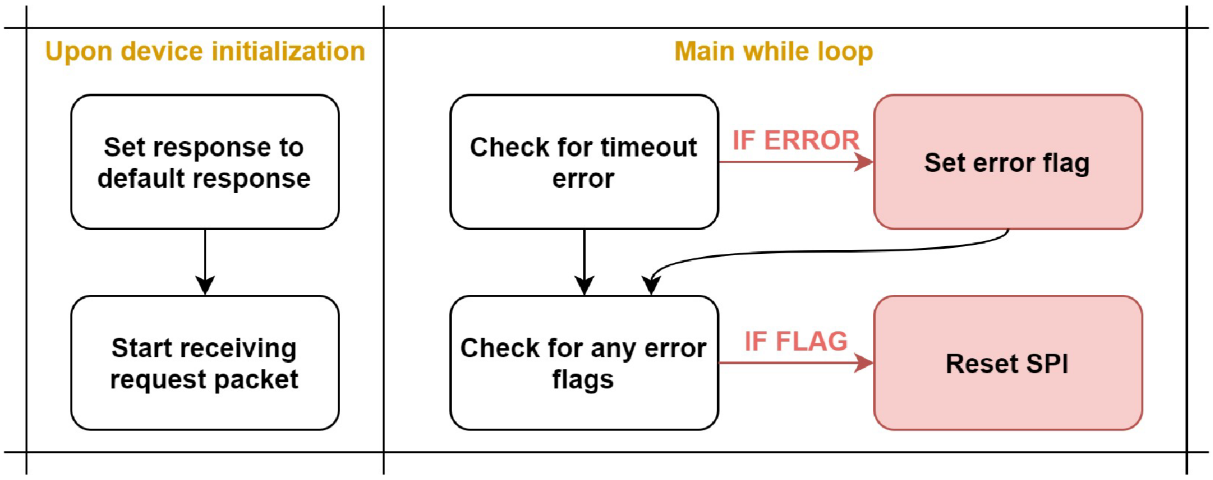

Upon device initialization
Set response to default response
Start receiving request packet
Main while loop
Check for timeout error
IF ERROR
Set error flag
Check for any error flags
IF FLAG
Reset SPI

Figure 5.6: Block diagram of the common algorithm for half-duplex and full-duplex mode

First, the algorithm for device or mode initialization is described:

- **Set response to default respone**
  The device does have a default response defined. This response will be transmitted either when the received request is not present inside the LUT or when the first transmission after initialization happens in full-duplex mode.

- **Start receiving/transmitreceiving request packet**
  Set SPI to receive or transmit-receive using DMA.

Next, the algorithm for the main while loop is described:

- **Check for timeout error**
  A timeout error occurs when no action on the SPI bus appears after SPI receive/transmit-receive initialization for a predefined time. The main reason for the existence of this error is to catch any unknown errors that would cause the SPI peripheral to not work correctly. If this error happens, error flag is set.

- **Check for error flag**
  Checks if any error flag is active. If any error flag is active, the SPI is reset. SPI reset procedure can be found in chapter 5.4. Possible error flags:

  – Timeout error

  This function exists because in earlier implementation, more errors existed and it is probable that in future development, more error flags will exist.

The core algorithms for half-duplex and full-duplex mode are different and will be described separately in the next sections.

Main algorithm - half-duplex

In half-fuplex operation, the SPI interface is only used one direction at a time. Meaning that for one transmission, only the MISO or the MOSI line is active while the other line is inactive. The core algorithm of this mode happens inside interrupts. There are two interrupts associated with this mode of operation. The receive and the transmit interrupt. The receive interrupt will trigger after a packet of a known size was received and the transmit interrupt will trigger after a packet of a known size was transmitted. Block diagram of the code for these algorithms can be seen in figure 5.7.

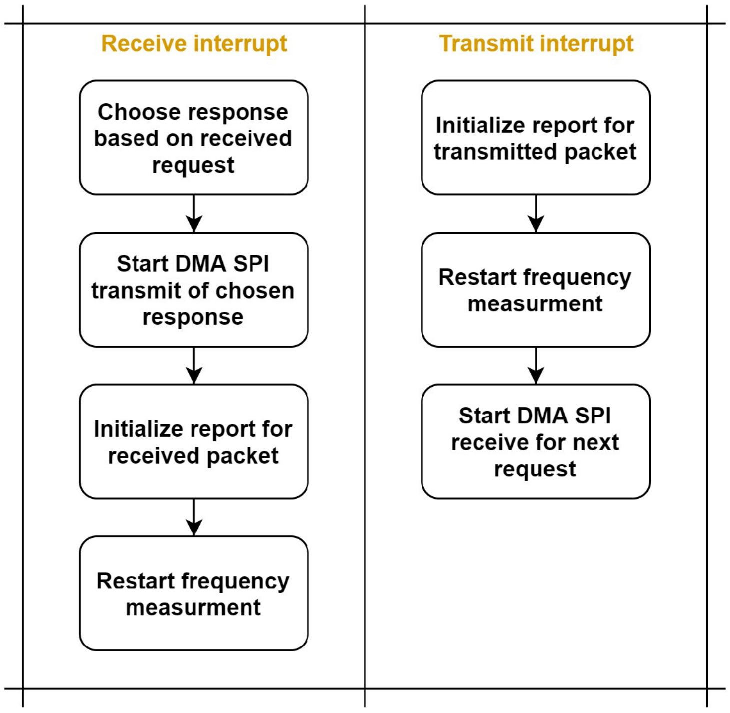

Receive interrupt
Choose response based on received request
Start DMA SPI transmit of chosen response
Initialize report for received packet
Restart frequency measurment
Transmit interrupt
Initialize report for transmitted packet
Restart frequency measurment
Start DMA SPI receive for next request

Figure 5.7: Block diagram of the half duplex mode algorithm for receive and transmit interrupts

First, the receive interrupt is described:

- **Choose the response based on the received request**
  This interrupt triggers after a packet was received. Based on this packet, an appropriate response needs to be chosen. The process of choosing this response was described in chapter 5.6.2.

- **Start the DMA SPI transmit of a chosen response**
  Once a response was chosen, SPI DMA transmit is initiated. This transmit will happen only after the master provides a clock signal and after the response packet is sent, a transmit interrupt will trigger.

- **Initialize a report for the received packet**
  A report message describing the received packet is formed. This report also contains the input capture data of frequency measurement. Complete report message forming and sending will be described later in chapter 5.7.

- **Restart the frequency measurement**
  The frequency measurement is restarted. The frequency measurement was described in section 5.1.

Next, the transmit interrupt is described:

- **Initialize a report for transmitted packet**
  A report message describing the transmitted packet is formed. This report also contains the input capture data of the frequency measurement. Complete report message forming and sending will be described later in chapter 5.7.

- **Restart the frequency measurement**
  The frequency measurement is restarted. The frequency measurement was described in section 5.1.

- **Start the DMA SPI receive for next request**
  After a response was sent, SPI needs to be set to receive the next request using DMA. After the next request will be received, a receive interrupt will trigger again.

In this operation mode, every request is received by itself followed by transmitting of a response. This action is performed repeatedly.

Main algorithm - full-duplex

In full-duplex operation, the SPI interface receives and transmits data at the same time. This means that both the MISO and the MOSI lines are being used for every clock pulse. The core algorithm of this mode happens inside an interrupt. In contrary to half-duplex mode, interrupts are not separated for receive and transmit and therefore only one interrupt is defined. This interrupt is labeled as a transmit-receive interrupt. Block diagram of the code for this interrupt can be seen in figure 5.8.

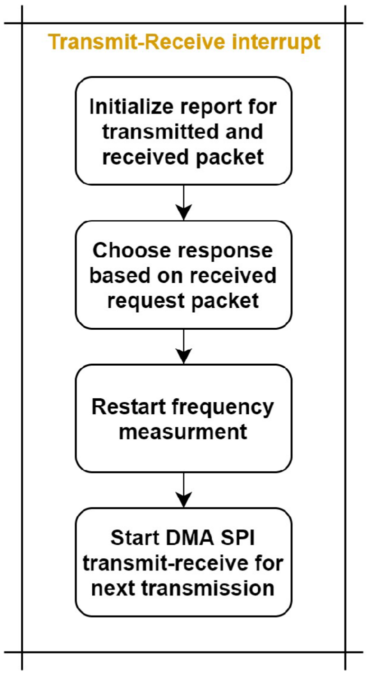

Transmit-Receive interrupt
Initialize report for transmitted and received packet
Choose response based on received request packet
Restart frequency measurment
Start DMA SPI transmit-receive for next transmission

Figure 5.8: Block diagram of half duplex algorithm for receive and transmit interrupts

Transmit-receive interrupt is described:

- **Initialize a report for transmitted and received packet**
  A report message describing transmitted and received packet is formed. This report also contains the input capture data of the frequency measurement. Complete report message forming and sending will be described later in chapter 5.7.
- **Choose a response based on the received request**
  In this transmit-receive transmission, a request packet is received. Based on this packet, an appropriate response needs to be chosen. The process of choosing this response was described in section 5.6.2. This response will be sent in the next transmit-receive transmission sequence.
- **Restart the frequency measurement**
  The frequency measurement is restarted. The frequency measurement was described in section 5.1.
- **Start DMA SPI transmit-receive for next transmission**
  After a transmit-receive transmission was executed, SPI needs to be set for another transmit-receive transmission using DMA.

In this mode, the response packet from the previous request is being transmitted while the current request packet is being received . This action is performed repeatedly.

5.6.3. LUT configuration

In order for this mode to be effective, LUT needs to be easily configured. In order for a LUT row to be configured, the row must either be not in use (is_used variable set to 0) or it must be cleared prior to writing. The specific commands for LUT configuration will be described in section 7.3.4

Clearing a LUT row

LUT can be cleared one row at a time and therefore a specific index needs to be provided. If a Clear LUT command is received, first, the errors are checked. If any error occurs with the command, the command is declared invalid and no action is performed. If no error occurs, request and response data and its sizes on LUT row, which index was specified in the command are cleared (set to 0). Also, is_used variable is set to 0 as well on the according index.

Possible clear LUT errors:

- **Index out of range**
  LUT index received in command is larger than LUT number of rows.

Writing a LUT row

LUT can be written one row at a time and therefore specific index needs to be provided. If a Write LUT command is received, first, the errors are checked. If any error occurs with the command, the command is declared invalid and no action is performed. If no error occurs, request and response data in LUT are changed according to the request and the response data received in command. The "sizes" variable is changed accordingly and the "is_used" variable is set to 1 on according index also.

Possible write LUT errors:

- **Index out of range**
  LUT index received in the command is larger than the LUT number of rows.
- **Row is used**
  LUT row on the index that was received in the command is already in use. To successfully write this row, this row needs to be cleared first.
- **Duplicate request found**
  The same request as the command provided was found present on another LUT row. No duplicate requests can exist in LUT.
- **Request size**
  The request size provided by the command is larger than the maximum request size.
- **Response size**
  The response size provided by the command is larger than the maximum response size.

5.6.4. Discussion and limitations

LUT mode is aimed for testing the most basic logic in real SPI masters. The responses for specific requests can be set and these responses will be transmitted by the slave device after a request is received.

One limitation is that this mode does not work with dynamic length packets. By dynamic length packets, packets that define theirs length in the beginning of the actual packet are meant. This kind of functionality was implemented in early design, but in order to accommodate this functionality, bytes in the packet needed to be received and processed individually one by one. This meant extreme SPI frequency limitation. If a continuous data stream was considered (time in-between bytes same as time in-between bits), the absolute maximum SPI clock frequency was in the range of 50 kHz, which is unacceptable as the maximum SPI frequency for this MCU is 18MHz.

SPI clock frequency therefore is not limited, but not all timings are without a limit. Packets need to be processed in interrupts in-between packets. In order for these packets to be processed successfully, there needs to be a minimal time in-between packets. This means that the frequency of the actual packets is limited. This maximum packet frequency was not specifically tested, but based on other tested interrupt times, this frequency is estimated to be in the range of 150-250 kHz, which would mean the time in-between packets would need to be atleast 4-6µs. For the future development, this interrupt time will need to be accurately measured.

5.6.5. Possible improvements

One useful improvement for the future development would be to add a configurable CRC to the packet, which would be checked and reported to the user. Most of the packets used nowadays use CRC for error checking and it would be useful to implement this feature. CRC polynomial would be configurable.

5.7. Command and report messages handling

Algorithm contains structs, that can completely describe received commands via UART interface or report and acknowledgement messages for every type of SPI transmission and command. The specific types of commands and report, acknowledgement messages will be described later in chapter 7.3.

5.7.1. Command handling

Commands are received via the UART interface during interrupts. Once a valid command is received, the command is not executed right away. Instead a struct instance is initiated with all the information regarding received command. This is because if the command was executed right away, some other action with higher priority could interrupt the execution of this command and in the meantime, another command would be received, which would cause an error as the first command did not finish executing yet. All commands will be executed asynchronically from the main while loop.

5.7.2. Report message handling

Same as the commands, the report messages are not sent at the time when the SPI transmission ended, but rather a struct instance is initiated allowing the report messages to be sent later asynchronically in the main while loop. This way is chosen because if a report messages would be sent right away, the MCU could receive another SPI transmission in the time, when previous report message is being sent. A report message array also contains acknowledgement messages.

5.7.3. Main while loop commands and report messages handler

The commands and the report messages are stored in two arrays where each element of the array represents either a command instance of struct or a report message instance of a struct based on which array is being referenced. Both command and report messages arrays have two pointers. One pointer points to the index, where the next instance will be written (write pointer) and the second pointer points to the next instance, which should be executed/reported (execute pointer). Both command and report message handlers are executed in every iteration of the main while loop.

Command handler

Algorithm of the command handler can be described as:

- **Check if the current command instance of a struct has not been executed yet**
  The current command instance is defined by the execute pointer.
- **Execute command**
  If the current command instance has not been executed yet, execute it. Check errors before execution if any errors could exist based on specific command.

- **Increment execute pointer**
  The execute pointer is incremented.

Report message handler

The report message algorithm of the report message handler can be described as:

- **Check if the current report message instance of a struct has not been sent yet**
  The current report message instance is defined by the execute pointer.

- **Send message**
  If the current message instance has not been sent yet, send it. Calculate the measured frequency before every report message that includes frequency measurement data.

- **Increment execute pointer**
  The execute pointer is incremented.

5.7.4. Discussion and limitations

Report messages are stored inside a buffer. Each element of this buffer is a struct representing one SPI transmission. The buffer does not have infinite size and therefore there is a maximum number of report messages that can be stored at the same time. This means that if report messages are being created faster than they are being sent, this buffer could overflow and some report messages could be lost. If this situation occurs, internal error flag is thrown and report messages will no longer be created. In order for this situation to happen, a large number of SPI transmissions would need to be performed continuously.

6. Advanced functionality - EEPROM emulator mode

This whole chapter describes the EEPROM emulator operational mode. This mode is the last of the operational modes of this device and is considered advanced as its main purpose is not to test a general SPI master device, but to test a specific SPI master device that is made for controlling one specific slave device. The main idea of this mode is that the device behaves identically to the real SPI EEPROM device, which means that it does not matter if the master is connected to an emulator device or a real EEPROM device from the master's point of view. The process of enabling this mode can be found in chapter 7.1.1.

6.1. EEPROM Selection

Before developing an EEPROM emulator mode, first a real EEPROM needs to be chosen that will be emulated. *25AA160D* EEPROM was chosen manufactured by the *Microchip* company. This EEPROM was chosen for various reasons. One of the criteria for choosing a real EEPROM model was that it must be possible to simulate the whole EEPROM memory in MCU's RAM memory. The second criterion was that the EEPROM must have standardized instruction set, that is common amongst other EEPROM and FLASH devices.

6.1.1. EEPROM attributes

25AA160D has these attributes:

- **Size**
  Total memory capacity is 2kB or 2048 bytes. This capacity should be easily simulated in *STM32F303K8* MCU RAM memory.
- **Word length**
  This device contins 8bit words, therefore on one address of memory, one byte is stored.
- **Operating voltage**
  The operating voltage of this device is 1.8-5.5V. This means that for interfacing with an SPI slave device, no logic level shifters will be required.
- **SPI clock frequency**
  The maximum SPI clock frequency for this device is 10MHz. This device is a SPI slave.

More details about this specific EEPROM device can be found in [21].

6.1.2. Hardware pins

There are 6 data pins in total on this EEPROM device. These data hardware pins are:

- **CS**
  Chip select pin. Used for SPI
- **MISO**
  MISO pin used for SPI.
- **MOSI**
  MOSI pin used for SPI.
- **SCK**
  Clock pin used for SPI.
- **WP**
  Write protect pin. This pin is ignored in this emulator design.
- **HOLD**
  Hold pin. This pin is ignored in this emulator design.

6.2. EEPROM features

While creating a model of this EEPROM, it is important to note features that need to be replicated.

6.2.1. Memory organization

Memory is organized in a 2048 byte array. Every byte has its own address. The address is a 11 bit value, but in instructions, 16 bit value will be provided and 5 MSB bits will be ignored. The memory array is separated into pages. In this specific EEPROM type, page size is 32 bytes. This means that the memory array has 64 pages in total.

6.2.2. Status register

The status register has 8 bits and is used to store the current configuration of the EEPROM. The status register can be read by the RDSR command and R/W (Readable/Writable) bits can be configured by the WRSR command.

Bit number	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Access	R/W	-	-	-	R/W	R/W	R	R
Bit name	WPEN	-	-	-	BP1	BP0	WEL	WIP

Table 6.1: Status register

*Note - R/W (readable/writable) — R (read only)

WPEN

This bit is writable and enables the WP (write protect) hardware pin. WP pin is not used in the designed emulator, therefore this bit is not needed.

BP0 and BP1 and block protect

BP0 and BP1 bits are used for the block protect functionality. These bits are R/W. By defining these bits, it is possible to protect a block of memory data. If a block is protected, it is impossible to write to these memory locations. Three possible block protect configurations:

- **BP1 = 0 & BP0 = 0**
  Block protect inactive.
  Protected addresses : None
- **BP1 = 0 & BP0 = 1**
  Upper quarter of memory array protected
  Protected addresses : 0x0600 - 0x07FF
- **BP1 = 1 & BP0 = 0**
  Upper half of memory array protected.
  Protected addresses : 0x0400 - 0x07FF
- **BP1 = 1 & BP0 = 1**
  Entire memory array protected.
  Protected addresses : 0x0000 - 0x07FF

WEL

Read only bit indicating current status of the write enable latch. The write enable latch can be set using the WREN command. The write enable latch is reset when these 3 conditions are met:

- **Succesfull WRDI instruction is executed**
- **Automatically after succesfull WRITE operation**
- **Automatically after succesfull WRSR operation**

WIP

Read only bit that indicates if the EEPROM device is currently writing data. If the EEPROM is in WIP (write in progress) state, value of this bit will be 1 and the only possible instruction that can be executed in this state is RDSR.

6.2.3. Instruction set

Operation name	**Operation code**	**Description**
READ	0x03	Read data from memory
WRITE	0x02	Write data to memory
WRDI	0x04	Reset write enable latch
WREN	0x06	Set the write enable latch
RDSR	0x05	Read status register
WRSR	0x01	Write status register

Table 6.2: Instruction set for 25AA160C/D EEPROM [21]

Timing diagrams for each operation can be found at [21].

Chip select pin assertion

There are 6 instructions available for this EEPROM device. Every instruction must have the same chip select pin procedure. Instruction starts by putting the chip select pin to logical LOW. After this, the command data are transmitted. To sucesfully terminate current instruction, the chip select pin must be turned HIGH. This means that in-between every instruction, the chip select pin must be logical HIGH for a certain amount of time. Description of these 6 instructions are as follows.

READ

After the chip select is turned logical LOW, SPI master must transmit an operation code with the value of **0x03**. After the operation code, 16 bit address needs to be transmitted by the SPI master. 5 MSB bits of this address are "don't care" bits as the memory capacity is 2048 bytes. After the operation code and the address are successfully received by the EEPROM slave, SPI master must provide clock pulses and the memory data stored at the specified address will be shifted onto the MISO line. If master proceeds to provide clock pulses for more than 1 byte, subsequent bytes in the EEPROM memory will be shifted onto the MISO line, while internal address pointer increments automatically. If the internal address pointer reaches the highest address, it will roll over and start at address zero again. This means that data can be read indefinitely. Operation is terminated by putting the chip select pin to logical HIGH level.

WRITE

Prior to any WRITE attempt, write enable latch must set. This is done by WREN command and therefore successful WREN command must be clocked onto the SPI bus before the WRITE command. After the chip select is turned logical LOW, SPI master must transmit an operation code with the value of **0x02**. After the operation code, 16 bit address needs to be transmitted by the SPI master. 5 MSB bits of this address are "don't care" bits as memory capacity is 2048 bytes. After the operation code and the address are successfully received by an EEPROM slave, SPI master must transmit data that will actually be stored in the memory array at the specified memory address. Master can transmit more than one byte and subsequent bytes will be written to the subsequent memory locations, while address will increment with every byte. This WRITE command functions as a page write, therefore the write boundaries for one WRITE operation are limited. Only page, to which the specified address belongs, can be modified with one WRITE instruction. If the write address automatically increments to the current page end, the address rolls over to the beginning of that same page. Same address cannot be written twice in the same command, therefore the number of bytes that can be written in a single WRITE command are limited to the page size, which is in this specific instance 32. If more bytes are shifted onto the MOSI line, every byte after 32nd will be ignored. In order to terminate the instruction, the chip select pin must be brought logical HIGH. It is important to note that data will be actually written into the memory array only after the chip select is brought logical HIGH. Writing takes some time and therefore after WRITE command, a certain time is reserved for write in progress (WIP). If EEPROM is in WIP state, bit 0 of the status register will have value 1. In this state, status register can be polled for WIP bit, but all the other instructions will be ignored. After EEPROM exits WIP state, the write enable latch is automatically reset, therefore there is no need to execute WRDI instruction.

WRDI

Resets the write enable latch and therefore disables any following write operations. For any write operation, the write enable latch must be set using WREN command. The write enable latch can be read from the status register as bit number 1 with RDSR command. For WRDI instruction to execute, first the chip select pin must be brought to logical LOW, followed by master transmitting the operation code with the value **0x04**. Lastly, the instruction is terminated by bringing the chip select to logical HIGH again.

WRDI

Sets the write enable latch and therefore enables any following write operations. The write enable latch can be read from the status register as bit number 1 with RDSR command. For WRDI instruction to execute, first the chip select pin must be brought to logical LOW, followed by master transmitting the operation code with the value **0x06**. Lastly, the instruction is terminated by bringing the chip select to logical HIGH again.

RDSR

RDSR command reads the actual status register inside an EEPROM device. First, the chip select pin must be brought to logical LOW, the operation code of value **0x03** must be transmitted by master. After the operation code is transmitted by master, EEPROM device shifts the byte value of the status register on the MISO line. The operation is terminated by bringing the chip select pin HIGH.

WRSR

Prior to any WRSR command, the write enable latch must be set, otherwise the command will be ignored. WRSR command writes data to the status register bits with R/W accessibility inside the EEPROM device. First, the chip select pin must be brought to logical LOW, the operation code of value **0x01** must be transmitted by the master followed by transmitting the status register byte. Only the R/W bits are written into the status register, all other bits are "don't care". The operation is terminated by bringing the chip select pin HIGH. The actual data write happens after the chip select is brought HIGH and therefore time is needed for WIP state described in WRITE instruction. The status register can be polled in this state. After EEPROM exits WIP state, the write enable latch is automatically reset, therefore there is no need to execute WRDI instruction.

6.3. Implementation

In the EEPROM emulator mode, bytes will be received and transmitted one by one using the DMA method with a rare exception when two byte address is being received. Receiving one by one impacts the maximum SPI clock frequency, but this approach is necessary as instruction type varies and therefore the number of bytes in an instruction varies. Also, some instructions demand the slave to transmit data and some demand to receive data. An interrupt will be triggered after every SPI transmission. An interrupt will also trigger after the chip select goes HIGH signalling the end of instruction. The chip select interrupt is accomplished using the timer in the input capture mode, where the rising edge will be detected on the chip select pin.

6.3.1. State machine

EEPROM emulator mode can be interpreted as a state machine. There are 4 states in total.

- **OP_CODE RECEIVE**
  In this state, the device is currently receiving operation code. Once the op_code is received, the state is changed according to the received op_code. Possible next states according to the op_code:
  - **READ or WRITE op_code** : ADDRESS RECEIVE
  - **WRDI or WREN op_code** : No other possible state, the device is waiting for the chip select going HIGH.
  - **WRSR op_code** : DATA RECEIVE
  - **RDSR op_code** : DATA TRANSMIT
- **ADRESS RECEIVE**
  In this state, the memory address is being received. After the address is received, state is changed according to the op_code. Possible next states according to the op_code:
  - **READ op_code** : DATA TRANSMIT

  - **WRITE op_code** : DATA RECEIVE

- **DATA RECEIVE**
  In this state, the actual data is being received. This state can be active in WRITE or WRSR command. No possible next state, the device is waiting for the chip select going HIGH.

- **DATA TRANSMIT**
  In this state, the actual data is being transmitted. This state can be active in READ or RDSR command. No possible next state, the device is waiting for the chip select going HIGH.

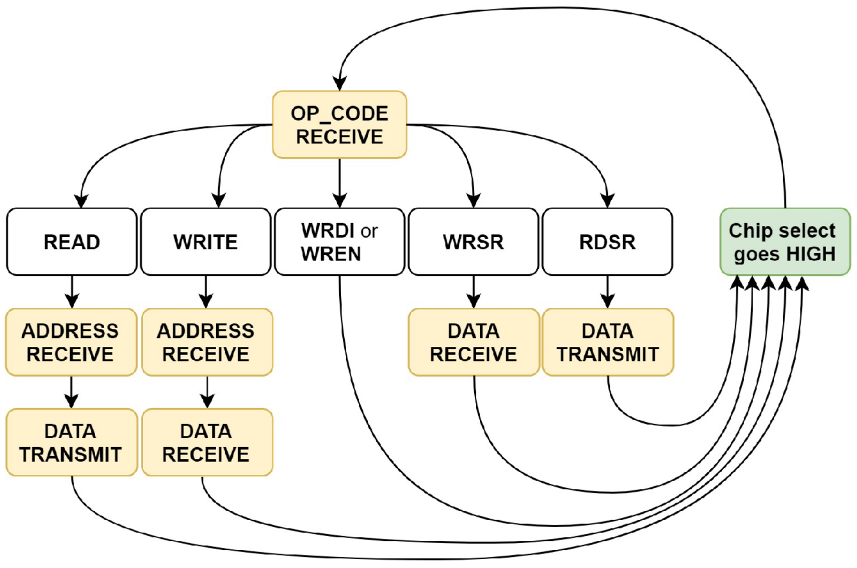


Figure 6.1: Block diagram of eeprom emulator state machine

State machine block diagram can be seen in figure 6.1. Upon device or EEPROM emulator mode initialization, the device is set to OP_CODE RECEIVE state. States are changed appropriately based on the current instruction and SPI transmissions. After the instruction is terminated by the chip select pin going logical HIGH, the state is returned to OP_CODE RECEIVE.

6.3.2. Main algorithm

Block diagram of the main algorithm can be seen in figure 6.2. No action will be performed in the main while loop. This mode is SPI receival/transmitting event driven and therefore all functionality will be performed inside interrupts.

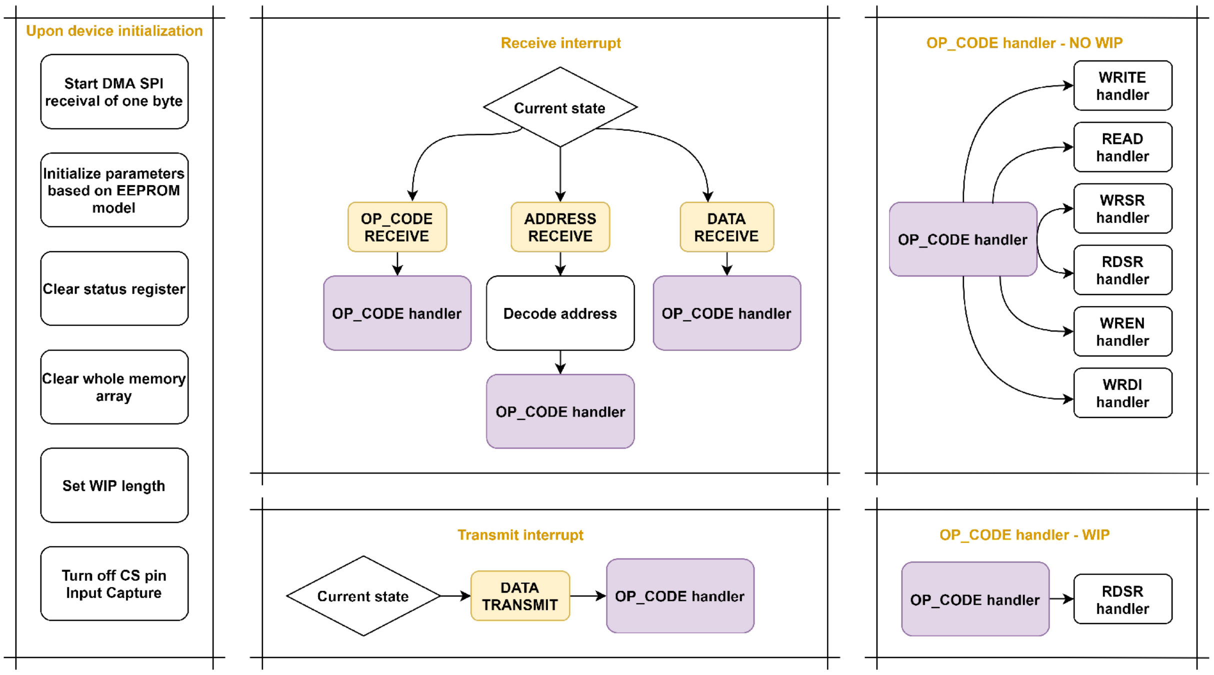


Figure 6.2: Block diagrams of the main algorithm of EEPROM emulator mode

Device initialization

Firstly, upon device or EEPROM emulator mode initialization, multiple actions need to be performed.

- **Parameters initialization**
  Parameters have to be initialized based on the specified EEPROM type. In current implementation, only one EEPROM is supported and therefore only one can be defined. In future development, it is expected to support more EEPROM devices that can be easily defined. The parameters are things such as the total memory size, page size, block size, status register etc. .

- **Clear memory and status register**
  Reset value of the memory is 0xff. The whole memory array is set to this value. Status register value is set to the value of 0.

- **Set WIP length**
  After any succesfull WRITE or WRSR command, a delay must occur while the WIP bit in the status register is set to 1. In this delay, only RDSR command can be executed. WIP delay time on this device is configurable and can be defined in microseconds. WIP delay is implemented using a timer in normal mode. After the timer is enabled, specified time elapses and interrupt is triggered.

- **Turn off the chip select pin interrupt**
  Interrupt on the rising edge of the chip select pin is used to terminate instructions. The interrupt is disabled in this state in case of unwanted noise on the signal line. The interrupt will be enabled later in the last state of the instruction.

- **Start DMA SPI receival of one byte**
  Current state upon the initialization is OP_CODE RECEIVE, therefore one byte receive must be started using DMA method.

Receive interrupt

Receive interrupt will trigger after:

- op_code is received
- two byte address is received
- memory byte or status register byte is received

This interrupt therefore can exist in three device states. Action based on the state is as follows.

- **OP_CODE RECEIVE state**
  In this state, the OP_CODE handler is called. All logic will be resolved in this handler.
- **ADDRESS RECEIVE state**
  In this state, firstly, the received address is decoded. Decoding includes applying a mask to the received address, so that 5 MSB are ignored and also the data counter is zeroed. Data counter represents how many data bytes were transferred in a single instruction, op_code and address are not classified as data bytes. OP_CODE handler is called after address decoding. All logic will be resolved in this handler.
- **DATA RECEIVE state**
  In this state, the OP_CODE handler is called. All logic will be resolved in this handler.

Transmit interrupt

This interrupt can exist only in one state, which is DATA TRANSMIT. In this state, only one action is required and that action is calling the OP_CODE handler. All logic will be resolved in this handler.

OP_CODE handler

OP_CODE handler's main purpose is to call specific handlers. The handler that is called is dependant on the received op_code in the current instruction. Which handlers are allowed to be called depends on if a write is currently in progress (WIP bit in the status register).

- **NO write in progress (no WIP)**
  If a write is not in progress, the possible handlers based on op_code are:
  - **WRITE handler**
  - **READ handler**

  - **WRSR handler**
  - **RDSR handler**
  - **WREN handler**
  - **WRDI handler**

- **write in progress (WIP)**
  If a write is in progress, only RDSR command is allowed and therefore the possible handlers are limited to:
  - **RDSR handler**

6.3.3. Write handler

Write handler can be called by the OP_CODE handler if the op_code belongs to the WRITE instruction. The main purpose of this handler is to receive address and to receive and write data to the memory array. This handler must oversee if the byte that is being currently written into memory is not out of boundaries of the starting page (starting address page) and also to check if the total number of bytes that were written does not exceed the page size, which would mean that the same address was written twice, which is forbidden. Block diagram of this handler can be found in figure 6.3. This handler can exist in three device states.

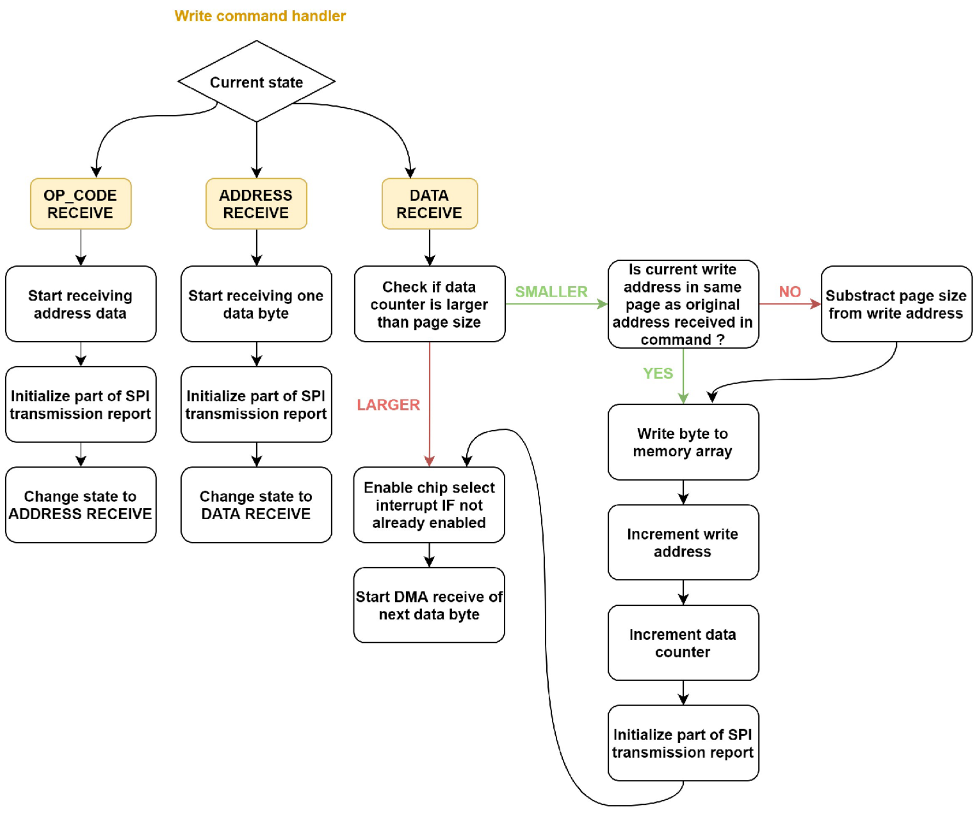

Write command handler
Current state
OP_CODE RECEIVE
ADDRESS RECEIVE
DATA RECEIVE
Start receiving address data
Initialize part of SPI transmission report
Change state to ADDRESS RECEIVE
Start receiving one data byte
Initialize part of SPI transmission report
Change state to DATA RECEIVE
Check if data counter is larger than page size
SMALLER
LARGER
Enable chip select interrupt IF not already enabled
Start DMA receive of next data byte
Is current write address in same page as original address received in command ?
NO
YES
Substract page size from write address
Write byte to memory array
Increment write address
Increment data counter
Initialize part of SPI transmission report

Figure 6.3: Block diagram of a write handler in EEPROM emulator mode

OP_CODE RECEIVE state

In this state, the op_code receival interrupt is triggered. Next action is to start SPI DMA receival of two address bytes. State is changed to ADDRESS RECEIVE. Also, a report message is initalized, where struct is initialized and prepared for the next data in the WRITE instruction.

ADDRESS RECEIVE state

In this state, the two byte address receival interrupt is triggered. Address was decoded prior to this handler. Next action is to start SPI DMA receival of one data byte that will be written to the write address. The report message is updated with the starting address and state is changed to DATA RECEIVE.

DATA RECEIVE state

In this state, the data byte receival interrupt is triggered. Multiple actions need to be performed in this state. These actions can be seen in figure 6.3 and can be described as:

- **Check if the data counter is larger than the page size**
  WRITE command can write multiple bytes, but no more than the page size. To oversee this, first the data counter indicating how many bytes were written is checked. If this data counter is larger than the page size, write is no longer allowed and device is waiting for the chip select going HIGH indicating the end of the instruction. If the data counter is smaller than the page size, write process follows.
- **Check if the current write address is in correct page**
  If data counter is smaller than the page size, write process begins with this action. WRITE instruction has a page boundary, meaning that all written bytes in the same instruction must belong to the same page as did the starting address. This fact is checked. If the current write address is outside the valid page, page size is substracted from the write address. This returns the write address to the start of the correct page.
- **Write byte to memory array**
  Once the write address is in correct page, the actual byte received is written to the memory array on actual write address. While writing data byte to the memory array, the write enable latch in the status register is checked. If the write enable latch is not set, no data will be written into the memory.
- **Increment the data counter and write address**
  Data counter indicating how many bytes were transmitted in the current instruction is incremented by the value of one. Write address indicating on which address to write the current byte is incremented by the value of one.
- **Initialize part of SPI report message**
  Data byte is added to the report message.
- **Enable chip select interrupt if not already enabled**
  Chip select interrupt will indicate the termination of the instruction. This interrupt

was not enabled before in case of any unwanted noise. In order to detect the end of the transmission, this interrupt needs to be enabled.

- **Start SPI DMA receival of the next data byte**
  SPI is set to receive the next data byte. State is not changed as multiple bytes can be written in this instruction and only way to change state at this point is by the chip select interrupt.

6.3.4. Read handler

Read handler can be called by the OP_CODE handler if the op_code belongs to READ instruction. The main purpose of this handler is to receive address and to transmit correct data. The handler must oversee if the read address is not larger than the memory size. Block diagram of this handler can be found in figure 6.4. This handler can exist in three device states.

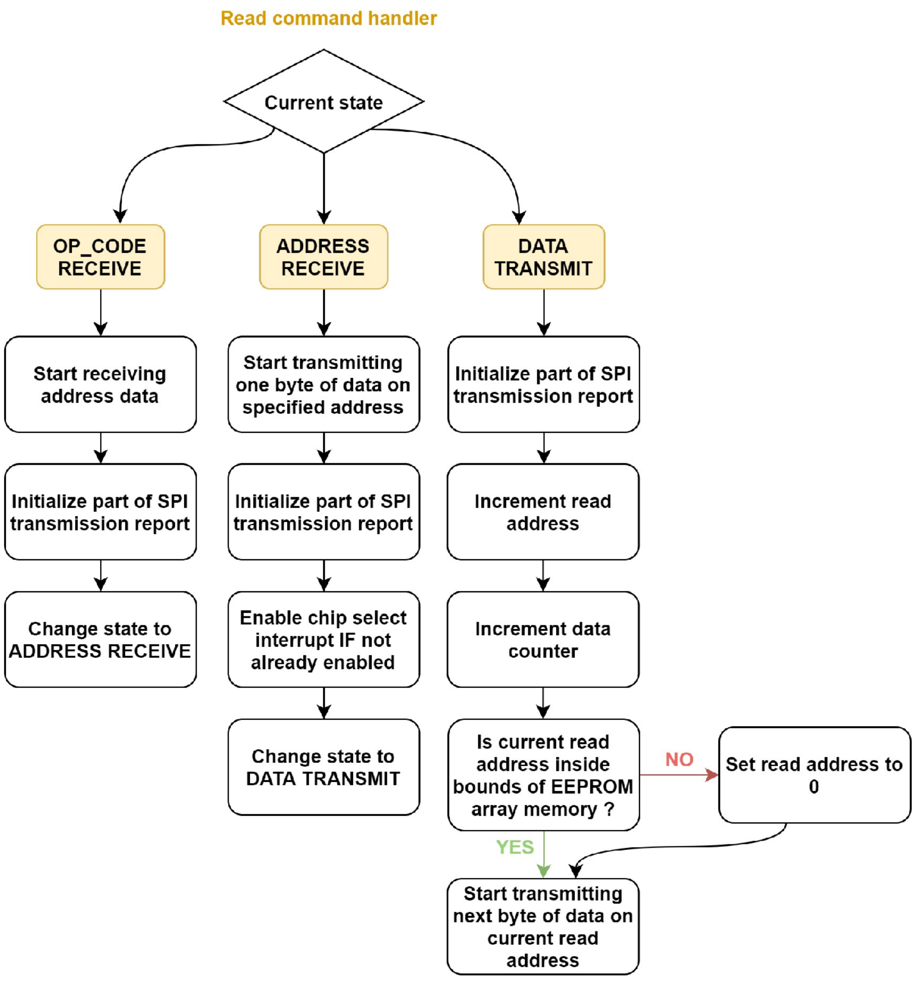

Read command handler
Current state
OP_CODE RECEIVE
ADDRESS RECEIVE
DATA TRANSMIT
Start receiving address data
Initialize part of SPI transmission report
Change state to ADDRESS RECEIVE
Start transmitting one byte of data on specified address
Initialize part of SPI transmission report
Enable chip select interrupt IF not already enabled
Change state to DATA TRANSMIT
Initialize part of SPI transmission report
Increment read address
Increment data counter
Is current read address inside bounds of EEPROM array memory ?
NO
Set read address to 0
YES
Start transmitting next byte of data on current read address

Figure 6.4: Block diagram of read handler in EEPROM emulator mode

OP_CODE RECEIVE state

In this state, the op_code receival interrupt is triggered. Next action is to start SPI DMA receival of two address bytes. State is changed to ADDRESS RECEIVE. Also, a report message is initalized, where struct is initialized and prepared for the next data in READ instruction.

ADDRESS RECEIVE state

In this state, the two byte address receival interrupt is triggered. The address was decoded prior to this handler. Next action is to start SPI DMA transmission of one data byte which was specified by the received starting read address. The report message is updated with the starting address and state is changed to DATA TRANSMIT. Another important action is to the enable chip select interrupt if not already enabled. This will allow detection of the instruction termination.

DATA TRANSMIT state

In this state, the interrupt indicating sucesfull transmission of one byte is triggered. There are multiple actions that need to be performed in this interrupt:

- **Initialize a part of the SPI transmission report**
  The report message is updated with the transmitted byte.
- **Increment the read address and data counter**
  Read address from which the data will be transmitted is incremented by the value of one. Data counter indicating how many bytes were transmitted is also incremented by the value of one.
- **Check if read address is larger than memory size**
  Previously incremented address is checked if it is inside a valid range of addresses. If the read address is larger than the memory size, read address is set to 0.
- **Start transmission of next byte**
  Once the read address is valid, SPI DMA transmission of the next byte located on the read address is started.

6.3.5. WRSR, RDSR, WREN and WRDI handlers

All the other handlers apart from the chip select interrupt handler can be seen in figure 6.5.

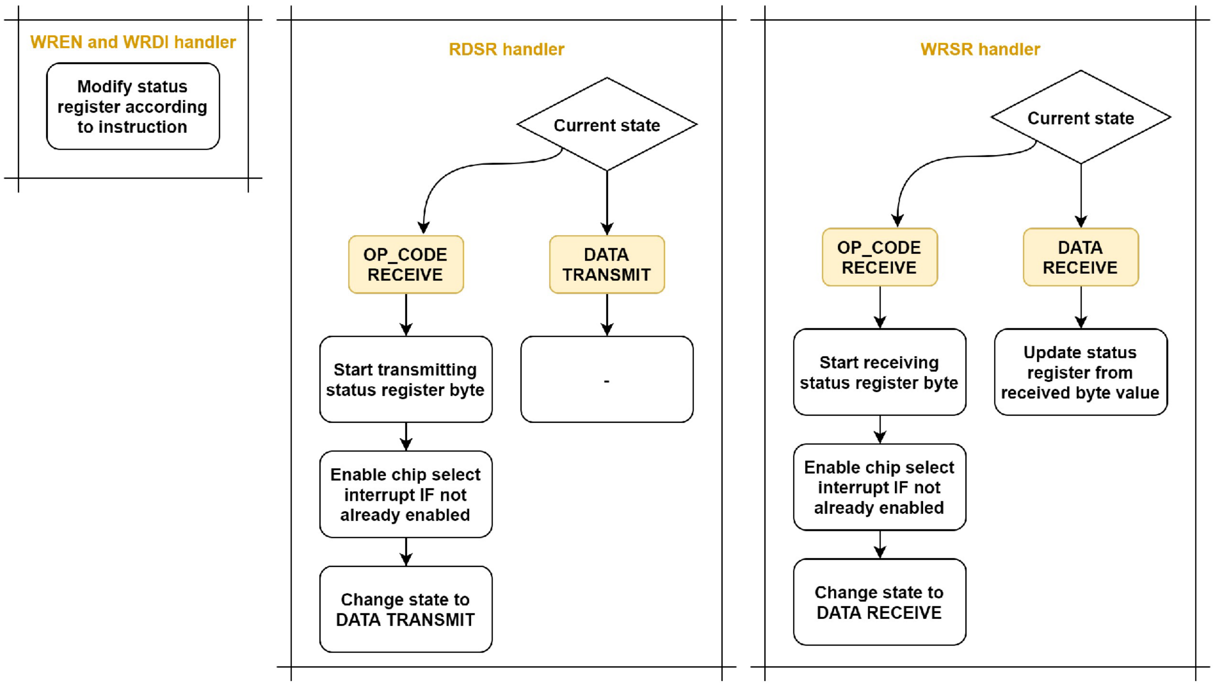

WREN and WRDI handler
Modify status register according to instruction
RDSR handler
Current state
OP_CODE RECEIVE
DATA TRANSMIT
Start transmitting status register byte
-
Enable chip select interrupt IF not already enabled
Change state to DATA TRANSMIT
WRSR handler
Current state
OP_CODE RECEIVE
DATA RECEIVE
Start receiving status register byte
Update status register from received byte value
Enable chip select interrupt IF not already enabled
Change state to DATA RECEIVE

Figure 6.5: Block diagram of RDSR,WRSR,WREN and WRDI handlers in EEPROM emulator mode

RDSR handler

RDSR handler can exist in two states:

- **OP_CODE RECEIVE state**
  This interrupt happens upon op_code receival. Following needed action is to start SPI DMA transmission of the status register byte. The chip select interrupt is enabled if not already enabled. State is changed to DATA TRANSMIT.

- **DATA TRANSMIT state**
  This interrupt happens upon the transmission of the status register byte. No action is required and device is waiting for the chip select rising edge interrupt indicating the end of the instruction.

WRSR handler

WRSR handler can exist in two states:

- **OP_CODE RECEIVE state**
  This interrupt happens upon op_code receival. Following needed action is to start SPI DMA receival of the status register byte. The chip select interrupt is enabled if not already enabled. State is changed to DATA RECEIVE.

- **DATA TRANSMIT state**
  This interrupt happens upon the receival of the status register byte. This byte is written as the new status register. While writing this byte, the write enable latch is checked and also a mask is applied to this byte in order to write only bits that are writable. Non writable bits stay unmodified.

WREN, WRDI handler

In both WREN and WRDI handlers, the only possible device state is OP_CODE RECEIVE. Following needed action is to modify the WEN bit in the status register according to what command was received. The chip select interrupt is enabled if not already enabled.

6.3.6. Chip select interrupt handler

The chip select interrupt triggers on the rising edge of the chip select signal and indicates termination of the instruction. Block diagram of the algorithm can be seen in figure 6.6.

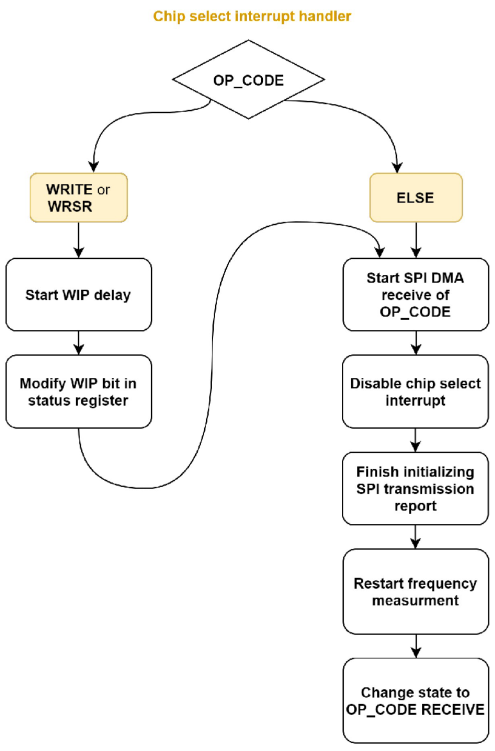

Chip select interrupt handler
OP_CODE
WRITE or WRSR
ELSE
Start WIP delay
Modify WIP bit in status register
Start SPI DMA receive of OP_CODE
Disable chip select interrupt
Finish initializing SPI transmission report
Restart frequency measurment
Change state to OP_CODE RECEIVE

Figure 6.6: Block diagram of chip select interrupt handler in EEPROM emulator mode

Blocks of this algorithm can be described as:

- **IF WRITE or WRSR command**
  If current instruction is WRITE or WRSR, the write in progress delay needs to be started. Timer that handles the WIP delay is started and WIP bit in the status register is set to 1. To end the WIP delay, timer will trigger an interrupt after set time. WIP delay was first measured on a real EEPROM device and WIP delay on the EEPROM emulator device was set to the same value, even though write operation could take less time on the EEPROM emulator. Final value of the WIP delay was set to 2.75 ms.

- **OP_CODE SPI receival is started**
  SPI DMA receival of the op_code is started and state is changed to OP_CODE RECEIVE.

- **Chip select interrupt is disabled**
  Chip select interrupt is disabled in case of unwanted noise.

- **SPI report is initialized**
  SPI report is initialized with the current instruction. In case of WRITE or READ instruction, the main parts of the report were already initialized by parts in every interrupt of this instruction. In case of WRSR and RDSR instruction, the whole report is created here. Reports about WRDI, WREN instructions do not exist in this implementation. For all reports created, the input capture data of the frequency measurement are appended to the report message.

- **Restart the frequency measurement**
  The measurement of frequency is restarted. Every instruction will be measured separately.

6.4. Optimization - SPI clock speed

The described EEPROM emulator implementation was not developed in one iteration, so naturally there are some important changes/optimizations that could be noted. This and following sections will contain what are considered to be the most important optimizations.

6.4.1. Problem description

In the EEPROM emulator mode, SPI data is being received and transmitted one byte at a time with the rare exception of address receive when 2 bytes are received. This method is required due to the instructions having different byte lengths as well as that the next data byte transmission is dependant on the previous byte received and therefore all logic needs to be implemented in-between the bytes in interrupts. The disadvantage of this approach is that the SPI clock frequency is severely limited if a continuous data stream is expected. A continuous data stream provides the worst case of the time in-between bytes as the time in-between bytes is equal to the time in-between bits and therefore to the time of a one clock pulse. This means that the SPI clock frequency is limited in order to provide enough time in-between bytes for the logic.

6.4.2. Base measurement

The first iteration of the implementation used the HAL drivers for SPI peripheral as well as the basic setting of a GCC compiler. Simple way to describe the function of a GCC compiler is that it is responsible for ”translating” programming language into machine code, meaning it converts the source code to executable instruction file for the MCU. For this measurement, a WRITE command was performed, where bytes [1, 2, 3, 4, 5] were written at the address with value 0x00. The SPI clock freuqency of the master was set to 54 kHz. A benchmark GPIO pin was set to output HIGH value at the start of the interrupt and output LOW value after the next SPI DMA transmission is setup. The time was measured from the rising edge of the last clock pulse in a byte to benchmark pin going LOW. Result of this measurement can be seen in figure 6.7.

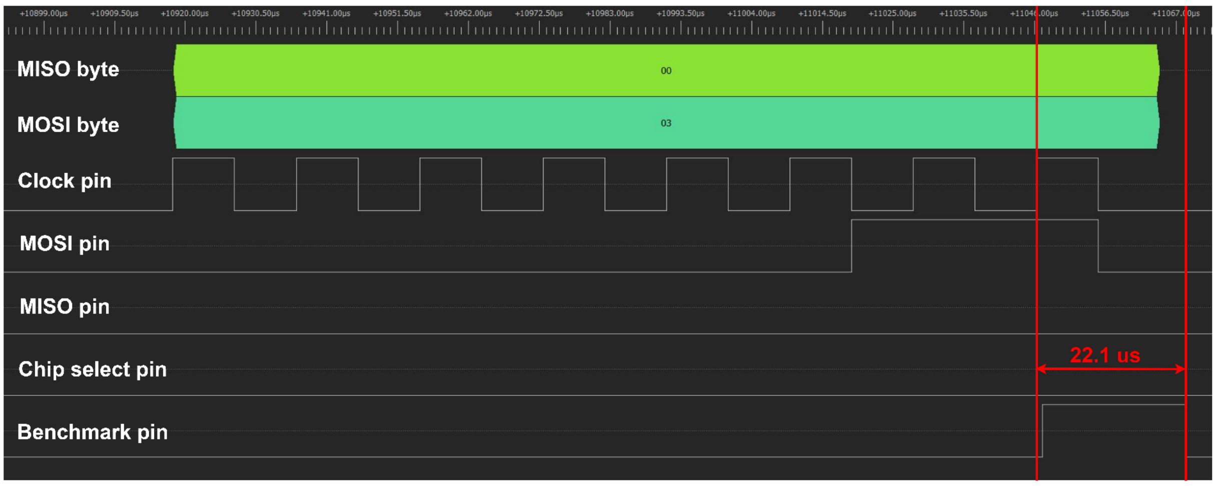


Figure 6.7: Base measurement for SPI clock speed optimization

Red markers indicate the last clock pulse of a byte and the benchmark pin going low. Time of this period is 22.1 µs, which translates to the maximum clock frequency of 45.2 kHz.

6.4.3. GCC setting

The first iteration of optimization included modifying a GCC compiler setting. Setting was set to -Ofast. Settings of the GCC compiler as well as its overall function are described in [22]. By applying this setting, the measured time went down to 12.45 µs, which meant increase in the maximal SPI clock frequency to 80 kHz.

6.4.4. SPI drivers optimization

The next iterations of optimization involved modifying the SPI drivers. Originally used HAL drivers need to take into consideration every possibility of the SPI peripheral, which is not needed in this case. By modifying the SPI drivers and leaving the absolute minimum necessary for the correct action, measured time decreases vastly. Drivers were modified in multiple steps, but only the final result will be shown. The final result is measured while using the SPI drivers described in section 5.3. While the base measurement was done only for WRITE command, the final measurement will be done for WRITE command as well as for READ command, which will ensure that both receive and transmit functionality will be tested.

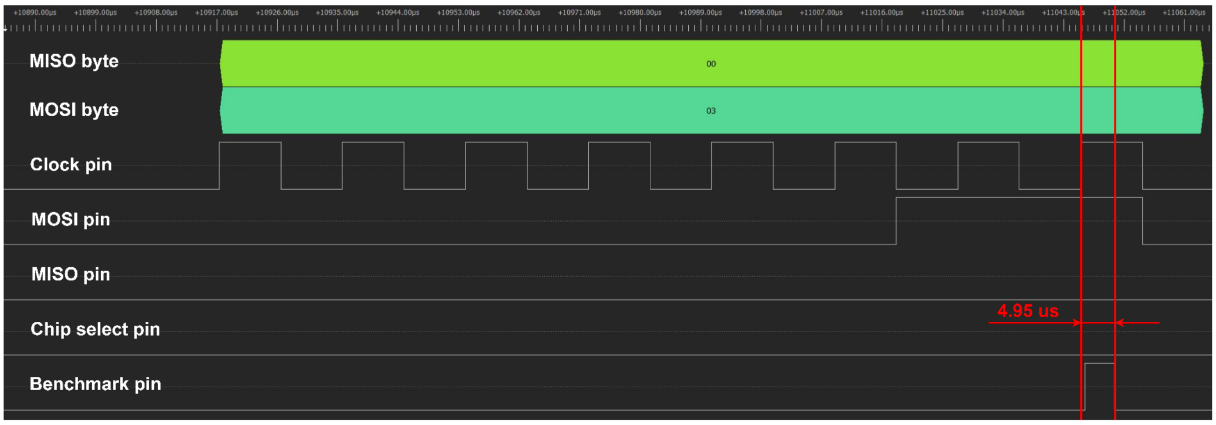


Figure 6.8: Final measurement for SPI clock speed optimization for WRITE instruction

The final measurement for the same WRITE command as in the base measurement can be seen in figure 6.8. The measured time is marked by the red markers and the final measured time is 4.95 µs, which translates to a maximum clock frequency of 202kHz.

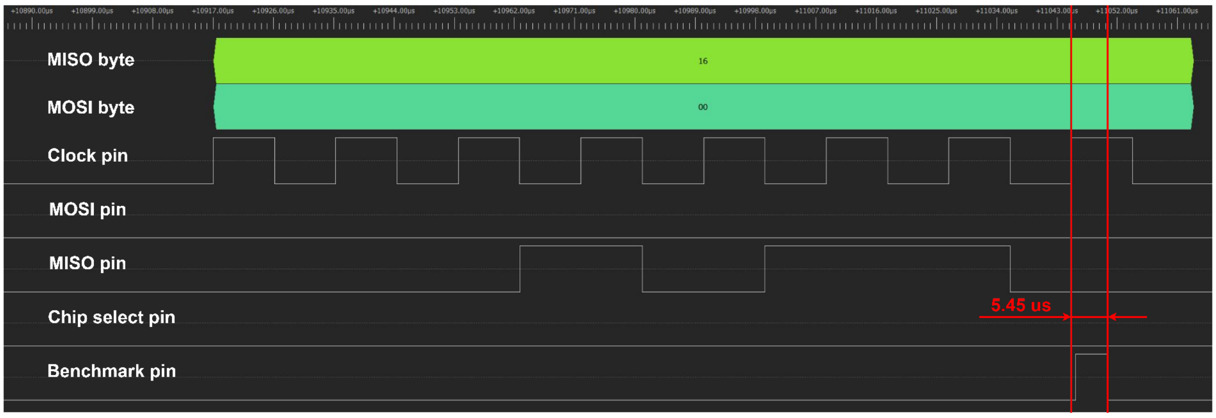


Figure 6.9: Final measurement for SPI clock speed optimization for READ instruction

Final measurement for READ command can be seen in figure 6.9. The measured time is marked by the red markers and the final measured time is 5.45 µs, which translates to a maximum clock frequency of 183 kHz. Base measurement for READ command was not performed, but it is expected to be slightly worse than base measurement for WRITE command.

6.4.5. Optimization conclusion

The SPI clock frequency is by nature majorly limited for this device mode. In order to improve the situation, optimization was done by changing the GCC compiler settings and by rewriting SPI peripheral drivers using the Low Layer (LL) drivers. From the base value of 45 kHz, the maximum frequency was improved to 202kHz for WRITE instruction and 183kHz for READ instruction. The base measurement can be seen in figure 6.7 and can be compared with the final measurements in figure 6.8 and figure 6.9.

6.5. Optimization - Chip select interrupt time

6.5.1. Problem description

In previous optimization, the SPI clock frequency was optimized, but the time in-between instructions was not taken into consideration. The minimum time in-between instructions exactly correlates to the time of the chip select interrupt. This time is the time from the rising edge of the chip select to the next rising edge of the clock signal. It was chosen that this time would ideally be no longer than the time of one clock pulse.

6.5.2. Base measurement

An early implementation of this mode had slightly different code inside the chip select interrupt handler. Report messages for every instruction were completely initialized only in this interrupt and also, the frequency was calculated in this interrupt if the last time of calculation was larger than 20 ms. This meant that the chip select interrupt time was strongly dependant on the type of instruction. In all the measurements, the worst case was measured and the worst case for this measurement was determined to be for WRITE command, when the maximum number of bytes possible were written. Benchmark pin was set to go HIGH at the beginning of the chip select interrupt and go LOW in the end of the chip select interrupt. Measurement was done from the rising edge of the chip select signal to the falling edge of the benchmark pin. The base measurement can be seen in figure 6.10.

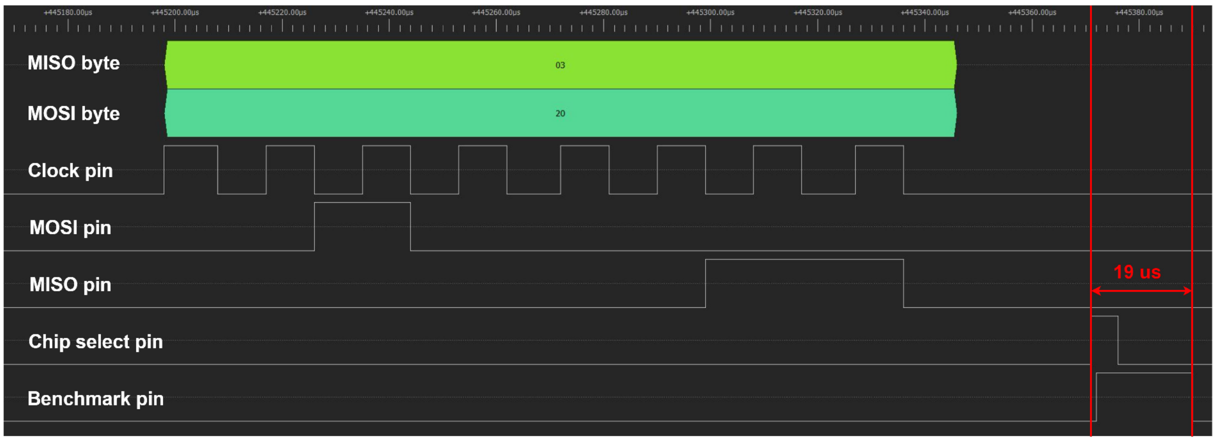


Figure 6.10: Base measurement for chip select interrupt time optimization

The base measurement is indicated by red markers and the time of measurement was 19 µs. Target time is around 5 µs. It is important to note, that when the actual frequency calculation is done in this interrupt, the interrupt time increases by around 450 µs. This happens once every 20 ms.

6.5.3. Final optimization

The first step of optimization was to remove frequency calculation from this interrupt. In the final implementation, the frequency is calculated for every report message just before the report message is sent. This means that in the interrupt, only the input capture frequency measurement data are being written to every report message. This also means that every report message will have separately measured SPI clock frequency, while before frequency was only measured once every 20 ms.

The next step of optimization was to divide the report message initialization into parts and initialize these parts not at once in the chip select interrupt, but rather initialize a part after every byte of the instruction. This would massively improve the chip select interrupt time and also make it a constant for WRITE instruction.

The final measurement can be seen in figure 6.11.

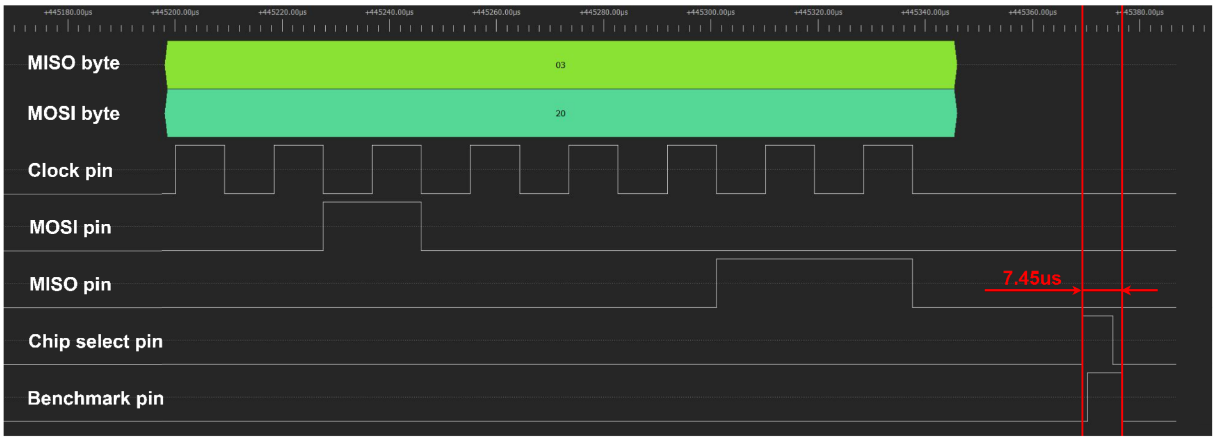


Figure 6.11: Final measurement for chip select interrupt time optimization

As can be seen from the figure, measurement indicated by the red markers is equal to 7.45 µs. This is the absolute worst case and every other type of command has better result.

6.6. Discussion and limitations

The EEPROM emulator mode is used to create a model of an existing EEPROM device. The goal was to create a device, which behaviour would be the same as the real EEPROM device if both of these devices were connected to an SPI master capable of transmitting valid EEPROM instructions from the instruction set. Such device was created and a detailed verification tests as well as a comparison tests with the real EEPROM device will be provided later in chapter 8.

Emulator cannot be 100% same as the real device, some limitations occur. In case of the actual logic, the behaviour is completely the same. One difference is that the real EEPROM memory array and the status register are non-volatile, meaning data will retain after the device is disconnected from the power supply. All data of EEPROM emulator are volatile, because they are stored in the MCU's RAM memory. This means that after device reboot, no data will remain. This is not considered to be a problem as the device is not designed as a "part for part" replacement of a real EEPROM device, but rather as a test device, that can test the logic of EEPROM SPI masters.

The next difference is when the instruction data is actually being written into the memory. On the real EEPROM device, data is written into the memory only after the chip select pin is brought HIGH. On the EEPROM emulator, data is written into the memory at the same moment as it is received by the instruction. This is also not considered to be a problem as it should not pose any noticeable difference from the "outside", meaning no difference to the SPI master or state of the memory. The only situation when there could be a difference in functionality is if the device was powered off during an instruction. In this case real EEPROM would not keep the data from that instruction in memory, but the EEPROM emulator would. But in reality, the memory of the EEPROM emulator is volatile. The fact that the EEPROM emulator memory is volatile negates this problem because the behaviour is already vastly different on reboot.

One major limitation is the SPI clock frequency. The worst case of an SPI transmission is considered, which is a continuous data stream. A continuous data stream means that time in-between bytes is the same as the time in-between bits (clock pulses). All logic is executed in the time between bytes and therefore the minimum time has to elapse in order for the device to function properly. The maximum SPI clock frequency was only 45 kHz in earlier implementation and was brought up to 200kHz for WRITE command and 183kHz for READ command with optimization. These values are vastly different than the maximum clock frequency of a real EEPROM device, which is 10 MHz. This massive difference is not ideal, but priority of this emulator device is not to test the electrical characteristics of an SPI master, but to test logic. In reality, SPI master's SPI clock frequency can be easily turned down. Another important fact to note is that this emulator device can be classified as a prototype or "proof of concept" and in future development, a much faster MCU (up to 10x faster system clock) can be used and therefore the maximum SPI clock frequency would increase significantly also. Maybe it would be more fitting to emulate EEPROM devices which use I2C communication interface instead of the SPI interface as generally speed of I2C interface does not exceed 400kHz, which is much closer to the current emulator maximum clock frequency. One other important realization is that in reality continuous data stream is almost never the case, which means that the maximum SPI clock frequency can be much higher depending on the format of data transmission.

In order to match the behaviour of a real EEPROM device, some parameters were also slowed down. There is a write in progress delay on this EEPROM type and according to datasheet [21], its maximum value is 5ms. This time was measured on a real EEPROM device and the rough measured value was determined to be 2.75 ms . EEPROM emulator does not naturally have this delay and therefore it was artificially added to match the behaviour of a real EEPROM device.

6.7. Possible improvements

One major improvement in the future development would be to use a much faster MCU which would impact the maximum SPI clock frequency as the clock frequency is majorly limited in this state of development.

Other plans for the future development are to support more EEPROM device. The goal is to be able to generally define any possible EEPROM device with a simple parameters definition inside a device struct in C language. The parameters would describe the total memory size, page, sector, block size as well as instruction set and WIP delay.

7. Python API and configuration

7.1. Device configuration

7.1.1. Configuration tree

There are three parameters that define which mode the device is currently in. These parameters are:

- **stream_mode_enable**
- **eeprom_mode_enable**
- **full_duplex_enable**

The configuration tree can be seen in figure 7.1. If the stream_mode parameter is set, the device will be in stream mode and the other two parameters are ignored. If the stream_mode is not set, the next parameter which is being evaluated is eeprom_mode_enable. If this parameter is set, the device will enter EEPROM emulator mode and the last parameter is ignored. If the eeorom_mode_enable is not set, the device is automatically in LUT mode. The parameter full_duplex_enable determines if the LUT mode will be in full-duplex or half-duplex configuration.

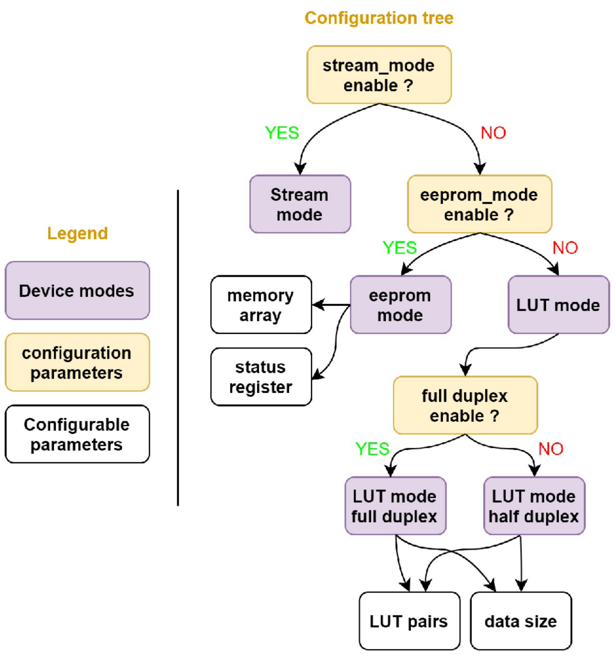


Figure 7.1: Configuration tree of the device

In stream mode, no internal parameters can be configured.

EEPROM emulator mode configuration

In EEPROM mode, the memory array and the status register can be modified from the python API.

LUT mode configuration

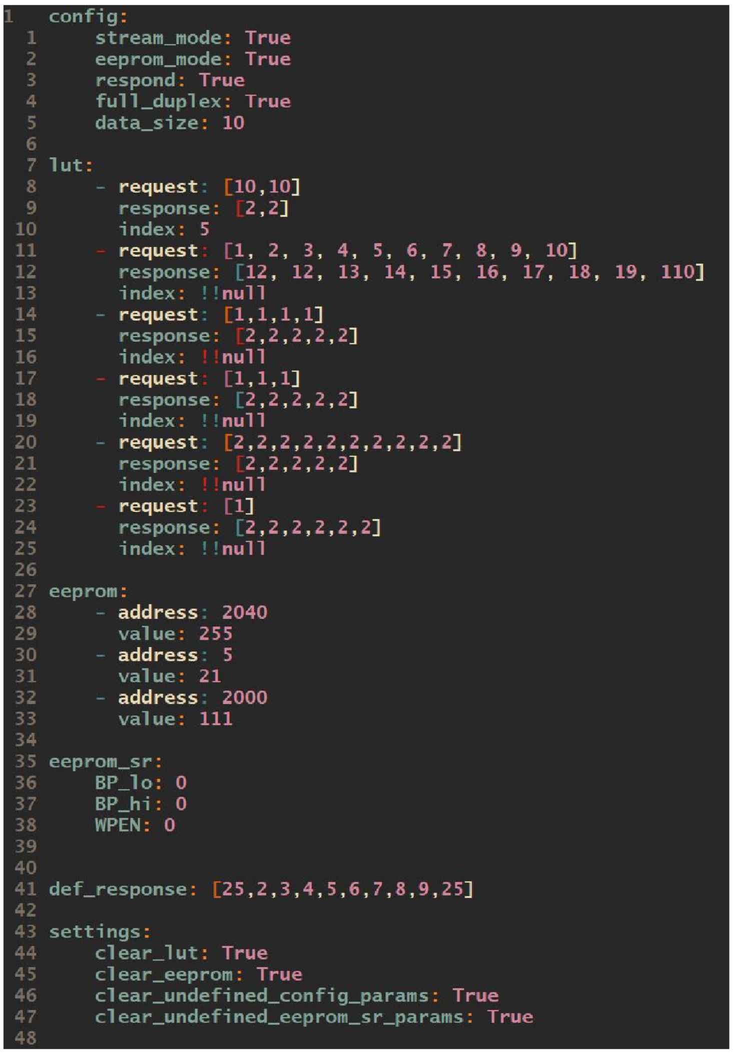
In LUT mode, LUT pairs and data size can be modified from the python API.

7.1.2. Discussion

The device can be configured according to the configuration tree. In current version of implementation, the core settings of the actual SPI peripheral such as SPI mode cannot be configured. It is necessary to add SPI peripheral configuration in future development.

7.2. YAML file

YAML (YAML Ain't Markup Language) is a data serialization language which is used mostly for configuration files. In this specific case, YAML is used to create a configuration python dictionary that is later decoded by python API and this configuration is applied to the embedded device. More information about YAML can be found at [23].



```yaml
config:
    stream_mode: True
    eeprom_mode: True
    respond: True
    full_duplex: True
    data_size: 10

lut:
    - request: [10,10]
      response: [2,2]
      index: 5
    - request: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      response: [12, 12, 13, 14, 15, 16, 17, 18, 19, 110]
      index: !!null
    - request: [1,1,1,1]
      response: [2,2,2,2,2]
      index: !!null
    - request: [1,1,1]
      response: [2,2,2,2,2]
      index: !!null
    - request: [2,2,2,2,2,2,2,2,2,2]
      response: [2,2,2,2,2]
      index: !!null
    - request: [1]
      response: [2,2,2,2,2,2]
      index: !!null

eeprom:
    - address: 2040
      value: 255
    - address: 5
      value: 21
    - address: 2000
      value: 111

eeprom_sr:
    BP_lo: 0
    BP_hi: 0
    WPEN: 0


def_response: [25,2,3,4,5,6,7,8,9,25]

settings:
    clear_lut: True
    clear_eeprom: True
    clear_undefined_config_params: True
    clear_undefined_eeprom_sr_params: True

```

Figure 7.2: Example of a YAML file

Example of a YAML file can be seen in figure 7.2. This is a fully defined configuration and no other expressions can be used. It is possible to add more LUT pairs and to add more EEPROM data definition. YAML file can also be defined partially and in that case, only the defined config parameters will be applied to the embedded device. It is possible to set YAML decoder so that the undefined parameters will be cleared. While decoding a YAMl file, all possible errors such as YAML syntax and parameter types are checked.

7.3. Python API

7.3.1. Main API tasks and characteristics

API (application programming interface) can be generally described as a code library that allows the user to control a device/application, for which the API was created.

API programming language selection

There are multiple viable programming languages for creating APIs. Amongst the most well known ones are C++, Python, VBS or even graphical languages like LabView. For this projec, python was chosen as the preferred programming language. The main reasons for choosing this language were previous experience with this language, readily available modules for very wide variety of applications and functionalities.

Main tasks

Main tasks of API can be distinguished as:

- **Sending configuration commands through the serial port interface.**
  For configuration, different commands must be sent to the device. API should provide documented methods that are able to send valid commands to the embedded device. These commands must be error checked by CRC and by receiving acknowledgement message back from the embedded device. This acknowledgement message must contain the same information as the command, otherwise the command will not be classified as successfully executed.

- **Receiving report and acknowledgement messages through the serial port interface.**
  The embedded device will send two kinds of messages to a PC. One is a report type of message, the second is an acknowledgement type of message. Report message is received by the API everytime a transmission happens on the SPI bus of the embedded device. These report messages will contain specific information regarding what happened on SPI data lines. Acknowledgement messages will be received everytime a valid command is sent from the PC API to the embedded device.

- **Loading and applying config specified by the user upon device connection.**
  User can specify wanted complete device configuration in YAML file. Upon device serial port connection, this configuration will be loaded into the embedded device.

- **Storing the current state of the embedded device.**
  While the embedded device is connected, API must know its current state of configuration. This is done by reading the configuration at device connection and also updating the configuration upon receival of every valid acknowledgement message.

API characteristics

API will be created as set of python methods and classes available for the user. These methods will be documented so that their function is completely understandable and should be able to completely control embedded device and all its functionalities. Documentation can be found at [24]

7.3.2. Implementation

In this section, the algorithms for each main API tasks will be described.

Connecting to a device

For the serial connection, pyserial module is used. This module allows the user to control and configure the serial port. Figure 7.3 represents a block diagram of the algorithm that is used for connecting to an embedded device. First, a connection is attempted on the serial port specified by the user. If this connection is unsuccessful, error is returned and the device status is changed appropriately. If the connection is successful, first a YAML configuration is loaded and decoded from the YAML file. Next, the loaded and decoded YAML configuration is sent to the embedded device using commands. Lastly, the current configuration is read from the embedded device and stored.

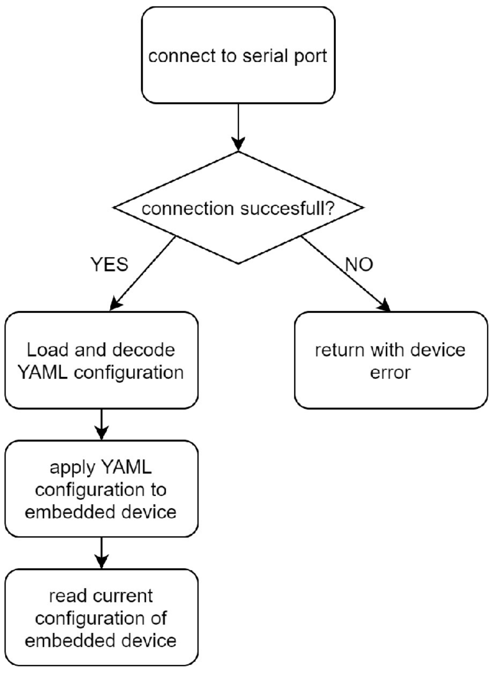

connect to serial port
connection succesfull?
YES
NO
Load and decode YAML configuration
return with device error
apply YAML configuration to embedded device
read current configuration of embedded device

Figure 7.3: Block diagram of device connection algorithm

Receiving and decoding messages

The main handler of the module needs to be called frequently in the main while loop of a script that uses this module. In this handler, bytes are received from the USB receive buffer and messages are decoded. Every valid message is initiated to Message() object. Upon message initiation, the CRC value of that same message is checked. If the CRC value is not correct, the message is ignored. If the message is of acknowledgement type, message is used for validation of the current command that is in the process of validation. If the message is of report type, data from the report message are printed into the console. Every report and acknowledgement message is stored in a buffer after processing. This way any message received can be accessed later from this buffer.

Transmitting commands

Commands can be transmitted in blocking mode or non-blocking mode. In blocking mode, commands are sent inside the called method. In non-blocking mode, commands are appended to a command buffer and executed later asynchronically in the main handler in the main while loop of a parent script. Every command needs to receive an acknowledgement message. In order for acknowledgement message to be valid, the message must contain the actual data of the related command and the CRC value must be correct. If the acknowledgement message is not received within the specified time, two more tries are attempted. After the third try of the same command is sent and no acknowledgement message is received within the specified timeout time, the command is labeled as timed out.

7.3.3. General packet description

UART packet can be generally described as can be seen in table 7.1. Packet consists of:

- **Start - Stop byte**
  Byte value to indicate start of a packet. Value is 0xBB.
- **Total length of packet**
  Total length of a packet. This it not present in a packet sent from PC to STM32. In the future development, the packets will be modified so that every packet is completely same.
- **Op_code**
  Byte that describes type of message/command.
- **Command/message data**
  Actual data of a command or message. The length of this part of the packet can vary based on the specific message/command.
- **Two bytes of CRC**
  Sixteen bit CRC is used for error checking with polynomial value 0x8810. CRC is calculated from the whole packet up to this point, but without the start-stop byte.
- **Start - Stop byte**
  Byte value to indicate the end of a packet. Value is 0xBB.

First byte	Start - Stop Word byte
...	Total packet length
...	Op_code
...	Command/message data
...	Two bytes of CRC
Last byte	Start - Stop byte

Table 7.1: General packet for python API transmissions

*Note - Total packet length is not included in PC to STM32 packet.

7.3.4. Specific commands description

Every command has a blocking mode attribute that determines if it will execute in blocking or non-blocking mode as well as some timeout values. Every command needs to be accompanied by a successful receival of a valid acknowledgement message. Requested data from every read command are in the acknowledgement message. Commands can be described as:

Command name	**Description**
Config write	Writes the config parameters described in the configuration tree as well as the data size for LUT mode.
Config read	Reads the config parameters described in the configuration tree as well as the data size for LUT mode.
LUT write	Writes a single LUT pair.
LUT clear	Cleares a single LUT pair.
LUT read	Reads a single LUT pair.
Set response	Sets the default response for LUT mode.
Write EEPROM	Writes bytes to the EEPROM emulator memory array at the specified address.
Read EEPROM	Reads bytes from the EEPROM emulator memory array at the specified address.
Wrsr EEPROM	Writes the status register byte to the EEPROM emulator.
Rdsr EEPROM	Reads the status register byte from the EEPROM emulator.
Clear chip EEPROM	Resets the whole memory array of the EEPROM emulator to 0xFF.

Table 7.2: Python API commands

7.3.5. Specific acknowledgement messages description

Acknowledgement messages are received after successful command transmission by the python API. Every python API command has got its own specific acknowledgement message.

7.3.6. Specific report messages description

Report messages are received after a successful SPI transmission. The list of all the possible report messages follows.

Message name	Description
Stream report	Report containing information about one received byte in stream mode.
Rx report	Report containing information about received bytes in half-duplex LUT mode.
Tx report	Report containing information about transmitted bytes in half-duplex LUT mode.
TxRx report	Report containing information about transmitted and received bytes in full-duplex LUT mode.
EepWrite report	Report containing information about EEPROM WRITE command received via SPI interface.
EepRead report	Report containing information about EEPROM READ command received via SPI interface.
EepWrsr report	Report containing information about EEPROM WRSR command received via SPI interface.
EepRdsr report	Report containing information about EEPROM RDSR command received via SPI interface.

Table 7.3: Report messages

7.4. Discussion

Configuration of the device and the main python API features were described. In reality, function of python API is more complex. More information about how the python API functions and more information about commands and messages can be found at [24].

8. Verification

Once the stream, LUT and EEPROM modes are developed and functioning along with the python API, it is important to perform appropriate verification tests for these modes. In order to test the SPI slave device, an SPI master is needed.

8.1. Test master design

Configurable SPI master is needed to sucesfully test the SPI slave device that is being developed. For this purpose, a separate SPI master was developed. For hardware, same NUCLEO board with the same MCU was used. For software, same python API with C framework was used as was described in Python API section 7.3. Important features that can be accessed by the master's python API are:

- `spi_master.transmit( transmit_list )`
- `spi_master.receive( receive_length )`
- `spi_master.transmitreceive( transmitreceive_list )`
- `spi_master.write_eep( address, data )`
- `spi_master.read_eep( address, number_of_bytes )`
- `spi_master.rdsr_eep(  )`
- `spi_master.wrsr_eep( status_register_byte )`
- `spi_master.clear_chip_eep(  )`

Transmit method

Transmit function can be called by expression:

```
spi_master.transmit( transmit_list )
```

, where spi_master is an SPI master object and transmit_list is an integer list representing bytes to be transmitted, therefore values of this list have to be of type 8 bit unsigned integer. The length of this list is limited to the value 50. Once this method is called, the SPI master transmits these bytes on the MOSI SPI line.

Receive method

Receive function can be called by expression:

```
spi_master.receive( receive_length )
```

, where spi_master is an SPI master object and receive_length is an integer value representing the number of bytes to be received. This value is limited to value 50. Once this method is called, the SPI master provides clock signal and receives the specified number of bytes from the MISO SPI line.

Transmit receive method

Transmit-Receive function can be called by expression:

```
spi_master.transmitreceive( transmitreceive_list )
```

, where spi_master is an SPI master object and transmitreceive_list is an integer list representing bytes to be transmitted, therefore values of this list have to be of type 8 bit unsigned integer. In this method, also data is received. The length of the transmitreceive_list indicates the number of bytes to be received and the length of this list is limited to the value 50. Once this method is called, the SPI master provides clock signal and receives the specified number of bytes from the MISO SPI line and transmits the provided list on the MOSI SPI line.

write_eep method

write_eep method can be called by expression:

```
spi_master.write_eep( address, data )
```

, where spi_master is an SPI master object, address is the EEPROM memory starting address where data will be written and data is a list of bytes. Size of this list determines how many bytes will be written. Once this method is called, the SPI master transmits WREN command, followed by WRITE command. After WRITE command, the status register is polled with RDSR command until the WIP bit is equal to 0.

read_eep method

read_eep method can be called by expression:

```
spi_master.read_eep( address, number_of_bytes )
```

, where spi_master is an SPI master object, address is the EEPROM memory starting address from where data will be read and number_of_bytes is the size of the READ command. Once this method is called, the SPI master transmits READ command and receives specified number of bytes.

rdsr_eep method

rdsr_eep method can be called by expression:

```
spi_master.rdsr_eep(  )
```

, where spi_master is an SPI master object. Once this method is called, the SPI master transmits RDSR command and receives the status register byte.

wrsr_eep method

wrsr_eep method can be called by expression:

```
spi_master.wrsr_eep( status_register_byte )
```

, where spi_master is an SPI master object, status_register_byte is a byte that will be written into the status register. Once this method is called, the SPI master transmits WREN command, followed by WRSR command. After WRSR command, the status register is polled with RDSR command until the WIP bit is equal to 0.

clear_chip_eep method

clear_chip_eep method can be called by expression:

```
spi_master.clear_chip_eep(  )
```

, where spi_master is an SPI master object. Once this method is called, the SPI master transmits WRITE commands with all the needed accessories such as WEN command, polling of the status register with RDSR command. These write commands will write the value 0xFF to every byte in the EEPROM memory. Single WRITE command will write one complete page, therefore the number of WRITE commands will be equal to the number of pages in EEPROM device.

8.2. Logic analyzer tests - Stream and LUT mode

In order to test the logic functionality in depth, first we must ensure that what is being reported about the SPI interface by the MCU is actually correct and present on the actual data lines. For the slave and master, the same type of MCU is used and therefore there could be some type of unknown internal error that would cause the MCUs to report correct values, but the actual SPI transmission would be flawed and both of these MCUs would have the same decoding error causing them to report data correctly. In order to test this, a DSLogic logic analyzer was used manufactured by the DreamSourceLab company [25].

8.2.1. Stream mode verification

First, the stream mode is verified using the logic analyzer. The slave is set to stream mode. The master is set to transmit 10 bytes = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. Clock frequency of the master is 54kHz. Logic analyzer output was analyzed by DSView software and can be seen in figure 8.1. As can be seen from the figure, 10 bytes clocked on the SPI clock pin and the MOSI line data are correct. The MISO line is empty because the device does not transmit data in stream mode.

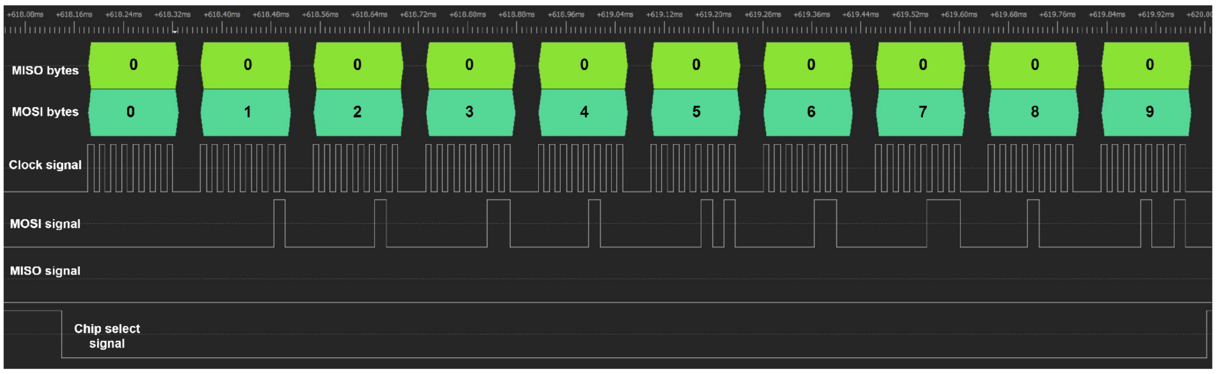
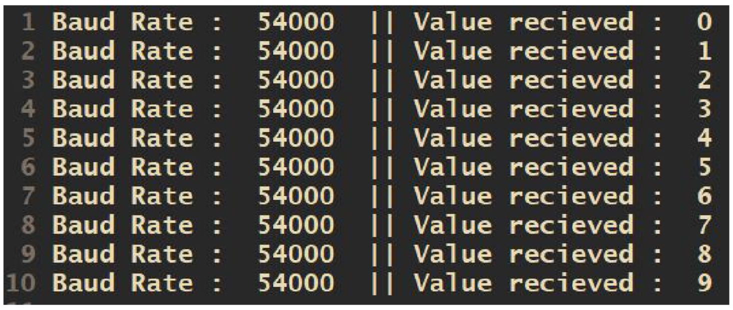


Figure 8.1: Logic analyzer test result for stream mode verification

In figure 8.2, output of the python API can be seen. As can be seen from the figure, API receives reports about 10 bytes received by the slave device and their values as well as the SPI clock frequency are valid.



```
Baud Rate :  54000  || Value recieved :  0
Baud Rate :  54000  || Value recieved :  1
Baud Rate :  54000  || Value recieved :  2
Baud Rate :  54000  || Value recieved :  3
Baud Rate :  54000  || Value recieved :  4
Baud Rate :  54000  || Value recieved :  5
Baud Rate :  54000  || Value recieved :  6
Baud Rate :  54000  || Value recieved :  7
Baud Rate :  54000  || Value recieved :  8
Baud Rate :  54000  || Value recieved :  9
```

Figure 8.2: Python API output for logic analyzer stream mode verification

8.2.2. LUT mode half duplex verification

Next, the LUT mode in half-duplex configuration is verified using the logic analyzer. The slave is set to LUT half-duplex mode. One row of LUT is configured to include:

- **request = [1, 2, 3, 4, 5]**
- **response = [5, 4, 3, 2, 1]**

Master is set to transmit 5 bytes equal to the existing request in LUT = [1, 2, 3, 4, 5]. After these 5 bytes are transmitted by the master, the master is set to receive 5 bytes. The clock frequency of master is 54kHz. Logic analyzer output was analyzed by DSView software and can be seen in figure 8.3. As can be seen by the figure, 5 bytes from the first packet are transmitted by the master and are present on the MOSI line, MISO line is empty. Second packet resembles the response sent by the SPI slave device. Correct 5 bytes are present on the MISO line.

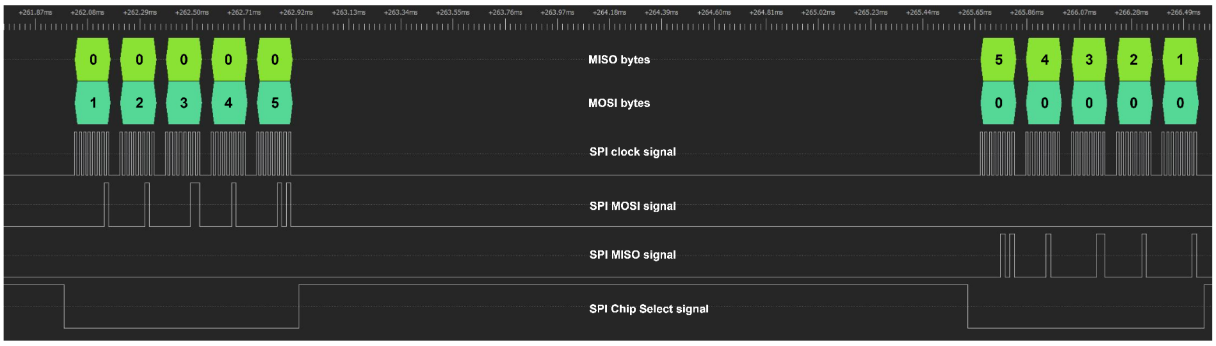
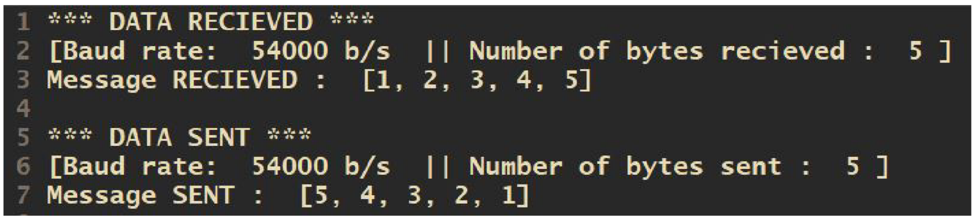


Figure 8.3: Logic analyzer test result for LUT half-duplex mode verification

In figure 8.4, output of the python API can be seen. As can be seen from the figure, first a receival of 5 bytes is reported. Byte values are correct and matching to the request in LUT and the SPI clock frequency is correct also. Next, the transmission of 5 bytes is reported. The byte values are correct and matching to the response in LUT and also the SPI clock frequency is correct.



```
*** DATA RECIEVED ***
[Baud rate:   54000 b/s   || Number of bytes recieved :  5 ]
Message RECIEVED :   [1, 2, 3, 4, 5]

*** DATA SENT ***
[Baud rate:   54000 b/s   || Number of bytes sent :  5 ]
Message SENT :   [5, 4, 3, 2, 1]
```

Figure 8.4: Python API output for logic analyzer LUT half-duplex mode verification

8.2.3. LUT mode full duplex verification

Next, the LUT mode in full-duplex configuration is verified using logic analyzer. The slave is set to LUT full-duplex mode. **Default response** is set to : [**0, 0, 0, 0, 0**] and one row of LUT is configured to include:

- **request** = [**1, 2, 3, 4, 5**]
- **response** = [**5, 4, 3, 2, 1**]

The master is set to transmitreceive 5 bytes equal to the existing request in LUT = [1, 2, 3, 4, 5]. In this transmitreceive, default response is expected to be received. After these 5 bytes are transmitreceived by the master, the master is set to transmitreceive 5 bytes, which values does not matter for this test. Response from LUT is expected to be received in this transmission. The clock frequency of the master is 54kHz. Logic analyzer output was analyzed by DSView software and can be seen in figure 8.5. As can be seen by the figure, 5 bytes from the first packet are transmitted by master and are present on the MOSI line while at the same time, the default response is received on the MISO line. The second packet resembles the response sent by the SPI slave device on the MISO line and the MOSI line contains 5 bytes sent by the SPI master that represent the next request in LUT and their values are not important for this test.

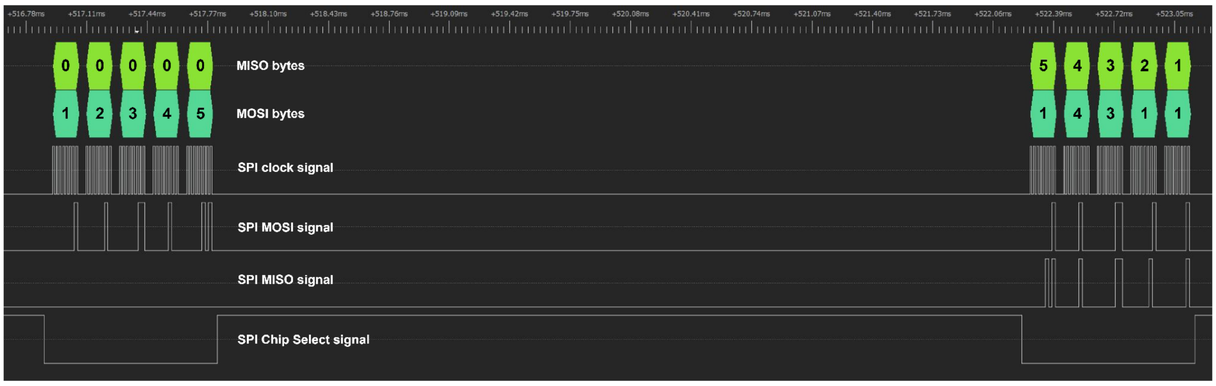
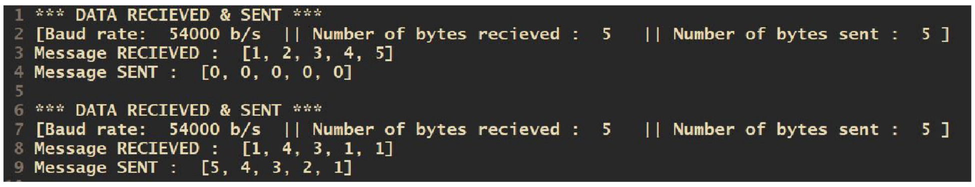


Figure 8.5: Logic analyzer test result for LUT full-duplex mode verification

In figure 8.6, output of the python API can be seen. As can be seen from the figure, transmitreceival of 5 bytes is reported. Request inside LUT was received and default response was transmitted. The byte values are correct and the SPI clock frequency is correct also. Next, transmitreceive of 5 bytes is reported. Response from LUT is transmitted, while 5 bytes are received, which values are not important for this test. Byte values are correct and also the SPI clock frequency is correct.



```
*** DATA RECIEVED & SENT ***
[Baud rate:  54000 b/s  || Number of bytes recieved :  5   || Number of bytes sent :  5 ]
Message RECIEVED :  [1, 2, 3, 4, 5]
Message SENT :  [0, 0, 0, 0, 0]

*** DATA RECIEVED & SENT ***
[Baud rate:  54000 b/s  || Number of bytes recieved :  5   || Number of bytes sent :  5 ]
Message RECIEVED :  [1, 4, 3, 1, 1]
Message SENT :  [5, 4, 3, 2, 1]
```

Figure 8.6: Python API output for logic analyzer LUT full-duplex mode verification

8.3. More detailed tests - Stream and LUT mode

For more detailed testing of the stream and LUT mode, a python test structures were implemented. There are 5 tests in total that are needed to test functionality of the modes from this chapter. All tests have the same test structure, and these tests are differentiated with the slave configuration, what the SPI master is transmitting and what is the valid output of the test. In all described tests, the master's SPI clock frequency was set to 54kHz.

8.3.1. Common test structure

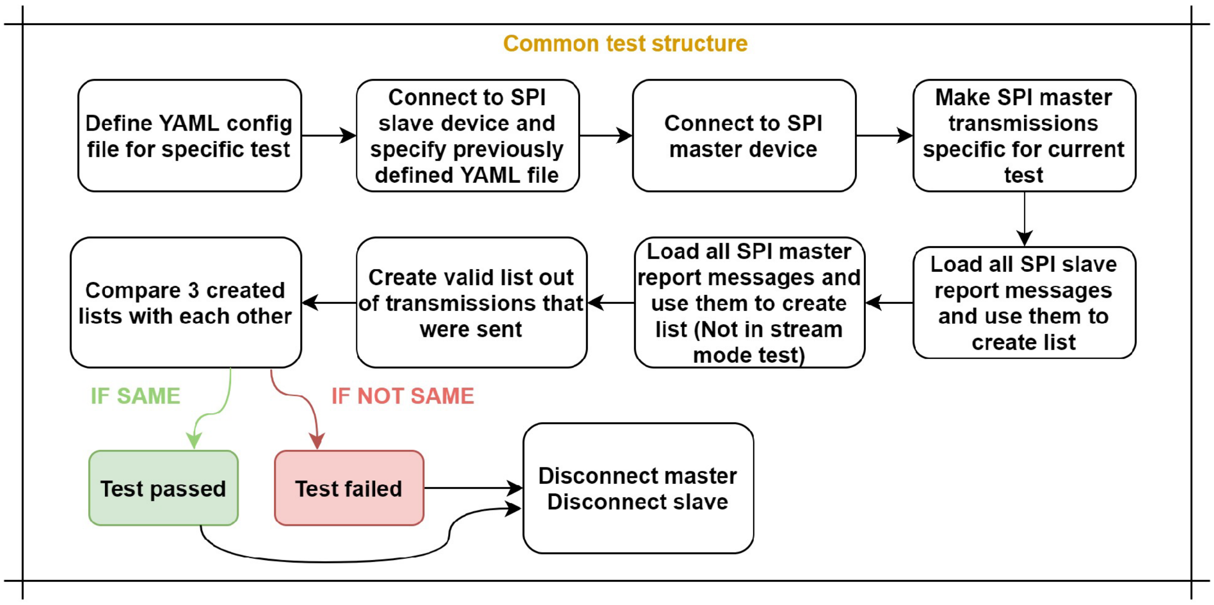

Common test structure
Define YAML config file for specific test
Connect to SPI slave device and specify previously defined YAML file
Connect to SPI master device
Make SPI master transmissions specific for current test
Load all SPI slave report messages and use them to create list
Load all SPI master report messages and use them to create list (Not in stream mode test)
Create valid list out of transmissions that were sent
Compare 3 created lists with each other
IF SAME
IF NOT SAME
Test passed
Test failed
Disconnect master Disconnect slave

Figure 8.7: Common test structure for modes in this chapter

Common test structure can be seen in figure 8.7 and consists of:

- **Define YAML file**
  In other words, create a slave configuration for the specific test. This configuration is defined by creating a YAML file that will be later used to configure the SPI slave device upon connection. This part of algorithm is test specific.

- **Connect to SPI slave device**
  Connect to the SPI slave device and specify the created YAML file as the input configuration. Set print_info parameter to False.

- **Connect to SPI master device**
  Connect to the SPI master device. Set print_info parameter to False.

- **Create SPI master transmissions**
  Every test will require different transmissions to be initiatet by the master device. Commands will be sent to the master about these transmissions and these SPI transmissions will execute in blocking mode. This part of algorithm is test specific.

- **Load all SPI slave report messages and create a list**
  After every transmission, SPI slave device will send a report message to the python API. These messages are loaded and a list is created from these messages. List contains all data from the messages appended together in chronological order.

- **Load all SPI master report messages and create a list**
  After every transmission, SPI master device will send a report message to the python API. These messages are loaded and a list is created from these messages. List contains all data from the messages appended together in chronological order. This part will happen in all tests except for stream mode test.

- **Create a valid list out of tranmsmissions**
  Both the list created from the slave reports and the list created from the master reports have only one possible correct value based on the specific test and what was transmitted via the SPI interface. A new list is created that represents this only correct possibility.

- **Compare 3 created lists**
  All 3 lists are compared. For the test to pass, all 3 lists must be absolutely the same. If any list differs from another, test is declared as failed. For the stream mode test, only 2 lists are compared.

- **Disconnect master and slave**
  Slave and master devices are disconnected.

8.3.2. Stream mode test

YAML configuration

Slave configuration for the stream mode is simple. Only thing that is needed is to enable stream mode.

SPI transmissions

In the stream mode test, the SPI transmissions can be completely random, meaning random data, random size and random type. There is no expected action from the SPI slave. In the recorded test, transmissions were defined in this order:

```
master.transmit( [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] )
master.receive( 5 )
master.transmitreceieve( [4, 5, 6] )
```

Slave list creation

Slave list will consist out of the stream report messages. Data from each report will be appended together in chronological order.

Master list creation

In this test, no master list will be created.

Valid list creation

Three possible master commands can be set to execute:

- **Transmit**
  The actual bytes of the transmission are appended to the list.

- **Receive**
  Zeros are appended to the list. The number of zeros to append is the same as the length of the receive command. In this command, the master will clock the set number of zeros on MOSI line.

- **TransmitReceive**
  The actual bytes of the transmission are appended to the list.

8.3.3. LUT half duplex test

YAML configuration

Slave configuration for this test is a little bit more complex. First, the slave device is set to LUT mode and the full-duplex parameter is set to False. Next, the LUT pairs need to be defined. These LUT pairs will be configured into the slave device and also used to generate transmissions, that will be sent by the SPI master. The data size is not defined yet, because every LUT pair could have different size, therefore the data size will be set before every SPI master transmission. In the recorded test, LUT pairs displayed in table 8.1 were defined:

LUT row	**Request packet**	**Response packet**
0	[1, 2, 3, 4, 5]	[5, 4, 3, 2, 1]
1	[11, 66]	[55, 65]
2	[11, 66, 8, 8, 8]	[55, 65, 1, 2, 3]
3	[8]*10	[4]*10
4	[3]*7	[200]*7

Table 8.1: Lookup table for LUT half-duplex test

*Note - [3]*7 means list with 7 bytes, where all values are equal to 3

SPI transmissions

The transmissions sent by the master device are dependant on LUT pairs. For every LUT pair, two SPI transmissions will happen. First, the request will be transmitted by the master and after that, a receive will be initiated by the master for the same number of bytes as is the length of the response in the current LUT pair. Before every LUT pair transmissions, the SPI slave device must be configured for the proper data size that is equal to the request or response length in the current LUT pair.

Slave list creation

Slave list will consist out of receive and transmit reports. Data from each report will be appended together in chronological order.

Master list creation

Master list will consist out of transmit and receive reports. Data from each report will be appended together in chronological order.

Valid list creation

Valid list will contain all the LUT data appended together in order. For every LUT row, first the request will be appended followed by the response.

8.3.4. LUT full duplex test

YAML configuration

First, the slave device is set to LUT mode and full-duplex parameter is set to True. Next, the LUT pairs need to be defined. These LUT pairs will be configured into the slave device and also used to generate transmissions that will be sent by the SPI master. The data size is configured also. The data size for every LUT pair has to be the same because in full-duplex mode, the response for the previous request will be received at the same time as next request is being sent. In the recorded test, default response is set to all zeros and LUT pairs displayed in table 8.2 were defined:

LUT row	Request packet	Response packet
0	[1, 2, 3, 4, 5]	[5, 4, 3, 2, 1]
1	[1, 4, 3, 1, 1]	[88, 2, 1, 1, 1]
2	[4, 4, 4, 4, 4]	[255, 200, 255, 1, 1]
3	[2, 2, 2, 2, 2]	[82, 82, 82, 82, 82]

Table 8.2: Lookup table for LUT full-duplex test

SPI transmissions

Transmissions sent by the master device are dependant on LUT pairs. Every LUT pair will be tested. Only the transmitreceive commands will be executed by the SPI master. The absolute first transmitreceive command will transmit first LUT request and receive default response of the slave device. Every transmitreceive command after the first one will receive the response for the last request sent and also at the same time transmit the next LUT request. This is the reason that the data size must be a constant for all the LUT pairs in this test.

Slave list creation

Slave list will consist out of transmitreceive reports. Both the request and the response data from each command will be appended together into one list in chronological order. From every command, the first data that was sent on the MOSI line will be appended and after that, the data from the MISO line will be appended.

Master list creation

Master list will be created the same way as the slave list, but from the master device reports and not the slave device reports.

Valid list creation

Valid list will contain all LUT data appended together in order. For every LUT row, first the request will be appended followed by the response.

8.3.5. Default response test - LUT half duplex mode

YAML configuration

In this test, only the default response is tested. The slave device is configured to LUT mode with full-duplex parameter set to False. Data size is clearly defined and it is the same as the length of the default response that is also defined. In the recorded test, default response was set to: [1, 2, 3, 9, 200, 255].

SPI transmissions

Only 2 SPI transmissions are performed by the master device in this test. First command is a transmit command. Value of bytes does not matter as the slave LUT is cleared, but the length of this transmit command must be the same as the default response length. The next command is a receive command of the same number of bytes as is the default_response length. In this receive command, the default response is expected to appear on the MISO line.

Slave list creation

Slave list will consist of transmit and receive reports. Data from these reports will be appended together in chronological order.

Master list creation

Slave list will consist of transmit and receive reports. Data from these reports will be appended together in chronological order.

Valid list creation

Valid list will firstly contain zeros. The number of these zeros is equal to the default response length. After these zeros, the default response is appended.

8.3.6. Default response test - LUT full duplex mode

YAML configuration

In this test, only the default response is tested. Slave device is configured to LUT mode with full-duplex parameter set to True. Data size is clearly defined and it is the same as the length of the default response that is also defined. In the recorded test, default response was set to: [1, 3, 9, 200, 255].

SPI transmissions

Only 1 SPI transmission is performed by the master in this test. One transmitreceive command is executed by the SPI master with the length same as is the length of the default response. In this command, default response is expected to appear on the MISO line. Transmitted data are zeros as this data is not important.

Slave list creation

Slave list will consist of one transmitreceive report. First, the data on the MOSI line will be appended, then, the data on the MISO line will be appended.

Master list creation

Master list will consist of one transmitreceive report. First, the data on the MOSI line will be appended, then, the data on the MISO line will be appended.

Valid list creation

Valid list will firstly contain zeros. The number of these zeros is equal to the default response length. After these zeros, default response is appended.

8.3.7. Tests conclusion

The tests were performed in such way to test every functionality of stream and LUT mode. All tests passed. For changing the test data, it is possible to just edit the transmission definitions for each test. More complex as well as simpler transmissions were also tried for each test with the same result. In described tests, the SPI clock frequency is relatively low, therefore different SPI clock frequencies were tried and result was the same.

8.4. Logic analyzer verification - EEPROM emulator mode

First, the basic command functionality of the EEPROM emulator must be checked. Real 25AA160D EEPROM device was acquired and connections to the SPI master were made in such way, that the emulator and the real device could be easily swapped. The SPI clock frequency was set to 54kHz. All six instructions were performed on the emulator device and on the real device and compared using the same logic analyzer as in previous tests. Volatility of the memory was taken into consideration and instructions were executed in such way that both the real device and the EEPROM emulator should have the same memory and status register state for every instruction. Both the emulator and the real device results for each instruction are displayed below. Logic analyzer signals of the emulator are completely the same as signals of the real EEPROM device. All signal items in following figures from top to bottom are:

- MISO bytes
- MOSI bytes
- Decoded command data
- Command description
- Clock signal
- MOSI signal
- MISO signal
- Chip select signal

8.4.1. WRITE

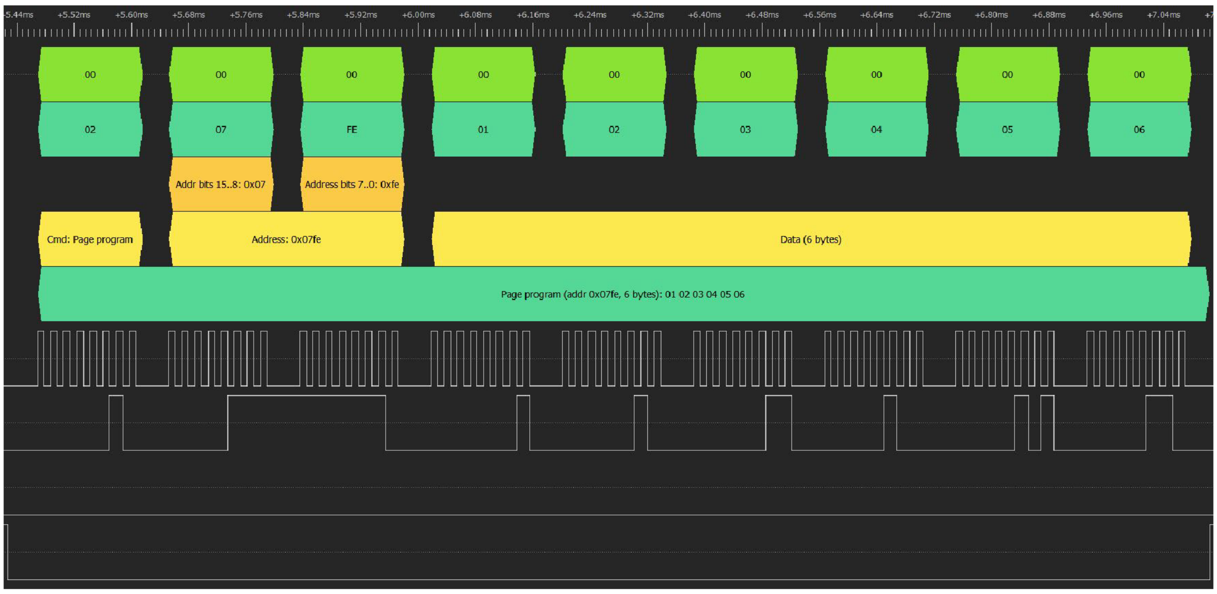


Figure 8.8: Write command executed on emulator device

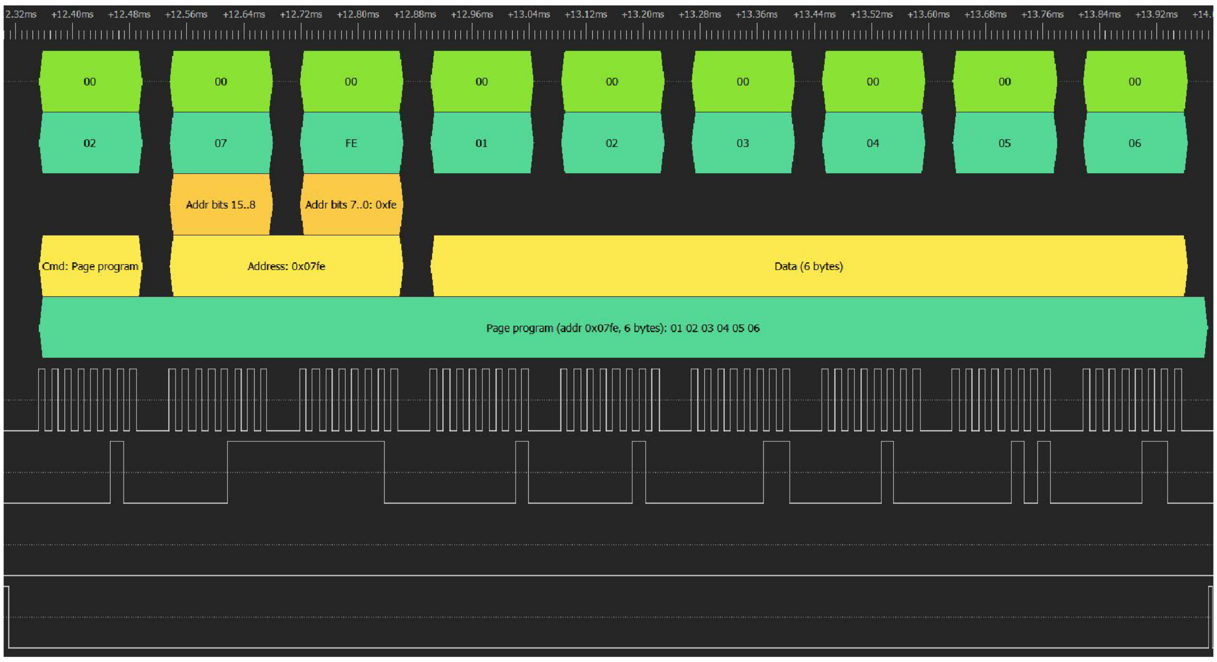


Figure 8.9: Write command executed on real device

8.4.2. READ

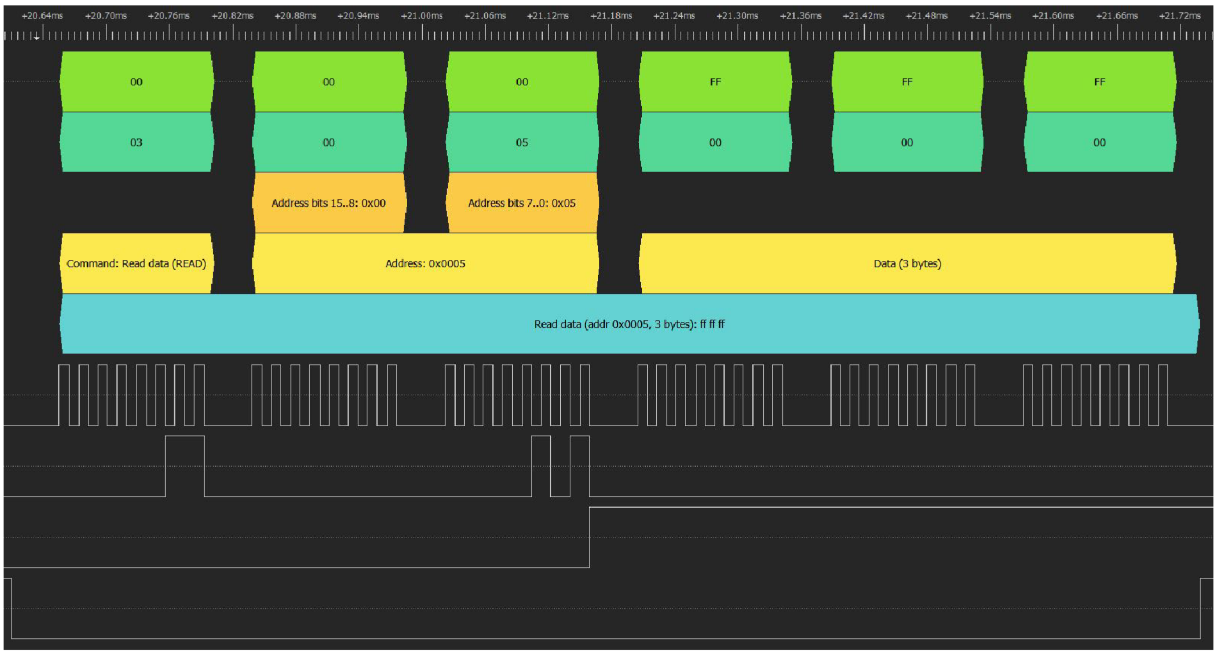


Figure 8.10: Read command executed on emulator device

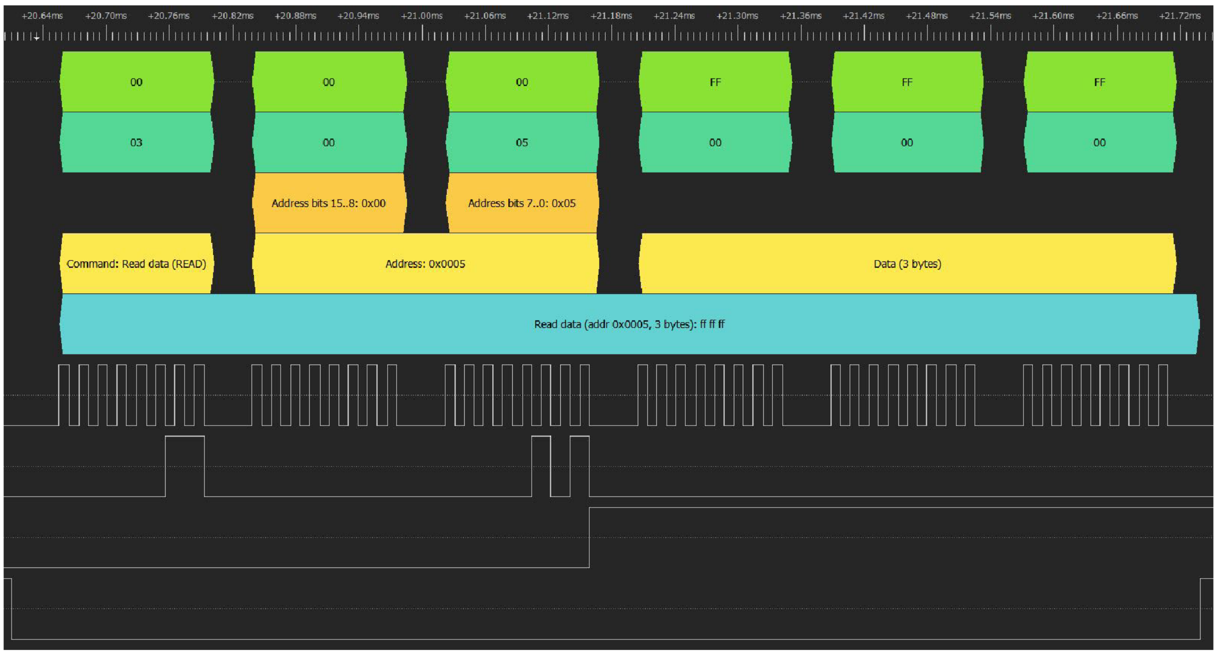


Figure 8.11: Read command executed on real device

8.4.3. WRSR

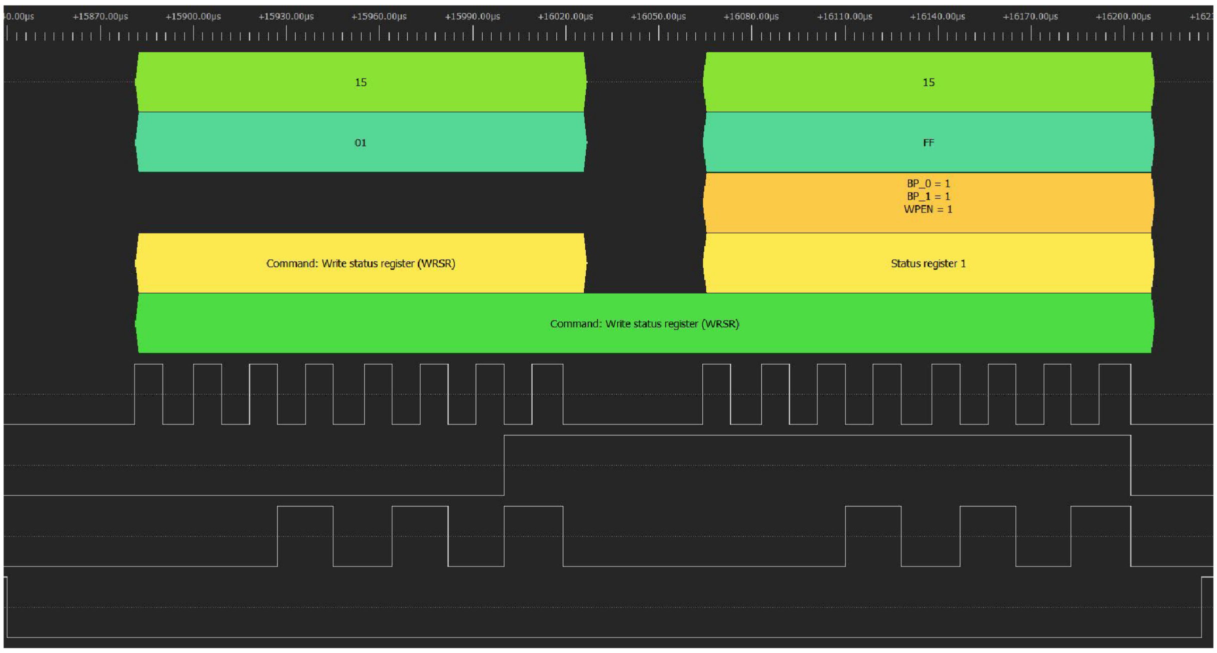

15
01
15
FF
BP_0 = 1
BP_1 = 1
WPEN = 1
Command: Write status register (WRSR)
Status register 1
Command: Write status register (WRSR)

Figure 8.12: Wrsr command executed on emulator device

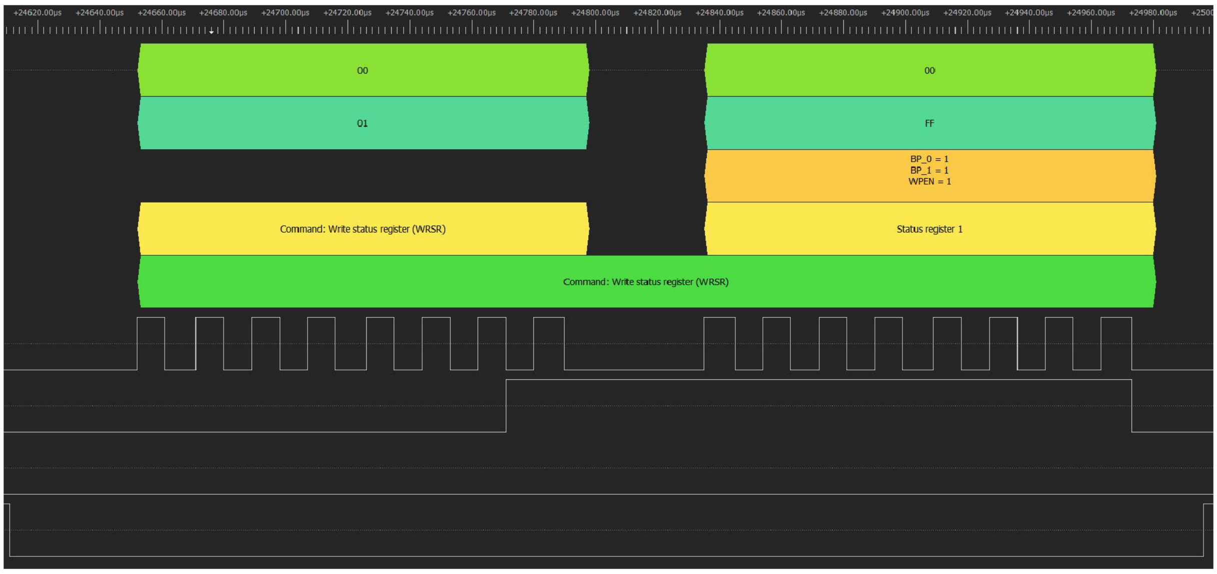

00
01
00
FF
BP_0 = 1
BP_1 = 1
WPEN = 1
Command: Write status register (WRSR)
Status register 1
Command: Write status register (WRSR)

Figure 8.13: Wrsr command executed on real device

*Note : MISO values are different because in the time of measurement, emulator had problem with clearing the internal SPI Tx buffer. This problem was resolved in later development. All wrong MISO values are when a byte is not being sampled as no transmit from the slave device is expected.

8.4.4. RDSR

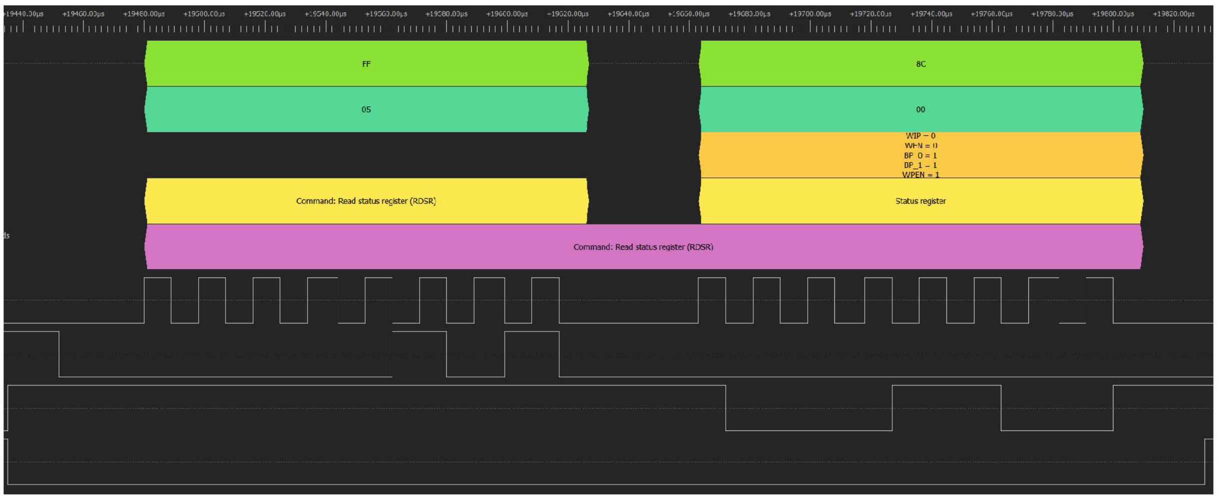

FF
05
Command: Read status register (RDSR)
8C
00
WIP = 0
WEN = 0
BP_0 = 1
BP_1 = 1
WPEN = 1
Status register
Command: Read status register (RDSR)

Figure 8.14: Rdsr command executed on emulator device

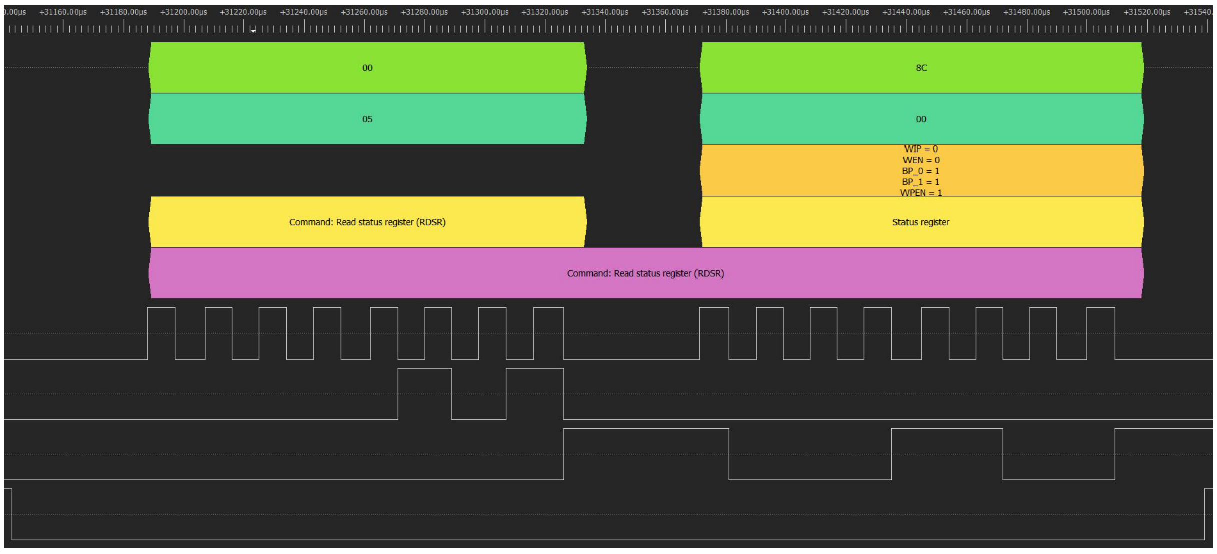

00
05
Command: Read status register (RDSR)
8C
00
WIP = 0
WEN = 0
BP_0 = 1
BP_1 = 1
WPEN = 1
Status register
Command: Read status register (RDSR)

Figure 8.15: Rdsr command executed on real device

*Note : MISO values are different because in the time of measurement, emulator had problem with clearing the internal SPI Tx buffer. This problem was resolved in later development. All wrong MISO values are when a byte is not being sampled as no transmit from the slave device is expected.

8.4.5. WREN

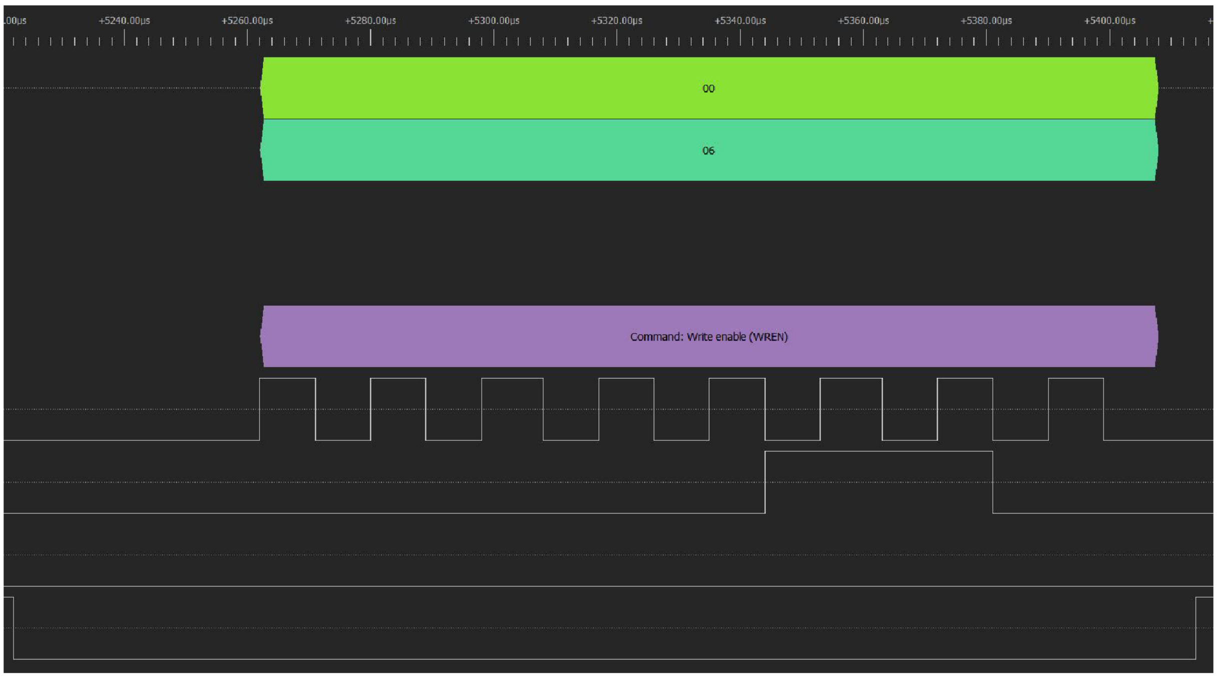


Figure 8.16: Wren command executed on emulator device

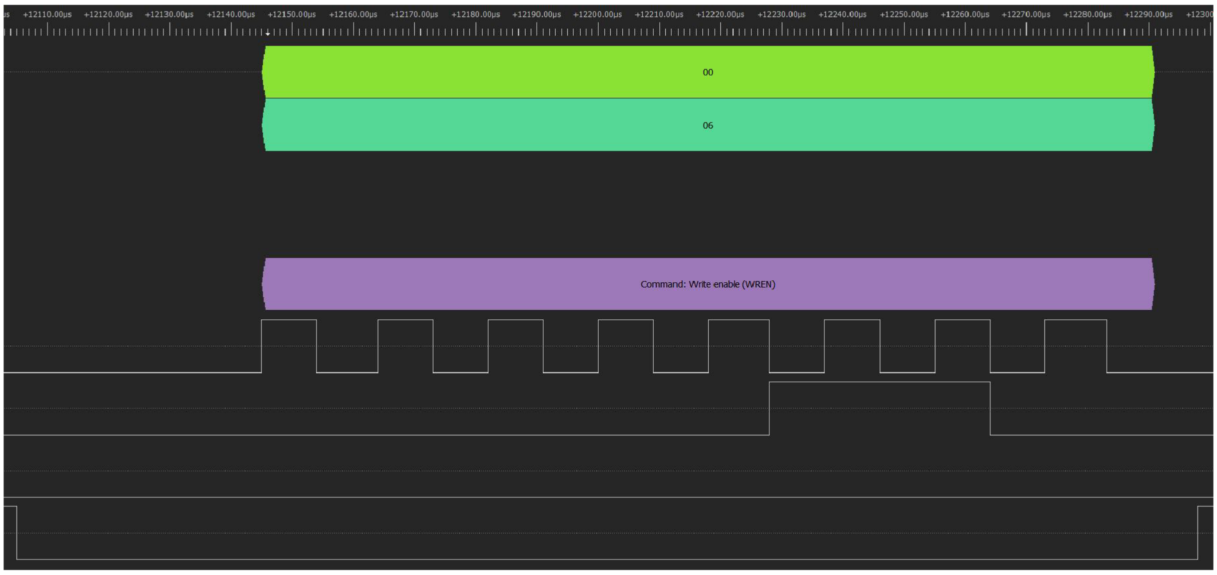


Figure 8.17: Wren command executed on real device

8.4.6. WRDI

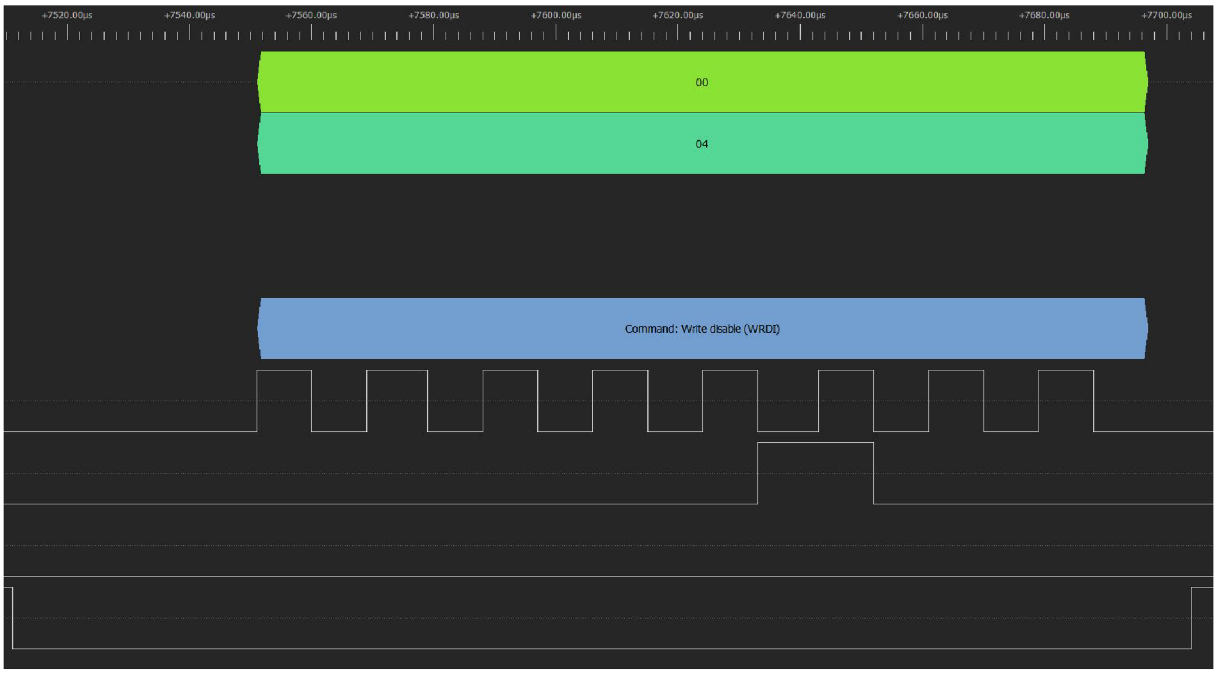


Figure 8.18: Wrdi command executed on emulator device

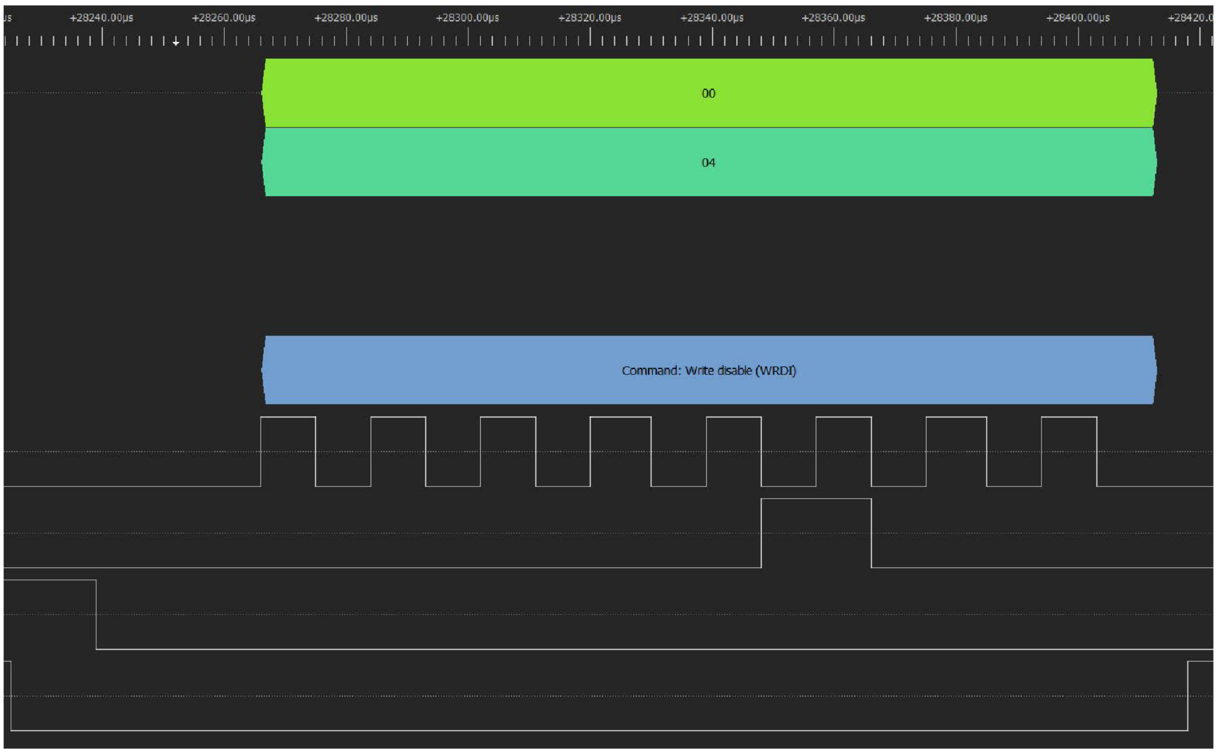


Figure 8.19: Wrdi command executed on real device

8.5. More detailed Tests - EEPROM emulator mode

For more thorough testing of the EEPROM emulator device, multiple tests were invented. These tests should test most of the possible ways an EEPROM may be used. All the tests will firstly execute on the EEPROM emulator device. Next, all the tests will execute on the real EEPROM device. If all tests pass, it means that the behaviour of the emulator and the real device is completely the same.

8.5.1. Common structure

All tests have a common structure and the core functions of this structure are changed according to the specific test. The common structure can be seen in figure 8.20.

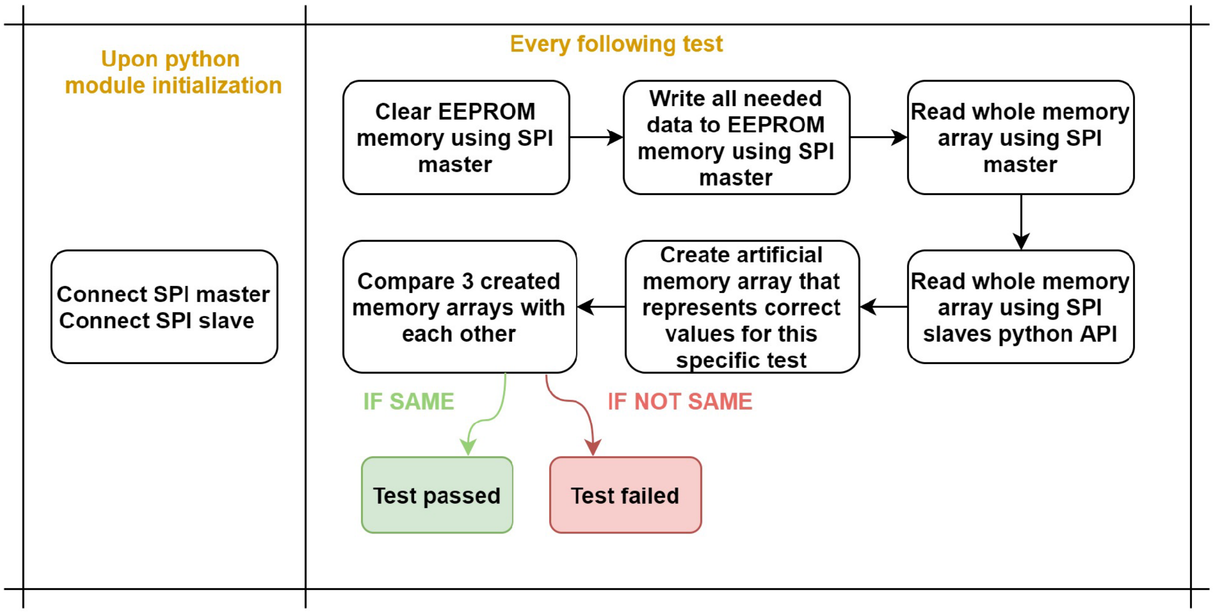

Upon python module initialization
Connect SPI master Connect SPI slave
Every following test
Clear EEPROM memory using SPI master
Write all needed data to EEPROM memory using SPI master
Read whole memory array using SPI master
Read whole memory array using SPI slaves python API
Create artificial memory array that represents correct values for this specific test
Compare 3 created memory arrays with each other
IF SAME
IF NOT SAME
Test passed
Test failed

Figure 8.20: Common structure for EEPROM emulator more advanced tests

- **Upon module initialization**
  Upon the python module initialization, the SPI master and the SPI slave devices need to be connected to the python APIs. For the SPI slave, a YAML file is defined with the parameters: eeprom_mode: True, clear_eeprom: False, clear_undefined_config_params: True.

- **Clear EEPROM memory using the SPI master**
  This part of the algorithm clears the whole EEPROM memory using the SPI master and therefore using the instructions transferred via the SPI interface. This part is a test itself, meaning that for every test, actually two tests are executed. First, the clear chip test is executed followed by the actual test. This is not the case for pure clear chip test as the clear chip test would execute two times subsequently.

- **Write all the needed data to the EEPROM memory using the SPI master**
  In this part, the actual test data are written to the EEPROM using the SPI interface. This part is test specific.

- **Read whole memory array using the SPI master**
  The whole EEPROM memory is read through READ instructions over the SPI interface. Data are read one page at a time. The whole memory array is saved into a list.
- **Read whole memory array using the SPI slave's python API**
  The whole EEPROM memory is read through the UART using the slave's python API. Data are read one page at a time. Whole memory array is saved into a list. This part does not execute when the real EEPROM device is connected as there is no connection through UART.
- **Create artificial memory array**
  Every test has a valid state of the whole memory array after all the instructions were transmitted. This state is test specific. A list representing the valid state of the whole memory array is artificially created.
- **Compare all memory arrays**
  In case when the emulator is connected, all three memory array lists are compared. Test passes if they are completely same. In case when the real device is connected, only two memory array lists are compared.

8.5.2. Clear chip test

Clear chip test clears the whole memory array to value 0xFF.

SPI master commands

Only needed master command is

```
spi_master.clear_chip( )
```

This command clears the whole memory array one page at a time.

Valid memory array

Valid memory array for this test is an array with value of every byte equal to 0xFF.

8.5.3. Full page write test

Writes every byte in every page.

SPI master commands

Pages will be written one page at a time. The starting address of WRITE instruction will always be the first byte of a page, therefore no page overflow can happen. Every byte in every page will be written. Every byte in the same page will be equal to the current page index.

Valid memory array

Memory array is recreated where every page contains only the index of that same page.

8.5.4. Full page write with offset test

Writes every byte in every page.

SPI master commands

Pages will be written one page at a time. The starting address of WRITE instruction is not the first byte of a page, therefore page overflow will happen. The starting index of a page can be specified. Every byte in every page will be written. Every byte in the same page will be equal to the current page index.

Valid memory array

Memory array is recreated where every page contains only index of that same page.

8.5.5. Partial page write test

Every page will be written, but not all bytes in the page will be written.

SPI master commands

Pages will be written one page at a time. The starting address of WRITE instruction can be specified as well as the number of bytes to write can be specified. Every written byte in one page will be equal to the current page index.

Valid memory array

Memory array is created where every written byte equals to the current page index and all bytes that were not written equal to 0xFF.

8.5.6. Partial page read test

Full page write test is performed but not every byte in every page is read.

SPI master commands

Full page write test is executed, but instead of reading the whole memory array, only the bytes specified by the starting index and the number of bytes to be read are read. Bytes starting from the specified index are read from every page.

Valid memory array

Every page contains the index of that same page. For every page, only the specified number of bytes are appended to list.

8.5.7. Block protect test

Tests all the possible configurations of block protect.

SPI master commands

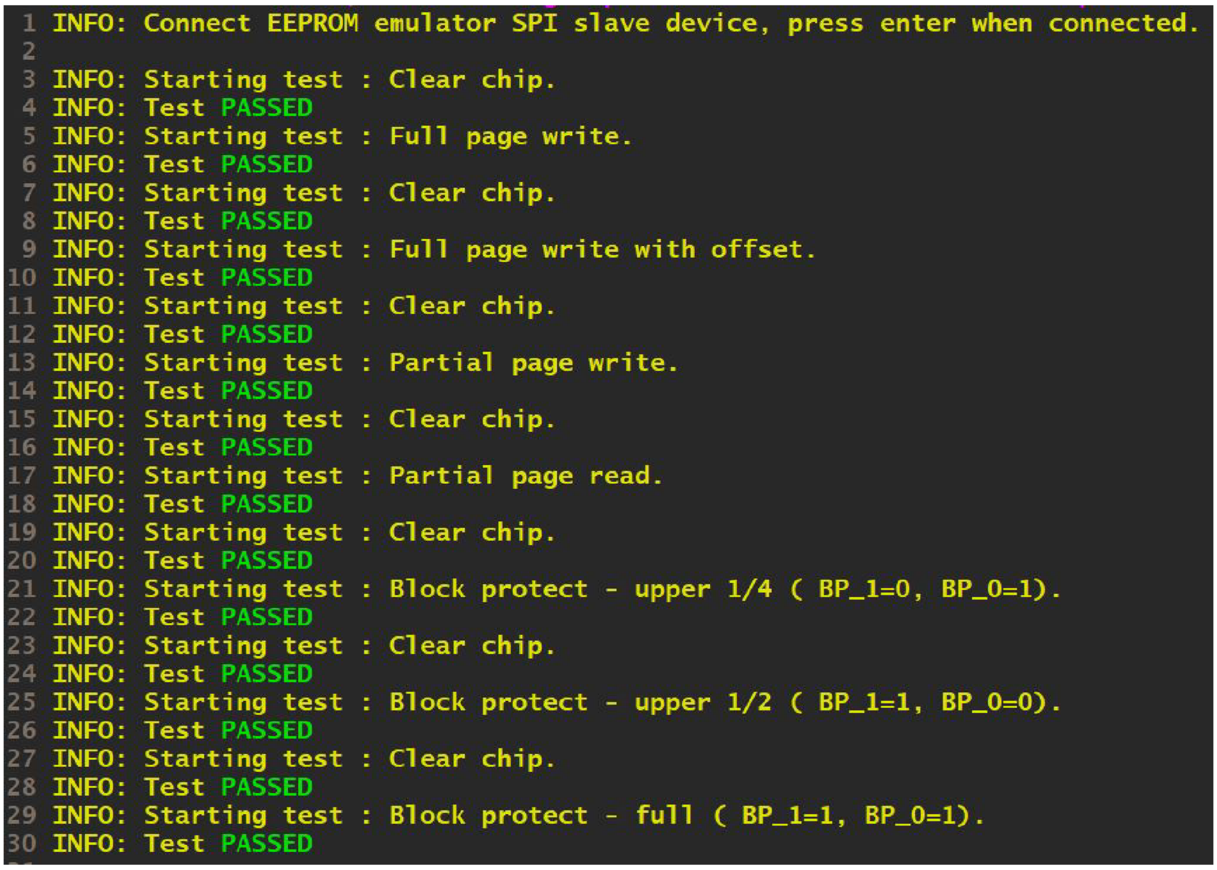
Write full pages test is executed, but prior to that, block protect is set by WRSR command. This test executes 3 times for every possibility of the enabled block protect.

Valid memory array

Every byte that is not protected by the current setting of the block protect contains the index of the page to which the byte belongs. Every byte that is protected by the current setting of the block protect has value 0xFF.

8.5.8. Results

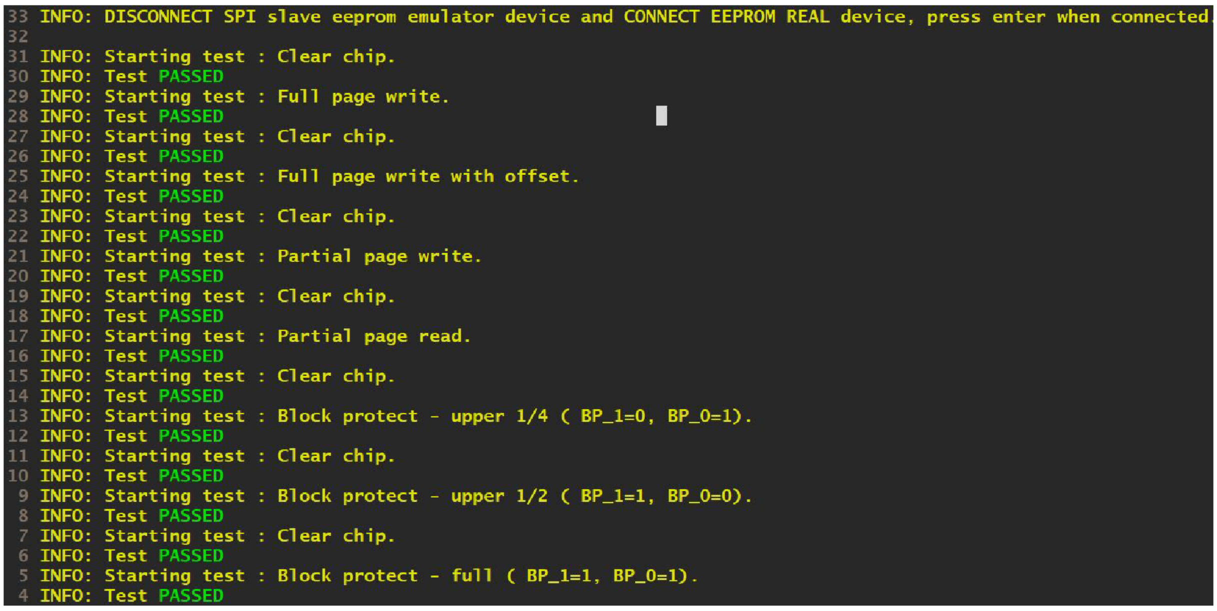
Logs of test results can be seen in figures 8.21 and 8.22.



```
INFO: Connect EEPROM emulator SPI slave device, press enter when connected.

INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Full page write.
INFO: Test PASSED
INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Full page write with offset.
INFO: Test PASSED
INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Partial page write.
INFO: Test PASSED
INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Partial page read.
INFO: Test PASSED
INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Block protect - upper 1/4 ( BP_1=0, BP_0=1).
INFO: Test PASSED
INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Block protect - upper 1/2 ( BP_1=1, BP_0=0).
INFO: Test PASSED
INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Block protect - full ( BP_1=1, BP_0=1).
INFO: Test PASSED
```

Figure 8.21: Test results for EEPROM emulator device



```
INFO: DISCONNECT SPI slave eeprom emulator device and CONNECT EEPROM REAL device, press enter when connected.

INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Full page write.
INFO: Test PASSED
INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Full page write with offset.
INFO: Test PASSED
INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Partial page write.
INFO: Test PASSED
INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Partial page read.
INFO: Test PASSED
INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Block protect - upper 1/4 ( BP_1=0, BP_0=1).
INFO: Test PASSED
INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Block protect - upper 1/2 ( BP_1=1, BP_0=0).
INFO: Test PASSED
INFO: Starting test : Clear chip.
INFO: Test PASSED
INFO: Starting test : Block protect - full ( BP_1=1, BP_0=1).
INFO: Test PASSED
```

Figure 8.22: Test results for real EEPROM device

8.6. Reliability and WIP delay tests - EEPROM emulator mode

Last test performed is about the overall reliablity of write and following read operations. This test also tests the value of the write in progress delay.

8.6.1. Main idea of the test

Random writes with according reads

The main idea of the test is to perform a large number of random writes and verify written data with read command. All commands are done by the SPI master, SPI slave python API is not used. First, a random write data are generated. Data are written on a random address, with random data size and every byte of the write instruction is random. Same data is read from every memory location where the data was written. If written data has gone out of the page boundary, two read instructions must be executed because the write address rolled back to the starting address of the page. Read data is then compared with written data and if they are not same, the current random write is considered to be not sucesfull. The overall success rate is evaluated in percentage.

WIP delay

In this test, also a python script was developed that analyzes data from logic analyzer, where all the random writes and reads are captured. Data is exported from the logic converter file into a .csv file. Python script then analyzes .csv file and measures the average write in progress delay for every write command of the same size. In the end, the write in progress delay was not dependant on the write command byte length and therefore one average value of the write in progress delay was measured. It is important to note that resolution of the measurement was one RDSR command, which takes relatively long time at this SPI clock frequency (390 µs), therefore in reality, the write in progress delay can be dependant on the write command byte length, but it could not be detected with this measurement resolution.

8.6.2. Results

For all devices tested, 7500 random writes with according reads were executed. First, two different real EEPROM devices were tested. First device was minorly flawed as the success rate for this device was **97%**. The second real EEPROM device had success rate of **100%**. The average write in progress delay for both real devices was measured as **2.75 ms**, which translates to 7 RDSR instructions until the WIP bit had value 0. This WIP delay was configured into the EEPROM emulator device and the test was executed on the emulator device. Success rate was **100%** and the analyzed average delay was **2.75 ms**.

9. Conclusion

The main goals of this thesis were to create a testing device for SPI master devices. This testing device should have general testing capability and also the capability to emulate one specific real SPI slave device. STM32F303K8 microcontroller was chosen as a platform for this device. NUCLEO development board was chosen containing this specific MCU. The main characteristic of this device is that it must be configurable from a personal computer. The device is connected to PC's serial USB port via UART interface, while using NUCLEO's ST-LINK as UART-USB adapter. Python API was created that can communicate with the embedded device. The main tasks of the API are to apply configuration onto the device as well as to receive report and acknowledgement messages from the device. Acknowledgement messages are expected to be received after every command. For easy device configuration, a YAML file is supported. The YAML file is created by the user to specify wanted configuration. The configuration from the YAML file is applied onto the device upon connection to the API. Before designing any method that can test the logic of an SPI master device, a feature that measures the SPI clock frequency was developed. The clock frequency of an SPI master can be measured with adequate resolution. First mode that tests the general function of an SPI master is stream mode. Stream mode is receive only mode and the main purpose of this mode is to stream every byte received via SPI directly to the python API. Some problems occurred due to difference in the SPI and UART interface speeds, but most of the problems were resolved by using a circular buffer with head and tail pointers. Using this buffer ensures that reporting of the SPI bytes, that were received, happens asynchronically from actually receiving the SPI bytes. Next method for testing the basic logic of an SPI master is LUT (Look Up Table) mode. This mode allows the user to define LUT pairs. One LUT pair consists of a request that will be received via the SPI interface from the SPI master and a response that will be sent to the SPI master if the according request is received. This mode can function either in full-duplex or half-duplex mode. For more detailed testing of a specific SPI master, one real SPI slave device was chosen and its emulator was developed. An SPI EEPROM was chosen as a real SPI slave device. Emulator was created that accurately emulates logic behaviour of said real SPI slave device. One major limitation of this emulator device is that the maximum clock frequency is much lower than the maximum SPI clock frequency of a real EEPROM device. This is because all logic of the emulator device must be resolved inside interrupts, which is causing this limitation. This limitation should not be considered as major problem, because the developed device can be labeled as "proof of concept" or a prototype. For future development, MCU with much higher frequency could be used, which would massively improve this problem. Also, this limitation was measured while considering continuous bit stream, which is highly unlikely in real situations. Verification tests were developed and executed for all existing features. Results of the tests were successful and all features are functioning as they should.

There are a few possible improvements for future development. One of the major improvements for stream mode would be to modify the process of sending reports. Currently, one report contains only one received byte. It would be more effective if one report contained all the bytes that were not reported yet in that time moment. The plans for the EEPROM emulator mode are to add support for more EEPROM devices. Ideally develop some method how to easily define an universal EEPROM device with one instance in a struct.

All set goals were accomplished and some of them were surpassed. The result of this thesis is a device that can be used to test existing SPI masters. It is possible to test universal SPI master using stream and LUT mode and it is also possible to test specific SPI EEPROM master using EEPROM emulator mode. Device can be configured from a PC via python API.

Bibliography

[1] Aardvark I2C/SPI Host Adapter [online]. Total phase, 2021 [cit. 2021-5-18]. Dostupné z: https://www.totalphase.com/products/aardvark-i2cspi/

[2] USB-SPI Master & Slave Adapter - DLN-4S. Diolan [online]. [cit. 2021-5-18]. Dostupné z: https://diolan.com/dln-4s

[3] Microcontroller. Wikipedia [online]. 2021 [cit. 2021-5-18]. Dostupné z: https://en.wikipedia.org/wiki/Microcontroller

[4] Embedded system. Wikipedia [online]. 2021 [cit. 2021-5-18]. Dostupné z: https://en.wikipedia.org/wiki/Embedded_system

[5] Chip-design compiler keys on 2.5, 3D multi-die system-in-a-package developments. Microcontrollertips [online]. [cit. 2021-5-18]. Dostupné z: https://www.microcontrollertips.com/chip-design-3dic-compiler-keys-on-2-5-3d-multi-die-system-in-a-package-developments/

[6] PIC16F887. Microchip [online]. [cit. 2021-5-18]. Dostupné z: https://www.microchip.com/wwwproducts/en/PIC16F887

[7] Serial Peripheral Interface. Wikipedia [online]. 2021 [cit. 2021-5-18]. Dostupné z: https://en.wikipedia.org/wiki/Serial_Peripheral_Interface

[8] SPI Tutorial. Corelis [online]. [cit. 2021-5-18]. Dostupné z: https://www.corelis.com/education/tutorials/spi-tutorial/

[9] CAMPBELL, Scott. BASICS OF UART COMMUNICATION. Circuitbasics [online]. [cit. 2021-5-18]. Dostupné z: https://www.circuitbasics.com/basics-uart-communication/

[10] UART Explained. Electricimp [online]. [cit. 2021-5-18]. Dostupné z: https://developer.electricimp.com/resources/uart

[11] Universal asynchronous receiver-transmitter. Wikipedia [online]. 2021 [cit. 2021-5-18]. Dostupné z: https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

[12] Cyclic redundancy check. Wikipedia [online]. 2021 [cit. 2021-5-18]. Dostupné z: https://en.wikipedia.org/wiki/Cyclic_redundancy_check

[13] EEPROM. Wikipedia [online]. 2021 [cit. 2021-5-18]. Dostupné z: https://en.wikipedia.org/wiki/EEPROM

[14] STM32 32-bit Arm Cortex MCUs. St [online]. [cit. 2021-5-18]. Dostupné z: https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html

[15] STM32F3 Series. St [online]. [cit. 2021-5-18]. Dostupné z: https://www.st.com/content/st_com/en/products/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus/stm32-mainstream-mcus/stm32f3-series.html

[16] STM32F303. St [online]. [cit. 2021-5-18]. Dostupné z: https://www.st.com/content/st_com/en/products/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus/stm32-mainstream-mcus/stm32f3-series/stm32f303.html

[17] STM32 Nucleo Boards. St [online]. [cit. 2021-5-18]. Dostupné z: https://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-mpu-eval-tools/stm32-mcu-mpu-eval-tools/stm32-nucleo-boards.html

[18] STM32 Nucleo-32 development board with STM32F303K8 MCU. St [online]. [cit. 2021-5-18]. Dostupné z: https://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-mpu-eval-tools/stm32-mcu-mpu-eval-tools/stm32-nucleo-boards/nucleo-f303k8.html

[19] Arduino compare. Arduino [online]. [cit. 2021-5-18]. Dostupné z: https://www.arduino.cc/en/products.compare

[20] Description of STM32F4 HAL and low-layer drivers - User manual. St [online]. 2020 [cit. 2021-5-18]. Dostupné z: https://www.st.com/resource/en/user_manual/dm00105879-description-of-stm32f4-hal-and-ll-drivers-stmicroelectronics.pdf

[21] 25AA160C/D, 25LC160C/D: 16K SPI Bus Serial EEPROM. Microchip [online]. [cit. 2021-5-18]. Dostupné z: https://ww1.microchip.com/downloads/en/DeviceDoc/22150B.pdf

[22] GCC Wiki. Gnu [online]. 2020 [cit. 2021-5-18]. Dostupné z: https://gcc.gnu.org/wiki

[23] YAML. Wikipedia [online]. 2021 [cit. 2021-5-18]. Dostupné z: https://en.wikipedia.org/wiki/YAML?oldid=441923691

[24] Spi_slave. Gitlab [online]. 2021 [cit. 2021-5-18]. Dostupné z: https://gitlab.com/pzencar/spi_slave

[25] DSLogic Series USB-based Logic Analyzer. Dreamsourcelab [online]. 2020 [cit. 2021-5-18]. Dostupné z: https://www.dreamsourcelab.com/product/dslogic-series/

[26] STM32F303x6/x8: Datasheet. St [online]. [cit. 2021-5-19]. Dostupné z: https://www.st.com/resource/en/datasheet/stm32f303k8.pdf

[27] Reference manual: STM32F303x6/8. St [online]. [cit. 2021-5-19]. Dostupné z: https://www.st.com/resource/en/reference_manual/dm00043574-stm32f303xbcde-stm32f303x68-stm32f328x8-stm32f358xc-stm32f398xe-advanced-armbased-mcus-stmicroelectronics.pdf

[28] STM32F303x6/x8 Errata sheet. St [online]. [cit. 2021-5-19]. Dostupné z: https://www.st.com/resource/en/errata_sheet/dm00109011-stm32f303x6x8-rev-z-device-limitations-stmicroelectronics.pdf

List of acronyms

API	Application Programming Interface
WIP	Write In Progress
YAML	YAML Ain't Markup Language
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
DMA	Direct Memory Access
CRC	Cyclic Redundancy Check
LUT	LookUp Table
EEPROM	Electrically Erasable Programmable Read-Only Memory
I2C	Inter-Integrated circuit
GPIO	General Purpose Input-Output
PC	Personal Computer
CPU	Central Processing Unit
RAM	Random Access memory
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
PWM	Pulse Width Modulation
IDE	Integrated Development Environment
MCU	Microcontroller Unit
SCK	Serial Clock
MOSI	Master Out, Slave In
MISO	Master In, Slave Out
SS	Slave Select
CPOL	Clock Polarity
CPHA	Clock Phase
WRSR	Write Status Register

RDSR	Read Status Register
WREN	Write Enable
WRDI	Write Disable
PCB	Printed Circuit Board
APB	Advanced Peripheral Bus
LL	Low Layer
HAL	Hardware Abstraction Layer
WIP	Write In Progress
MSB	Most Significant Bit
LSB	Least Significant Bit
FIFO	First In First Out

List of Figures

List of Tables

Appendix

The thesis contains following appendixes:

- C source codes:
  - spi_main.c
  - spi_main.h
  - uart_main.c
  - uart_main.h
  - uart_cmd_buff.c
  - uart_cmd_buff.h
  - eeprom_devices.c
  - eeprom_devices.h
  - eeprom_emulator.c
  - eeprom_emulator.h
  - LUT.c
  - LUT.h
  - main.c
  - main.h
  - stm32f3xx_it.c
  - stm32f3xx_it.h
- python source codes:
  - spi_slave.py
  - eeprom_unit_test.py
  - eeprom_delay_test.py
  - debugmode_unit_test.py
- STM32CubeMx project file:
  - spi_slave.ioc