PRAXE

Datum: 09.01. 2023	Střední průmyslová škola, Chomutov, Školní 50, příspěvková organizace	Třída: V4
Číslo úlohy: 3.	I^2C – Čtení EEPROM paměti	Jméno: Tomáš Bartoš

Zadání:

S využitím vývojového kitu AVR a paměti I2C řady 24Cxx sestavte v jazyku C program implementující čtení a zápis paměti pomocí I2C sběrnice na základě požadavku ze sériové linky.

Výsledné řešení musí obsahovat následující funkcionalitu:

- Implementace I2C komunikace na úrovni softwaru nebo využitím hardwarové jednotky I2C v procesoru (s úplným pochopením principů, nikoliv copy & paste nalezeným fragmentem kódu)
- Podporu čtení a zápisu libovolné paměťové buňky v adresním rozsahu jednoho čipu.
- Zpracování požadavku na čtení a zápis paměti zaslaný terminálem ze sériové linky ve formátu R 0xAAAA pro čtení dané adresy a W 0xAAAA 0xBB pro zápis daného bytu BB na zadanou adresu AAAA
- Korektní reakci na neplatný vstup nebo absenci ACK bitu ze strany paměti

Teorie:

Řešení této úlohy, kde je hlavním cílem naimplementovat komunikaci s EEPROM paměti skrz I2C protokol zahrnuje 2 části: část softwarovou a hardwarovou.

Po softwarové stránce řešení je zde ke komunikaci na sběrnici I2C použita metoda bit-bangingu. Bit-banging je metoda komunikace na sběrnici, kdy není využito žádné abstrakční vrstvy ve formě pomocných registrů na procesoru, ale přímá manipulace s jednotlivými bity posílanými na sběrnici. Tato metoda tak dovoluje psát programátorovi mnohem laicky čitelnější kód a vyšší kontrolu nad sběrnicí.

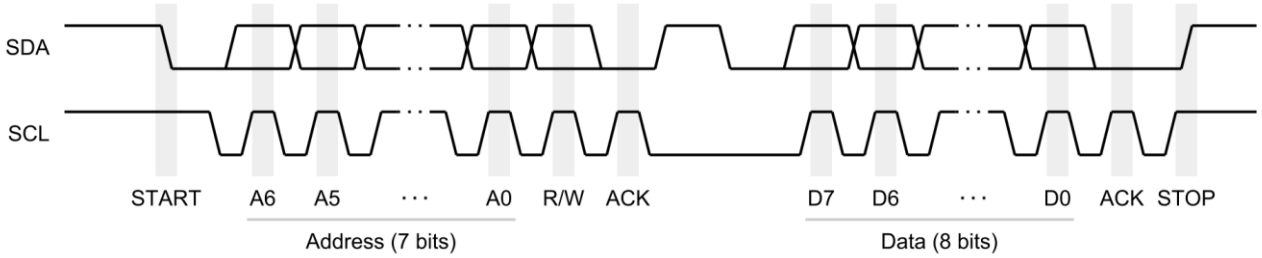
Co se samotného komunikačního protokolu I2C týče, jedná se o protokol vyvinutý firmou Philips v roce 1982 pro komunikaci mezi integrovanými obvody. Původně byl určen pro interní použití v zařízeních Philips, ale později byl zpřístupněn i ostatním výrobcům. Protokol I2C je poloduplexní a pracuje se dvěma vodiči:

- SDA (Serial Data) = přenos dat
- SCL (Serial Clock) = taktovací signály

Stav těchto vodičů se v průběhu komunikace mění, ale platí obecné pravidlo, že SDA je v klidovém stavu na logické úrovni 1, zatímco SCL je ponecháno na logické úrovni 0. Důvod platnosti tohoto pravidla je, že I2C sběrnice je sběrnice se zařízeními pracujícími na principu obvodů s otevřeným kolektorem a stažení log. 1 k zemi je pro ně velmi výhodné.

Z hlediska rolí připojených zařízení může na I2C sběrnici být vždy jen jeden MASTER, přičemž ostatní zařízení se označují jako SLAVE. Pokud chce MASTER získat informace od SLAVE, či s ním, jakkoliv jinak manipulovat musí ho nejdříve „oslovit“ jeho7 bitovou I2C adresou a read/write bitem. Jakmile SLAVE obdrží svou adresu, je povinen ohlásilo svou přítomnost stažením SDA linky do nuly. Tento proces se nazývá acknowledge a v průběhu komunikace je po SLAVE zařízení vyžadován vždy po odeslání jakýchkoliv dat.

Diagram komunikace na I2C rozhraní:



[1]

Co se hardwarových specifik této úlohy týče, je zde použita EEPROM paměť s označením 24c16 s propojovacím rozhraním I2C. Způsob princip funkce této paměti je založen na jednotlivých paměťových buňkách, které jsou uspořádány v matici a jejich základním principem je uchovávání elektrického náboje. Tento proces uchování je založený na využití tranzistorů s tzv. plovoucím hradlem, jež dovoluje „permanentní“ uložení náboje za pomocí jevu tunelování (přenos náboje na izolovaný foloating gate).

Adresování paměťových buněk v paměti funguje na principu 16 bitové adresy a 8 bitového datového slova, které se nachází na dané adrese. Při zápisu je pak vždy zaspán jeden byte dat.

[1] https://articles.saleae.com/logic-analyzers/spi-vs-i2c-protocol-differences-and-things-to-consider

Po stránce hardwaru samotného mikrokontroleru, tak je při řešení této úlohy nejdůležitější vše správně zapojit a nakonfigurovat. Při zapojování je nutno přepnout pomocí jumperu na desce výstup sériové linky ne výstupní RS-232 rozhraní a při konfiguraci MakeFilu nastavit následující parametry:

- Typ procesoru = ATmega128
- Typ programátoru = jtagmk1
- Cílové soubory ke kompilaci = main.c usart.c
- Frekvenci procesoru = 14745600Hz
- Podporu scanf a printf pro čísla s plovoucí čárkou

Po nastavení těchto parametrů je mikrokontroler připraven komunikovat s EEPROM pamětí, co však nesmí chybět je řešení sériové komunikace na rozhraní RS-232. Pro plnou funkčnost tohoto rozhraní je nutné si z webových stránek https://kolousek.spselectronic.cz/v4/ stáhnout avr-demo.zip, kde se nachází soubor usart.c a usart.h, jež je nutné přesunout do složky s main.c a v hlavním zdrojovém kódu knihovnu naimportovat následujícím příkazem:

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

V souboru usart.h je pak nutné přepsat řádek:

```
FILE _usart_io;
```

Na:

```
FILE usart_io;
```

Následná komunikace na rozhraní RS-232 pak ve zkratce probíhá tak, že data jsou posílána sériově bit po bitu. Každý byte je tvořen start bitem, datovými bity, případně paritním bitem a stop bitem. Start bit signalizuje začátek bytu, stop bity ukončují přenos.

Softwarová stránka implementace řešení běžící na mikrokontroleru ATmega128 je v základu postavena na práci s těmito třemi registry:

- Registr směru = DDRx
- Datový /výstupní registr = PORTx
- Vstupní registr = PINx

Popis programu:

Fáze inicializace:

Ihned po překladu kódu a jeho následném nahrání do mikrokontroleru nastane fáze inicializace. V této fázi probíhají primárně 2 hlavní činnosti. První z těchto činností je import interních knihoven pro samotnou práci s mikrokontrolerem společně s importem knihovny pro sériovou komunikaci. Další částí je deklarace globálních proměnných programu, přímo související s I2C rozhraním, konkrétně se pak jedná o proměnné následující:

- SDA = adresa pinu pro posílání dat
- SCL = adresa pinu pro posílání hodin
- SLAVE_ADDR = definuje adresu zařízení
- I2C_DELAY = pauza mezi operacemi na sběrnici
- MIN a MAX ADDR = minimální a maximální adresa v EEPROM paměti.

Krom těchto globálních proměnných je v první fázi také deklarovaná datová struktura UserInput, sloužící jen pro uložení a následně zjednodušenou práci s uživatelský vstupem.

Druhou a poslední fází inicializace je vyvolání metody usart_setup z knihovny usart.h, která slouží k inicializaci komunikace na sériové lince jak po stránce hardwarové, tak softwarové. Fáze dva pak končí nastavením pinů SDA a SCL jakožto výstupních

Hlavní smyčka programu:

Po vykonání metody inicializace se běh programu přesouvá do nekonečné smyčky, která slouží jakožto uživatelský interface pro ovládání komunikace s EEPROM pamětí. Prvním krokem pro zadávání příkazu generátoru je prompt posílaný po sériové lince pomocí metody printf a argumentu „Enter an EEPROM command “, následuje další smyčka porovnávající aktuální stisknutou klávesu uživatele (proměnná lastChar) s enterem tak, aby se předešlo validaci vstupu ještě před jeho nekompletností. Ve smyčce získávající vstup od uživatele je vždy stisknutá klávesa uložena do proměnné lastChar pokud není nulová ukládá se do pole buffer, jež se indexuje pomocí

proměnné bufferIndex. V případě, že je bufferIndex vyšší než velikost pole, pole se přemaže metodou memset a bufferIndex anuluje.

Jakmile je zmáčknuta klávesa enter, vnořená smyčka se přeruší a program přechází k validaci uživatelského vstupu. Vstup od uživatele neboli pole buffer, je předáno společně s adresou globální proměnné userData jakožto argument metodě parseUserInput. Daná metoda nejdříve anuluje dílčí proměnné proměnné userData, jejímž datovým typem je custom struktura UserInput, a následně vstup od uživatele rozdělí metodou strtok a delimetru , ‘. Následuje iterace skrz rozdělené prvky a jejich postupné ukládání do dílčích proměnných proměnné userData. Po tomto rozdělení vstupu od uživatele do přívětivější formy jsou za pomocí řady podmínek používajících metodu strcmp či jednoduchý compare charakterů validovaná převedené data. Pokud jsou data nekompletní či špatná je uživatel metodou printf notifikován, proměnné buffer a bufferIndex resetovány a stav programu je navrácen na počátek hlavní nekonečné smyčky, kde je uživatel tázán na EEPROM příkaz. V případě, že jsou data v pořádku je za pomocí porovnávání podproměnné operation struktury UserData zjištěno, zdali chce uživatel data do paměti zapisovat či číst.

Pokud je zvolena operace čtení, uživatel je notifikován o probíhajícím čtení z jeho vyžadované adresy a je volána metoda readFromEEPROM, která jako návratovou hodnotu vrací přečtená data a zároveň stav správnosti vykonání indikuje nastavením proměnné error_flag na příslušnou hodnotu. Pokud je error_flag nastaven na 1, program dá uživateli vědět, že nastala chyba čtení. V případně opačném jsou uživateli navrácena data z dané adresy.

Vzhledem k architektuře paměti je nemožné číst paměť ze specifikované adresy v příkazu jen za pomocí jednoho standardizovaného „dotazu“ na paměť. Je tedy nutné před každým čtením vyvolat falešný zápis na adresu stránky, kterou chce uživatel přečíst a následně na něj bez odesílání stop bitu (metoda stopBit) navázat start bitem (startBit) a adresou zařízení uzpůsobenou pro čtení (metoda sendAddr s parametrem readWrite nastaveným na 1). Po úspěšném přijetí ack bitu (metoda readBit) nám pak EEPROM paměť při odeslání tiku hodin sama navrátí data na oné „zápisové“ adrese (čtení bytu -> metoda readByte).

Při volbě operace zápisu je opět uživatel notifikován, že probíhá zápis informace na předem zvolenou adresu a následně je volána metoda writeToEEPROM, která vrací int hodnotu značící úspěch operace. Pokud je návratová hodnota rovna 1, uživateli je oznámeno, že vše proběhlo úspěšně, pokud však jednička navrácena nebyla je vrácena zpráva o neúspěšnosti kroku.

Co se samotného zápisu do paměti pomocí metody writeToEEPROM týče, jedná se o velmi jednoduchá proces. Na začátku se pouze adresuje slave start bitem a adresou, přičemž je read/write bit nastaven na log. 0. Poté se jen rozdělí 16 bitová adresa stránky pomocí shiftu bitů na 2 osmi bitové součástí, které se odešlou za využitím metody sendByteAndAck. Pokud odeslání adres proběhne úspěšně, je odeslán i byte s daty a celý proces zakončen stop bitem.

Rozbor Proměnných a metod:

Metody:

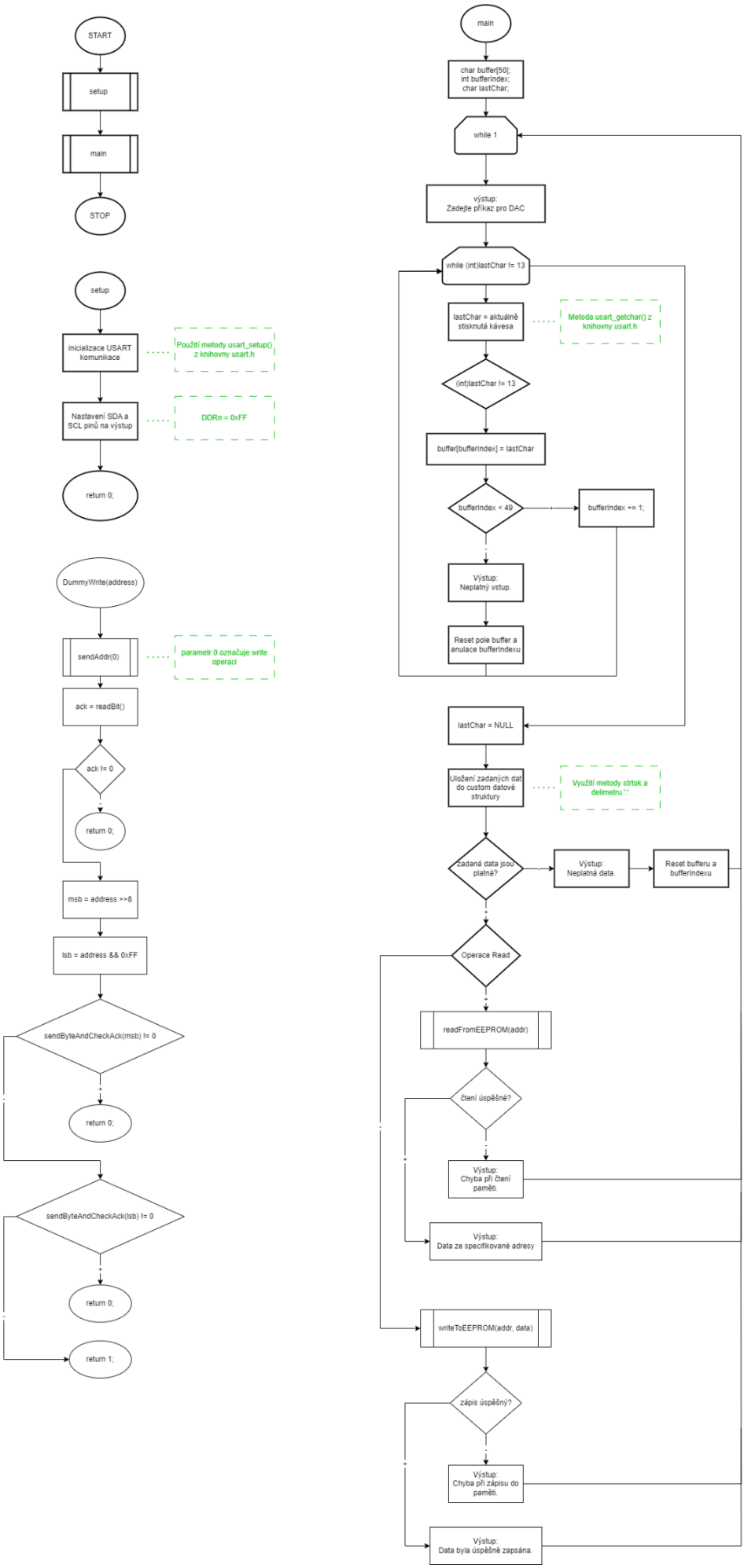
main.c			
Typ	**Metoda**	**Parametry**	**Účel**
void	startBit	void	Odeslání start bitu
void	stopBit	void	Odeslání stop bitu
void	sendBit	int bit	Odeslání jednoho bitu
void	sendByte	uint8_t byte	Odeslání jednoho bytu
int	readBit	void	Čtení jednoho bitu
void	tick	bool wait	Taktování komunikace
void	sendAddr	int readWrite	Odeslání I2C adresy s čtecím/zápisovým bitem
uint8_t	readByte	void	Čtení jednoho bytu
int	communicationStart	int readWrite	Zahájení komunikace s EEPROM
int	sendByteAndCheckAck	uint8_t byte	Odeslání bytu s ověřením ACK
int	writeToEEPROM	uint16_t address, uint8_t data	Zápis bytu do EEPROM
int	dummyWrite	uint16_t address	Simulace zápisu do EEPROM
uint8_t	readFromEEPROM	uint16_t address, int *errorFlag	Čtení bytu z EEPROM
int	parseUserInput	UserInput *result, char *data	Rozklad uživatelského vstupu
int	getCommandValidity	UserInput *input	Kontrola validity uživatelského vstupu
int	getAddr	char *data, uint16_t *saveAddr	Převod řetězcové reprezentace adresy na uint16_t

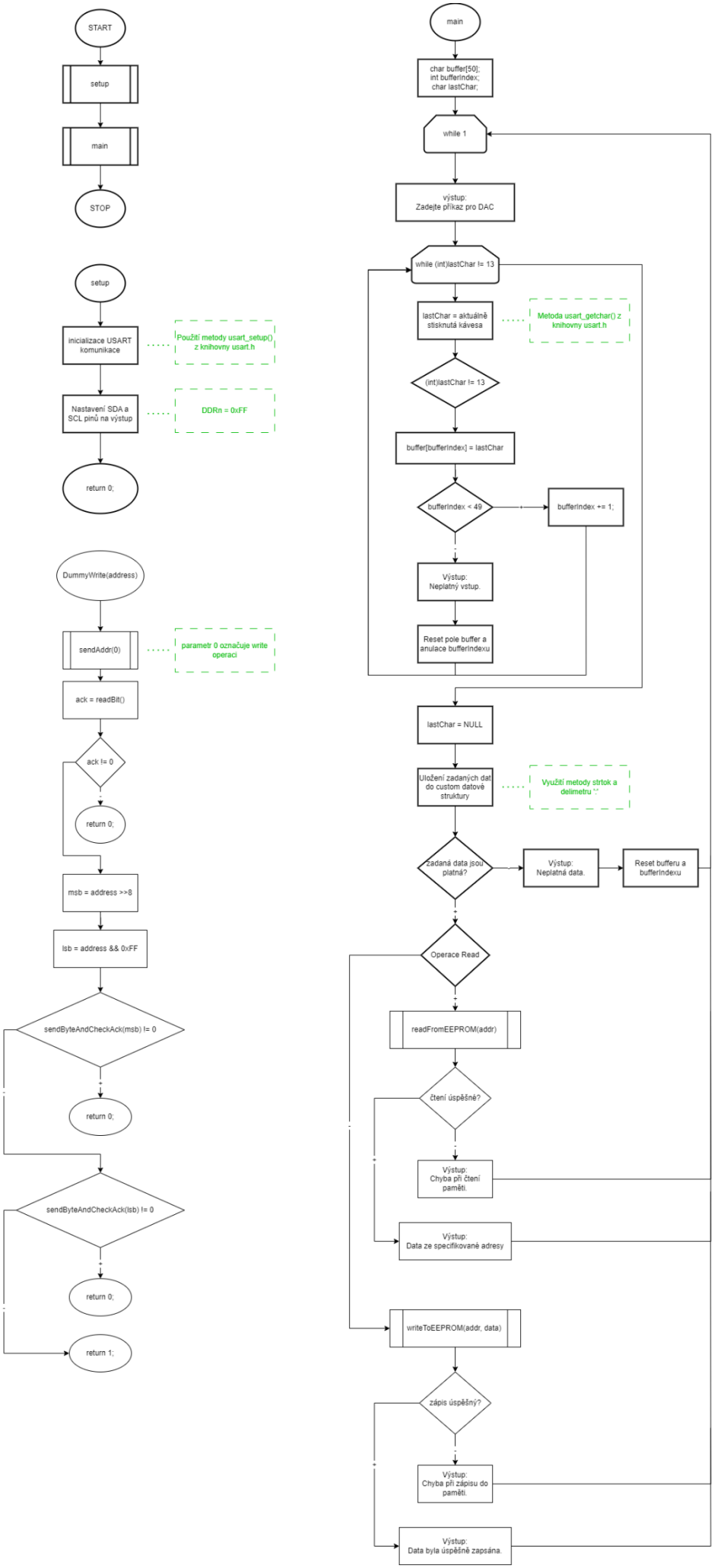
Proměnné:

main.c		
Typ	**Název**	**Účel**
UserInput	userData	Custom datová struktura pro ukládání vstupu od uživatele
char[]	buffer	Buffer pro ukládání dat od uživatele.
int	bufferIndex	Index pro práci s bufferem.
char	lastChar	Proměnná pro ukládání poslední stisklé klávesy.

config.h		
Typ	**Název**	**Účel**
define	I2C_CONF	Registr označující konfigurační registr I2C pinů
define	I2C_OUT	Registr pro výstup dat na I2C piny
define	I2C_IN	Registr pro vstup dat na I2C piny
define	SDA	Adresa SDA pinu
define	SCL	Adresa SCL pinu
define	SLAVE_ADDR	Adresa paměti EEPROM
define	I2C_DELAY	Doba zpoždění při I2C komunikaci (v mikrosekundách)
define	MIN_ADDR	Minimální adresa paměti EEPROM
define	MAX_ADDR	Maximální adresa paměti EEPROM
define	MAX_CMD_LEN	Maximální délka konzolového příkazu
define	WRITE_VAL	Logická hodnota zápisového bitu pro EEPROM
define	READ_VAL	Logická hodnota čtecího bitu pro EEPROM
define	ENTER_CODE	ASCII kód klávesy Enter
define	DEBUG_OUTPUT	Bool hodnota pro zorazení debug hlášek

Vývojový diagram:




START
setup
main
STOP
setup
inicializace USART komunikace
Použití metody usart_setup() z knihovny usart.h
Nastavení SDA a SCL pinů na výstup
DDRn = 0xFF
return 0;
DummyWrite(address)
sendAddr(0)
parametr 0 označuje write operaci
ack = readBit()
ack != 0
return 0;
msb = address >>8
lsb = address && 0xFF
sendByteAndCheckAck(msb) != 0
return 0;
sendByteAndCheckAck(lsb) != 0
return 0;
return 1;
main
char buffer[50];
int bufferIndex;
char lastChar;
while 1
výstup:
Zadejte příkaz pro DAC
while (int)lastChar != 13
lastChar = aktuálně stisknutá kávesa
Metoda usart_getchar() z knihovny usart.h
(int)lastChar != 13
buffer[bufferIndex] = lastChar
bufferIndex < 49
bufferIndex += 1;
Výstup:
Neplatný vstup.
Reset pole buffer a anulace bufferIndexu
lastChar = NULL
Uložení zadaných dat do custom datové struktury
Využití metody strtok a delimetru ';'
Zadaná data jsou platná?
Výstup:
Neplatná data.
Reset bufferu a bufferIndexu
Operace Read
readFromEEPROM(addr)
čtení úspěšné?
Výstup:
Chyba při čtení paměti.
Výstup:
Data ze specifikované adresy
writeToEEPROM(addr, data)
zápis úspěšný?
Výstup:
Chyba při zápisu do paměti.
Výstup:
Data byla úspěšně zapsána.

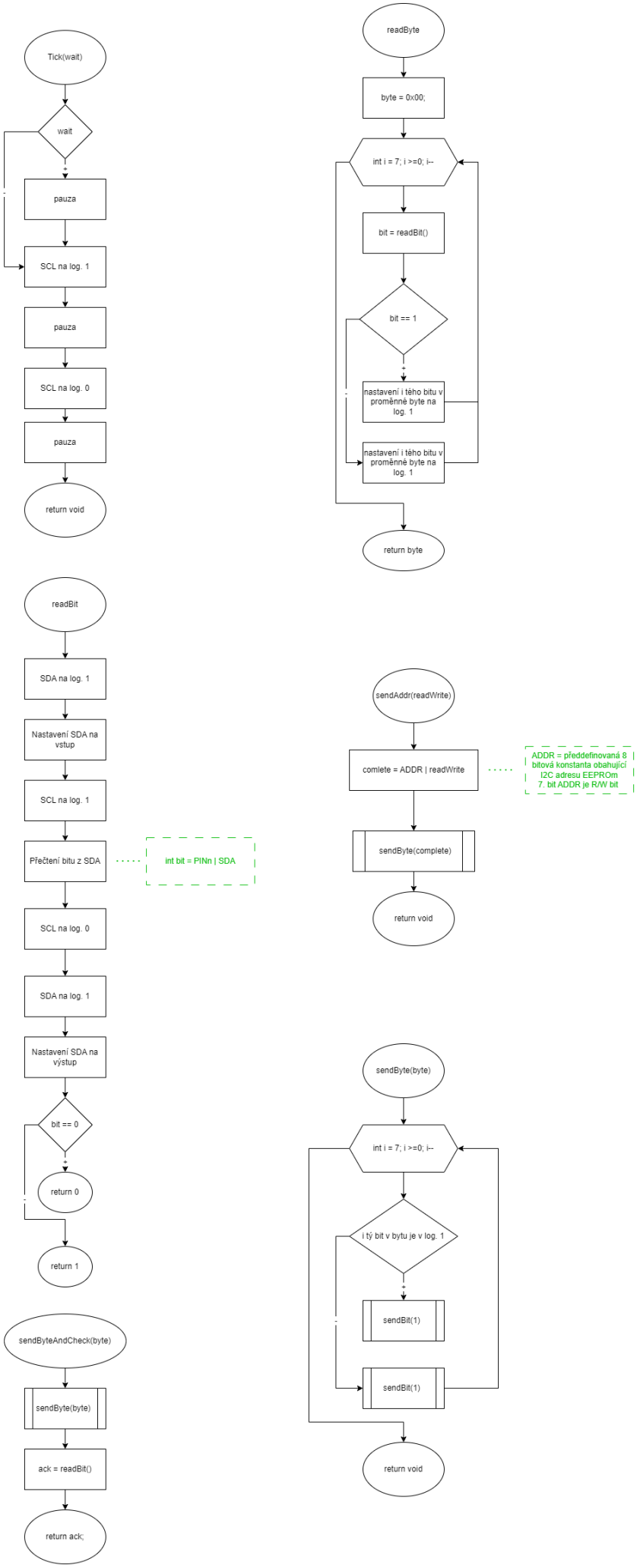
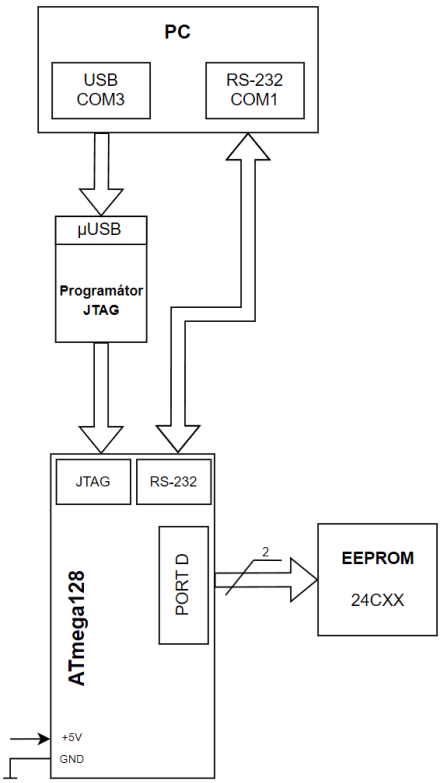

Tick(wait)
wait
+
-
pauza
SCL na log. 1
pauza
SCL na log. 0
pauza
return void
readByte
byte = 0x00;
int i = 7; i >=0; i--
bit = readBit()
bit == 1
+
-
nastavení i tého bitu v proměnné byte na log. 1
nastavení i tého bitu v proměnné byte na log. 1
return byte
readBit
SDA na log. 1
Nastavení SDA na vstup
SCL na log. 1
Přečtení bitu z SDA
int bit = PINn | SDA
SCL na log. 0
SDA na log. 1
Nastavení SDA na výstup
bit == 0
+
-
return 0
return 1
sendByteAndCheck(byte)
sendByte(byte)
ack = readBit()
return ack;
sendAddr(readWrite)
comlete = ADDR | readWrite
ADDR = předdefinovaná 8 bitová konstanta obahující I2C adresu EEPROm
7. bit ADDR je R/W bit
sendByte(complete)
return void
sendByte(byte)
int i = 7; i >=0; i--
i tý bit v bytu je v log. 1
+
-
sendBit(1)
sendBit(1)
return void

Schéma zapojení:



Komentovaný výpis programu:

main.c:

```
#include "config.h"

// custom data structure for effectively stroing user input
typedef struct
{
    char operation;
    uint16_t operand_0;
    uint16_t operand_1;
} UserInput;

// function prototypes for I2C
void startBit(void);
void stopBit(void);
void sendBit(int bit);
void sendByte(uint8_t byte);
int readBit(void);
void tick(bool wait);
void sendAddr(int readWrite);
uint8_t readByte(void);
uint8_t readFromEEPROM(uint16_t address, int *errorFlag);
int writeToEEPROM(uint16_t address, uint8_t data);
int sendByteAndCheckAck(uint8_t byte);

// function protoypes for serial comm.
int parseUserInput(UserInput *result, char *data);
int getCommandValidity(UserInput *input);
int getAddr(char *data, uint16_t *saveAddr);
```

```
// global objects
UserInput userData = {'\0', 0x0000, 0x0000};

// user buffer related variables
char buffer[MAX_CMD_LEN];
int bufferIndex = 0;
char lastChar = '\0';

/**
 * @brief Method for setting up the communication as well as the initial ppin states.
 */
void setup(void)
{
    // serial communication setup
    usart_setup();

    // setup SDA and SCL as output pins
    I2C_CONF = (SDA | SCL);
}

/**
 * @brief Main method of the program
 */
void main(void)
{
    setup();
    printf("Welcome to E2PROM interface!\n");
    printf("Each command has to have this structure: \n");
    printf("\t W ADDR DATA or R ADDR  (eg. W 0x0000 0xFF)\n");

    while (1)
    {
        printf("Enter an EEPROM command: ");

        while ((int)lastChar != ENTER_CODE)
        {
            // while not enter

            // save the currently pressed character
            lastChar = usart_getchar();

            buffer[bufferIndex] = lastChar;

            if (bufferIndex < MAX_CMD_LEN - 1)
            {
                // if the buffer index in bounds -> add
                bufferIndex++;
            }
            else
            {
                // buffer would be out of bounds -> reset buffer
                printf("Maximum command length exceeded, resetting to empty buffer
...\n");
                memset(buffer, 0, MAX_CMD_LEN);
```

```
            bufferIndex = 0;
        }
    }

    // command was submitted -> parse the buffer

    lastChar = '\0'; // anulate the last charr
    buffer[bufferIndex] = lastChar;

    int valid = parseUserInput(&userData, buffer);

    // reset the buffer array and index
    memset(buffer, 0, MAX_CMD_LEN);
    bufferIndex = 0;

    if (valid == 0)
    {
        // inpit was invalid -> show error and return to start
        printf("Invalid command. Try again...\n");
        continue;
    }
    // printf("operand 0 = %d\n", userData.operand_0);
    //  validate the input
    if (getCommandValidity(&userData) == 0)
    {
        printf("Invalid command. Try again...\n");
        continue;
    }

    // input fully valid -> check for opperation and issue the I2C commands
    if (userData.operation == 'R')
    {
        // read operation
        printf("Reading from address %x\n", userData.operand_0);
        int errorFlag = 0;
        uint8_t data = readFromEEPROM(userData.operand_0, &errorFlag);
        if (errorFlag == 1)
        {
            printf("Error occured while reading from EEPROM\n");
        }
        else
        {
            printf("\nData read: %x\n", data);
        }
    }
    else
    {
        // write operation
        printf("Writing %x to address %x\n", userData.operand_1, userData.operand_0);
        int success = writeToEEPROM(userData.operand_0, userData.operand_1);

        if (success == 0)
        {
            printf("Error occured while writing to EEPROM\n");
```

```
            }
            else
            {
                printf("Data written successfully\n");
            }
        }
    }
}

/**
 * @brief Method for writing a byte to the EEPROM
 * @param address The address to write to
 * @param data The data to write
 */
int writeToEEPROM(uint16_t address, uint8_t data)
{
    // function for writing a byte to the EEPROM
    // writing works by sending 2x 8 it address and than the data
    int ack = communicationStart(WRITE_VAL);

    if (DEBUG_OUTPUT)
    {
        printf("Write ack: %d \n", ack);
    }

    if (ack != 0)
    {
        // raise an error when ACK not received
        return 0;
    }

    // split the address into 2 8 bit numbers
    uint8_t msbAddr = (uint8_t)(address >> 8);
    uint8_t lsbAddr = (uint8_t)(address & 0xFF);

    // send the memory cell address by sending 2 bytes
    if ((sendByteAndCheckAck(msbAddr) != 0) || (sendByteAndCheckAck(lsbAddr) != 0))
    {
        return 0;
    }

    // send the actual data and send stop bit
    if (sendByteAndCheckAck(data) != 0)
    {
        return 0;
    }

    stopBit();

    // everything went well -> return 1 to indicate success
    return 1;
}

/**
```

```
 * @brief Method for sending the address user want to read to the EEPROM
 */
int dummyWrite(uint16_t address)
{
    // function for writing a byte to the EEPROM
    // writing works by sending 2x 8 it address and than the data
    int ack = communicationStart(WRITE_VAL);

    if (ack != 0)
    {
        // raise an error when ACK not received
        return 0;
    }

    // split the address into 2 8 bit numbers
    uint8_t msbAddr = (uint8_t)(address >> 8);
    uint8_t lsbAddr = (uint8_t)(address & 0xFF);

    // send the memory cell address by sending 2 bytes
    if ((sendByteAndCheckAck(msbAddr) != 0) || (sendByteAndCheckAck(lsbAddr) != 0))
    {
        return 0;
    }
    // everything went well -> return 1 to indicate success
    return 1;
}

/**
 * @brief Method for reading a byte from the EEPROM
 * @param address The address to read from
 * @param errorFlag Pointer to the error flag variable
 */
uint8_t readFromEEPROM(uint16_t address, int *errorFlag)
{
    dummyWrite(address);

    startBit();
    sendAddr(READ_VAL);
    int ack = readBit();
    if (ack != 0)
    {
        // set an error flag to true
        *errorFlag = 1;
    }

    uint8_t data = readByte();

    stopBit();

    return data;
}

/**
 * @brief Method for sending a start bit to the slave
```

```
 */
void startBit(void)
{
    // function for sending the start bit condition to the I2C slave

    // SDA and SCL on
    I2C_OUT = SDA | SCL;
    // wait a bit
    _delay_us(I2C_DELAY);
    // SDA off
    I2C_OUT &= ~(SDA);
    // delay
    _delay_us(I2C_DELAY);
    // SCL off
    I2C_OUT &= ~SCL;
    // wait
    _delay_us(I2C_DELAY);
    printf("|START");
}

/**
 * @brief Method for sending a stop bit to the slave
 */
void stopBit(void)
{
    // function for sending the stop bit condition to the I2C slave

    // SDA off and SCL off
    I2C_OUT &= ~(SCL | SDA);
    // wait a bit
    _delay_us(I2C_DELAY);
    // SCL ON
    I2C_OUT |= SCL;
    // delay
    _delay_us(I2C_DELAY);
    // SDA ON
    I2C_OUT |= SDA;
    // delay
    _delay_ms(I2C_DELAY);
    printf("STOP|");
}

/**
 * @brief Method for starting the communication with the EEPROM
 * @param readWrite The read/write bit to be sent
 * @return Int statig whether or not the mcu received an ACK bit
 */
int communicationStart(int readWrite)
{
    startBit();
    sendAddr(readWrite);
    int ack = readBit();
    return ack;
}
```

```
/**
 * @brief Method for sending an address and the rad write bit
 * @param readWrite The read/write bit to be sent
 */
void sendAddr(int readWrite)
{
    // function for sending an address and the rad write bit
    uint8_t completeByte = SLAVE_ADDR | readWrite;
    sendByte(completeByte);
}

/**
 * @brief Method for sending one bit of DATA on the I2C line
 * @param bit The bit to be sent on the bus
 */
void sendBit(int bit)
{
    printf("%d", bit);
    if (bit == 0)
    {
        I2C_OUT &= ~SDA;
    }
    else
    {
        I2C_OUT |= SDA;
    }

    // wait a little and tick
    tick(true);
}

/**
 * @brief Method for sending one byte to the I2C bus
 * @param byte The actual data to send
 * @param MSBFirst Determines whether the most significant bit should go  firs
 */
void sendByte(uint8_t byte)
{
    printf("|");
    for (int i = 7; i >= 0; i--)
    {
        if (isHigh(byte, i) == 1)
        {
            // send log. 1
            sendBit(1);
        }
        else
        {
            // send log. 0
            sendBit(0);
        }
    }
    printf("|");
```

```
}

/**
 * @brief Method for sending one byte to the I2C bus and checking for ACK
 * @param byte The actual data to send
 * @param MSBFirst Determines whether the most significant bit should go  firs
 * @return Int statig whether or not the mcu received an ACK bit
 */
int sendByteAndCheckAck(uint8_t byte)
{
    sendByte(byte);
    int ack = readBit();
    if (DEBUG_OUTPUT)
    {
        printf("Send byte and check ack: %d \n", ack);
    }
    return ack;
}

/**
 * @brief Method for reading a bit from the I2C bus
 */
int readBit(void)
{
    // SDA high
    I2C_OUT |= SDA;

    // I2c_conf SDA pin TO INPUT ->* SDA bit set to zero
    I2C_CONF &= ~(SDA);

    // clock tick -> keep the clock high and read the data
    I2C_OUT |= (SCL);
    _delay_us(1);

    // read data while the clock is high
    int bit = (I2C_IN & SDA);

    // set the clock down
    I2C_OUT &= ~(SCL);

    // set the original SDA state
    I2C_OUT |= SDA;

    I2C_CONF |= (SDA);

    printf("%d", bit);

    if (bit == 0)
    {
        return 0;
    }

    return 1;
}
```

```
uint8_t readByte()
{
    // function for reading a byte from the I2C bus
    uint8_t byte = 0x00;
    printf("|[");
    for (int i = 7; i >= 0; i--)
    {
        // read the bit
        int bit = readBit();
        if (bit != 0)
        {
            byte = setBit(byte, i);
        }
        else
        {
            byte = nullBit(byte, i);
        }
    }
    printf("]");
    printf("|");
    return byte;
}

/**
 * @brief Method for ticking with the clock
 * @param wait Parameter determining if the method waits for the previsously set signal to
stabilise
 */
void tick(bool wait)
{
    if (wait)
    {
        // wait for the previously set signal to stabilise on the pin
        _delay_ms(I2C_DELAY / 2);
    }

    I2C_OUT |= SCL;
    _delay_us(I2C_DELAY);
    I2C_OUT &= ~(SCL);
    _delay_us(I2C_DELAY);
}

/**
 * @brief Parse user input and return a UserInput struct with the appropriate data.
 * @param result pointer to the global UserInput variable
 * @param data User input string.
 */
int parseUserInput(UserInput *result, char *data)
{
    // reset the current userInput values
    result->operation = '\0';                   // Initialize to a default char value
    result->operand_0 = result->operand_1 = 0; // Initialize to a default integer value
```

```
    int splitID = 0;

    char *token = strtok(data, " ");

    while (token != NULL)
    {
        // printf("Token %d: %s\n", splitID, token ? token : "NULL");
        if (splitID == 0)
        {
            // either R or W
            if ((token[0] == 'R' || token[0] == 'W') && token[1] == '\0')
            {
                // printf("Assigning operation: %s\n", token);
                result->operation = token[0];

                if (DEBUG_OUTPUT)
                {
                    printf("Assigned operation character %c\n'", result->operation);
                }
            }
            else
            {
                // printf("Invalid operation\n");
                return 0;
            }
        }
        else
        {
            uint16_t addr;
            if (getAddr(token, &addr) != 0)
            {
                // conversion occured successfully -> save
                if ((splitID == 1))
                {
                    result->operand_0 = addr;
                }
                else if ((splitID == 2))
                {
                    result->operand_1 = addr;
                }
                else
                {
                    // invalid address length
                    return 0;
                }
            }
            else
            {
                // error occured -> return invalid flag
                if (DEBUG_OUTPUT)
                {
                    printf("addr error\n");
                }
                return 0;
```

```
            }
        }
        token = strtok(NULL, " ");
        splitID++;
    }

    // everything went smoothly -> success flag
    return 1;
}

/**
 * @brief CHeck the validity of user data before sending
 * @param input pointer to the UserInput datatype storing all input related vars
 * @return int assessing the validity of the user input
 */
int getCommandValidity(UserInput *input)
{
    if (input->operand_0 < MIN_ADDR || input->operand_0 > MAX_ADDR)
    {
        if (DEBUG_OUTPUT)
        {
            printf("base case 1\n");
        }
        return 0;
    }

    // more compley checks
    if (input->operation == 'R')
    {
        // read operation issued
        return 1;
    }

    if (input->operation == 'W')
    {
        // write operation issued
        if (input->operand_1 != NULL)
        {
            if (MIN_ADDR <= input->operand_1 && input->operand_1 <= MAX_ADDR)
            {
                return 1;
            }
        }
    }
    return 0;
}

/**
 * @brief Convert the string representation of 16 bit addr. to uint_8.
 * @param data String representation of the address
 * @param saveAddr pointer to the final variable to save the data to
 * @return int stating the status of conversion
 */
int getAddr(char *data, uint16_t *saveAddr)
```

```
{
    if (data == NULL)
    {
        return 0;
    }

    // convert a char* to a 16 base number

    char *ptr;
    unsigned long ret;

    ret = strtoul(data, &ptr, 16);

    // convert the data back to unsigned 16 bit int
    uint16_t result = (uint16_t)ret;
    // save data to the final specified destination
    *saveAddr = result;
    return 1;
}
```

config.h:

```
#ifndef CONFIG_H
#define CONFIG_H

// Include necessary libraries
#include <stdio.h>
#include <math.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

// include custom liraries
#include "usart.h"
#include "bin_lib.h"

// global registers for DDR, PORT, PIN
#define I2C_CONF (DDRD)
#define I2C_OUT (PORTD)
#define I2C_IN (PIND)

// Global pin setup
#define SDA 0x01
#define SCL 0x02

// Global variable setup
#define SLAVE_ADDR 0b10100000 // EEPROM memory address
#define I2C_DELAY 5           // delay of 5 microseconds
#define MIN_ADDR 0x0000       // minimum EEPROM memory address
#define MAX_ADDR 0xFFFF       // maximum EEPROM memory address
```

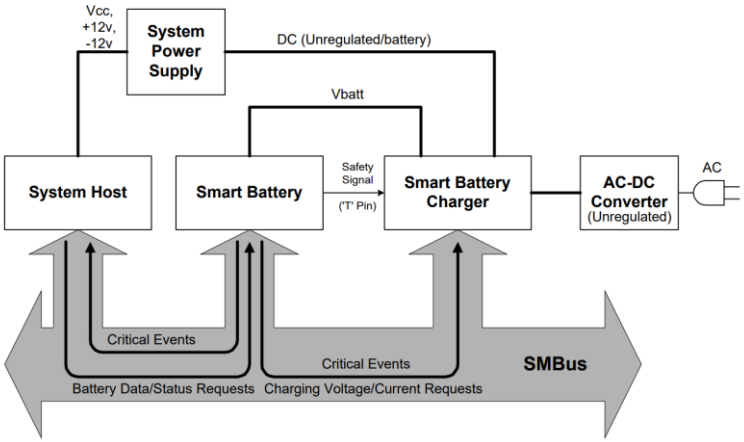
```
#define MAX_CMD_LEN 50        // maximum length of command
#define WRITE_VAL 0           // the log. value of write bit for the EEPROM
#define READ_VAL 1            // the log. value of read bit for the EEPROM
#define ENTER_CODE 13         // ASCII code for enter key
#define DEBUG_OUTPUT false

#endif // CONFIG_H
```

Odpovědi na otázky:

1. I2C rozhraní v upravené podobě je použito také ve standardu SMBus v každém počítači. Dohledejte si stručnou specifikaci tohoto rozhraní a popište jednu konkrétní oblast (zodpovědnost), kterou tento standard definuje.

   Konkrétní oblastní, která je definována standardem SMBus jsou chytré bateriové systémy, které byli primárním katalyzátorem vzniku tohoto rozhraní. Původní motivací k implementaci rozhraní sloužícího ke komunikaci komponent spravující baterie zařízení byl požadavek na real time hodnoty kapacity baterie, odhadovanou dobu nabíjení a popřípadě i nabíjecí výkon.

   Zařízení obsáhlá v tomto smart battery systému jsou: system host (počítač, notebook), chytrá baterie a chytrá nabíječka baterií.



[2]

   Komunikace probíhá pomocí zpráv, které jsou posílány mezi masterem a slavy. Tyto zprávy mohou obsahovat požadavky na data, jako je aktuální stav baterie, požadovaný nabíjecí proud nebo informace o stavu nabíjení. Slavy na tyto zprávy odpovídají posíláním relevantních dat zpět masteru.

   Díky standardizovanému rozhraní SMBus je možné jednoduše integrovat chytré bateriové systémy do různých zařízení a zajišťovat tak efektivní správu a monitorování baterií. Tato technologie přináší uživatelům výhody v podobě delší životnosti baterií, lepšího výkonu a optimalizace nabíjecích procesů.

2. V případě použití I2C rozhraní pro monitorování teploty (např. obvodem LM75) je pro reakci na kritické stavy (např. přehřátí systému) periodické čtení a vyhodnocení aktuální teploty neefektivní. Pokuste se navrhnout vhodnější řešení.

   Nejvhodnějším řešením tohoto problému je na teplotním čidle nastavit hodnotu Temperature Overlimit Shutdown na jeho konfiguračním registru na teplotu, po jejímž přesažení bude vykonána příslušná akce k zchlazení systému. V takovém případě je pak třeba jen periodicky nahlížet do stavového registru dané součástky a kontrolovat hodnotu uloženou pod názvem Overtemperature Shutdown. Vyhodnocení přehřátí je pak mnohem rychlejší a v ušetřeném mezičase (oproti konstantnímu čtení aktuální teploty) může mikrokontroler komunikovat s ostatními zařízeními.[3]

[2] http://sbs-forum.org/specs/sbdat110.pdf

[3] https://www.ti.com/lit/ds/symlink/lm75b.pdf?ts=1706252688004&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FLM75B%252Fpart-details%252FLM75BIMX-3%252FNOPB

3. Každá buňka EEPROM paměti má omezený počet zápisů před vyčerpáním spolehlivosti. Pokuste se navrhnout jak byste tento limit obešli např. při implementaci počítadla ujetých kilometrů v automobilu (uvažujte řádově miliony zápisů).

   Možné řešení tohoto problému je například implementovat systém, kde bude paměť typu EEPROM sloužit pouze jako „spádová“ paměť. To znamená, že je jí předsazená paměť RAM, kam se průběžně ukládací informace o ujeté vzdálenosti a v moment, kdy je jízda vyhodnocena jako ukončená se počet kilometrů zapíše do nevolatilní paměti EEPROM. Tento systém tak po ujetí 100 km vzdálenosti zapíše do EEPROM paměti pouze jednou, oproti minimálně 100 zápisům bez využití tohoto systému.

Závěr:

Výsledkem řešení této úlohy je plně funkční implementace protokolu I2C za užití metody bit-bangingu. Krom vyřešení základních operací na sběrnici obsahuje výsledné řešení i interface pro komunikaci s oblohou, který je řešen skrz sériovou komunikaci a obsahuje základní validační pravidla, tudíž by teoreticky mohl být využit v praktickém prostředí.