PRAXE

Datum: 28.11. 2023	Střední průmyslová škola, Chomutov, Školní 50, příspěvková organizace	Třída: V4
Číslo úlohy: 2.	Generátor průběhů	Jméno: Tomáš Bartoš

Zadání:

S využitím vývojového kitu AVR a dvou paralelních DAC převodníků sestavte v jazyku C program fungující jako jednoduchý dvoukanálový funkční generátor ovládaný ze sériové linky. Ověření parametrů generovaného průběhu provádějte osciloskopem.

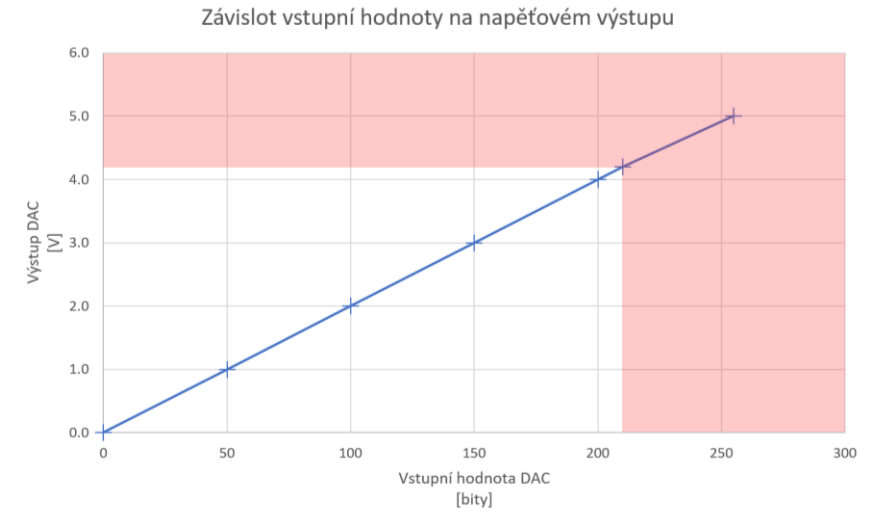
Výsledné řešení musí obsahovat následující funkcionalitu:

- Nezávislé nastavení parametrů pro každý kanál v reálných jednotkách (Hz, mV)
- Podpora tří různých průběhů na kanál. Vyžadován sinus a dva další dle vlastního výběru
- Generování průběhu v reálném čase pro oba kanály současně
- Nastavení parametrů průběhu pomocí textových příkazů z terminálu připojeného sériovou linkou
- (např. CH1:TYPE:SIN, CH2:FREQ:12, CH1:AMP:1500)

Teorie:

Řešení úlohy generátoru dvou na sobě nezávislých průběhů za využití paralelních DAC převodníků a mikrokontroleru ATmega128 vyžaduje znalost principu generování signálu na DAC převodnících a znalost způsobu programování a interních funkcí použitého mikrokontroleru.

Použité DAC převodníky fungují tak, že pro výstup analogové informace (napětí) na výstup je třeba přivést na jejich datovou vstupní sběrnici určitou hodnotu. Daná hodnota je díky 8 bitovému rozlišení převodníku dána v rozsahu od 0 do 255 a je ze stejného důvodu přiváděna za pomocí 8 vodičů. Tato hodnota je následně interní logikou vyhodnocena a přeložena na analogové výstupní napětí, které je pomocí vodičů dvou vyvedena na výstup převodníku. Důležité je však podotknout, že ač je DAC převodník určený například pro 5V, po přivedení čísla 255 na paralelní vstup bude výstupní napětí dosahovat velikosti okolo 4,2V. Daný pokles napětí je daný stavem saturace, který nastane při požadavku stejného napětí, jaké bere DAC převodník za referenční. Převodníky jsou tudíž omezeny na generální signálu o maximální amplitudě 4,2V (4200mV).



Co se hardwaru samotného mikrokontroleru týče, 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 generovat signál za pomocí výstupu na příslušné porty, 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.

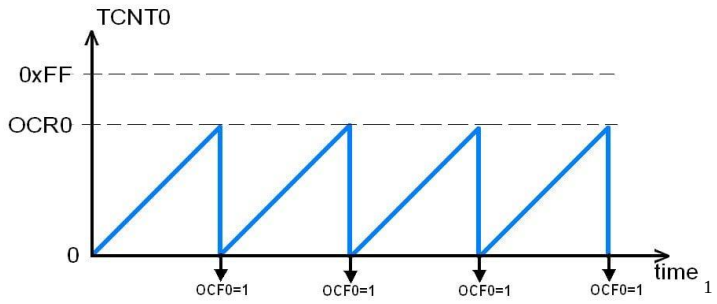
Softwarová stránka implementace řešení běžící na mikrokontroleru ATmega128 je v základu postavena na práci s těmito třemi registry:

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

Nadále je v řešení implementovaný čítač/časovač v režimu CTC, který funguje na principu nastavení maximální hodnoty tiků, po jejichž dosažení se jejich hodnota anuluje a přičítání pokračuje znovu. Způsob konfigurace pro konkrétní řešení je vypsán níže. V konfiguraci je zvolena předdělička o velikosti 1, což znamená že maximální hodnota odpovídá počtu tiků při vykonávání obsluhy přerušení a maximální velikost je díky zpětné kompatibilitě s 8 bitovými čítači nastavena na 255 (2**8).

```
TCCR1B = (1 << WGM12) | (1 << CS10); // CTC mode and prescaler equal to 1
TCNT1 = 0; // reset of the timer
OCR1A = 255;  // Set TOP value for CTC mode
TIMSK |= (1 << OCIE1A);  // Enable Timer 1 interrupt on Compare Match A
```

Diagram průběhu CTC čítače:



[1]

Popis programu:

Fáze inicializace a generování signálu:

Ihned po překladu a nahrání programu se krom importu interních knihoven a knihovny pro komunikci po sériové lince nadefinují důležité globální proměnné uchovávající nejdůležitější parametry generátoru, jakýmu jsou:

- Kanálové porty pro konfiguraci a výstup
- Maximální dovolená amplituda v mV
- Maximální dovolená amplituda v bitech vyjádřena decimálně
- Počet vzorků na signál
- Maximální délka uživatelského vstupu

Krom těchto globálních konfiguračních konstant jsou pak dále v programu velmi důležité datové struktury, jež zjednodušují celkový přístup k datům. První z daných struktur je: UserInput sloužící pro uchovávání parametrů zadaných uživatelem. Z důvodu možnosti uchování pouze pro mikrokontroler nedešifrovatelných dat je tu i

[1] https://www.electronicwings.com/storage/PlatformSection/TopicContent/56/description/ttimer%20compare%20mode.png

struktura druhá s názvem Channel. Struktura Channel je díky svým exaktně definovaným podproměnným přímo určena k manipulaci s parametry výstupního signálu, a tudíž je z globálního pohledu konečným článkem řetězce převodu uživatelského vstupu na reálnou akci mikrokontroleru.

Po zpracování všech deklarací proměnných procesor pokračuje vykonáním metody setup. Ta ihned po spuštění volá metodu uasrt_setup z knihovny usart.c tak, aby se inicializovala komunikace jak po softwarové, tak po hardwarové stránce. Následně jsou nastaveny oba porty kanálů na výstupní pomocí registru DDR a je pomocí interních registrů nakonfigurován interní čítač/časovač. Konfigurace čítače a časovače je nastavena tak, aby pracoval v režimu CTC a vykonával se každých 255 tiků procesoru. Tento čítač je pak centrální částí celého procesu generování, jelikož nezávisle na hlavní smyčce odpočítává počet tiků procesoru a generuje si přerušení, jehož obslužná funkce pak obstarává funkce spojené s výstupem signálu na kanál.

Kvůli rychlému generování signálů jsou následně naplněny statická pole s hodnotami pro průběh sinus a úsečku, ovšem ke kompletnosti inicializace je dále nutné spočítat modulo hodnotu pro čítač/časovač a povolit globální přerušení pomocí metody sei.

Obsluha přerušení pak funguje na tom principu, že je při každém vyvolání inkrementována hodnota proměnné označená jako internalCounter pro každý kanál a následně je srovnávána s před vypočítaným počtem přerušení k výstupu signálu o určité frekvenci, který je vykonáván metodou getModulo. Jakmile se „modulo“ shoduje s počtem přerušení je proměnná internalCounter anulována a volá se metoda CHnOutputFunction, mající na starosti výpočet a výstup nutné decimální hodnoty pro aktuální čas a aktuálně zvolenou funkci. Výstupní signál je v případě sinu a lineárního průběhu generován pomocí před vypočítaných tabulek a indexu outputCounter, jež se vždy přičítá po provedení metody CHnOutputFunction. Pro průběh obdélníkový jsou použity dvě podmínky a outputCounter, přičemž je na výstup přivedena hodnota amplitudy, pokud je outputCounter menší než polovina počtu vzorků a nula, pokud větší.

Hlavní smyčka programu:

Po vykonání metody inicializace se běh programu přesouvá do nekonečné smyčky, která slouží jakožto uživatelský interface pro ovládání jinak nezávislého generátoru signálů. Prvním krokem pro zadávání příkazu generátoru je prompt posílaný po sériové lince pomocí metody printf a argumentu „Enter DAC 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 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 DAC příkaz. V případě, že jsou data v pořádku je metodou strcmp zjištěno jakého kanálu se zadané parametry týkají a na základě toho je vyvolána metoda setChannelVariables, jež převede char * data na čísla.

Posledním krokem je přepočet modulo hodnoty pro daný kanál za využití metody getModulo a reset proměnných buffer a buffeerIndex.

Rozbor Proměnných a metod:

Metody:

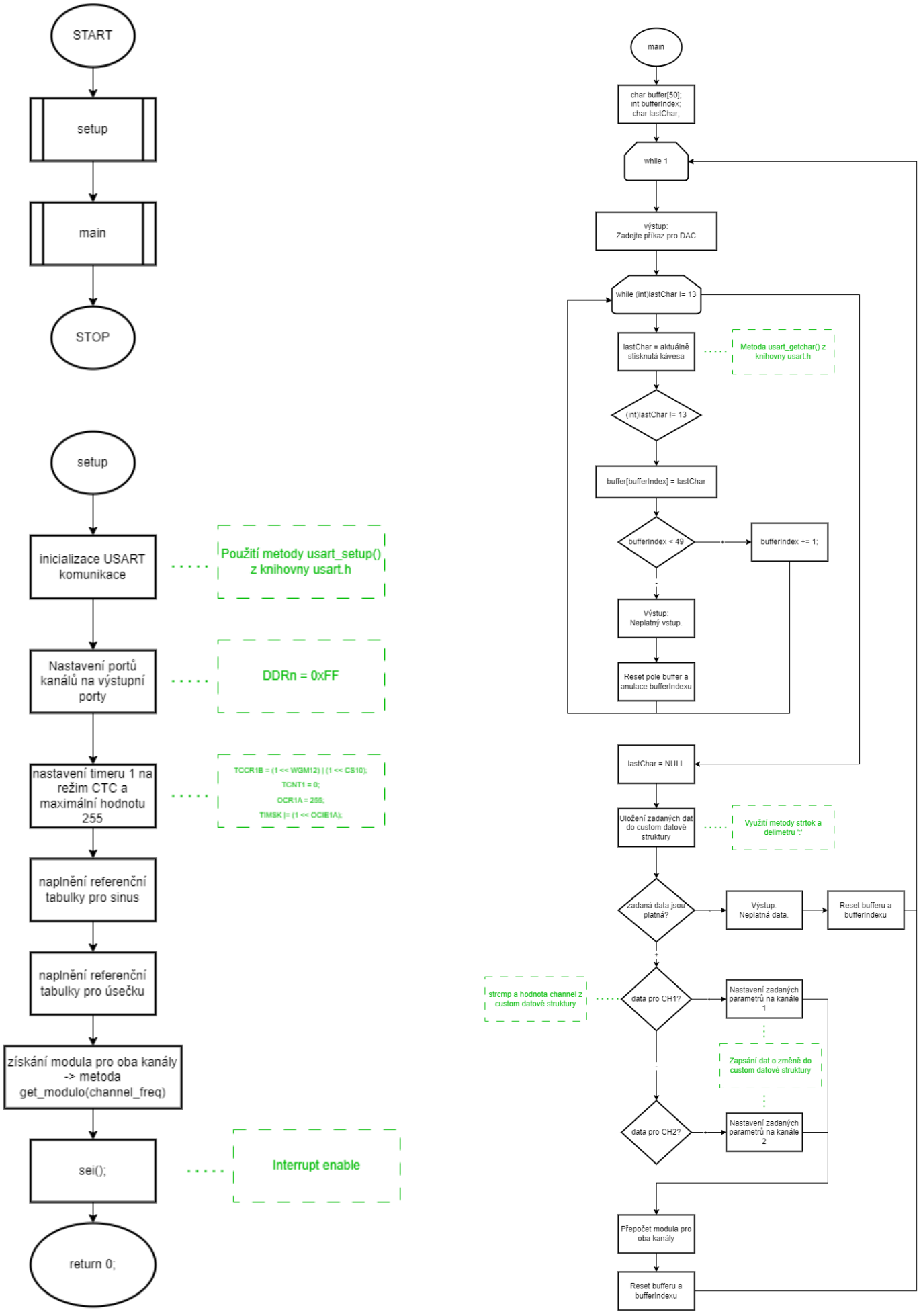
main.c			
Typ	Metoda	Parametry	Účel
void	setup		Nastavení výstupních portů a komunikace.
int	main		Hlavní smyčka programu.
void	setChannelVariables	Channel *channel, UserInput input	Metoda pro uložení dat o kanálu do custom struktury.
UserInput	parseUserInput	char *data	Metoda pro parsování vstupu od uživatele.
bool	GetCommandValidity	UserInput *input	Metoda rpo kontrulu uživatelského vstupu.
void	CH1OutputFunction		Metoda pro výstup příslušné informace na kanál 1.
void	CH2OutputFunction		Metoda pro výstup příslušné informace na kanál 2.
void	fillSineTable		Metoda pro naplnění tabulky pro sinus průběh.
void	fillLinearTable		Metoda pro naplnění tabulky pro linerní průběh.
uint32_t	GetModulo	uint32_t frequency	Metoda pro získání počtu pulzů mezi jednotlivými výstupy na převodník.
void	outputToDAC	volatile uint8_t *port, uint8_t value	Metoda pro zapsání informace na přílušný port.
int	convertAmplitude	int oldAmplitude	Metoda pro řevod amplitudy z mV na číslo od 0 do 210.

Proměnné:

main.c		
Typ	Název	Účel
struct	UserInput	Uchovává vstupní data od uživatele.
struct	Channel	Spravuje hodnoty související s kanály.
float[]	sineTable	Obsahuje předpočítané hodnoty sinové funkce.
float[]	lineTable	Obsahuje předpočítané hodnoty lineární funkce.
char[]	buffer	Ukládá aktuální vstup od uživatele.
int	bufferIndex	Udržuje aktuální index v bufferu.
char	lastChar	Ukládá poslední načtený znak ze vstupu.
struct	CH1	Reprezentuje hodnoty kanálu 1.
struct	CH2	Reprezentuje hodnoty kanálu 2.
struct	userData	Obsahuje aktuální vstupní data od uživatele.

config.h		
Typ	Název	Účel
define	CH1_CONF	Konfigurační registr pro výstupní port CH1.
define	CH2_CONF	Konfigurační registr pro výstupní port CH2.
define	CH1_OUT	Výstupní registr pro port CH1.
define	CH2_OUT	Výstupní registr pro port CH2.
define	DIGITAL_MAX_AMP	Maximální amplituda v digitálním rozsahu.
define	ANALOG_MAX_AMP	Maximální amplituda v analogovém rozsahu.
define	SAMPLING_RATE	Frekvence vzorkování.
define	MAX_CMD_LEN	Maximální délka příkazu od uživatele.

Vývojový diagram:



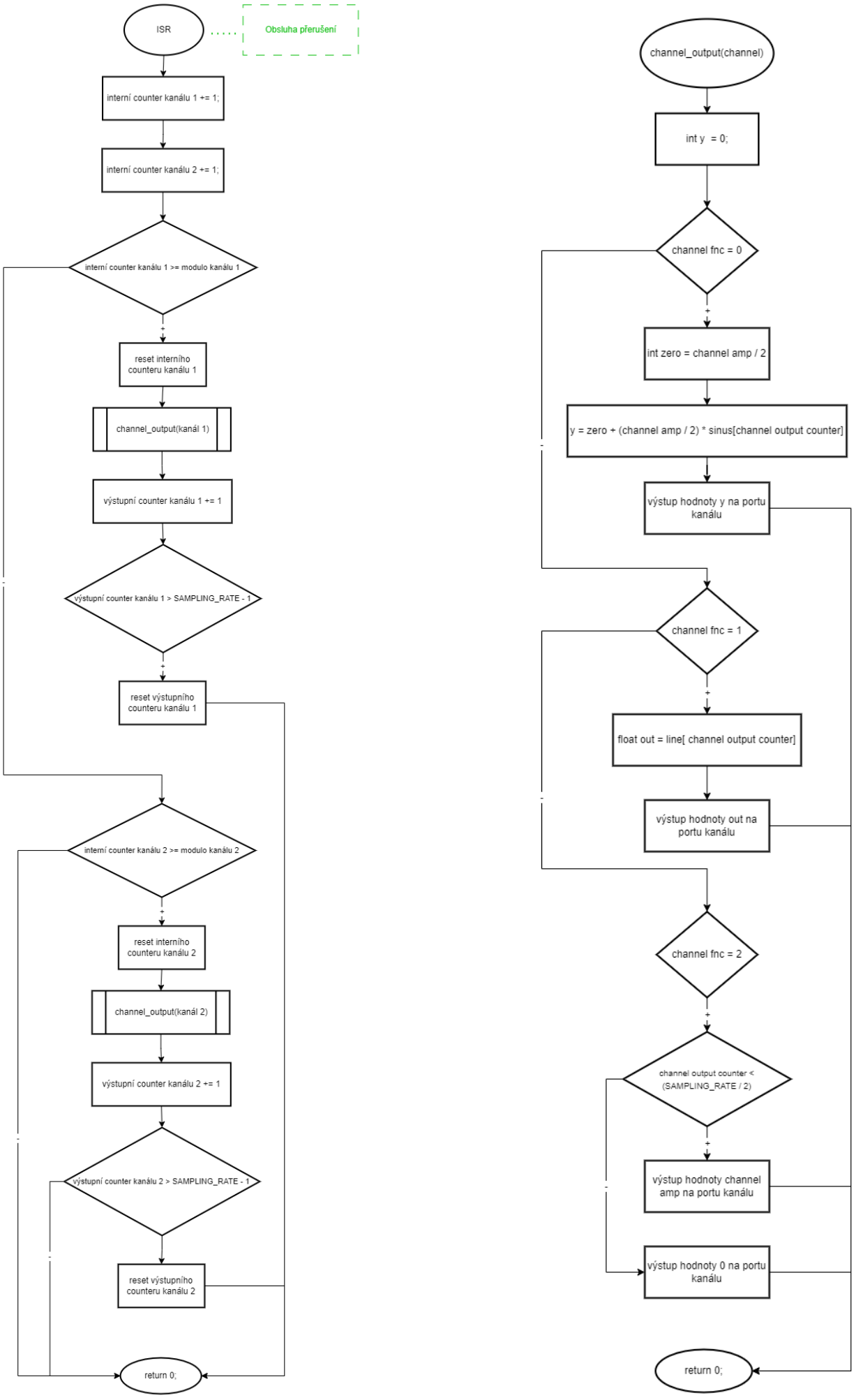
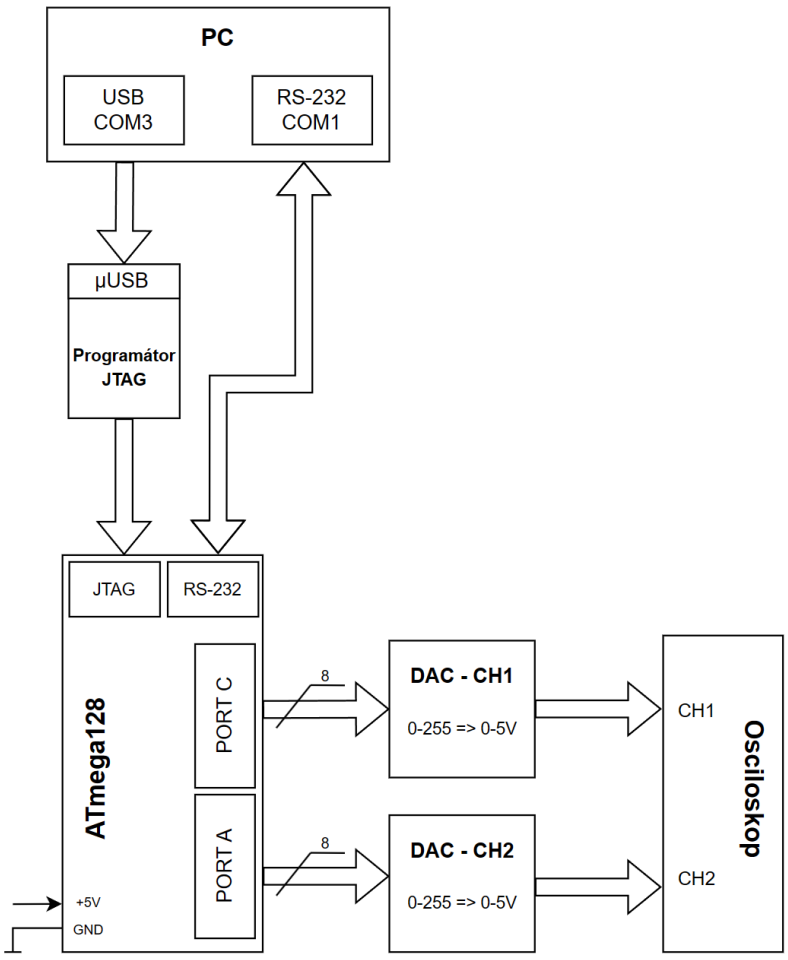

ISR
Obsluha přerušení
interní counter kanálu 1 += 1;
interní counter kanálu 2 += 1;
interní counter kanálu 1 >= modulo kanálu 1
reset interního counteru kanálu 1
channel_output(kanál 1)
výstupní counter kanálu 1 += 1
výstupní counter kanálu 1 > SAMPLING_RATE - 1
reset výstupního counteru kanálu 1
interní counter kanálu 2 >= modulo kanálu 2
reset interního counteru kanálu 2
channel_output(kanál 2)
výstupní counter kanálu 2 += 1
výstupní counter kanálu 2 > SAMPLING_RATE - 1
reset výstupního counteru kanálu 2
return 0;
channel_output(channel)
int y = 0;
channel fnc = 0
int zero = channel amp / 2
y = zero + (channel amp / 2) * sinus[channel output counter]
výstup hodnoty y na portu kanálu
channel fnc = 1
float out = line[channel output counter]
výstup hodnoty out na portu kanálu
channel fnc = 2
channel output counter < (SAMPLING_RATE / 2)
výstup hodnoty channel amp na portu kanálu
výstup hodnoty 0 na portu kanálu
return 0;

Schéma zapojení:



Komentovaný výpis program:

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"

// Global register setup
#define CH1_CONF (DDRC)
#define CH2_CONF (DDRA)

#define CH1_OUT (PORTC)
#define CH2_OUT (PORTA)


// Global variable setup
#define DIGITAL_MAX_AMP 210
#define ANALOG_MAX_AMP 4200
#define SAMPLING_RATE 360 // always an even number
#define MAX_CMD_LEN 50

#endif // CONFIG_H
```

main.c:

```
/**
 * @file main.c
 * @brief Main program file for the signal generation and communication.
 */

#include "config.h"

// custom struct for better representation of user data
typedef struct {
    char *channel;
    char *parameter;
    char *value;
} UserInput;

// custom struct for better management of the channel related values
typedef struct {
    int function;
    int frequency;
```

```
    int amplitude;
    uint32_t modulo;
    volatile int internalCounter;
    volatile int outputCounter;
} Channel;

// function prototypes
void setChannelVariables(Channel *channel, UserInput parsedInput);
void parseUserInput(UserInput *result, char* data);
bool GetCommandValidity(UserInput *input);
void CH1OutputFunction(void);
void CH2OutputFunction(void);
uint32_t GetModulo(uint32_t frequency);
void outputToDAC(volatile uint8_t *port, uint8_t value);
void fillSineTable();
void fillLinearTable();

// Output function predefined arrays

float sineTable[SAMPLING_RATE];
float lineTable[SAMPLING_RATE];
//float roofTable[SAMPLING_RATE];

// blank representation of the channels
//            fnc   freq      amp    mod   iCntr  oCntr
Channel CH1 = {0,   1,        210,    0,    0,       0};
Channel CH2 = {0,   0,        210,    0,    0,       0};


// blank data for the UserInput
UserInput userData = {"", "", ""};

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


/**
 * @brief Set up the basic port and communication.
 * This function initializes the basic port and communication settings.
 */
void setup(void)
{
    // Setup the communication protocol.
    usart_setup();

    CH1_CONF = 0xFF; // Setup of channel 1.
    CH2_CONF = 0xFF; // Setup of channel 2.

    // Configure Timer 1 in CTC mode with the prescaler = 1
    TCCR1B = (1 << WGM12) | (1 << CS10); // CTC mode and prescaler equal to 1
    TCNT1 = 0; // reset of the timer
    OCR1A = 255;  // Set TOP value for CTC mode
```

```
    TIMSK |= (1 << OCIE1A);  // Enable Timer 1 interrupt on Compare Match A


    // fill in the function tables
    fillSineTable();
    fillLinearTable();

    // calulate the initial modulos
    CH1.modulo = GetModulo(CH1.frequency);
    CH2.modulo = GetModulo(CH2.frequency);

    // Enable global interrupts
    sei();

}
/**
 * @brief The main entry point of the program.
 * @return The program exit status.
 */
int main(void)
{
    // Initialization part of the program.
    setup();

    printf("Welcome to a DigiWave generator!\n");
    printf("Each command for the DAC has to have this structure: \n");
    printf("\tCHANNEL:PARAMETER:VALUE (eg. CH1:TYPE:SIN)\n");

    // The main loop.
    while (1)
    {
        printf("Enter DAC command: ");

        while((int)lastChar != 13)
        {
            lastChar = usart_getchar();

            if(lastChar != 13)
            {
                buffer[bufferIndex] = lastChar;

                if(bufferIndex < MAX_CMD_LEN - 1)
                {
                    bufferIndex++;
                }
                else
                {
                    printf("Maximum command length exceeded, reseting to empty buffer
...\n");
                    memset(buffer, 0, MAX_CMD_LEN);
                    bufferIndex = 0;
                }
            }
        }
```

```
        lastChar = NULL;

        // enter pressed
        parseUserInput(&userData, buffer);

        if(!GetCommandValidity(&userData))
        {
            // invalid command -> clear the buffer
            printf("Invalid command entered. Try again.\n");
            memset(buffer, 0, MAX_CMD_LEN);
            bufferIndex = 0;
            continue;
        }

        if (strcmp(userData.channel, "CH1") == 0)
        {
            setChannelVariables(&CH1, userData);
        }

        if (strcmp(userData.channel, "CH2") == 0)
        {
            setChannelVariables(&CH2, userData);
        }

        // assign the modulos
        CH1.modulo = GetModulo(CH1.frequency);
        CH2.modulo = GetModulo(CH2.frequency);

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

    return 0;
}

/**
 * @brief Set channel-specific variables based on user input.
 * @param channel Pointer to the Channel struct.
 * @param parsedInput UserInput struct containing parsed user input.
 */
void setChannelVariables(Channel *channel, UserInput parsedInput)
{
    if (strcmp(parsedInput.parameter, "TYPE") == 0)
    {
        // set the current function
        if (strcmp(parsedInput.value, "SIN") == 0)
        {
            channel->function = 0;
        }
        else if (strcmp(parsedInput.value, "LINE") == 0)
        {
            channel->function = 1;
        }
```

```
        else if (strcmp(parsedInput.value, "SQR") == 0)
        {
            channel->function = 2;
        }
    }
    else
    {
        // FREQ or AMP -> need to convert the char* to int
        int value = atoi(parsedInput.value);

        if (strcmp(parsedInput.parameter, "FREQ") == 0)
        {
            channel->frequency = value; // frequency is by default in Hz
        }
        else if (strcmp(parsedInput.parameter, "AMP") == 0)
        {
            channel->amplitude = convertAmplitude(value); // convert from mV to interval
<0;210>
        }
    }
}


/**
 * @brief Parse user input and return a UserInput struct.
 * @param input User input string.
 * @return Parsed UserInput struct.
 */
void parseUserInput(UserInput *result, char* data)
{
    // init the fields as empty
    result->channel = result->parameter = result->value = NULL;

    int id = 0;

    // split the user input with the ':' delimeter
    char *token = strtok(data, ":");

    // go through the split string
    while (token != NULL) {
        if(id == 0)
        {
            result->channel = token;
        }
        if(id == 1)
        {
            result->parameter = token;
        }
        if(id == 2)
        {
            result->value = token;
        }
        token = strtok(NULL, ":");
        id++;
```

```
    }
}


/**
 * @brief Check the validity of a user command.
 * @param input UserInput struct to be validated.
 * @return True if the command is valid, false otherwise.
 */
bool GetCommandValidity(UserInput *input)
{
    if (input->channel == NULL || input->parameter == NULL || input->value == NULL)
    {
        // invalid input -> incomplete command
        return false;
    }

    bool valid = false;

    if (strcmp(input->channel, "CH1") == 0 || strcmp(input->channel, "CH2") == 0)
    {
        // channel is entered correctly
        valid = true;
    }

    if(strcmp(input->parameter, "TYPE") == 0 || strcmp(input->parameter, "FREQ") == 0 ||
strcmp(input->parameter, "AMP") == 0)
    {
        // parameter has the correct type
        valid = true;
    }
    else
    {
        valid = false;
    }


    // check if the entered type is correct
    if(strcmp(input->parameter, "TYPE") == 0)
    {
        if(strcmp(input->value, "SIN") == 0 || strcmp(input->value, "LINE") == 0 ||
strcmp(input->value, "SQR") == 0)
        {
            valid = true;
        }
        else
        {
            valid = false;
        }
    }
    else
    {
        // value has to be a number as the param is either freq or sin
        // convert the char* to int -> ASCII to integer
```

```
        int valueNumber = atoi(input->value);

        if(strcmp(input->parameter, "AMP") == 0)
        {
            if(valueNumber > 0 && valueNumber <= ANALOG_MAX_AMP)
            {
                valid = true;
            }
            else
            {
                valid = false;
            }
        }
        else
        {
            // input.parameter == FREQ
            if(valueNumber > 0)
            {
                valid = true;
            }
            else
            {
                valid = false;
            }
        }
    }

    return valid;
}

/**
 * @brief ISR for Timer 1 for generating the signals on both channels
 */
ISR(TIMER1_COMPA_vect)
{
    CH1.internalCounter++;
    CH2.internalCounter++;

    // CH1 code
    if(CH1.internalCounter >= CH1.modulo)
    {
        // if it is time to generate -> reset internal counter
        CH1.internalCounter = 0;

        // output the proper values for channel one
        CH1OutputFunction();

        CH1.outputCounter++; // increase the array index

        if(CH1.outputCounter > SAMPLING_RATE - 1)
        {
            // if index out of bounds -> reset it
            CH1.outputCounter = 0;
        }
```

```
    }

    // CH2 code
    if(CH2.internalCounter >= CH2.modulo)
    {
        // if it is time to generate -> reset the internal counter
        CH2.internalCounter = 0;

        // output the proper values for ch2
        CH2OutputFunction();

        CH2.outputCounter++; // increase the array index

        if(CH2.outputCounter > SAMPLING_RATE - 1)
        {
            // if the array index out of bounds -> reset it
            CH2.outputCounter = 0;
        }
    }
}

/**
 * @brief Output function for Channel 1.
 */
void CH1OutputFunction()
{

    int yCH1 = 0;

    if(CH1.function == 0)
    {
        // sine function
        int virtualZero = CH1.amplitude / 2;
        yCH1 = virtualZero + (CH1.amplitude / 2) * sineTable[CH1.outputCounter];
        outputToDAC(&CH1_OUT, (yCH1));
    }

    if(CH1.function == 1)
    {
        // line function
        float out = lineTable[CH1.outputCounter]; // temporary variable due to
incompatible variabe type of yCH1
        outputToDAC(&CH1_OUT, CH1.amplitude * out);
    }
    if(CH1.function == 2)
    {
        // square function -> duty cycle 50%
        if(CH1.outputCounter < (SAMPLING_RATE / 2))
        {
            outputToDAC(&CH1_OUT, CH1.amplitude);
        }
        else
        {
            outputToDAC(&CH1_OUT, 0);
```

```
        }
    }
}


/**
 * @brief Output function for Channel 2.
 */

void CH2OutputFunction()
{
    int yCH2 = 0;
    if(CH2.function == 0)
    {
        // sine function
        int virtualZero = CH2.amplitude / 2;
        yCH2 = virtualZero + (CH2.amplitude / 2) * sineTable[CH2.outputCounter];
        outputToDAC(&CH2_OUT, (yCH2));
    }

    if(CH2.function == 1)
    {
        // linear function
        float out = lineTable[CH2.outputCounter];
        outputToDAC(&CH2_OUT, CH2.amplitude * out);
    }

    if(CH2.function == 2)
    {
        // square function -> duty cycle 50%
        if(CH2.outputCounter < (SAMPLING_RATE / 2))
        {
            outputToDAC(&CH2_OUT, CH2.amplitude);
        }
        else
        {
            outputToDAC(&CH2_OUT, 0);
        }
    }
}

/**
 * @brief Function that fills the global sine table with data to speed up the runtime.
 * the calculated value is in range <-1; 1> => need to set up the virtual zero
 */
void fillSineTable()
{
    for(int i=0; i < SAMPLING_RATE; i++)
    {
        // convert deg to rad and calculate the sine of that value
        sineTable[i] = sin((M_PI / 180.0) * i);
    }
}
```

```c
/**
 * @brief Function that fills the global linear table with data to speed up the runtime.
 */
void fillLinearTable()
{
    for(int i=0; i < SAMPLING_RATE; i++)
    {
        lineTable[i] = (i * (1.0/SAMPLING_RATE)); // 1/SAMPLING_RATE = 0.00277
    }
}


/**
 * @brief Function that gets the number of ticks between each DAC output to controll the
frequency.
 * @param frequency = desired frequency of the signal
 * @param sampleRate = the resolution of the output signal
 */
uint32_t GetModulo(uint32_t frequency) {
    // Calculate the number of ticks for a given frequency
    uint32_t ticks = ((F_CPU / 256) / frequency) / SAMPLING_RATE; // 256 = number of
possible outputs of the DAC -> 2**8
    return ticks;
}

/**
 * @brief function that outputs a given value to a predefined port.
 * @param port = address to an output port
 * @param value = decimal 8 bit value to output on the given port
 */
void outputToDAC(volatile uint8_t *port, uint8_t value)
{
    // assign the value to the port
    *port = value;
}


/**
 * @brief Converts the amplitude from mV to decimal [0, DIGITAL_MAX_AMP].
 * @param oldAmplitude The amplitude in mV.
 * @return The converted amplitude in the range [0, DIGITAL_MAX_AMP].
 */
int convertAmplitude(int oldAmplitude)
{
    return (oldAmplitude * DIGITAL_MAX_AMP) / (ANALOG_MAX_AMP);
}
```

Odpovědi na otázky:

1. Rozhodněte, jaké rozlišení převodníku budete potřebovat, pokud budete chtít generovat signál v rozsahu ±2.5V s nejmenším rozlišením (velikostí kroku) alespoň 1.5mV.

   Obecný vzorec pro výpočet rozlišení převodníku v mV:

   $$rozlišení = \frac{\Delta U}{n}$$

   Rozlišení = rozlišení (nejmenší krok) v mV
   ΔU = napěťový rozsah
   n = počet stavů na vstupu DAC převodníku

   Po dosazení a vyjádření:

   $$1.5 \cdot 10^{-3} = \frac{5}{n} => n = \frac{5}{1.5 \cdot 10^{-3}} \cong 3333.33$$

   Výpočet bitového rozlišení převodníku:
   Důvod převodu n na reálné rozlišení (x) je, že n je pouze výsledkem umocnění čísla 2 bitový rozlišením.

   $$x = \log_2 3333.33 => x \cong 12\ bit\text{ů}$$

   Dle výpočtu je zřejmé, že je třeba převodník s maximálně 12 bitovým rozlišením. Ke splnění podmínky je však nutné zvolit převodník 8 bitový, jelikož 12 bitový převodník má minimální změnu napětí zhruba 1.22 mV.

2. Vyberte vhodný integrovaný DAC vyhovující požadavkům v předcházejícím bodu a uveďte nejdůležitější parametry z jeho katalogového listu. (tip: použijte vyhledávač některého velkého prodejce součástek, např. Mouser nebo Farnell).

   Mnou navrhovaný DAC: DAC0800LCN/NOPB od Texas Instuments

   Parametry:
   - Rozlišení = 8bitů
   - Počat kanálů = 1
   - Doba ustálení = 100ns
   - Výstupní napětí = -10 -> +18V

3. Pokuste se pro požadavek na generování sinusového průběhu o frekvenci 1MHz s alespoň 16 vzorky na periodu odvodit nejdelší dobu převodu, kterou může použitý převodník mít (tip: použijte Nyquistův teorém a přepočet f na T).
   Nyquistův teorém:

   $$f_{vzorkovací} \geq 2 \cdot f_{signálu}$$

   Minimální vzorkovací frekvence:

   Nejmenší možná vzorkovací frekvence:

   $$f_{vzorkovací} \geq 2 \cdot 10^6\ Hz$$

   Perioda signálu:

   $$T = \frac{1}{f_{vzorkovací}} = \frac{1}{2 \cdot 10^6} = 5 \cdot 10^{-7}\ s$$

   Perioda vzorku:

   $$T_{vzorku} = \frac{T}{16} = \frac{5 \cdot 10^{-7}}{16} = 31{,}25 \cdot 10^{-9}\ s$$

   Z výpočtů uvedených výše vyplívá, že nejdelší doba převodu je 31,25ns.

Závěr:

Výsledkem této úlohy je plně funkční implementace generátoru 2 na sobě nezávislých průběhů, jejichž parametry se dají dynamicky měnit za běhu. Jediné, co řešení chybí je vyšší rozsah podporovaných frekvencí signálu, ten se aktuálně pohybuje od 0 do zhruba 100Hz, ale dal by se upravit změnou počtu vzorků na průběh. Čím nižší by byl počet vzorků na průběh, tím vyšší frekvence by generátor dosáhl.