Politechnika Świętokrzyska

Wydział Mechatroniki i Budowy Maszyn

Katedra Automatyki i Robotyki

Instrukcja do zajęć laboratoryjnych z przedmiotów

„Układy Cyfrowe i Programowalne”

„Układy Mikroprocesorowe w Sterowaniu”

„Mikrokontrolery i Komputery Jednoukładowe”

opracował:

dr inż. Dawid Pietrala

Kielce, 2021

Spis treści

INFORMACJE WSTĘPNE 3

ZAJĘCIA NR 1 WPROWADZENIE. WYJŚCIA CYFROWE 4

ZAJĘCIA NR 2 OBSŁUGA WEJŚĆ CYFROWYCH. 16

ZAJĘCIA NR 3 OBSŁUGA WYŚWIETLACZA SEGMENTOWEGO LED. 22

ZAJĘCIA NR 4 OBSŁUGA WYŚWIETLACZA ALFANUMERYCZNEGO. 27

ZAJĘCIA NR 5 USART – OBSŁUGA PODSTAWOWA 31

ZAJĘCIA NR 6 USART – OBSŁUGA Z WYKORZYSTANIEM PRZERWAŃ. 35

ZAJĘCIA NR 7 USART – OBSŁUGA Z WYKORZYSTANIEM KONTROLERA DMA 40

ZAJĘCIA NR 8 I2C – OBSŁUGA PODSTAWOWA. AKCELEROMETR LSM303C 46

ZAJĘCIA NR 9 I2C – PRZERWANIA. AKCELEROMETR LSM303C 54

ZAJĘCIA NR 10 I2C – OBSŁUGA Z WYKORZYSTANIEM DMA. AKCELEROMETR LSM303C 60

ZAJĘCIA NR 11 PRZETWORNIK ANALOGOWO – CYFROWY. 66

ZAJĘCIA NR 12 PRZERWANIA ZEWNĘTRZNE. 71

ZAJĘCIA NR 13 UKŁADY LICZNIKOWE PODSTAWOWE. POMIAR CZASU. ULTRADŹWIĘKOWY CZUJNIK ODLEGŁOŚCI. 75

ZAJĘCIA NR 14 UKŁADY LICZNIKOWE OGÓLNEGO ZASTOSOWANIA. GENEROWANIE SYGNAŁU PWM. 81

ZAJĘCIA NR 15 UKŁADY LICZNIKOWE OGÓLNEGO ZASTOSOWANIA – SILNIK DC Z ENKODEREM. 84

ZAJĘCIA NR 16 SERWONAPĘD DC 87

Informacje wstępne

Zagadnienia omówione w tej instrukcji zostały również przedstawione w formie materiałów wideo umieszczonych w serwisie YouTube. Lista tych materiałów dostępna jest pod adresem: https://www.youtube.com/playlist?list=PLoRtN4ki_qPCNAwfnr_b2gYybPPj6rXz4

Zezwala się na kopiowanie i powielanie w niezmienionej formie bez ograniczeń niniejszej instrukcji, z zastrzeżeniem obowiązku umieszczenia informacji o jej autorze.

Zezwala się na kopiowanie i powielanie w niezmienionej formie bez ograniczeń niniejszych materiałów wideo, z zastrzeżeniem obowiązku umieszczenia informacji o ich autorze.

Zajęcia nr 1 Wprowadzenie. Wyjścia cyfrowe

Wprowadzenie

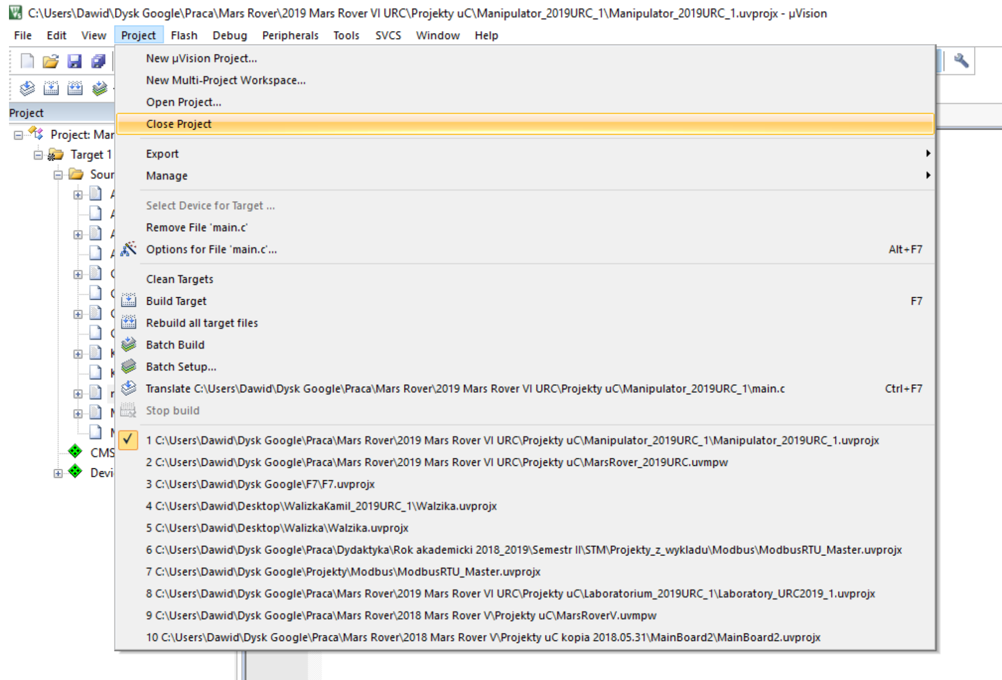
Podczas pierwszych zajęć student zapozna się ze środowiskiem programistycznym (IDE) Keil uVision wykorzystywanym do programowania mikrokontrolerów STM32L4 firmy STMicroelectronics. Student zapozna się ze sposobem tworzenia projektu w ww. środowisku programistycznym, pozna strukturę plików i katalogów powstałych przy tworzeniu projektu. Następnie student przygotuje i napisze pierwszy program w języku C na mikrokontroler, w którym zapozna się z wykorzystaniem cyfrowych wyprowadzeń mikrokontrolera do sterowania diodami LED, pozna metody programowania pamięci Flash mikrokontrolera za pomocą dedykowanego programatora STLink v2 oraz przetestuje działanie programu.

Wszystkie niezbędne informacje, dotyczące źródeł, z których można pobrać środowisko programistyczne Keil uVision, sterowniki do programatora STLink v2 oraz informacje dotyczące modułu testowego STM32L4 „Kameleon” zostaną przedstawione studentom w ramach zajęć wykładowych.

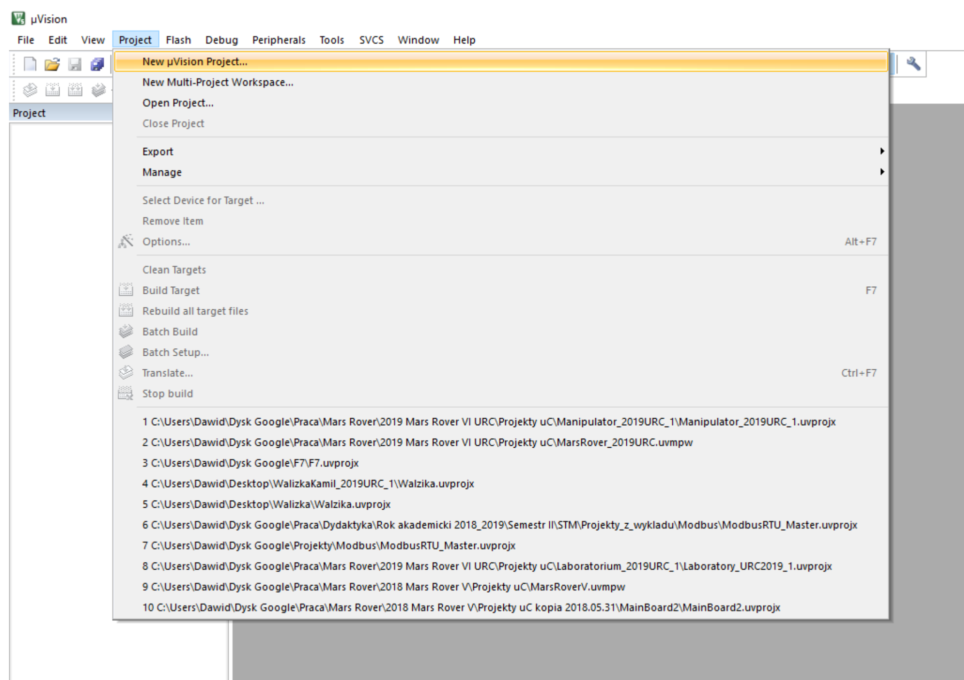
Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków”.

Tworzenie projektu w środowisku Keil uVision

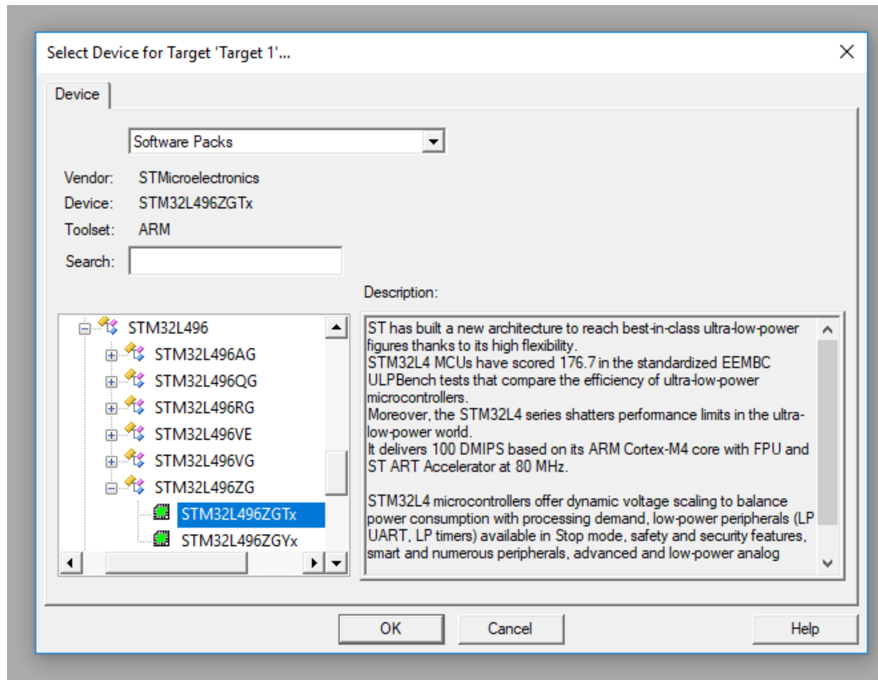
Następnie przystępujemy do utworzenia projektu w środowisku programistycznym Keil uVision. W tym celu uruchamiamy środowisko. Jeżeli wraz z uruchomieniem otwarty został poprzedni projekt należy zamknąć go za pomocą polecenia *Project/Close Project*.



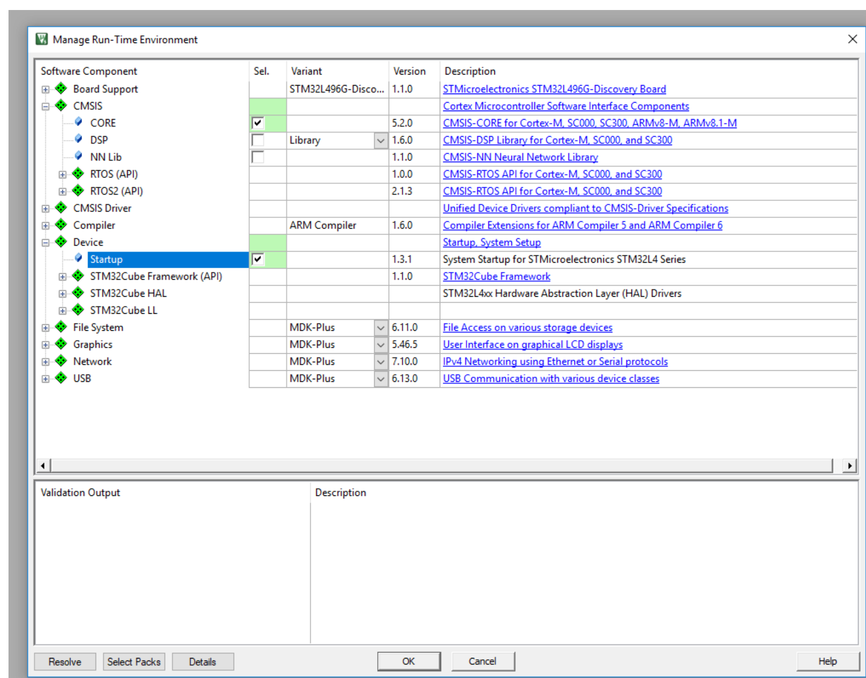
Następnie za pomocą polecenia *Project/New uVision Project…* tworzymy nowy projekt i zapisujemy go w utworzonym wcześniej podkatalogu.



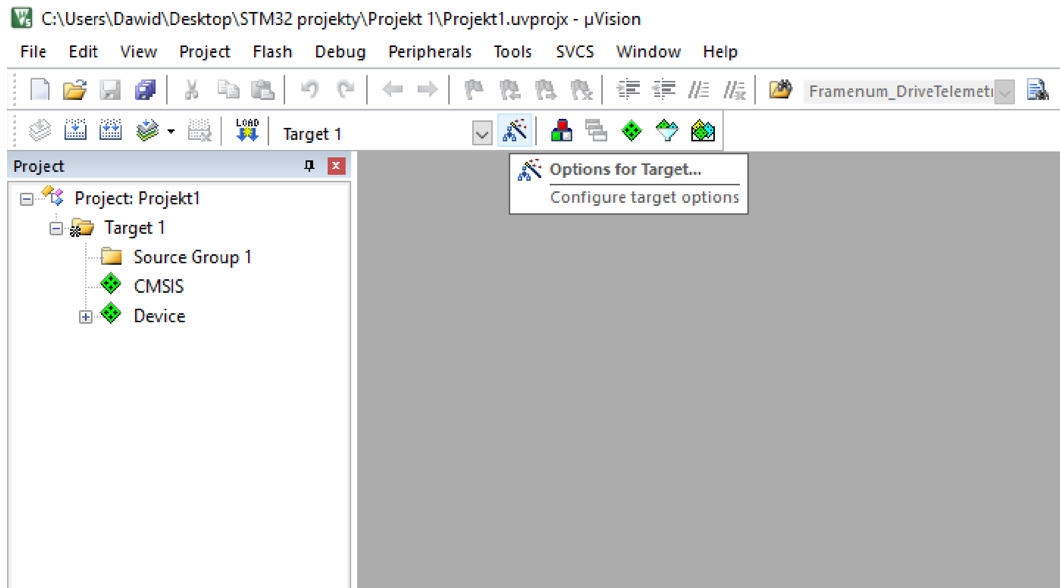
Nazwa projektu nie powinna zawierać „polskich znaków”. Następnie w otwartym oknie wybieramy model procesora, który będziemy programować. W naszym przypadku jest to STM32L496ZGT i potwierdzamy *OK*.



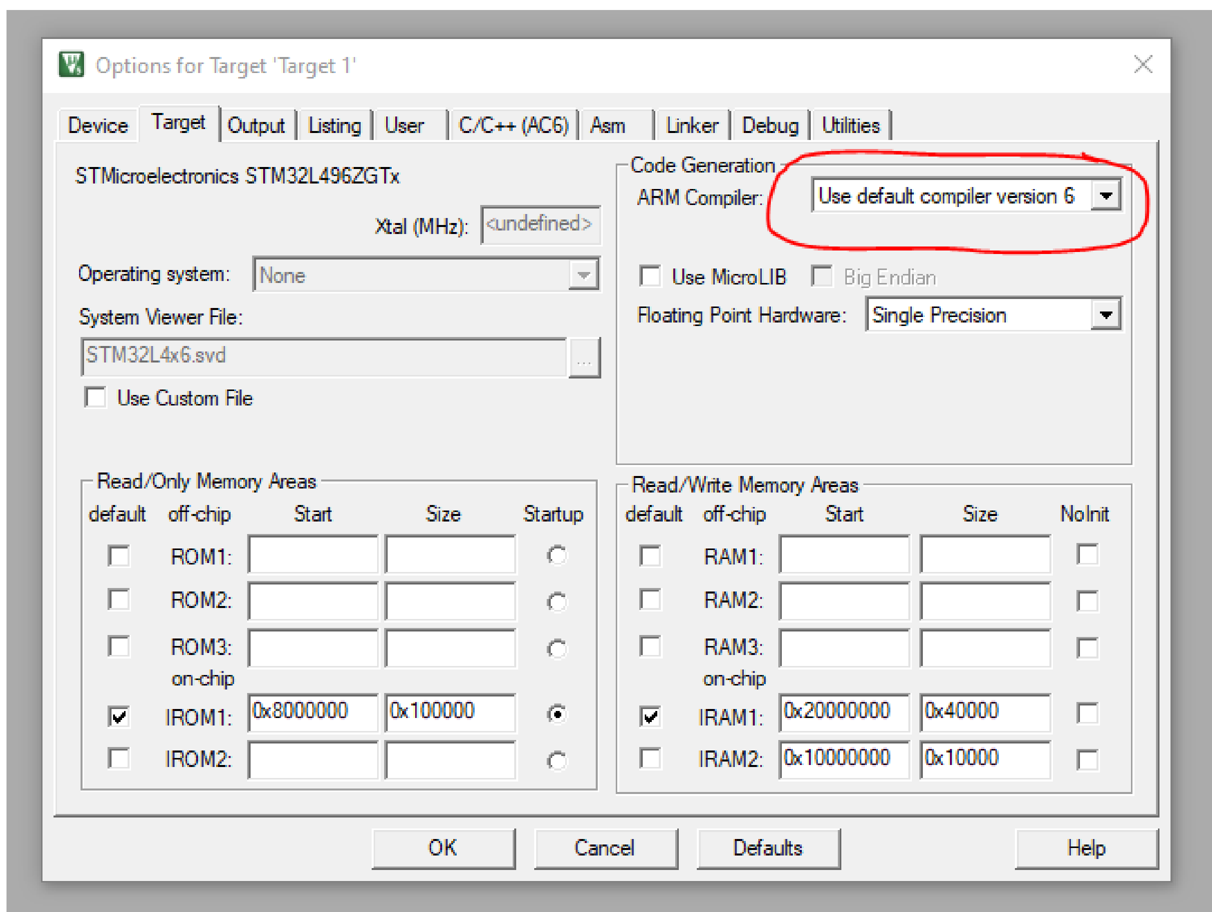
Następnie z otwartego okna wybieramy niezbędne biblioteki i moduły, jakie musimy dołączyć do projektu. Z grupy *CMSIS* wybieramy *Core*, a z grupy *Device* wybieramy *Startup*. Potwierdzamy przyciskiem OK.



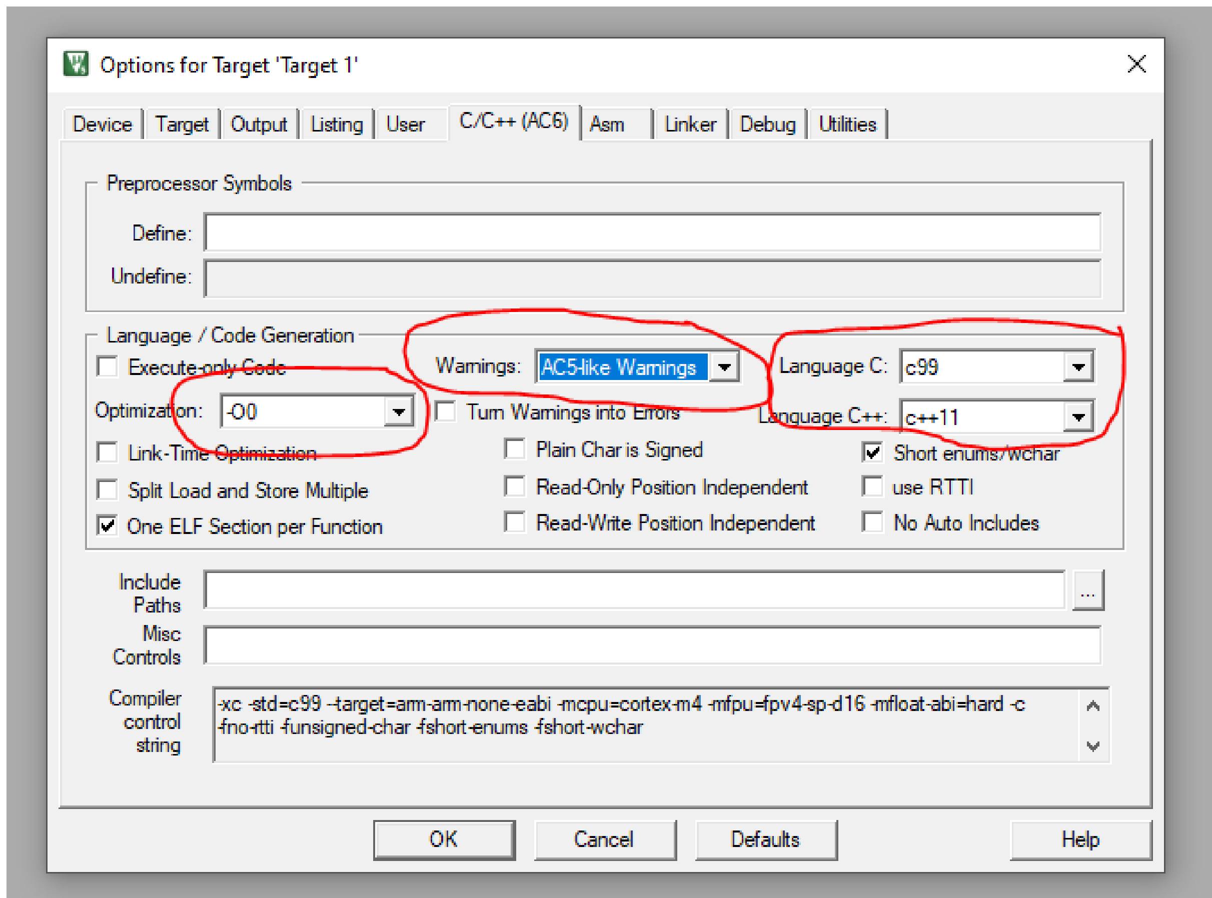
W ten sposób projekt został utworzony. Musimy jeszcze skonfigurować w projekcie kilka ustawień. W tym celu otwieramy okno *Options for Target* naciskają ikonę jak na zdjęciu.



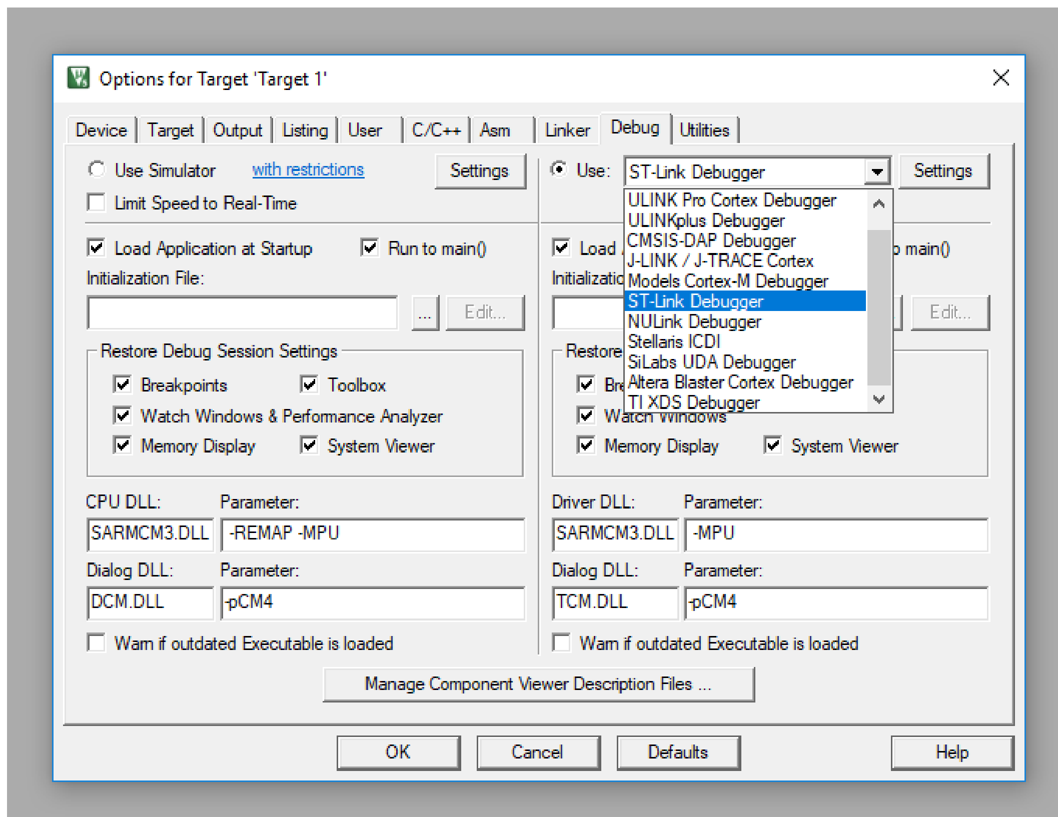
Następnie, w zakładce *Target,* wybieramy odpowiedni kompilator (*Use default compiler ver 6*).



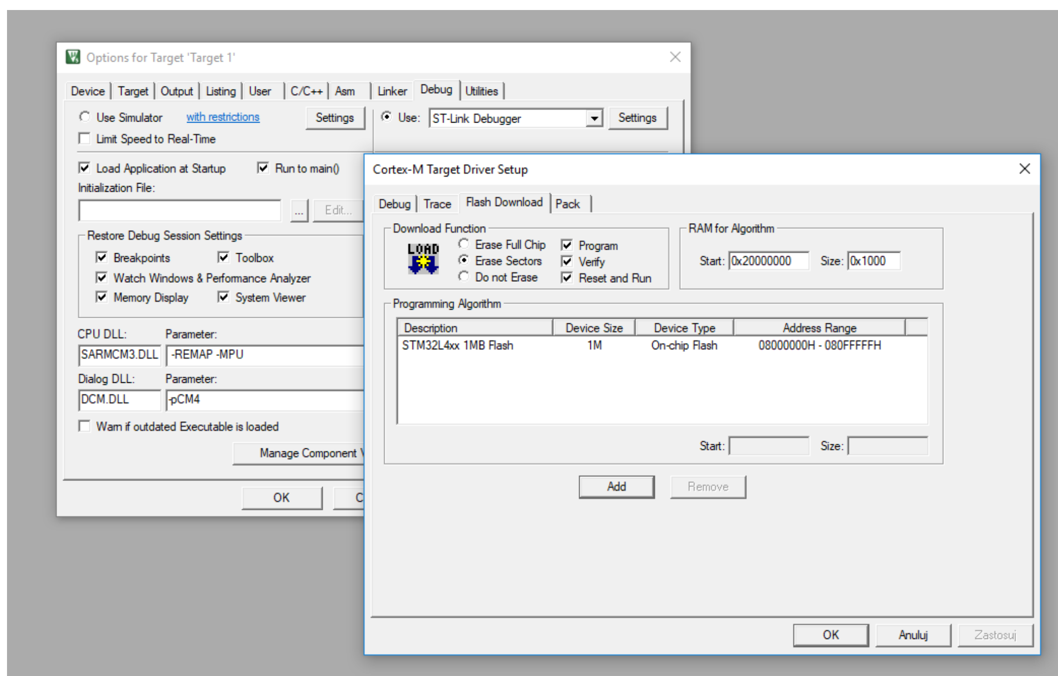
Dalej, w zakładce *C/C++* zaznaczamy opcję *C99 Mode, AC5-like Warnings* i brak optymalizacji.



Przechodzimy do zakładki *Debug*, wybieramy z listy programator *ST-Link Debuger* i naciskamy przycisk *Settings*.



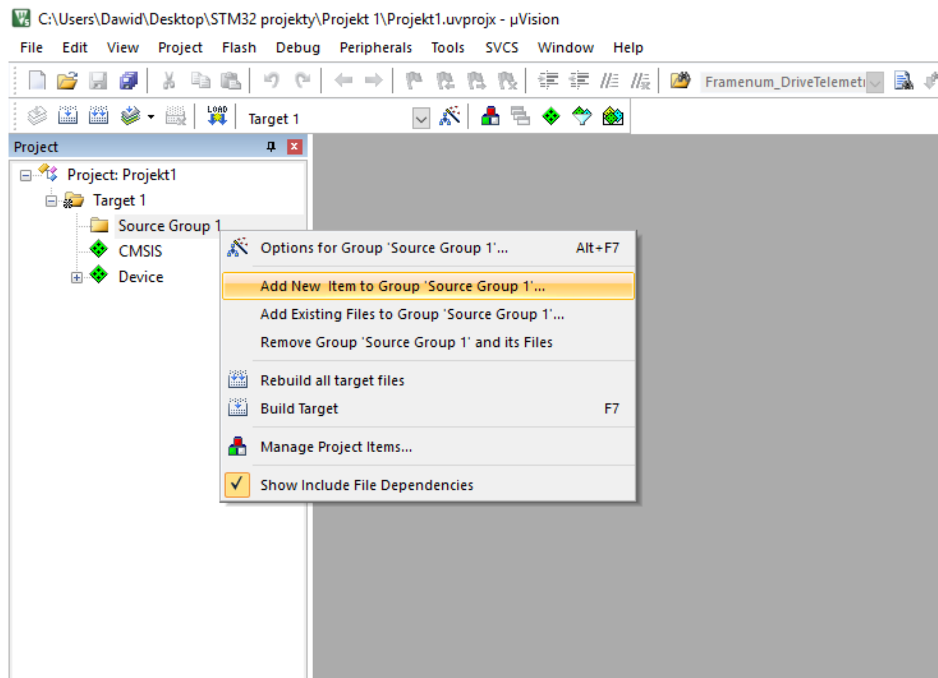
Otworzy się okno ustawień dla programatora. Przechodzimy do zakładki *Flash Download* i zaznaczamy checkbox *Reset and Run*.



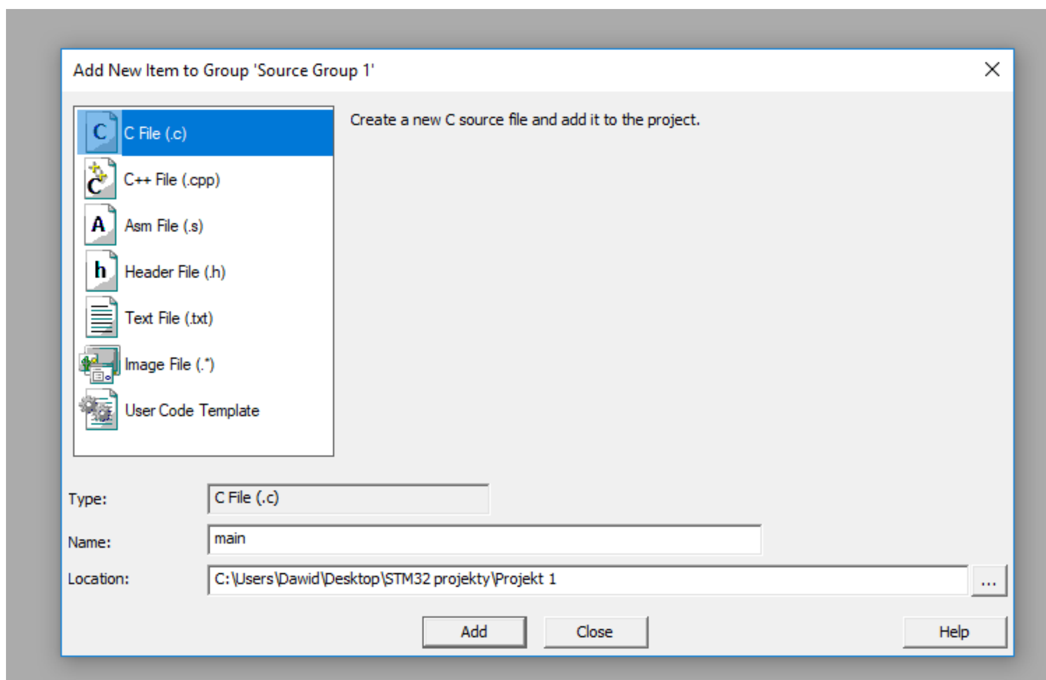
Potwierdzamy zmiany poprzez naciśnięcie przycisków *Ok* w obydwu oknach. Nasz projekt jest gotowy i możemy przystąpić do pisania programu.

Pierwszy program

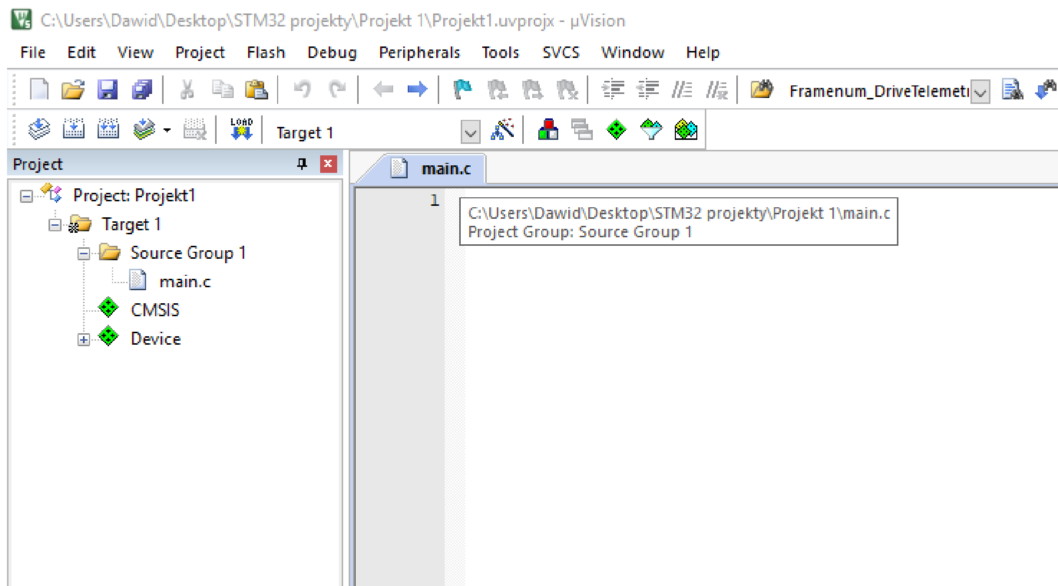
Nasze środowisko jest gotowe do pracy, jednak nasz projekt nie zawiera jeszcze żadnego pliku, w którym moglibyśmy umieścić program. Musimy zatem taki plik utworzyć. W tym celu klikamy PPM (prawy przycisk myszy) w drzewku projektu na napis *Source Group 1* i wybieramy opcję *Add New Item to Group „Source Group 1”…*



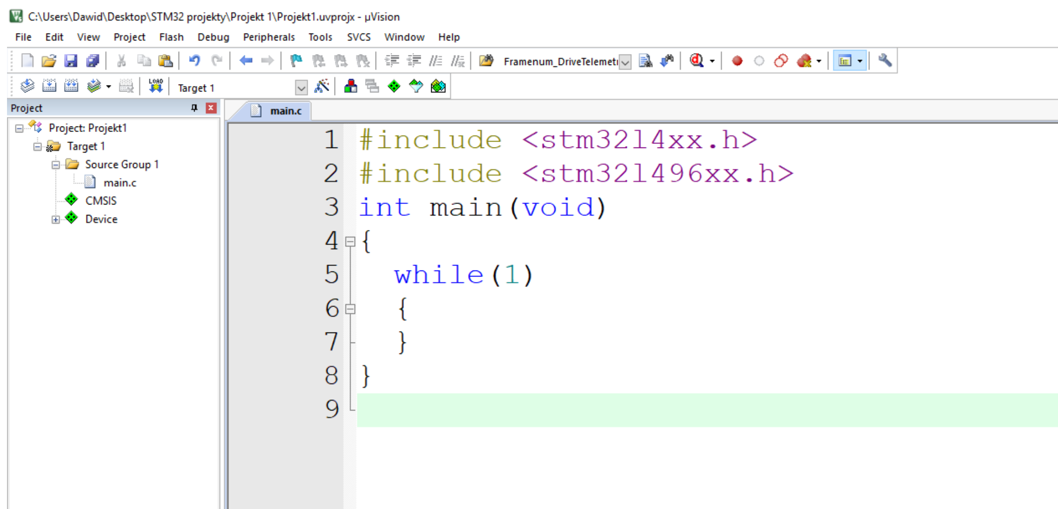
W otwartym oknie wybieramy rodzaj pliku jako *C File (.c)*, nadajemy nazwę *main* i potwierdzamy przyciskiem *Add*.



Utworzony plik pojawił się w drzewku projektu oraz został on otwarty. Możemy zacząć pisać w nim nasz pierwszy program.



Każdy program w języku C na mikrokontroler musi zawierać funkcję *main(){}* i powinien zawierać nieskończoną pętlę *while(1){}*. Do programu dołączamy również niezbędne biblioteki zawierające makrodefinicje dla danego mikrokontrolera.



Następnie uruchomimy wbudowany w rdzeń procesora układ licznikowy *Systick* i wykorzystamy go do generowania przerwań z okresem 1ms, co pozwoli na odmierzanie stosunkowo precyzyjnych odstępów czasu. W tym celu napiszemy program jak poniżej.

```
#include <stm32l4xx.h>
#include <stm32l496xx.h>
volatile uint32_t tick = 0;
void delay_ms(uint32_t ms)
{
	tick=0;
```

```
	while(tick < ms);
}
void Led_Conf(void)
{
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOCEN;
	GPIOC->MODER &= ~GPIO_MODER_MODE6;
	GPIOC->MODER |= GPIO_MODER_MODE6_0;
}
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	while(1)
	{
	}
}
void SysTick_Handler(void)
{
	tick++;
}
```

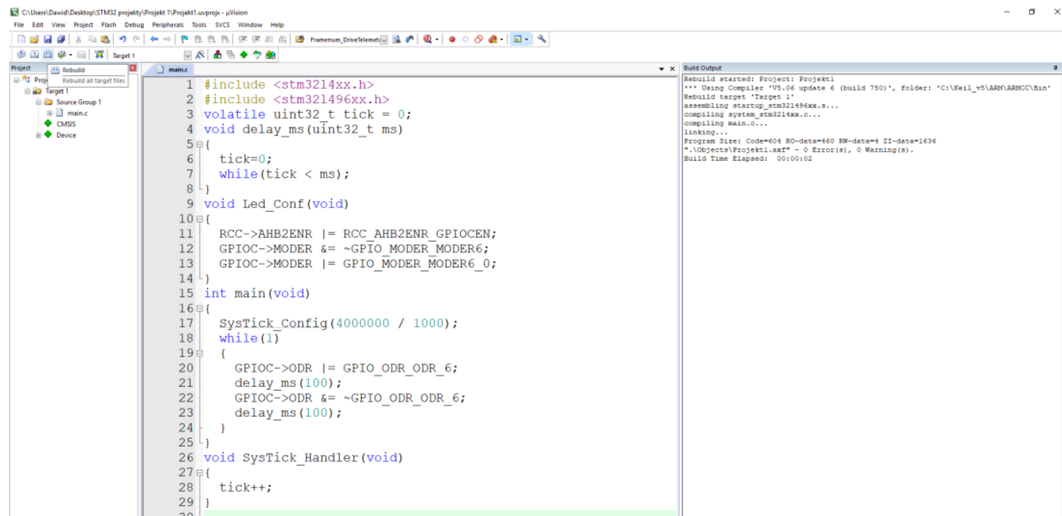
Instrukcja *Systick_Config()* uruchamia układ licznikowy. Jej parametr definiuje z jaką częstotliwością ma być generowane przerwanie. Funkcja *SysTick_Handler()* stanowi wektor obsługi przerwania dla tego układu licznikowego. W jej wywołaniu następuje inkrementacja *tick*. Natomiast funkcja *delay_ms()* wstrzymuje procesor na okres czasu (w milisekundach) podany jako parametr tej funkcji. Następnie przystępujemy do skonfigurowania wyprowadzenia numer 6 w porcie C *(GPIOC.6)* mikrokontrolera jako cyfrowego wyjścia, które zostanie wykorzystane do sterowania pracą diody LED. W tym celu tworzymy funkcję o nazwie *Led_Conf()*. Jej definicja zawiera następujące elementy. W pierwszej kolejności włączamy taktowanie zegarowe dla portu C. Następnie czyścimy aktualny stan bitów odpowiadających pinowi szóstemu w rejestrze *MODER* portu *GPIOC*. Następnie możemy skonfigurować pin *GPIOC.6* jako cyfrowe wyjście.

Dysponując tak przygotowanymi funkcjami możemy przetestować działanie naszego programu. W tym celu uzupełniamy funkcję *main()* kodem, który naprzemiennie włącza i wyłącza diodę LED podłączoną do *GPIOC.6.*

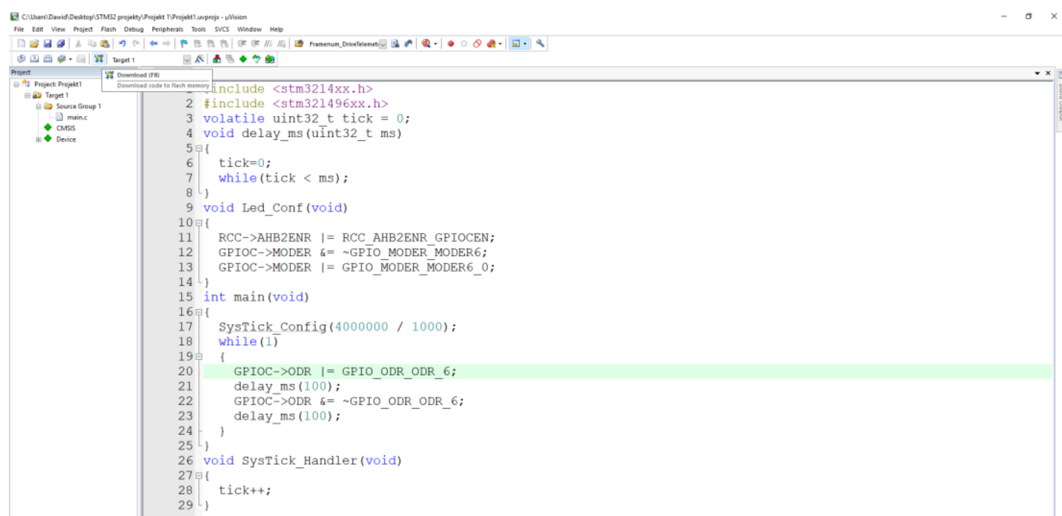
```
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	while(1)
	{
		GPIOC->ODR |= GPIO_ODR_OD6;
		delay_ms(100);
		GPIOC->ODR &= ~GPIO_ODR_OD6;
		delay_ms(100);
	}
}
```

Ustawienie stanu wysokiego dla danego pinu, a tym samym włączenie diody, odbywa się poprzez ustawienie odpowiedniego bitu w rejestrze *ODR* dla danego *GPIO*. Tym samym wyłączenie diody, czyli ustawienie stanu niskiego na danym pinie odbywa się poprzez wyczyszczenie odpowiedniego bitu w rejestrze *ODR*.

Następnie kompilujemy nasz program naciskając ikonę *Rebuild*.



Jeżeli proces kompilacji przebiegł prawidłowo możemy wgrać nasz pierwszy program do pamięci Flash mikrokontrolera i przetestować jego działanie. W tym celu używamy ikony *Download*.



Zadanie 1:

Jeżeli praca programu przebiega prawidłowo, a powyższe zagadnienia są dla studenta zrozumiałe, należy przystąpić do wykonania następującego zadania. Łącznie w zestawie dydaktycznym znajduje się 8 diod LED, podłączonych do następujących wyprowadzeń: *GPIOC.6*, *GPIOC.7*, *GPIOC.8*, *GPIOC.9*, *GPIOE.4*, *GPIOD.3*, *GPIOE.5*, *GPIOE.6*.

Student powinien uzupełnić funkcję *Led_Conf()* o kod konfigurujący do pracy pozostałe 7 wyprowadzeń mikrokontrolera, do których podłączone są pozostałe diody LED. Następnie student powinien przetestować działanie wszystkich wyprowadzeń.

Następnie przystępujemy do rozszerzenia naszego programu o elementy pozwalające w łatwy sposób zarządzać diodami LED w zestawie ewaluacyjnym. Elementy te będą również przydatne w przyszłych projektach realizowanych na tym zestawie dydaktycznym. W pierwszej kolejności utworzymy typ wyliczeniowy zawierający nazwy definiujące stan diody. Następnie utworzymy funkcję o nazwie *Led_OnOff(uint8_t num, eLed state),* która będzie wykorzystywana do zmiany stanu danej diody.

```
typedef enum{LedOff = 0, LedOn = 1, LedTog = 2}eLed;
void Led_OnOff(uint8_t num, eLed state)
{
	switch(num)
	{
		case 0:
			if(state == LedOff)			GPIOC->ODR &= ~GPIO_ODR_OD6;
			else if(state == LedOn)		GPIOC->ODR |= GPIO_ODR_OD6;
			else if(state == LedTog)		GPIOC->ODR ^= GPIO_ODR_OD6;
			break;
	}
}
```

Funkcja ta jako parametry przyjmuje numer diody, od 1 do 8, oraz stan jaki dana dioda ma osiągnąć: wyłączona, włączona lub zmieniony na przeciwny. Następnie możemy przetestować działanie tej funkcji uzupełniając funkcję *main()* następującym fragmentem programu.

```
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	while(1)
	{
		Led_OnOff(0, LedTog);
		delay_ms(100);
	}
}
```

Zadanie 2:

Student powinien uzupełnić funkcję Led_OnOff() o kod pozwalający sterować pozostałymi siedmioma diodami LED oraz przetestować działanie.

Zadanie 3:

Student powinien napisać program, który umożliwi przesuwanie się od lewej do prawej i z powrotem linijki włączonych diod LED, przy czym jednocześnie powinny być włączone maksymalnie trzy diody.

Zadanie 4:

Student powinien przygotować program umożlwiający wyświetlanie liczb zakodowanych w naturalnym kodzie binarnym za pomocą 8 diod LED. Program ten powinien wyświetlać w pętli *for()* kolejne liczby od 0 do 255 z odstępem czasu równym 1 sekundy.

Zadanie domowe:

Należy uzupełnić funkcję *Led_OnOff()* o możliwość łatwej zmiany stanu wszystkich diod jednocześnie.

Zajęcia nr 2 Obsługa wejść cyfrowych.

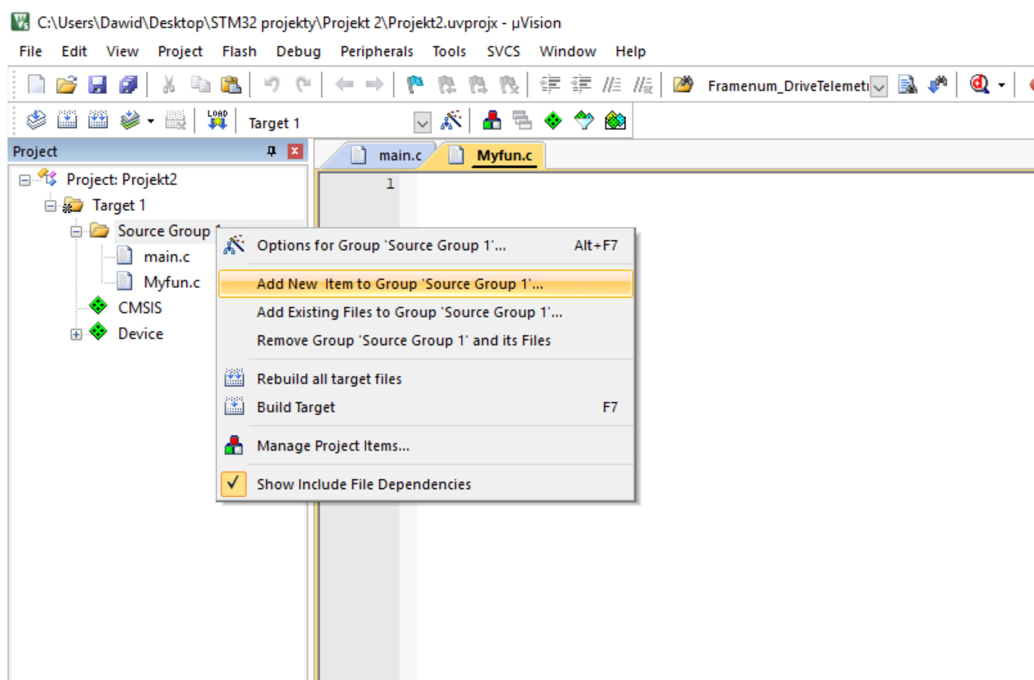
Wprowadzenie

Podczas drugich zajęć student zapozna się z tworzeniem projektu zawierającego więcej niż jedne plik źródłowy oraz pliki nagłówkowe. Student zdobędzie wiedzę na temat tworzenie własnych modułów składających się z plików źródłowych i nagłówkowych oraz wykorzystania ich w kolejnych projektach. Następnie student zapozna się z wykorzystaniem wejść cyfrowych mikrokontrolera do obsługi przycisków.

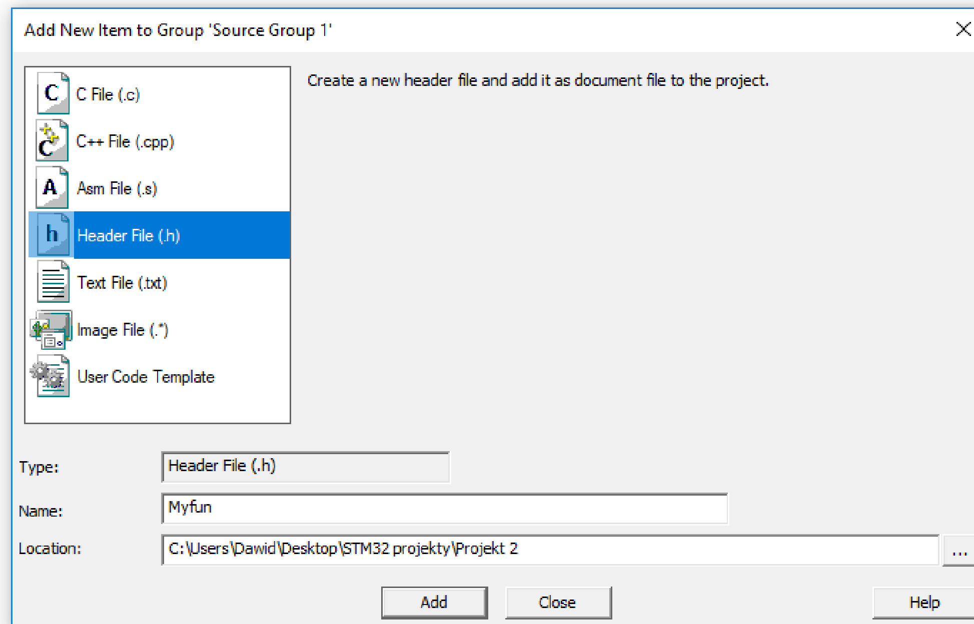
Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków”.

Tworzenie projektu w środowisku Keil uVision

Na wstępie tworzymy projekt według schematu postępowania przedstawionego w instrukcji do pierwszych zajęć. Następnie do projektu dodajemy trzy nowe pliki: dwa pliki typu *C File (.c) main.c* i *Myfun.c*



oraz jeden plik nagłówkowy typu *Header File (.h)* o nazwie *Myfun.h.*



Następnie w poszczególnych plikach wpisujemy następujące fragmenty kodu. W pliku *Myfun.h* wpisujemy kod:

```
#ifndef _MYFUN
#define _MYFUN
#include <stm32l4xx.h>
#include <stm32l496xx.h>
//…tutaj możemy wpisywać pozostały kod programu dla tego pliku…
#endif
```

W pliku *Myfun.c* wpisujemy kod:

```
#include "Myfun.h"
//…tutaj mozemy wpisywac pozostaly kod programu dla tego pliku...
```

W pliku *Myfun.c* wpisujemy kod:

```
#include "Myfun.h"
int main(void)
{
	while(1)
	{
	}
}
```

W ten sposób otrzymujemy własny moduł, który będziemy mogli uzupełniać o nowe funkcje konfigurujące do pracy poszczególne układy peryferyjne naszego mikrokontrolera. Moduł ten będziemy wykorzystywać we wszystkich przyszłych projektach. Zatem ważne jest aby przygotować go ze szczególną starannością, ponieważ ułatwi to przyszłe prace programistyczne.

Następnie uzupełniamy nasz moduł poznanymi na poprzednich zajęciach funkcjami, których zadaniem jest obsługa wbudowanych w zestaw dydaktyczny diod LED. A także uzupełniamy go o funkcje wykorzystujące układ licznikowy *Systick* do generowania opóźnień. Po ukończeniu tych zadań pliki *Myfun.c, Myfun.h* i *main.c* powinny wyglądać następująco.

Plik *Myfun.c:*

```
#include "Myfun.h"
volatile uint32_t tick = 0;
void delay_ms(uint32_t ms)
{
	tick=0;
	while(tick < ms);
}
void Led_Conf(void)
{
	RCC->AHB2ENR |=		RCC_AHB2ENR_GPIOCEN;
	GPIOC->MODER &=		~GPIO_MODER_MODER6 & ~GPIO_MODER_MODER7 &
				~GPIO_MODER_MODER8 & ~GPIO_MODER_MODER9;
	GPIOC->MODER |=		GPIO_MODER_MODER6_0 | GPIO_MODER_MODER7_0 |
				GPIO_MODER_MODER8_0 | GPIO_MODER_MODER9_0;
	RCC->AHB2ENR |=		RCC_AHB2ENR_GPIODEN;
	GPIOD->MODER &=		~GPIO_MODER_MODER3;
	GPIOD->MODER |=		GPIO_MODER_MODER3_0;
	RCC->AHB2ENR |=		RCC_AHB2ENR_GPIOEEN;
	GPIOE->MODER &=		~GPIO_MODER_MODER4 & ~GPIO_MODER_MODER5 &
				~GPIO_MODER_MODER6;
	GPIOE->MODER |=		GPIO_MODER_MODER4_0 | GPIO_MODER_MODER5_0 |
				GPIO_MODER_MODER6_0;
}
void Led_OnOff(uint8_t num, eLed state)
{
	switch(num)
	{
		case 0:
			if(state == LedOff)		GPIOC->ODR &= ~GPIO_ODR_OD6;
			else if(state == LedOn)		GPIOC->ODR |= GPIO_ODR_OD6;
			else if(state == LedTog)	GPIOC->ODR ^= GPIO_ODR_OD6;
			break;
		case 1:
			if(state == LedOff)		GPIOC->ODR &= ~GPIO_ODR_OD7;
			else if(state == LedOn)		GPIOC->ODR |= GPIO_ODR_OD7;
			else if(state == LedTog)	GPIOC->ODR ^= GPIO_ODR_OD7;
			break;
		case 2:
			if(state == LedOff)		GPIOC->ODR &= ~GPIO_ODR_OD8;
			else if(state == LedOn)		GPIOC->ODR |= GPIO_ODR_OD8;
			else if(state == LedTog)	GPIOC->ODR ^= GPIO_ODR_OD8;
			break;
		case 3:
			if(state == LedOff)		GPIOC->ODR &= ~GPIO_ODR_OD9;
			else if(state == LedOn)		GPIOC->ODR |= GPIO_ODR_OD9;
			else if(state == LedTog)	GPIOC->ODR ^= GPIO_ODR_OD9;
			break;
		case 4:
			if(state == LedOff)		GPIOE->ODR &= ~GPIO_ODR_OD4;
			else if(state == LedOn)		GPIOE->ODR |= GPIO_ODR_OD4;
```

```
                else if(state == LedTog)   GPIOE->ODR ^= GPIO_ODR_OD4;
                break;
        case 5:
                if(state == LedOff)        GPIOD->ODR &= ~GPIO_ODR_OD3;
                else if(state == LedOn)    GPIOD->ODR |= GPIO_ODR_OD3;
                else if(state == LedTog)   GPIOD->ODR ^= GPIO_ODR_OD3;
                break;
        case 6:
                if(state == LedOff)        GPIOE->ODR &= ~GPIO_ODR_OD5;
                else if(state == LedOn)    GPIOE->ODR |= GPIO_ODR_OD5;
                else if(state == LedTog)   GPIOE->ODR ^= GPIO_ODR_OD5;
                break;
        case 7:
                if(state == LedOff)        GPIOE->ODR &= ~GPIO_ODR_OD6;
                else if(state == LedOn)    GPIOE->ODR |= GPIO_ODR_OD6;
                else if(state == LedTog)   GPIOE->ODR ^= GPIO_ODR_OD6;
                break;
    }
}
void SysTick_Handler(void)
{
    tick++;
}
```

Plik Myfun.h:

```
#ifndef _MYFUN
#define _MYFUN
#include <stm32l4xx.h>
#include <stm32l496xx.h>
typedef enum{LedOff = 0, LedOn = 1, LedTog = 2}eLed;
void delay_ms(uint32_t ms);
void Led_Conf(void);
void LED_OnOff(uint8_t num, eLed state);
#endif
```

Plik *main.c:*

```
#include "Myfun.h"
int main(void)
{
    SysTick_Config(4000000 / 1000);
    Led_Conf();
    while(1)
    {
        Led_OnOff(0, LedTog);
        delay_ms(100);
    }
}
```

Warto zauważyć, że w pliku *Myfun.h* umieszczane są wszystkie deklaracje funkcji, które mają być dostępne poza modułem. Jedynie deklaracje wektorów przerwań nie są tutaj umieszczane, ponieważ zostały już umieszczone w pliku startup_stm32l496xx.s. Jako deklarację funkcji należy tutaj rozumieć jej pierwszą linijkę opatrzoną dodatkowo średnikiem. Natomiast w pliku *Myfun.c* umieszczane są definicje funkcji (czyli tzw. „ciała funkcji”). Możemy teraz

skompilować nasz projekt i wgrać go do pamięci naszego mikrokontrolera. Efektem działania programu powinno być miganie jednej z diod z częstotliwością 10Hz.

Przystępujemy teraz do zapoznania się z konfiguracją wyprowadzeń mikrokontrolera jako wejść cyfrowych i wykorzystaniem ich do odczytu stanu przycisków podłączonych do urządzenia. W zestawie dydaktycznym znajduje się pięciopozycyjny joystick, który de facto składa się z pięciu przycisków monostabilnych podłączonych do wyprowadzeń *GPIOE.0, GPIOE.1, GPIOE.2, GPIOE.3, GPIOE.15.* Wciśnięcie przycisku powoduje zwarcie odpowiadającego mu wyprowadzenia do masy. Tym samym w celu wykrycia wciśnięcia przez użytkownika przycisku należy sprawdzić czy na danym wyprowadzeniu pojawił się stan niski (domyślnie jest tam stan wysoki). W pliku *Myfun.c* tworzymy funkcje *Joy_Conf()* i konfigurujemy w niej poszczególne wyprowadzenia mikrokontrolera jako wejścia cyfrowe. Poniższy kod przedstawia tą funkcję ze skonfigurowanym jednym wyprowadzeniem. Student powinien uzupełnić konfiguracje pozostałych czterech wyprowadzeń. Wejście cyfrowe wymaga wyczyszczenia w rejestrze *MODER* danego *GPIO* wszystkich bitów dla danego wyprowadzenia. Należy pamiętać o umieszczeniu deklaracji tej funkcji w pliku nagłówkowym.

```
void Joy_Conf(void)
{
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOEEN;
	GPIOE->MODER &= ~GPIO_MODER_MODE0;
}
```

Możemy teraz wywołać tę funkcję w funkcji *main()* i odczytać stan przycisku. W tym celu uzupełnimy nasz plik *main.c* o następujący kod:

```
#include "Myfun.h"
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	Joy_Conf();
	while(1)
	{
		if((GPIOE->IDR & GPIO_IDR_ID0) == RESET)	Led_OnOff(0, LedOn);
		else						Led_OnOff(0, LedOff);
		delay_ms(10);
	}
}
```

Następnie przygotujemy funkcję do łatwego odczytu stanu przycisków. W tym celu w pliku nagłówkowym utworzymy typ wyliczeniowy a w pliku z kodem źródłowym funkcję *Joy_Read()*:

```
typedef enum{JoyNull = 0, Up = 1, Down = 2, Left = 3, Right = 4, Center = 5}eJoy;
eJoy Joy_Read(void)
{
```

```
	eJoy state = JoyNull;
	if((GPIOE->IDR & GPIO_IDR_ID0) == RESET)	state = Right;
	return state;
}
```

Należy pamiętać o umieszczeniu deklaracji tej funkcji w pliku nagłówkowym.

Zadanie 1:

Student powinien uzupełnić funkcję odczytu o pozostałe wyprowadzenia, przetestować jej działanie i upewnić się o odpowiednim przypisaniu kierunków pracy joystick'a do wyprowadzeń mikrokontrolera.

Zadanie 2:

Napisać program, w którym wykorzystywany jest joystick i linijka diod LED. Program ma umożliwiać użytkownikowi decydowanie o kierunku poruszania się włączonej diody LED (Left lub Right) oraz o szybkości poruszania się włączonej diody (Up lub Down). Należy określić minimalną częstotliwość przełączania diod na 1Hz i maksymalną na 100Hz.

Zadanie domowe:

Wbudowany w zestaw dydaktyczny joystick umożliwia jednoczesne uzyskanie dwóch stanów: wciśnięcie środka oraz jednego z pozostałych. Jednakże napisana przez nas funkcja pozwala na odczyt jedynie jednego. Student powinien zastanowić się jak rozszerzyć jej możliwości aby możliwy był odczyt większej liczby stanów joystick'a.

Zajęcia nr 3 Obsługa wyświetlacza segmentowego LED.

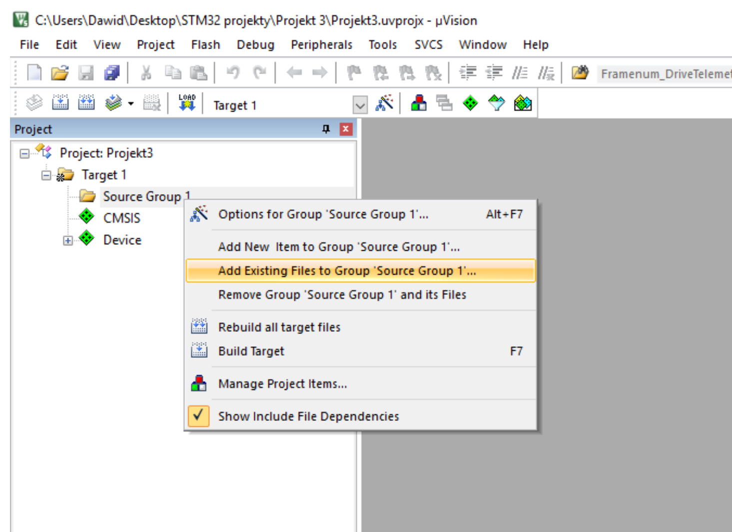
Wprowadzenie

Podczas tych zajęć student wykorzysta nabytą wiedzę dotyczącą wejść i wyjść cyfrowych do sterowania wyświetlaczem 7-segmentowym, złożonym z czterech modułów, w sposób statyczny i dynamiczny.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków".

Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu trzy pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1"…*

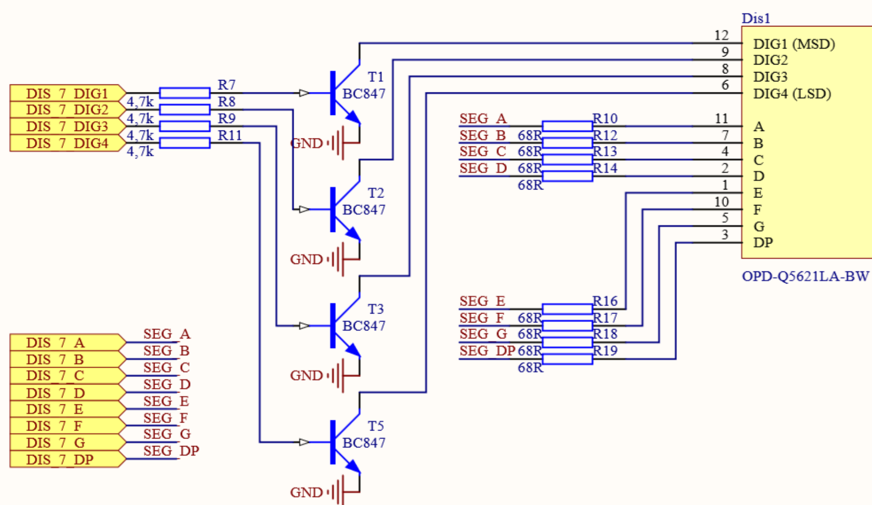


Następnie modyfikujemy plik *main.c* do postaci:

```
#include "Myfun.h"
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	Joy_Conf();
	while(1)
```

```
	{
		Led_OnOff(0, LedOn);
		delay_ms(100);
	}
}
```

Kompilujemy nasz program, wgrywamy i testujemy działanie. Jeżeli wszystko pracuje prawidłowo możemy przejść do obsługi wyświetlacza. W naszym zestawie edukacyjnym znajduje się wyświetlacz siedmiosegmentowy złożony z czterech modułów. Każdy moduł może wyświetlać jedną cyfrę i zbudowany jest z ośmiu diod LED. Siedem z nich służy do wyświetlania cyfry a ósma dioda to separator dziesiętny. Schemat połączenia wyświetlacza przedstawiono na zdjęciu.



Każdy z modułów posiada 8 wyprowadzeń podłączonych po jednym do każdej diody LED. Są one połączone równolegle pomiędzy modułami. Ponadto każdy moduł posiada jedno wyprowadzenie służące do wyboru modułu. W celu włączenia np. segmentu B w module 3 należy podać stan wysoki na wyprowadzenie oznaczone *DIS_7_DISG3* aby aktywować moduł trzeci oraz stan wysoki na wyprowadzenie *DIS_7_B*. Poszczególne wyprowadzenia wyświetlacza podłączono do następujących wyprowadzeń mikrokontrolera: *DIS_7_DIS1 = GPIOB.2, DIS_7_DIS2 = GPIOB.3, DIS_7_DIS3 = GPIOB.4, DIS_7_DIS4 = GPIOB.5, DIS_7_A = GPIOG.0, DIS_7_B = GPIOG.1, DIS_7_C = GPIOG.2, DIS_7_D = GPIOG.3, DIS_7_E = GPIOG.4, DIS_7_F = GPIOG.5, DIS_7_G = GPIOG.6, DIS_7_DP = GPIOG.9.*

Zadanie 1:

Proszę utworzyć w pliku *Myfun.c* funkcję o nazwie *Led7seg_Conf(),* w której 12 wyprowadzeń mikrokontrolera potrzebnych do sterowania wyświetlaczem zostanie skonfigurowanych jako cyfrowe wyjścia. Proszę pamiętać o włączeniu taktowania zegarowego dla wszystkich używanych portów. Proszę pamiętać o umieszczeniu deklaracji funkcji w pliku nagłówkowym.

Następnie proszę wywołać utworzoną funkcję w funkcji *main()* i przetestować działanie. Proszę sprawdzić, czy wszystkie segmenty we wszystkich modułach pracują prawidłowo.

Uwaga: ponieważ niektóre z użytych tutaj wyprowadzeń wymagają dodatkowo włączenia zasilania należy na początku tej funkcji umieścić następujący fragment kodu:

```
void Led7seg_Conf(void)
{
	RCC->APB1ENR1 |= RCC_APB1ENR1_PWREN;
	PWR->CR2 |= PWR_CR2_IOSV;
//Tutaj proszę wpisać resztę kodu dla tej funkcji
}
```

Teraz napiszemy program pozwalającą w łatwy sposób wyświetlać pojedyncze cyfry na danym module wyświetlacza. W tym celu utworzymy funkcję według schematu:

```
void Led7seg_WriteDigit(uint8_t pos, uint8_t num)
{
	//tutaj proszę wpisać brakujący fragment kodu.
}
```

Funkcja ta przyjmuje dwa parametry: *pos* czyli numer modułu wyświetlacza od 0 do 3 oraz *num* czyli cyfrę, która ma zostać wyświetlona.

Zadanie 2:

Proszę uzupełnić ww. funkcję tak aby posiadała następujące zdolności: na początku ustawić w stan niski wszystkie wyprowadzenia wykorzystywane do sterowania wyświetlaczem, następnie proszę zaimplementować algorytm pozwalający na ustawienie w stan wysoki wyprowadzenia aktywującego odpowiedni moduł wyświetlacza, w zależności od wartości parametru *pos*, na zakończenie proszę zaimplementować algorytm pozwalający na aktywowanie odpowiednich wyprowadzeń mikrokontrolera, tak aby włączyć segmenty odpowiadające zadanej cyfrze (parametr *num*). Proszę pamiętać o umieszczeniu deklaracji funkcji w pliku nagłówkowym.

Zadanie 3:

Proszę wykorzystać napisaną wcześniej funkcję do wyświetlania kolejnych cyfr z odstępem czasowym 500ms na kolejnych modułach wyświetlacza. W tym celu można wykorzystać dwie zagnieżdżone pętle *for*.

Przygotowany w ten sposób kod pozwala na obsługę wyświetlacza jedynie w sposób statyczny. To znaczy nie pozwala na wyświetlanie różnych cyfr na różnych modułach „w tym samym czasie". Aby wyświetlać liczby składające się z kilku cyfr musimy zastosować dynamiczną obsługę wyświetlacza. W tym celu możemy wykonać wstępnie następujący eksperyment:

Zadanie 4:

Proszę napisać program, który w pętli *while(1){}* funkcji *main()* umieści kolejną pętle for(){} wyświetlająca na czterech modułach wyświetlacza różne cyfry, tak aby utworzyły liczbę czterocyfrową np. 1234. Wyświetlanie powinno się odbywać w taki sposób, aby w jednej iteracji pętli for była wyświetlana tylko jedna cyfra. Pomiędzy iteracjami należy dodać opóźnienie o wartości 100ms. Proszę przetestować działanie programu, a następnie sprawdzić jego działanie przy mniejszych czasach opóźnienie, aż do 1ms. Poniżej wstępny szablon takiego programu, w którym należy uzupełnić parametry funkcji *Led7seg_WriteDigit().*

```
#include "Myfun.h"
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	Joy_Conf();
	Led7seg_Conf();
	while(1)
	{
		for(uint8_t i=0;i<4;i++)
		{
			Led7seg_WriteDigit(.........);
			delay_ms(100);
		}
	}
}
```

Jak wynika z przeprowadzonych testów przy większych opóźnieniach pomiędzy iteracjami zauważalne jest migotanie wyświetlacza. Ponadto przy dynamicznej obsłudze każdy z modułów włączony jest przez zaledwie 25% czasu. Skutkuje to czterokrotnie niższą jasnością świecenia wyświetlacza. Jest to zjawisko, którego nie można uniknąć. Przygotujemy teraz mechanizm pozwalający na łatwą obsługę wyświetlacza w sposób dynamiczny.

Zadanie 5:

W pierwszej kolejności napiszemy funkcję, której zadaniem będzie rozłożenie przekazanej liczby na poszczególne cyfry i wpisanie ich do utworzonej tablicy. Następnie na odpowiedniej pozycji wyświetlacza zostanie wyświetlona tylko jedna cyfra. Globalna zmienna *led7seg_pos* przechowuje informację o aktualnie zajmowanej pozycji wyświetlacza. Natomiast globalna zmienna *led7seg_value* to liczba, która ma zostać wyświetlona. Liczba może posiadać maksymalnie cztery cyfry. Wstępny szablon takiej funkcji przedstawiono poniżej.

```
volatile uint8_t led7seg_pos;
volatile uint16_t led7seg_value;
void Led7seg_WriteNumber(void)
{
	uint8_t tab[4];
```

```
		//tutaj proszę wpisać kod, który do poszczególnych elementów tablicy tab wpisze wartości równe
		//poszczególnym cyfrom skłądowym parametru value
		Led7seg_WriteDigit(…………);
		//Tutaj proszę uzupełnić kod zwiększający wartość zmiennej ldpos o jeden przy każdym wywołaniu
		//tej funkcji. Należy pamiętać aby nie przekroczyć wartości 3
}
```

Tak przygotowaną funkcję będziemy wywoływać z częstotliwością 1kHz w przerwaniu od licznika *Systick*, co przedstawia poniższy kod.

```
void SysTick_Handler(void)
{
	tick++;
	Led7seg_WriteNumber();
}
```

Natomiast globalna zmienna *led7seg_value* może być zmieniana w dowolnym miejscu programu. Po wprowadzeniu do niej nowej wartości zostanie ona automatycznie wyświetlona na wyświetlaczu. Należy pamiętać, że aby zmienna ta była widoczna np. w pliku *main.c* lub innym pliku źródłowym, należy w tym pliku umieścić jej deklarację wraz ze słowem kluczowym *extern*, tak jak poniżej.

```
#include "Myfun.h"
extern volatile uint16_t led7seg_value;
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	Joy_Conf();
	Led7seg_Conf();
	while(1)
	{
		led7seg_value++;
		delay_ms(100);
	}
}
```

Zadanie domowe:

Proszę zastanowić się jak zmienić funkcje Led7seg_WriteDigit() i Led7seg_WriteNumber() tak aby możliwe było wyświetlanie liczb niecałkowitych.

Zajęcia nr 4 Obsługa wyświetlacza alfanumerycznego.

Wprowadzenie

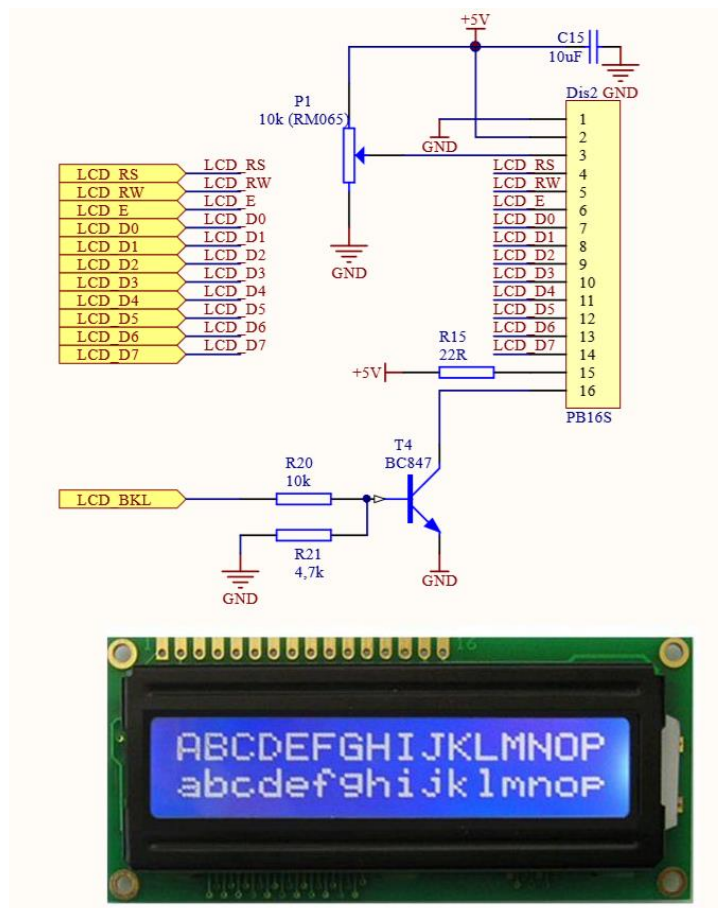
Podczas tych zajęć student zapozna się z działaniem wyświetlacza alfanumerycznego wyposażonego w sterownik HD44780. Student nauczy się obsługiwać taki wyświetlacz oraz wykorzystywać go do prezentacji stanu pracy mikrokontrolera.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków".

Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu trzy pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1"…* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie.

Przechodzimy teraz do zapoznania się z działaniem i obsługą wyświetlaczy alfanumerycznych ze sterownikiem HD44780. Jest to jeden z najbardziej popularnych sterowników dla takich wyświetlaczy, dlatego program, który przygotujemy będzie dopasowany do większości tego typu urządzeń. Nasz zestaw dydaktyczny wyposażony jest w taki wyświetlacz. Schemat podłączenia przedstawiono na poniższym rysunku. Wyświetlacz wyposażono w 16 wyprowadzeń. Część z nich służy do zasilania urządzenia, do zasilania podświetlenia lub do ustawienia kontrastu za pomocą potencjometru. Natomiast wyprowadzenia od 4 do 14 podłączone są do odpowiednich wyprowadzeń mikrokontrolera i służą do obsługi wyświetlacza. Wyprowadzenia od *LCD_D0* do *LCD_D7* służą do przesyłania równoległego jeden bajt danych do wyświetlacza lub z wyświetlacza do mikrokontrolera. W naszym przypadku wykorzystamy transmisje połówkową, czyli użyjemy tylko czterech starszych linii danych. Linia *LCD_RS* decyduje czy przesyłamy komendę czy wartość. Linia *LCD_RW* decyduje czy zapisujemy do wyświetlacza czy z niego czytamy. A linia *LCD_E* jest sygnałem strobe dla wyświetlacza.



W naszym programie wykorzystamy bibliotekę (a właściwie moduł składający się z dwóch plików: *HD44780.h i HD44780.c*). Biblioteka ta została pobrana ze strony Pana Radosława Kwietnia http://radzio.dxp.pl/ jako biblioteka dla mikrokontrolerów AVR. Została ona dopasowana do potrzeb mikrokontrolerów *STM32L4* przez prowadzącego zajęcia. Umieszczamy pliki z biblioteką w folderze z naszym projektem i dodajemy je do projektu. Następnie w pliku *main.c* dodajemy linię kodu dołączająca plik nagłówkowy *HD44780.h*. Przejdziemy teraz do omówienia działania najważniejszych elementów dołączonej biblioteki. W pliku nagłówkowym znajdują się makrodefinicje przypisujące poszczególne wyprowadzenia mikrokontrolera do poszczególnych wyprowadzeń wyświetlacza. Znajdują się tam również makrodefinicje poszczególnych komend dla wyświetlacza oraz deklaracje funkcji. Natomiast w pliku z kodem źródłowym znajdują się definicje wszystkich przygotowanych funkcji, w tym funkcji pomocniczych. Funkcja *LCD_delay50us()* służy do odmierzania pojedynczego odstępu czasu o długości 50us. Jej działanie zbliżone jest do działania znanej nam funkcji *delay_ms(),* jednak wykorzystuje inny układ licznikowy (co omówimy za chwilę). Funkcja *LCD_delayms(uint32_t ms)* bazuje na funkcji poprzedniej i odmierza czas w ms. Funkcja *LCD_Setbits(uint8_t byte)* ustawia linie *LCD_D4* do *LCD_D7* w odpowiednie stany na podstawie parametru funkcji. Funkcja *LCD_Write(uint8_t data)* zapisuje jedne bajt do

wyświetlacza w formie dwóch sekwencji po 4 bity. Funkcja *LCD_WriteCommand(uint8_t commandToWrite*) zapisuje komendę do wyświetlacza. Funkcja *LCD_WriteData(uint8_t dataToWrite)* zapisuje daną do wyświetlacza. Przedstawione powyżej funkcje są funkcjami pomocniczymi. Natomiast funkcja *LCD_Init()* służy do zainicjowania wyświetlacza odpowiednią sekwencją przesyłanych danych i komend. W tej funkcji uruchomiony jest licznik *TIM6*, który wykorzystywany jest do odmierzania czasu. Również w tej funkcji skonfigurowane są wszystkie niezbędne wyprowadzenia mikrokontrolera. Funkcja *TIM6_DAC_IRQHandler(void)* stanowi wektor przerwania dla *TIM6*. *Funckja LCD_Home()* służy do przenoszenia kursora na początek wiersza. Funkcja *LCD_GoTo(uint8_t x, uint8_t y)* służy do przenoszenia kursora w zadane miejsce. Funkcja *LCD_Clear()* służy do czyszczenia zawartości wyświetlacza. Natomiast funkcja *LCD_WriteText(char* text)* służy do wypisywania tekstu na wyświetlacz.

W tym momencie możemy przejść do uruchomienia wyświetlacza. W funkcji *main()* wywołujemy funkcję inicjującą wyświetlacz. Natomiast w pętli *while(1){}* możemy wypisać na LCD zadany tekst. Plik *main.c* powinien teraz wyglądać następująco:

```
#include "Myfun.h"
#include "HD44780.h"
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	Joy_Conf();
	LCD_Init();
	while(1)
	{
		LCD_Clear();
		LCD_WriteText("Kurs STM32L4!!!");
		Led_OnOff(0, LedOn);
		delay_ms(100);
	}
}
```

Zadanie 1:

Proszę przetestować działanie programu wpisując inny tekst. Proszę sprawdzić działanie programu przy wpisywaniu tekstu ale bez czyszczenia wyświetlacza. Proszę sprawdzić działanie pozostałych funkcji, takich jak *LCD_GoTo()* i *LCD_Home()*. Proszę utworzyć tablicę typu char zawierającą 16 elementów, wpisać do niej przykładowe wartości i przekazać ją jako parametr do funkcji wypisującej tekst.

Tak przygotowane oprogramowanie nie pozwala nam jednak na wyświetlanie zmiennych liczbowych. Musimy zatem napisać funkcję, która takie zdolności będzie posiadać.

Zadanie 2:

Proszę utworzyć w pliku *HD44780.c* funkcję według poniższego szablonu. Proszę pamiętać o umieszczeniu deklaracji funkcji w pliku nagłówkowym *HD44780.h.* Zadaniem tej funkcji jest wyświetlanie liczb całkowitych., Aby to zrobić takie liczby należy najpierw zamienić na tekst, a następnie wyświetlić za pomocą funkcji *LCD_WriteText()*. Należy zatem przygotować tablice typu char zawierającą co najmniej 16 elementów. Następnie można użyć jednej z wbudowanych funkcji z biblioteki *stdio.h* (tę bibliotekę należy najpierw dołączyć) jak np. *sprintf().* Na zakończenie wypełnioną tablicę należy przekazać jako parametr do funkcji wypisującej tekst.

```
void LCD_WriteNumber(int value)
{
	//tutaj proszę uzupełnić kod
}
```

Po przygotowaniu funkcji należy sprawdzić jej działanie. Proszę wyświetlić kolejne liczby z przedziału od 0 do 1000 z odstępem czasowym 10ms.

Zadanie 3:

Proszę napisać pogram odmierzający czas od włączenia mikrokontrolera i wyświetlający go na wyświetlaczu w formacie *h:m:s:ms* (godziny: minuty: sekundy: milisekundy).

Zadanie 4:

Proszę zmodyfikować powyższy program aby możliwe było zerowanie zegara z pomocą dowolnego przycisku, bez resetowania mikrokontrolera.

Zadanie 5:

Proszę zmodyfikować funkcję wyświetlającą liczby tak, aby możliwe było wyświetlanie liczb niecałkowitych. Można w tym celu nadal wykorzystywać funkcję *sprintf()* podając jako parametr specyfikator *„%f”*. Proszę również zmodyfikować tą funkcję tak, aby możliwe było wyświetlanie liczb w zadanym miejscu na wyświetlaczu.

Zadanie domowe:

Proszę zastanowić się jak wyświetlić na wyświetlaczu kolejne litery alfabetu tak jak na zdjęciu powyżej. Proszę zastanowić się czy jest możliwe zautomatyzowanie tego procesu, tak aby nie pisywać tekstu ręcznie.

Zajęcia nr 5 USART – obsługa podstawowa

Wprowadzenie

Podczas tych zajęć student zapozna się z działaniem interfejsu komunikacyjnego USART. Student zdobędzie praktyczną wiedzę dotyczącą obsługi tego układu peryferyjnego oraz nawiąże komunikację z komputerem klasy PC.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków”.

Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu te trzy pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1”…* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie.

Komunikacja bez wykorzystywania przerwań

Przystępujemy do skonfigurowania interfejsu *USART* do pracy. Nasz mikrokontroler posiada kilka interfejsów tego typu. My wykorzystamy interfejs *LPUART1*, którego dwie linie komunikacyjne dostępne są na wyprowadzeniach *GPIOC.0* (Rx) i *GPIOC*.1 (Tx). Podłączone są one do programatora STLink, dzięki czemu nie będziemy potrzebować żadnych dodatkowych urządzeń. Wystarczy, że podłączymy programator do gniazda USB w komputerze i w menedżerze urządzeń systemu Windows pojawi się nowy wirtualny port COM. Może to oczywiście wymagać instalacji sterowników. Następnie przystępujemy do obsługi interfejsu *LPUART1*. W pierwszym etapie uruchomimy interfejs w najprostszym trybie czyli bez wykorzystania przerwań i DMA. W tym celu tworzymy w pliku *Myfun.c* nową funkcję według poniższego szablonu. Należy pamiętać o umieszczeniu deklaracji funkcji w pliku nagłówkowym.

```
void LPUART1_Conf_Basic(void)
{
```

```
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOCEN;
	RCC->APB1ENR2		|= RCC_APB1ENR2_LPUART1EN;

	GPIOC->MODER		&= ~GPIO_MODER_MODER0 & ~GPIO_MODER_MODE1;
	GPIOC->MODER		|= GPIO_MODER_MODER0_1 | GPIO_MODER_MODE1_1;
	GPIOC->AFR[0]		|= 0x00000088;

	LPUART1->BRR		= (256 * 4000000) / 57600;
	LPUART1->CR1		|= USART_CR1_RE | USART_CR1_TE | USART_CR1_UE;
}
```

W funkcji tej mamy trzy grupy instrukcji. W pierwszej włączamy taktowanie dla *GPIOC*, taktowanie dla *LPUART1*. W drugiej konfigurujemy wyprowadzenia *GPIOC.0* i *GPIOC.1* w trybie *Alternate Function* oraz ze wskazaniem konkretnej alternatywnej funkcji. Natomiast w trzeciej grupie konfigurujemy sam interfejs. W pierwszej linijce ustawiamy prędkość komunikacji na 57600 bps (bitów na sekundę). Natomiast w drugiej włączamy część odbiorczą, nadawczą i cały interfejs. W tym momencie możemy rozpocząć komunikację. Wysyłanie pojedynczego bajta danych odbywa się poprzez wpisanie bajta do rejestru *LPUART1->TDR*. Odbiór polega na odczytaniu rejestru *LPUART1->RDR*.

Komunikację zaczniemy od wysyłania danych z zestawu dydaktycznego do komputera. W tym celu zmodyfikujemy plik *main.c*, tak aby umieścić w nim kod wysyłający pojedynczy bajt danych i przetestujemy działanie.

```
#include "Myfun.h"
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	LPUART1_Conf_Basic();
	while(1)
	{
		LPUART1->TDR = 123;
		Led_OnOff(0, LedTog);
		delay_ms(100);
	}
}
```

Następnie przygotujemy dwie funkcje pozwalające łatwiej wysyłać dłuższe sekwencje danych. Jednakże musimy zauważyć, że nie możemy wpisywać do rejestru *TDR* kilku bajtów jeden po drugim. Przed pisaniem kolejnego bajta musimy sprawdzić czy poprzedni został wysłany i czy rejestr jest pusty. W tym celu należy sprawdzić flagę *TXE* w rejestrze statusowym *ISR*. Możemy zatem napisać taką funkcję wysyłającą jeden bajt danych.

```
void ComSendChar(USART_TypeDef *USARTx, char c)
{
	while(!(USARTx->ISR & USART_ISR_TXE)){;}
		USARTx->TDR = c;
}
```

Funkcja ta przyjmuje dwa parametry: pierwszy to „nazwa usarta” a drugi do bajt danych do wysłania. Wadą tej funkcji, jak i całej pracy interfejsu w tak prostym trybie, jest oczekiwanie w bezczynności gdy poprzedni bajt nie został jeszcze wysłany. Możemy jeszcze napisać drugą funkcję, która korzysta z poprzedniej i służy do wysyłanie ciągu znaków. W tym miejscu musimy pamiętać o umieszczeniu deklaracji obydwu funkcji w pliku nagłówkowym.

```
void ComPuts(USART_TypeDef* USARTx, const char* str)
{
	while(*str)
		ComSendChar(USARTx, *str++);
}
```

Możemy teraz przetestować działanie tych funkcji wywołując je w pliku *main.c*.

```
#include "Myfun.h"
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	LPUART1_Conf_Basic();
	while(1)
	{
		ComPuts(LPUART1, "Kurs STM32L4. Politechnika Swietokrzyska\r\n");
		Led_OnOff(0, LedTog);
		delay_ms(100);
	}
}
```

Funkcję *ComPuts()* możemy wykorzystywać podobnie jak funkcję *Lcd_WriteText()* z poprzednich zajęć. Zatem możemy wysyłać liczby zamieniając je uprzednio na tekst, np. za pomocą funkcji *sprintf()*. Wykonanie odbioru danych w tak prostym trybie pracy interfejsu również nie jest optymalne i wymaga wstrzymania pracy procesora. Przed odczytaniem rejestru *RDR* musimy sprawdzić czy znajdują się tam jakieś dane czyli czy rejestr nie jest pusty. W tym celu sprawdzamy flagę *RXNE* w rejestrze statusowym *ISR* i jeżeli jest spełniony odpowiedni warunek to odczytujemy rejestr odbiorczy. Przedstawia to poniższy fragment kodu.

```
#include "Myfun.h"
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	LPUART1_Conf_Basic();
	while(1)
	{
		if((LPUART1->ISR & USART_ISR_RXNE) != RESET)
		{
			uint8_t data = LPUART1->RDR;
		}
		delay_ms(10);
	}
}
```

Zadanie 1:

Proszę przygotować program, który odbierze jeden bajt danych wysyłany z komputera i odeśle go z powrotem w formacie „Odebrałem: x", gdzie „x" to odebrany znak.

Zadanie 2:

Proszę napisać program, który będzie zliczał ile razy mikrokontroler odebrał literę „A" i odsyłał informację o aktualnym stanie licznika w formacie „Odebrałem x liter A".

Zadanie 3:

Proszę przygotować program, który pozwoli zdalnie uruchamiać diody LED w zestawie dydaktycznym. Użytkownik wysyła z komputera pojedynczy bajt danych o wartości od 0 do 7 co powoduje włączenie danej diody. Ponowne wysłanie tej samej liczby powoduje wyłączenie diody. Wysłanie liczby spoza zakresu skutkuje odesłaniem stosownego komunikatu.

Zajęcia nr 6 USART – obsługa z wykorzystaniem przerwań.

Wprowadzenie

Podczas tych zajęć student zapozna się z działaniem interfejsu komunikacyjnego *USART*. Student zdobędzie praktyczną wiedzę dotyczącą obsługi tego układu peryferyjnego oraz nawiąże komunikację z komputerem klasy PC.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków".

Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu te trzy pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1"…* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie.

Komunikacja z wykorzystaniem przerwań

Wysyłanie w najprostszym trybie wstrzymuje procesor, jednak jest to w pewnych sytuacjach akceptowalne, ponieważ to twórca oprogramowania decyduje kiedy dane będą wysyłane i może tak ułożyć strukturę programu, aby wstrzymanie procesora nie powodowało błędów w działaniu urządzenia. Przygotujemy teraz funkcję, która pozwoli nam wysyłać dane przez *LPUART1* z wykorzystaniem przerwań. Poniżej przedstawiono taką funkcję. Zmiana polega na ustawieniu w stan wysoki bitu *TXEIE* w rejestrze *CR1* oraz włączenia w kontrolerze przerwań *NVIC* zezwolenia dla przerwań od *LPUART1*. Jednakże włączenie przerwania *TXE* podczas konfiguracji sprawi, że dane zostaną od razu wysłane. Dlatego ustawienie i ewentualne wyczyszczenie bitu *TXEIE* będziemy realizować dopiero gdy będziemy chcieli wysyłać dane.

```
void LPUART1_Conf_Interrupt(void)
{
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOCEN;
	RCC->APB1ENR2		|= RCC_APB1ENR2_LPUART1EN;

	GPIOC->MODER		&= ~GPIO_MODER_MODE0 & ~GPIO_MODER_MODE1;
```

```
	GPIOC->MODER		|= GPIO_MODER_MODE0_1 | GPIO_MODER_MODE1_1;
	GPIOC->AFR[0]		|= 0x00000088;

	LPUART1->BRR		= (256 * 4000000) / 57600;
	LPUART1->CR1		|= USART_CR1_RE | USART_CR1_TE | USART_CR1_UE;
	NVIC_EnableIRQ(LPUART1_IRQn);
}
```

Tak przygotowany program zapewni generowanie przerwania za każdym razem, gdy rejestr nadawczy *LPUART1->TDR* będzie pusty. Będziemy mogli wtedy zapisać do niego kolejny bajt danych do wysłania. Potrzebujemy zatem jeszcze utworzyć tablicę z danymi do wysłania *bufwrite*, globalną zmienną będącą wskaźnikiem odczytu z tablicy *rp*, zmienną przechowującą informacje ile danych chcemy wysłać *rpmax* oraz musimy przygotować funkcję obsługi przerwania.

```
uint8_t bufwrite[100];
volatile uint16_t rp = 0;
volatile uint16_t rpmax = 10;
void LPUART1_IRQHandler(void)
{
	if((LPUART1->ISR & USART_ISR_TXE) != RESET)
	{
		LPUART1->TDR = bufwrite[rp];
		if(rp++ >= rpmax)
			LPUART1->CR1 &= ~USART_CR1_TXEIE;
		//Ewentualny dodatkowy kod
	}
}
```

W funkcji obsługi przerwania od *LPUART1* sprawdzane jest co jest źródłem przerwania. Jeżeli źródłem jest przerwanie *TXE* oznaczające pusty bufor nadawczy do rejestru *TDR* zapisywany jest kolejny bajt danych. Wskaźnik odczytu z tablicy jest inkrementowany. Gdy przekroczy on wartość zmiennej *rpmax* przerwanie zostanie wyłączone. Natomiast transmisja rozpoczynana jest za pomocą funkcji:

```
void LPUART1_SendWithInterrupt(const char* str, uint16_t len)
{
	for(int i=0;i<len;i++)
		bufwrite[i] = str[i];
	rp = 0;
	rpmax = len-1;
	LPUART1->CR1 |= USART_CR1_TXEIE;
}
```

Funkcja ta jako parametry przyjmuje dane do wysłania oraz ich liczbę. Dane do wysłania kopiowane są do tablicy *bufwrite*. Wskaźnik *rp* jest zerowany i uruchamiane jest zezwolenie na przerwanie od pustego bufora nadawczego. Transmisja rozpoczyna się natychmiast.

Aby przetestować to rozwiązanie należy użyć powyższej funkcji. Można to zrealizować następująco:

```
#include "Myfun.h"
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	LPUART1_Conf_Interrupt();
	while(1)
	{
		LPUART1_SendWithInterrupt("Dane do wyslania", 16);
		Led_OnOff(0, LedTog);
		delay_ms(100);
	}
}
```

Zadanie 1:

Przed przystąpienie do dalszych zajęć student proszony jest o przetestowanie działania dotychczasowego programu. Proszę zwrócić szczególną uwagę na wpływ wartości parametru *len* na ilość wysyłanych danych. Proszę przeanalizować co dzieje się w sytuacji, gdy ilość ta jest inna niż długość łańcucha podawanego jako pierwszy parametr. Proszę również sprawdzić działanie programu, gdy jako pierwszy parametr podawany jest wskaźnik do nowoutworzonej tablicy.

Skonfigurujemy teraz *LPUART1* w trybie pozwalającym odbierać dane z wykorzystaniem przerwań bez wstrzymywania procesora. Mamy możliwość skorzystania z trzech różnych źródeł przerwań dotyczących odbioru danych. Przerwanie *RXNE* jest generowane za każdym razem, gdy odebrany zostanie pojedynczy bajt. Przerwanie *IDLE* jest generowane jednorazowo gdy wystąpi stan bezczynności na linii odbiorczej. Natomiast przerwanie *CM* jest generowane gdy odebrany zostanie konkretny bajt danych. Zmodyfikujemy funkcję konfiguracyjną tak aby włączyć wszystkie wymienione przerwania.

```
void LPUART1_Conf_Interrupt(void)
{
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOCEN;
	RCC->APB1ENR2		|= RCC_APB1ENR2_LPUART1EN;

	GPIOC->MODER		&= ~GPIO_MODER_MODE0 & ~GPIO_MODER_MODE1;
	GPIOC->MODER		|= GPIO_MODER_MODE0_1 | GPIO_MODER_MODE1_1;
	GPIOC->AFR[0]		|= 0x00000088;

	LPUART1->BRR		= (256 * 4000000) / 57600;
	LPUART1->CR2		|= 123 << 24;
	LPUART1->CR1		|= USART_CR1_RE | USART_CR1_TE | USART_CR1_UE
				| USART_CR1_RXNEIE | USART_CR1_IDLEIE | USART_CR1_CMIE;
	NVIC_EnableIRQ(LPUART1_IRQn);
}
```

Zmiana polega na ustawieniu w stan wysoki bitów *RXNEIE, IDLEIE, CMIE* w rejestrze *CR1*. Ponadto przerwanie *CM* wymaga określenia wcześniej jakiego znaku oczekujemy. W tym celu należy zapisać wartość tego bajtu do rejestru *LPUART1->CR2*. W powyższym przykładzie

oczekujemy odebrania wartości 123. Musimy jeszcze przewidzieć realizację tych przerwań w funkcji obsługi przerwania.

```
void LPUART1_IRQHandler(void)
{
	if((LPUART1->ISR & USART_ISR_TXE) != RESET)
	{
		LPUART1->TDR = bufwrite[rp];
		if(rp++ >= rpmax)
			LPUART1->CR1 &= ~USART_CR1_TXEIE;
		//Ewentualny dodatkowy kod
	}
	if((LPUART1->ISR & USART_ISR_RXNE) != RESET)
	{
		uint8_t data = LPUART1->RDR;
		//Ewentualny dodatkowy kod
	}
	if((LPUART1->ISR & USART_ISR_IDLE) != RESET)
	{
		LPUART1->ICR |= USART_ICR_IDLECF;
		//Ewentualny dodatkowy kod
	}
	if((LPUART1->ISR & USART_ISR_CMF) != RESET)
	{
		LPUART1->ICR |= USART_ICR_CMCF;
		//Ewentualny dodatkowy kod
	}
}
```

Podczas obsługi przerwania od *RXNE* niezbędne jest odczytanie zawartości rejestru *RDR*, ponieważ powoduje to jednoczesne skasowanie flagi *RXNE* w rejestrze *ISR*. Podobna sytuacja ma miejsce podczas obsługi przerwania *TXE*. Wówczas zapis do rejestru *TDR* powoduje skasowanie flagi *TXE* w rejestrze statusowym *ISR*. Natomiast w pozostałych przerwaniach (*IDLE* i *CM*) musimy programowo skasować flagi poprzez zapis do rejestru *LPUART1->ICR*.

Zadanie 2:

Proszę przygotować program, który pozwoli zdalnie uruchamiać diody LED w zestawie dydaktycznym. Użytkownik wysyła z komputera pojedynczy bajt danych o wartości od 0 do 7 co powoduje włączenie danej diody. Ponowne wysłanie tej samej liczby powoduje wyłączenie diody. Wysłanie liczby spoza zakresu skutkuje odesłaniem stosownego komunikatu. Reakcja na odebrane dane powinna odbywać się w przerwaniu *RXNE*.

Zadanie 3:

Jest to bardziej zaawansowana wersja zadania numer 2. W tym zadaniu program ma zliczać ile dotychczas odebrano poszczególnych liter. Informacja ta ma być odsyłana w formie tabeli w przykładowym formacie:

A – 3, B – 7, C – 0.

Program powinien rozróżniać małe i wielkie litery.

Zadanie 4:

Jest to bardziej rozbudowana wersja zadania numer 2. Sterowanie diodami ma się odbywać poprzez wysyłanie bardziej złożonych sekwencji. Sekwencje powinny mieć następujący format:

#LED0_ON* - włączenie diody 0,

#LED7_OFF* - wyłączenie diody 7.

Specjalne znaki na początku (tzw. header) „#" i na końcu (tzw. terminator) „*" służą do oddzielenia poszczególnych komend od siebie. W celu realizacji tego zadania należy utworzyć tablicę na odbierane dane *bufread[100]* oraz zmienną globalną będącą wskaźnikiem zapisu do tablicy. Następnie w przerwaniu *RXNE* kopiować do tablicy odebrane dane i zwiększać wartość wskaźnika. Należy wykorzystać przerwanie CM wykrywające wystąpienie znaku terminatora i w tym przerwaniu zareagować na odebraną komendę.

Zajęcia nr 7 USART – obsługa z wykorzystaniem kontrolera DMA

Wprowadzenie

Podczas tych zajęć student zapozna się z działaniem kontrolera *DMA* (Direct Memory Access) i wykorzysta go do obsługi interfejsu komunikacyjnego *USART*, zarówno przy wysyłaniu jak i odbieraniu danych. Student zapozna się również ze sposobami na realizację komunikacji przy wykorzystaniu *DMA* i przerwań.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków".

Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu te trzy pliki klikając PPM na napis *Source Gropu 1* i wybierając opcję *Add Existing File to Group „Source Group 1"…* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie.

Wysyłanie danych z wykorzystaniem DMA

Mikrokontrolery STM32L496ZGT posiadają dwa niezależne kontrolery *DMA*. Każde z tych urządzeń posiada 7 kanałów (nazywanych w innych seriach mikrokontrolerów strumieniami). Zadaniem kontrolera jest kopiowanie danych pomiędzy obszarami pamięci. Ponieważ rejestry *TDR* i *RDR* dowolnego usarta są również pewnymi obszarami w pamięci możliwe jest takie skonfigurowanie kontrolera *DMA* aby zajmował się kopiowaniem danych z tych rejestrów lub do tych rejestrów. Ponadto *USART* jak i inne układy peryferyjne może ściślej współpracować z *DMA* wysyłając mu żądania do przeprowadzenia odpowiednich akcji.

W pierwszej kolejności zajmiemy się takim skonfigurowaniem kontrolera *DMA* aby z jego pomocą wysyłać dane przez *LPUART1*. Kontroler *DMA* ma 7 kanałów i jednocześnie może wykonywać operacje kopiowania w ramach jednego kanału. Natomiast w każdym kanale jest do ośmiu różnych żądań i podczas konfigurowania należy wybrać właściwe.

Table 46. Summary of the DMA2 requests for each channel

Request. number	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5	Channel 6	Channel 7
0	I2C4_RX	I2C4_TX	ADC1	ADC2	ADC3	DCMI	-
1	SAI1_A	SAI1_B	SAI2_A	SAI2_B	-	SAI1_A	SAI1_B
2	UART5_TX	UART5_RX	UART4_TX	-	UART4_RX	USART1_TX	USART1_RX
3	SPI3_RX	SPI3_TX	-	TIM6_UP DAC1	TIM7_UP DAC2	-	QUADSPI
4	SWPMI1_RX	SWPMI1_TX	SPI1_RX	SPI1_TX	DCMI	LPUART1_TX	LPUART1_RX
5	TIM5_CH4 TIM5_TRIG	TIM5_CH3 TIM5_UP	-	TIM5_CH2	TIM5_CH1	I2C1_RX	I2C1_TX
6	AES_IN	AES_OUT	AES_OUT	-	AES_IN	-	HASH_IN
7	TIM8_CH3 TIM8_UP	TIM8_CH4 TIM8_TRIG TIM8_COM	-	SDMMC1	SDMMC1	TIM8_CH1	TIM8_CH2

Na powyższym zdjęciu przedstawiono kanały i żądania dla *DMA2*. Można zauważyć, że *LPUART1_TX* dostępny jest w *DMA2* kanał 6 żądanie 4. Napiszemy teraz funkcję, która skonfiguruje do pracy *LPUART1* i *DMA2*.

```
volatile uint8_t dmabufwrite[100];
void LPUART1_Conf_DMA(void)
{
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOCEN;
	RCC->APB1ENR2		|= RCC_APB1ENR2_LPUART1EN;
	RCC->AHB1ENR		|= RCC_AHB1ENR_DMA2EN;

	GPIOC->MODER		&= ~GPIO_MODER_MODE0 & ~GPIO_MODER_MODE1;
	GPIOC->MODER		|= GPIO_MODER_MODE0_1 | GPIO_MODER_MODE1_1;
	GPIOC->AFR[0]		|= 0x00000088;

	LPUART1->BRR		= (256 * 4000000) / 57600;
	LPUART1->CR3		|= USART_CR3_DMAT | USART_CR3_DMAR;
	LPUART1->CR1		|= USART_CR1_RE | USART_CR1_TE | USART_CR1_UE;

	DMA2_Channel6->CPAR		= (uint32_t)&LPUART1->TDR;
	DMA2_Channel6->CMAR		= (uint32_t)dmabufwrite;
	DMA2_Channel6->CNDTR		= (uint16_t)100;
	DMA2_CSELR->CSELR		|= 0x00400000;//Channel 6
	DMA2_Channel6->CCR		|= DMA_CCR_MINC | DMA_CCR_DIR;
}
```

DMA zajmuje się kopiowaniem danych z jednego miejsca w pamięci do drugiego. Należy zatem utworzyć tablicę, w której będą się znajdować dane do wysłania. W tym przypadku jest to tablica *dmabufwrite*. W funkcji na początku włączono taktowanie dla *GPIOC*, *LPUART1* i *DMA2*. Następnie skonfigurowano kanał numer 6 ustawiając *LPUART1->TDR* jako miejsce, do którego *DMA* ma kopiować dane, *dmabufwrite* jako źródło danych. Ustawiono liczbę danych do jednorazowego przesłania na 100 i wybrano odpowiedni numer żądania. Włączono bit *MINC* (*Memory Increment*) oznaczający przesuwanie wskaźnika odczytu po tablicy *dmabufwrite* oraz ustalono kierunek kopiowania (z tablicy do *LPUART1*). Natomiast nie włączono jeszcze kontrolera *DMA*, ponieważ tablica nie została jeszcze wypełniona danymi.

Ponadto skonfigurowano odpowiednie wyprowadzenia portu *C* oraz skonfigurowano *LPUART1*. Nowością jest ustawienie bitu *DMAT* w rejestrze *LPUART1->CR3*. Bit ten zezwala na współpracę danego usarta z *DMA* przy wysyłaniu danych. W celu rozpoczęcia transmisji danych należy uzupełnić tablicę i uruchomić kontroler. W tym celu można napisać następującą funkcję:

```
void LPUART1_SendWithDMA(const char *str, uint16_t len)
{
	for(uint16_t i=0;i<len;i++)
		dmabufwrite[i] = str[i];
	DMA2_Channel6->CCR		&= ~DMA_CCR_EN;
	DMA2_Channel6->CNDTR	= (uint16_t)len;
	DMA2_Channel6->CCR		|= DMA_CCR_EN;
}
```

W funkcji tej dane do wysłania kopiowane są do tablicy. Następnie dany kanał *DMA* jest wyłączany aby móc go przekonfigurować. Czyszczona jest flaga, która mogła być sprzętowo ustawiona po zakończeniu poprzedniej transmisji, wprowadzana jest ilość danych do wysłania i uruchamiany jest kanał kontrolera. Funkcji tej możemy użyć następująco:

```
#include "Myfun.h"
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	LPUART1_Conf_DMA();
	while(1)
	{
		LPUART1_SendWithDMA("Wysylamy dane poprzez LPUART1 razem z DMA\r\n", 43);
		Led_OnOff(0, LedTog);
		delay_ms(100);
	}
}
```

Takie podejście do wysyłania „na pierwszy rzut oka” może wydawać się kłopotliwe i nic nie wnoszące, zwłaszcza w porównaniu z poznanymi wcześniej metodami. Jednakże jest ono najbardziej optymalne i wydajne zwłaszcza przy wysyłaniu dużych ilości danych. Od momentu uruchomienia transmisji procesor przestaje się zajmować obsługą interfejsu. Wszystkim zajmuje się *DMA*.

Zadanie 1:

Przed przystąpienie do dalszych zajęć student proszony jest o przetestowanie działania dotychczasowego programu. Proszę zwrócić szczególną uwagę na wpływ wartości parametru *len* na ilość wysyłanych danych. Proszę przeanalizować co dzieje się w sytuacji, gdy ilość ta jest inna niż długość łańcucha podawanego jako pierwszy parametr. Proszę również sprawdzić

działanie programu, gdy jako pierwszy parametr podawany jest wskaźnik do nowoutworzonej tablicy.

Oczywiście samo *DMA* również może zgłaszać przerwania. W tym przypadku najbardziej istotnymi są przerwania sygnalizujące zakończenie wysyłania połowy lub całego bloku danych (bity odpowiednio *HTIE* i *TCIE* w rejestrze *CCR* kontrolera *DMA*). Zmodyfikujmy funkcję konfiguracyjną tak, aby *DMA* zgłaszało przerwanie po zakończeniu wysyłania. Wystarczy ustawić bit *TCIE* i włączyć kontroler przerwań *NVIC*.

```
DMA2_Channel6->CCR  |= DMA_CCR_MINC | DMA_CCR_DIR | DMA_CCR_TCIE;
NVIC_EnableIRQ(DMA2_Channel6_IRQn);
```

Niezbędna jest również funkcja obsługi przerwania, która powinna wyglądać następująco:

```
void DMA2_Channel6_IRQHandler(void)
{
	if((DMA2->ISR & DMA_ISR_TCIF6) != RESET)
	{
		DMA2->IFCR |= DMA_IFCR_CTCIF6;
		//Ewentualny dodatkowy kod
	}
}
```

W funkcji tej czyścimy flagę przerwania, co również było robione wcześniej, gdy nie korzystaliśmy z przerwań, w funkcji wysyłającej.

Zadanie 2:

Proszę uruchomić program z obsługą przerwań i w przerwaniu od końca transmisji rozpocząć nowe wysyłanie. Proszę zastanowić się czym różni się działanie tego programu od poprzedniego.

Odbieranie danych z wykorzystaniem DMA i przerwań

Oczywiście możliwe jest wykorzystanie *DMA* do odbierania danych z interfejsu *USART*. Niezbędne jest skonfigurowanie odpowiedniego kanału, który współpracuje z *LPUART1_RX*. W tym wypadku będzie to kanał 7 żądanie 4. Niezbędne jest również utworzenie globalnej tablicy, do której *DMA* będzie kopiować odebrane dane. Poniżej funkcja posiadając odpowiednią konfiguracją.

```
volatile uint8_t dmabufwrite[100];
volatile uint8_t dmabufread[100];
void LPUART1_Conf_DMA(void)
{
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOCEN;
	RCC->APB1ENR2		|= RCC_APB1ENR2_LPUART1EN;
	RCC->AHB1ENR		|= RCC_AHB1ENR_DMA2EN;

	GPIOC->MODER		&= ~GPIO_MODER_MODE0 & ~GPIO_MODER_MODE1;
```

```
	GPIOC->MODER		|= GPIO_MODER_MODE0_1 | GPIO_MODER_MODE1_1;
	GPIOC->AFR[0]		|= 0x00000088;

	LPUART1->BRR		= (256 * 4000000) / 57600;
	LPUART1->CR3		|= USART_CR3_DMAT | USART_CR3_DMAR;
	LPUART1->CR1		|= USART_CR1_RE | USART_CR1_TE | USART_CR1_UE |
				USART_CR1_IDLEIE;
	NVIC_EnableIRQ(LPUART1_IRQn);

	DMA2_Channel6->CPAR		= (uint32_t)&LPUART1->TDR;
	DMA2_Channel6->CMAR		= (uint32_t)dmabufwrite;
	DMA2_Channel6->CNDTR		= (uint16_t)100;
	DMA2_CSELR->CSELR		|= 0x00400000;//Channel 6
	DMA2_Channel6->CCR		|= DMA_CCR_MINC | DMA_CCR_DIR | DMA_CCR_TCIE;
	NVIC_EnableIRQ(DMA2_Channel6_IRQn);

	DMA2_Channel7->CPAR		= (uint32_t)&LPUART1->RDR;
	DMA2_Channel7->CMAR		= (uint32_t)dmabufread;
	DMA2_Channel7->CNDTR		= (uint16_t)100;
	DMA2_CSELR->CSELR		|= 0x04000000;//Channel 7
	DMA2_Channel7->CCR		|= DMA_CCR_MINC | DMA_CCR_EN;
}
```

Podobnie jak poprzednio mamy tutaj określony adres peryferyjny (*LPUART1->RDR*), adres w pamięci (*bufread*), maksymalną ilość odbieranych danych (100), włączoną inkrementację wskaźnika w tablicy. Domyślnym kierunkiem kopiowania jest od układu peryferyjnego do tablicy w pamięci, więc nie musimy tutaj nic zmieniać. Ponadto, *DMA* jest od razu włączone, ponieważ od razu możemy spodziewać się przychodzących danych.

Jeżeli skonfigurowalibyśmy przerwanie od końca transmisji (bit *TCIE*), to *DMA* zgłosiłoby je po odebraniu 100 bajtów danych (ustawione w rejestrze *CNDTR*). Jednakże nigdy nie wiemy ile danych zostanie do nas przysłanych, dlatego skonfigurowaliśmy do pracy *LPUART1* z przerwaniem od stanu bezczynności na linii (bit *IDLEIE*). W tej sytuacji *DMA* będzie na bieżąco kopiować odbierane dane, natomiast *USART* zgłosi przerwanie, gdy dane przestaną spływać (pojawi się bezczynność na linii odbiorczej). Funkcja obsługi przerwania od *LPUART1* powinna wyglądać następująco:

```
void LPUART1_IRQHandler(void)
{
	if((LPUART1->ISR & USART_ISR_IDLE) != RESET)
	{
		//Ewentualny pozostały kod
		LPUART1_ReinitDMA();
		LPUART1->ICR |= USART_ICR_IDLECF;
	}
}
```

W funkcji obsługi przerwania powinniśmy reinicjować *DMA*, tak aby mogło ponownie odbierać dane. Powinniśmy również dokonać analizy danych które przyszły i wyczyścić bufor

odbiorczy. Funkcja ponownie inicjalizująca *DMA* sprowadza się do wyczyszczenia odpowiedniej flagi, gdyż możliwe, że została ustawiona i może wyglądać następująco:

```
void LPUART1_ReinitDMA(void)
{
	DMA2_Channel7->CCR		&= ~DMA_CCR_EN;
	DMA2_Channel7->CNDTR	= 100;
	DMA2_Channel7->CCR		|= DMA_CCR_EN;
}
```

Zadanie 3:

Proszę przygotować program pozwalający sterować diodami za pomocą następujących komend:

LED0_ON - włączenie diody 0,

LED7_OFF - wyłączenie diody 7.

Proszę zauważyć, że komendy te nie posiadają specjalnych znaków na początku (tzw. header) i na końcu (tzw. terminator) służących do oddzielenia poszczególnych komend od siebie, znanych z poprzednich zajęć. Należy zatem wykorzystać przerwanie IDLE wykrywające wystąpienie bezczynności na linii odbiorczej, a tym samym wykrywające koniec komendy i w tym przerwaniu zareagować na odebraną komendę.

Zajęcia nr 8 I2C – obsługa podstawowa. Akcelerometr LSM303C

Wprowadzenie

Podczas tych zajęć student zapozna się z działaniem interfejsu komunikacyjnego *I2C*. Student zdobędzie praktyczną wiedzę dotyczącą obsługi tego układu peryferyjnego oraz nawiąże komunikację z układem scalonym *LSM303C* zawierającym akcelerometr 3DoF i magnetometr 3DoF.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków”.

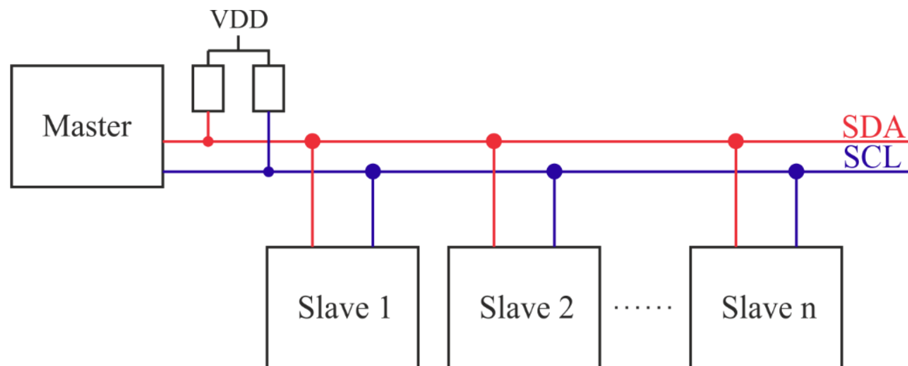
Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu te trzy pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1”…* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie.

Opis teoretyczny interfejsu I2C

I2C jest szeregową, synchroniczną magistralą komunikacyjną, pracującą w trybie half-duplex, mogącą przesyłać dane z częstotliwością do 100kHz (*Normal Mode*), 400 kHz (*Fast Mode*) lub 1000 kHz (*Fast Model Plus*). W najprostszym trybie pracy do magistrali podłączone jest jedno urządzenie master inicjujące transmisję i jedno lub wiele urządzeń *slave*. Możliwa jest również praca typu multimaster. Każde urządzenie slave posiada 7-bitowy unikatowy adres (możliwa praca z adresami 10-bitowymi). Adres 0x00 jest adresem rozgłoszeniowym. Magistrala składa się z dwóch linii: *SDA* – linia danych (ang. *Serial Data Line*) i *SCL* – linia zegara (ang. *Serial Clock Line*). Obydwie linie pracują w trybie *open drain* lub *otwarty kolektor*, co oznacza, że „1” zwalnia linię natomiast „0” ściąga linię do GND. Tym samym stan „0” jest stanem dominującym. Pozwala to na wykrywanie kolizji. Każde urządzenie nadając „1” jednocześnie sprawdza, czy na magistrali rzeczywiście pojawił się stan wysoki. Jeżeli tak nie jest, oznacza

to, iż inne urządzenie nadaje „0" w tym samym czasie i urządzenie zaprzestaje nadawania. Taka praca magistrali wymaga zastosowania dla każdej linii rezystora podciągającego *pull-up*. Poniżej przedstawiono schemat podłączania urządzeń do magistrali.



Komunikacja w magistrali odbywa się w jednym z dwóch kierunków:

- od urządzenia *master* do urządzenia *slave*,
- od urządzenia *slave* do urządzenia *master*.

Każda komunikacja rozpoczyna się od nadania bitu *STA* (start). Bit startu ma miejsce, gdy linia *SDA* zmienia swój stan z „1" na „0", podczas wysokiego stanu linii *SCL*. Nadanie bitu *STA* jest równoznaczne z rezerwacją magistrali przez dane urządzenie nadrzędne. Inne urządzenia muszą czekać z rozpoczęciem transmisji do czasu zakończenia obecnej. Po zakończeniu transmisji generowany jest bit stopu *STP*, czyli przejście linii *SDA* w stan wysoki przy wysokim stanie linii *SCL*. Zarówno bit *STA* jak i bit *STP* są zawsze generowane przez urządzenie *master*.

Zapis danych z układu *master* do układu *slave* odbywa się następująco. Wszystkie dane przesyłane są bajtowo po 8 bitów. Po sygnale startu *master* wysyła adres urządzenia *slave*. Adres ma zazwyczaj 7 bitów i nadawany jest począwszy od najmłodszego bitu. Natomiast najstarszy bit adresu ustawiony jest na „0", co oznacza, że *master* zapisuje dane do *slave*. Po każdym przesłanym bajcie urządzenie odbiorcze potwierdza jego otrzymanie bitem *ACK* (acknowledgment). Następnie *master* przesyła kolejne bajty danych otrzymując każdorazowo bit *ACK*. Po przesłaniu wszystkich danych *master* wystawia na magistralę bit *STP*, czym zwalnia magistralę i informuje układ *slave* o końcu transmisji.

Master	STA	Add+W		Data 1		Data n		STP
Slave			ACK		ACK		ACK	

Odczyt danych z układu *slave* do układu *master* odbywa się następująco. Wszystkie dane przesyłane są bajtowo po 8 bitów. Po sygnale startu master wysyła adres urządzenia *slave*. Adres ma zazwyczaj 7 bitów i nadawany jest począwszy od najmłodszego bitu. Natomiast najstarszy bit adresu ustawiony jest na „0", co oznacza, że *master* zapisuje dane do *slave*. Po każdym przesłanym bajcie urządzenie odbiorcze potwierdza jego otrzymanie bitem *ACK* (acknowledgment). Następnie *master* przesyła kolejne bajty danych otrzymując każdorazowo bit *ACK*. W bajtach tych mogą być zawarte np. informacje o tym jakie dane układ *slave* ma przygotować do transferu. Po przesłaniu wszystkich danych *master* ponownie wystawia na magistralę bit *STA*, czym informuje układ *slave* o nowej komunikacji. Po tym bicie *master* wysyła adres z najstarszym bitem ustawionym na „1" co oznacza odczyt z układu *slave*. Układ *slave* potwierdza bitem *ACK*, a następnie rozpoczyna transfer danych. Po każdym odebranym bajcie danych układ *master* potwierdza bitem ACK. Gdy *master* uzna, że odebrał już wszystkie dane ostatni bajt potwierdza bitem *STP* jednocześnie kończąc transmisję i zwalniając magistralę.

Master	STA	Add+W		Data 1		STA	Add+R			ACK		STP
Slave			ACK		ACK			ACK	Data 1		Data n	

Obsługa podstawowa interfejsu I2C w STM32L4

Mikrokontroler STM32L496ZGT6, w który wyposażono zestaw dydaktyczny, posiada cztery układy peryferyjne realizujące funkcje interfejsu *I2C*. Różnią się one między sobą szczegółami takimi jak możliwość wybudzania procesora. W zestawie dydaktycznym znajduje się czujnik MEMS LSM303C posiadający trzyosiowy akcelerometr i trzyosiowy magnetometr. Każde z tych urządzeń jest podłączone do magistrali *I2C3* naszego mikrokontrolera i jest osobnym układem *slave* z własnym adresem. Domyślny adres akcelerometru to 0x1d a magnetometru 0x1e. Magistrala posiada zewnętrzne rezystory podciągające *pull-up*. Podczas tych zajęć skonfigurujemy do pracy układ peryferyjny *I2C3* tak aby za jego pomocą odczytać wartość przyspieszeń poszczególnych osi akcelerometru. Akcelerometr posiada zestaw 27 ośmiobitowych rejestrów. Ich pełne zestawienie dostępne jest w dokumentacji układu. Poniżej przedstawiono kilka rejestrów, które musimy zapisać lub odczytać w celu poprawnej pracy z układem LSM303C.

Pierwszy rejestr to rejestr *WHO_AM_I_A* (adres rejestru 0x0f) zawierający stałą wartość 0x41. Można ją odczytać aby sprawdzić, czy nawiązaliśmy komunikację z układem.

WHO_AM_I_A (0Fh)

Accelerometer Who_AM_I register (r). This register is a read-only register. Its default value is 41h.

Table 20. WHO_AM_I_A register default value

0	1	0	0	0	0	0	1

Następnie w rejestrze konfiguracyjnym *CTRL_REG1_A* (adres rejestru 0x20) musimy ustawić częstotliwość odświeżania układu (bity *ODR2:0*) oraz włączyć pracę wszystkich trzech osi (bity *XEN*, *YEN* i *ZEN*). Dostępne częstotliwości odświeżania i odpowiadające kombinacje bitów *ODR* przedstawiono poniżej.

CTRL_REG1_A (20h)

Accelerometer control register 1 (r/w)

Table 23. CTRL_REG1_A register

HR	ODR2	ODR1	ODR0	BDU	ZEN	YEN	XEN

Table 25. ODR register setting

ODR2	ODR1	ODR0	ODR selection
0	0	0	Power down
0	0	1	10 Hz
0	1	0	50 Hz
0	1	1	100 Hz
1	0	0	200 Hz
1	0	1	400 Hz
1	1	0	800 Hz
1	1	1	N.A.

Warto również w rejestrze *CTRL_REG1_A* (adres rejestru 0x23) ustawić bit IF_ADD_INC dzięki czemu przy odczycie wielu rejestrów wskaźnik odczytu będzie automatycznie inkrementowany.

CTRL_REG4_A (23h)

Accelerometer control register 4 (r/w)

Table 31. CTRL_REG4_A register

BW2	BW1	FS1	FS0	BW_SCALE_ODR	IF_ADD_INC	I2C_DISABLE	SIM

Po skonfigurowaniu do pracy akcelerometru możemy już odczytać wartości przyspieszeń poszczególnych osi. Każda oś reprezentowana jest prze liczbę szesnastobitową (uint16_t) zapisaną w dwóch rejestrach ośmiobitowych o adresach od 0x28 do 0x2d, jak przedstawiono poniżej.

8.12 OUT_X_L_A (28h), OUT_X_H_A (29h)

Accelerometer x-axis output register (r)

Table 42. OUT_X_L_A register default values

0	0	0	0	0	0	0	0

Table 43. OUT_X_H_A register default values

0	0	0	0	0	0	0	0

8.13 OUT_Y_L_A (2Ah), OUT_Y_H_A (2Bh)

Accelerometer y-axis output register (r)

Table 44. OUT_Y_L_A register default values

0	0	0	0	0	0	0	0

Table 45. OUT_Y_H_A register default values

0	0	0	0	0	0	0	0

8.14 OUT_Z_L_A (2Ch), OUT_Z_H_A (2Dh)

Accelerometer z-axis output register (r)

Table 46. OUT_Z_L_A register default values

0	0	0	0	0	0	0	0

Table 47. OUT_Z_H_A register default values

0	0	0	0	0	0	0	0

Przystępujemy zatem do napisania funkcji konfiguracyjnej dla *I2C3*. Interfejs ten jest dostępny na wyprowadzenia *GPIOG.7* i *GPIOG.8*.

```
void I2C3_ConfBasic(void)
{
	RCC->APB1ENR1		|= RCC_APB1ENR1_PWREN;
	PWR->CR2		|= PWR_CR2_IOSV;
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOGEN;
	RCC->APB1ENR1		|= RCC_APB1ENR1_I2C3EN;

	GPIOG->MODER		&= ~GPIO_MODER_MODE8 & ~GPIO_MODER_MODE7;
	GPIOG->MODER		|= GPIO_MODER_MODE8_1 | GPIO_MODER_MODE7_1;
	GPIOG->AFR[0]		= 0x40000000;
	GPIOG->AFR[1]		|= 0x00000004;

	I2C3->TIMINGR		= 0x04;
	I2C3->CR1		= I2C_CR1_PE;
}
```

Użyte tutaj wyprowadzenia wymagają dodatkowo włączenia zasilania. Następnie włączamy taktowanie dla *GPIOG* oraz *I2C3*. W dalszej kolejności konfigurujemy wyprowadzenia w trybie *Alternate Function* oraz wybieramy odpowiedni numer *AF*. Pozostaje nam już tylko skonfigurować *I2C3*. Rejestr *TIMINGR* zawiera współczynniki odpowiedzialne za określenie prędkości pracy interfejsu. Ich wyznaczenie jest szczegółowo opisane w dokumentacji, jednak z uwagi na trudność ich wyznaczenia producent sugeruje skorzystanie z gotowych narzędzi wyliczających zawartość rejestru *TIMINGR*. Takim narzędziem jest darmowe oprogramowanie *STM32CubeMx* przygotowane przez producenta mikrokontrolerów. Akcelerometr może się komunikować z maksymalną częstotliwością 400kHz. Zatem po skonfigurowaniu w oprogramowaniu takiej prędkości, dla taktowania procesora częstotliwością 4MHz otrzymujemy, że do rejestru *TIMINGR* powinniśmy wpisać wartość 0x04. Następnie włączamy interfejs ustawiając bit *I2C_CR1_PE* w rejestrze *CR1*.

Następnie przygotujemy funkcję odczytującą zawartość rejestru *WHO_AM_I_A*.

```
uint8_t whoacc;
void LSM303_ACC_ReadWhoAmI(void)
{
	I2C3->CR2 = (1 << I2C_CR2_NBYTES_Pos) | (0x1d << (I2C_CR2_SADD_Pos + 1)) |
	I2C_CR2_START;

	while((I2C3->ISR & I2C_ISR_TXIS) == RESET)	{;}
	I2C3->TXDR = 0x0f;

	while((I2C3->ISR & I2C_ISR_TC) == RESET)	{;}
	I2C3->CR2 = (1 << I2C_CR2_NBYTES_Pos) | I2C_CR2_AUTOEND | (0x1d <<
	(I2C_CR2_SADD_Pos + 1)) | I2C_CR2_RD_WRN | I2C_CR2_START;

	while((I2C3->ISR & I2C_ISR_RXNE) == RESET)	{;}
	whoacc = I2C3->RXDR;
}
```

W celu rozpoczęcia komunikacji należy w rejestrze *CR2* ustawić wartość adresu układu *slave*, ustawić liczbę bajtów do wysłania (liczba ta nie uwzględnia bajtu adresu *slave*) oraz ustawić bit *I2C_CR2_START*. Po wykonaniu tego polecenia interfejs wyśle bit *STA*, następnie wyśle bit adresu, po otrzymaniu potwierdzenia *ACK* rozpocznie wysyłanie 1 bajtu danych. Zanim to nastąpi zostanie ustawiona flaga *I2C_ISR_TXIS* oznaczająca pusty rejestr nadawczy. Musimy wtedy uzupełnić rejestr *TXDR* wartością do wysłania. W tym przypadku jest to adres rejestru, który chcemy odczytać 0x0f. Następnie czekamy na zakończenie transmisji, co zostanie zasygnalizowane ustawieniem flagi *I2C_ISR_TC*. Następnie ponownie przygotowujemy transmisję. Do rejestru *CR2* wpisujemy liczbę bajtów, które chcemy odczytać, podajemy adres układu *slave*, ustawiamy bit *I2C_CR2_AUTOEND* oznaczający, że po zakończeniu transmisji ma zostać automatycznie wygenerowany bit *STP*, ustawiamy bit *I2C_CR2_RD_WRN*

oznaczający kierunek przesyłu od *slave* do *master* i ustawiamy bit *I2C_CR2_START* rozpoczynając transmisje. Następnie czekamy, aż układ *slave* prześle do nas jeden bajt danych. Zasygnalizuje to flaga *I2C_ISR_RXNE*. Po jej ustawieniu odczytujemy zawartość rejestru *RXDR*. Powinna się tam znajdować wartość 0x41.

Jeżeli wszystko przebiegło prawidłowo i otrzymaliśmy prawidłową wartość należy przystąpić do skonfigurowania akcelerometru. W tym celu napiszemy funkcję *LSM303_ACC_WriteConfig()*, która dokona zapisu do rejestrów konfiguracyjnych *CTRL_REG1_A* (adres rejestru 0x20) i *CTRL_REG4_A* (adres rejestru 0x23), Wpisując do nich odpowiednio wartości 0x67 i 0x04. Poniżej przedstawiono pierwszą część funkcji.

```
void LSM303_ACC_WriteConfig(void)
{
	I2C3->CR2 = (2 << I2C_CR2_NBYTES_Pos) | I2C_CR2_AUTOEND | (0x1d <<
	(I2C_CR2_SADD_Pos + 1)) | I2C_CR2_START;

	while((I2C3->ISR & I2C_ISR_TXIS) == RESET)    ;
	I2C3->TXDR = 0x20;

	while((I2C3->ISR & I2C_ISR_TXIS) == RESET)    ;
	I2C3->TXDR = 0x67;

	//Dalszy program zapisujący do drugiego rejestru
}
```

Zadanie 1:

Student powinien uzupełnić powyższą funkcję o fragment programu zapisujący dane do rejestru *CTRL_REG4_A* (adres rejestru 0x23, wartość 0x04).

Wszystkie powyższe funkcje należy uruchomić raz poprzez wywołanie ich w funkcji *main().*

```
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	I2C3_ConfBasic();
	LSM303_ACC_ReadWhoAmI();
	LSM303_ACC_WriteConfig();
	while(1)
	{
		Led_OnOff(0, LedTog);
		delay_ms(100);
	}
}
```

Następnie należy przygotować funkcję odczytującą wartość przyspieszenia wzdłuż osi X. W tym celu należy odczytać zawartość dwóch rejestrów: *OUT_X_L_*A (bajt młodszy, adres rejestru 0x28) i *OUT_X_H_A* (bajt starszy, adres rejestru 0x29) a następnie z dwóch zmiennych ośmiobitowych bez znaku (typ uin8_t) należy uzyskać jedną zmienną szesnastobitową ze

znakiem (typ int16_t). W tym celu należy wykorzystać operacje przesunięcia bitowego. Zarys funkcji przedstawiono poniżej. Przedstawiono również jej wykorzystanie polegające na cyklicznym wywoływaniu.

```
int16_t accx = 0;
void LSM303_ACC_Read(void)
{
	//uzupełnić o odczyt przyspieszenia wzdłuż osi X
}
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	I2C3_ConfBasic();
	LSM303_ACC_ReadWhoAmI ();
	LSM303_ACC_WriteConfig();
	while(1)
	{
		LSM303_ACC_Read();
		Led_OnOff(0, LedTog);
		delay_ms(10);
	}
}
```

Zadanie 2:

Student powinien napisać i przetestować działanie funkcji odczytującej zawartość dwóch rejestrów przechowujących wartość przyspieszenia wzdłuż osi X.

Zadanie 3:

Student powinien uzupełnić funkcję odczytującą o możliwość czytania wartości przyspieszenia wzdłuż pozostałych dwóch osi.

Zadanie 4:

Domyślnie akcelerometr mierzy przyspieszenie w zakresie -2g do 2g. Przy pionowym ustawieniu dowolnej osi powinna ona wskazać przyspieszenie równe -1g lub 1g oznaczające wartość przyspieszenia ziemskiego. Proszę zatem o przeskalowanie otrzymanych wcześniej wartości na zmienne niecałkowite w zakresie od -180,0 do 180,0 reprezentujące kąt nachylenia danej osi.

Zadanie 5:

Proszę spróbować przygotować drugi zestaw funkcji pozwalających na odczytanie wartości pola magnetycznego z magnetometru (*LSM303_MAG_ReadWhoAmI(), LSM303_MAG_WriteConfig(), LSM303_MAG_Read()*). Domyślny adres *slave* magnetometru to 0x1e, natomiast wartość rejestru *WHO_AM_I_M* wynosi 0x3d.

Zajęcia nr 9 I2C – przerwania. Akcelerometr LSM303C

Wprowadzenie

Podczas tych zajęć student zapozna się z działaniem interfejsu komunikacyjnego *I2C*. Student zdobędzie praktyczną wiedzę dotyczącą obsługi tego układu peryferyjnego z wykorzystaniem mechanizmu przerwań oraz nawiąże komunikację z układem scalonym *LSM303C* zawierającym akcelerometr 3DoF i magnetometr 3DoF.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków”.

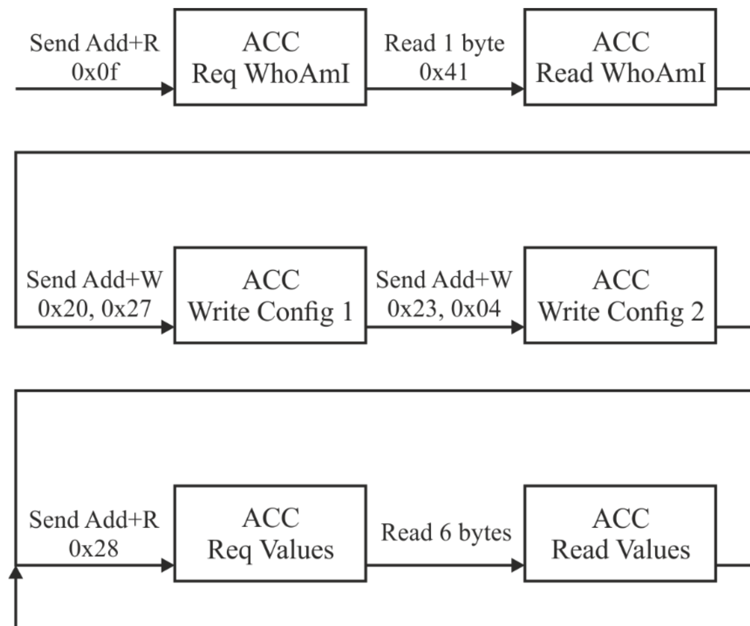
Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu te trzy pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1”…* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie.

Obsługa LSM303C z wykorzystaniem przerwań

Obsługa interfejsu komunikacyjnego *I2C* z wykorzystaniem przerwań może być znacznie trudniejsza niż obsługa interfejsu *USART* w oparciu o przerwania. Wynika to między innymi z faktu, że w *I2C* podczas transmisji układ odbiorczy musi każdorazowo potwierdzać odebranie pojedynczego bajtu danych wystawiając bit *ACK*, a układ nadawczy musi sprawdzić wystąpienie tego bitu. Ponadto odczyt danych z układu *slave* odbywa się poprzez rozpoczęcie komunikacji z trybie zapisu do *slave* z żądaniem przygotowania danych a następnie ponowny restart transmisji w trybie odczytu ze *slave*. Wymusza to na programiście przygotowanie mechanizmu maszyny stanu i odpowiedniego zarządzania aktualnym stanem magistrali *I2C*. Na poniższym rysunku przedstawiono graficznie schemat działania magistrali *I2C* w mikrokontrolerze *STM32L4* obsługującej akcelerometr wbudowany w układ *LSM303C*. Praca rozpoczyna się od odczytu zawartości rejestru *WHO_AM_I_A*. W tym celu należy

przesłać do *LSM303C* żądanie odczytu z rejestru 0x0f a następnie odczytać jeden bajt danych z *I2C*. Dalej należy zapisać do rejestru *CTRL_REG_1_A* wartość 0x67 i do rejestru *CTRL_REG_4_A* wartość 0x04. Po tym etapie można przystąpić do cyklicznego odczytu danych z układu. Należy cyklicznie wysyłać żądanie przesłania danych z adresu 0x28 a następnie odczytywać 6 bajtów z *I2C*.



Każdy z wymienionych etapów transmisji rozpoczyna się od odpowiedniej konfiguracji interfejsu *I2C* i ustawienia bitu *I2C_CR2_START*. Przed rozpoczęciem kolejnego etapu komunikacji należy odczekać na zakończenie poprzedniego co sygnalizowane jest ustawieniem flagi *TC* w rejestrze statusowym *ISR*. Jeżeli odpowiednie przerwanie jest włączone, zostanie ono wygenerowane.

Interfejs *I2C* w *STM32L4* posiada kilka źródeł przerwań: *TXE*, *TXIS*, *RXNE*, *ADDR*, *NACKF*, *STOPF, TC*. Podczas komunikacji z *LSM303C* wykorzystamy trzy z nich:

- *TXIS* – przerwanie generowane gdy rejestr nadawczy *TXDR* jest pusty i wymaga zapisu,
- *RXNE* – przerwanie generowane gdy rejestr odbiorczy *RXDR* jest pełny i wymaga odczytu,
- *TC* – przerwanie generowane gdy zakończono transmisję.

Utworzymy w pliku nagłówkowym typ wyliczeniowy z nazwami stanów magistrali, tak jak pokazano poniżej.

```
typedef enum{AccMag_Idle = 0, Acc_ReqWhoAmI, Acc_ReadWhoAmI, Acc_WriteConfig1,
Acc_WriteConfig2, Acc_ReqValues, Acc_ReadValues}eLSM303;
```

Następnie w pliku z kodem źródłowym utworzymy zmienną przechowującą aktualny stan magistrali *I2C lsm303_state*, tablicę zawierającą dane do wysłania *lsm303_tabw*, tablicę zawierającą dane odebrane z magistrali *lsm303_tabr*, zmienną przechowującą numer elementu w tablicy z danymi do wysłania *lsm303_wp* i zmienną przechowującą numer elementu w tablicy z danymi odebranymi *lsm303_rp*. Zdefiniujemy również tablicę na dane odebrane z akcelerometru *tabacc*, dane odebrane z magnetometru *tabmag* oraz zmienne przechowujące wartości z akcelerometru i magnetometru dla poszczególnych osi: *accx*, *accy*, *accz*, *magx*, *magy* i *magz*.

```
eLSM303     lsm303_state = AccMag_Idle;
uint8_t     lsm303_tabw[10];
uint8_t     lsm303_tabr[10];
uint16_t    lsm303_wp = 0;
uint16_t    lsm303_rp = 0;
uint8_t     tabacc[6];
uint8_t     tabmag[6];
int16_t     accx = 0;
int16_t     accy = 0;
int16_t     accz = 0;
int16_t     magx = 0;
int16_t     magy = 0;
int16_t     magz = 0;
```

Domyślny stan magistrali to *AccMag_Idle*. Następnie napiszemy funkcję konfigurującą *I2C3* w postaci:

```
void I2C3_ConfInterrupt(void)
{
	RCC->APB1ENR1		|= RCC_APB1ENR1_PWREN;
	PWR->CR2		|= PWR_CR2_IOSV;
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOGEN;
	RCC->APB1ENR1		|= RCC_APB1ENR1_I2C3EN;

	GPIOG->MODER		&= ~GPIO_MODER_MODE8 & ~GPIO_MODER_MODE7;
	GPIOG->MODER		|= GPIO_MODER_MODE8_1 | GPIO_MODER_MODE7_1;
	GPIOG->AFR[0]		|= 0x40000000;
	GPIOG->AFR[1]		|= 0x00000004;

	I2C3->TIMINGR		= 0x04;
	I2C3->CR1		= I2C_CR1_PE;
	NVIC_EnableIRQ(I2C3_EV_IRQn);
}
```

Jest ona bardzo podobna do funkcji z poprzednich zajęć. Zawiera dodatkowo instrukcję konfigurująca dla kontrolera przerwań. Nie zawiera ona natomiast włączenia poszczególnych przerwań. Wykonamy to przy inicjowaniu pierwszej transmisji do *LSM303C*. Potrzebujemy również funkcji obsługi przerwania dla *I2C3*. Zawierać ona będzie obsługę trzech źródeł przerwań: *TXIS*, *RXNE* i *TC*. W przerwaniu *TXIS* jesteśmy zobowiązani do wpisania do rejestru *TXDR* kolejnej wartość z tablicy z danymi do wysłania *lsm303_tabw*. W przerwaniu *RXNE*

jesteśmy zobowiązani do odczytu zawartości rejestru *RXDR* i zapisaniu tej danej do tablicy *lsm303_tabr*. W obydwu przypadkach musimy inkrementować wartości zmiennych przechowujących numery elementów w tablicy i pilnować aby nie przekroczyć rozmiaru danej tablicy. Natomiast w przerwaniu od *TC* zawarta będzie cała logika działania magistrali *I2C* przedstawiona na powyższym grafie. W przerwaniu tym sprawdzane jest jaki etap pracy magistrali właśnie zakończono i uruchamiany jest kolejny.

```
void I2C3_EV_IRQHandler(void)
{
	if((I2C3->ISR & I2C_ISR_TXIS) != RESET)
	{
		I2C3->TXDR = lsm303_tabw[lsm303_wp];
		if(lsm303_wp++ >= 10)
			lsm303_wp = 0;
	}
	if((I2C3->ISR & I2C_ISR_RXNE) != RESET)
	{
		lsm303_tabr[lsm303_rp] = I2C3->RXDR;
		if(lsm303_rp++ >= 10)
			lsm303_rp = 0;
	}
	if((I2C3->ISR & I2C_ISR_TC) != RESET)
	{
		if(lsm303_state == Acc_ReqWhoAmI)
			LSM303_Interrupt_ACC_ReadWhoAmI();
		else if(lsm303_state == Acc_ReadWhoAmI)
			LSM303_Interrupt_ACC_WriteConfig1();
		else if(lsm303_state == Acc_WriteConfig1)
			LSM303_Interrupt_ACC_WriteConfig2();
		else if(lsm303_state == Acc_WriteConfig2)
			LSM303_Interrupt_ACC_ReqValues();
		else if(lsm303_state == Acc_ReqValues)
			LSM303_Interrupt_ACC_ReadValues();
		else if(lsm303_state == Acc_ReadValues)
		{
			LSM303_Interrupt_ACC_Values();
			LSM303_Interrupt_ACC_ReqValues();
		}
	}
}
```

W celu uruchomienia każdego z etapów przygotowano odpowiednią funkcję, które opisano poniżej:

- *LSM303_Interrupt_ACC_ReqWhoAmI()* – wysłanie żądania odczytu rejestru *WHO_AM_I_A*, funkcja uruchamiany jednorazowo na początku w pliku *main.c*,
- *LSM303_Interrupt_ACC_ReadWhoAmI()* – odczyt zawartości rejestru *WHO_AM_I_A,*
- *LSM303_Interrupt_ACC_WriteConfig1()* – zapis do rejestru *CTRL_REG_1_A,*
- *LSM303_Interrupt_ACC_WriteConfig2()* – zapis do rejestru *CTRL_REG_4_A,*

- *LSM303_Interrupt_ACC_ReqValues()* – wysłanie żądania odczytu wartości z akcelerometru,
- *LSM303_Interrupt_ACC_ReadValues()* – odczyt wartości z akcelerometru
- *LSM303_Interrupt_ACC_Values()* – przeliczenie wartości z akcelerometru z 6 x uint8_t na 3 x int16_t (ta funkcja nie jest bezpośrednio związana z *I2C*).

Poniżej przedstawiono implementację poszczególnych funkcji. Niektóre z nich wymagają od studenta samodzielnego uzupełnienia.

```
void LSM303_Interrupt_ACC_ReqWhoAmI(void)
{
	lsm303_tabw[0]	= 0x0f;
	lsm303_wp	= 0;
	I2C3->CR1	|= I2C_CR1_TXIE | I2C_CR1_TCIE | I2C_CR1_RXIE;
	I2C3->CR2	= (1 << I2C_CR2_NBYTES_Pos) | (0x1d << (I2C_CR2_SADD_Pos + 1)) |
			I2C_CR2_START;
	lsm303_state	= Acc_ReqWhoAmI;
}
void LSM303_Interrupt_ACC_ReadWhoAmI(void)
{
	lsm303_rp	= 0;
	I2C3->CR2	= (1 << I2C_CR2_NBYTES_Pos) | (0x1d << (I2C_CR2_SADD_Pos + 1)) |
			I2C_CR2_RD_WRN | I2C_CR2_START;
	lsm303_state	= Acc_ReadWhoAmI;
}
void LSM303_Interrupt_ACC_WriteConfig1(void)
{
	lsm303_tabw[0]	= 0x20;
	lsm303_tabw[1]	= 0x67;
	lsm303_wp	= 0;
	I2C3->CR2	= (2 << I2C_CR2_NBYTES_Pos) | (0x1d << (I2C_CR2_SADD_Pos + 1)) |
			I2C_CR2_START;
	lsm303_state	= Acc_WriteConfig1;
}
void LSM303_Interrupt_ACC_WriteConfig2(void)
{
	//Uzupełnić samodzielnie
}
void LSM303_Interrupt_ACC_ReqValues(void)
{
	//Uzupełnić samodzielnie
}
void LSM303_Interrupt_ACC_ReadValues(void)
{
	//Uzupełnić samodzielnie
}
void LSM303_Interrupt_ACC_Values(void)
{
	//Uzupełnić samodzielnie
}
```

Po przygotowaniu wszystkich powyższych funkcji można przetestować działanie programu. W tym celu należy jednorazowo w pliku main.c uruchomić funkcję konfigurującą *I2C3* i funkcją inicjująca komunikację wysyłającą żądanie odczytu rejestru *WHO_AM_I_A*.

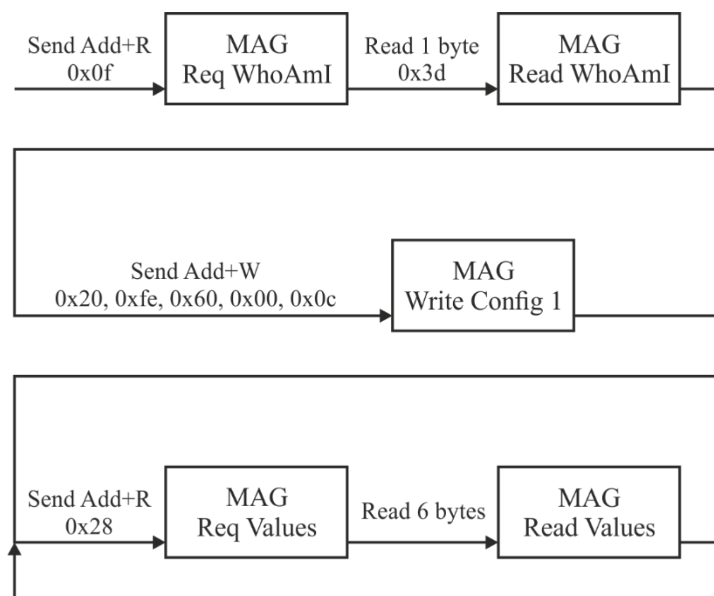
```
#include "Myfun.h"
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	I2C3_ConfInterrupt();
	LSM303_Interrupt_ACC_ReqWhoAmI();
	while(1)
	{
		Led_OnOff(0, LedTog);
		delay_ms(100);
	}
}
```

Zadanie 1:

Uzupełnić wszystkie powyższe funkcje, przetestować działanie programu wykorzystując oprogramowanie *STMStudio* do podglądu wartości zmiennych przechowujących wartości przyspieszeń odczytanych z akcelerometru.

Zadanie 2:

Poniżej przedstawiono graf obrazujący procedurę pracy magistrali przy odczycie danych z magnetometru. Jest ona bardzo podobna do procedury dla akcelerometru. Różni się jedynie zapisem do rejestrów konfiguracyjnych. Proszę przygotować zestaw nowych funkcji dla magnetometru oraz dopisać nowe wartości dla stanu pracy magistrali. Proszę przetestować działanie przy odczycie tylko z magnetometru.



Zajęcia nr 10 I2C – obsługa z wykorzystaniem DMA. Akcelerometr LSM303C

Wprowadzenie

Podczas tych zajęć student zapozna się z działaniem interfejsu komunikacyjnego *I2C*. Student zdobędzie praktyczną wiedzę dotyczącą obsługi tego układu peryferyjnego z wykorzystaniem kontrolera *DMA* oraz nawiąże komunikację z układem scalonym *LSM303C* zawierającym akcelerometr 3DoF i magnetometr 3DoF.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków".

Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu te trzy pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1"…* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie.

Obsługa LSM303C z wykorzystaniem kontrolera DMA

Obsługa czujnika *LSM303C* poprzez *I2C* z wykorzystaniem kontrolera DMA jest bardzo zbliżona do jego obsługi z wykorzystaniem mechanizmu przerwań, którą poznaliśmy podczas poprzednich zajęć. Z interfejsem *I2C3* współpracują kanał 2 i kanał 3 kontrolera *DMA1* wykorzystując żądanie 3. Kontroler *DMA* we współpracy z *I2C* można wykorzystać do przesyłu danych. Bit *STA*, adres układu *slave*, bit oznaczający kierunek przesyłu danych i bit *STP* muszą zostać wysłane przez pozostałe oprogramowanie. A zatem kontrolerem *DMA* można z powodzeniem zastąpić część programu zajmującą się obsługą przerwań *TXIS* i *RXNE*. Za każdym razem gdy flaga *TXE* w rejestrze statusowym *ISR* zostanie ustawiona, zostanie również wygenerowane żądanie do kontrolera *DMA* o zapis do rejestru *TXDR*. Za każdym razem gdy

flaga *RXNE* w rejestrze statusowym *ISR* zostanie ustawiona, zostanie również wygenerowane żądanie do kontrolera *DMA* o zapis do rejestru *RXDR*.

RM0351 **Direct memory access controller (DMA)**

Table 45. Summary of the DMA1 requests for each channel (continued)

Request. number	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5	Channel 6	Channel 7
2	-	USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
3	-	I2C3_TX	I2C3_RX	I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
4	TIM2_CH3	TIM2_UP	TIM16_CH1 TIM16_UP	-	TIM2_CH1	TIM16_CH1 TIM16_UP	TIM2_CH2 TIM2_CH4
5	TIM17_CH1 TIM17_UP	TIM3_CH3	TIM3_CH4 TIM3_UP	TIM7_UP. DAC2	QUADSPI	TIM3_CH1 TIM3_TRIG	TIM17_CH1 TIM17_UP
6	TIM4_CH1	-	TIM6_UP DAC1	TIM4_CH2	TIM4_CH3	-	TIM4_UP
7	-	TIM1_CH1	TIM1_CH2	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM15_CH1 TIM15_UP TIM15_TRIG TIM15_COM	TIM1_UP	TIM1_CH3

Przygotujmy zestaw zmiennych globalnych niezbędnych w tej wersji oprogramowania. Są one analogiczne do zmiennych z poprzednich ćwiczeń. Nie będziemy jednak potrzebować dwóch zmiennych przechowujących informacje o numerach elementów w tablicach na dane do wysłania i na dane odebrane. Pozostałe zmienne bez zmian.

```
eLSM303 lsm303_state = AccMag_Idle;
uint8_t lsm303_tabw[10];
uint8_t lsm303_tabr[10];
uint8_t tabacc[6];
uint8_t tabmag[6];
int16_t accx = 0;
int16_t accy = 0;
int16_t accz = 0;
int16_t magx = 0;
int16_t magy = 0;
int16_t magz = 0;
```

Następnie napiszemy funkcję konfigurującą *I2C* i *DMA1*.

```
void I2C3_ConfDMA(void)
{
	RCC->APB1ENR1		|= RCC_APB1ENR1_PWREN;
	PWR->CR2		|= PWR_CR2_IOSV;
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOGEN;
	RCC->APB1ENR1		|= RCC_APB1ENR1_I2C3EN;
	RCC->AHB1ENR		|= RCC_AHB1ENR_DMA1EN;

	GPIOG->MODER		&= ~GPIO_MODER_MODE8 & ~GPIO_MODER_MODE7;
	GPIOG->MODER		|= GPIO_MODER_MODE8_1 | GPIO_MODER_MODE7_1;
	GPIOG->AFR[0]		|= 0x40000000;
	GPIOG->AFR[1]		|= 0x00000004;

	DMA1_Channel2->CPAR		= (uint32_t)&I2C3->TXDR;
	DMA1_Channel2->CMAR		= (uint32_t)lsm303_tabw;
```

```
	DMA1_Channel2->CNDTR		= (uint16_t)10;
	DMA1_CSELR->CSELR		|= 0x00000030; //channel 2 req 3
	DMA1_Channel2->CCR		|= DMA_CCR_MINC | DMA_CCR_DIR;

	DMA1_Channel3->CPAR		= (uint32_t)&I2C3->RXDR;
	DMA1_Channel3->CMAR		= (uint32_t)lsm303_tabr;
	DMA1_Channel3->CNDTR		= (uint16_t)10;
	DMA1_CSELR->CSELR		|= 0x00000300; //channel 3 req 3
	DMA1_Channel3->CCR		|= DMA_CCR_MINC | DMA_CCR_EN;

	I2C3->TIMINGR			= 0x04;
	I2C3->CR1			= I2C_CR1_TXDMAEN | I2C_CR1_RXDMAEN | I2C_CR1_PE;
	NVIC_EnableIRQ(I2C3_EV_IRQn);
}
```

W funkcji tej nowymi elementami jest włączenie taktowania dla *DMA1*, skonfigurowanie kanału 2 do obsługi wysyłania przez *I2C3*, skonfigurowanie kanału 3 do obsługi odbioru przez *I2C3*, oraz ustawienie bitów *I2C_CR1_TXDMAEN* i *I2C_CR1_RXDMAEN* w rejestrze *CR1* zezwalających interfejsowi *I2C3* na wysyłanie żądań do *DMA1*. Ilość odbieranych i wysyłanych danych w rejestrach *NDTR* ustawiono wstępnie na 10. Natomiast ilość wysyłanych danych i tak jest kontrolowana przez I2C. Po wysłaniu lub odebraniu wszystkich danych *I2C* zgłosi przerwanie *TC*. Można również ustawić w rejestrze *NDTR* taką sama ilość danych do wysłania lub odebrania. Wtedy, o ile na to zezwolimy, *DMA* również zgłosi przerwanie *TC*. W naszym przypadku skorzystamy z pierwszego wariantu. Funkcja obsługi przerwania jest teraz prostsza, gdyż nie zawiera obsługi przerwania *TXIS* i *RXNE*. Poniżej jej przykład dla odczytu z akcelerometru. Oczywiście cała logika programu obsługująca procedurę zapisu i odczytu z akcelerometru nadal znajduje się w przerwaniu *TC*.

```
void I2C3_EV_IRQHandler(void)
{
	if((I2C3->ISR & I2C_ISR_TC) != RESET)
	{
		if(lsm303_state == Acc_ReqWhoAmI)
			LSM303_Dma_ACC_ReadWhoAmI();
		else if(lsm303_state == Acc_ReadWhoAmI)
			LSM303_Dma_ACC_WriteConfig1();
		else if(lsm303_state == Acc_WriteConfig1)
			LSM303_Dma_ACC_WriteConfig2();
		else if(lsm303_state == Acc_WriteConfig2)
			LSM303_Dma_ACC_ReqValues();
		else if(lsm303_state == Acc_ReqValues)
			LSM303_Dma_ACC_ReadValues();
		else if(lsm303_state == Acc_ReadValues)
		{
			LSM303_Dma_ACC_Values();
			LSM303_Dma_ACC_ReqValues();
		}
	}
}
```

Niewielkim zmianom uległy również funkcje inicjujące *I2C3* do realizacji poszczególnych kroków procedury odczytu. Usunięto z nich zerowanie zmiennych przechowujących numery elementów w tablicach z danymi do wysłania lub danymi odebranymi. Oprócz tego w każdej funkcji wysyłającej żądanie lub zapisującej dane do czujnika *LSM303C* umieszczono wywołanie funkcji reinicjującej *DMA*, która pokazano poniżej.

```
void I2C3_ReinitDMA(void)
{
	DMA1_Channel2->CCR &= ~DMA_CCR_EN;
	DMA1_Channel2->CNDTR = 10;
	DMA1_Channel2->CCR |= DMA_CCR_EN;

	DMA1_Channel3->CCR &= ~DMA_CCR_EN;
	DMA1_Channel3->CNDTR = 10;
	DMA1_Channel3->CCR |= DMA_CCR_EN;
}
```

Zmianie uległa też ilość konfigurowanych przerwań *I2C3*. Aktualnie wystarczające będzie włączenie jedynie przerwania *TC* poprzez ustawienie bitu *I2C_CR1_TCIE* w rejestrze *CR1*.

```
void LSM303_Dma_ACC_ReqWhoAmI(void)
{
	I2C3_ReinitDMA();
	lsm303_tabw[0]	= 0x0f;
	I2C3->CR1	|= I2C_CR1_TCIE;
	I2C3->CR2	= (1 << I2C_CR2_NBYTES_Pos) | (0x1d << (I2C_CR2_SADD_Pos + 1)) |
			I2C_CR2_START;
	lsm303_state = Acc_ReqWhoAmI;
}
void LSM303_Dma_ACC_ReadWhoAmI(void)
{
	I2C3->CR2	= (1 << I2C_CR2_NBYTES_Pos) | (0x1d << (I2C_CR2_SADD_Pos + 1)) |
			I2C_CR2_RD_WRN | I2C_CR2_START;
	lsm303_state	= Acc_ReadWhoAmI;
}
void LSM303_Dma_ACC_WriteConfig1(void)
{
	I2C3_ReinitDMA();
	lsm303_tabw[0]	= 0x20;
	lsm303_tabw[1]	= 0x27;
	I2C3->CR2	= (2 << I2C_CR2_NBYTES_Pos) | (0x1d << (I2C_CR2_SADD_Pos + 1)) |
			I2C_CR2_START;
	lsm303_state	= Acc_WriteConfig1;
}
void LSM303_Dma_ACC_WriteConfig2(void)
{
	I2C3_ReinitDMA();
	lsm303_tabw[0]	= 0x23;
	lsm303_tabw[1]	= 0x04;
	I2C3->CR2	= (2 << I2C_CR2_NBYTES_Pos) | (0x1d << (I2C_CR2_SADD_Pos + 1)) |
			I2C_CR2_START;
	lsm303_state	= Acc_WriteConfig2;
}
void LSM303_Dma_ACC_ReqValues(void)
{
	I2C3_ReinitDMA();
```

```
		lsm303_tabw[0]	= 0x28;
		I2C3->CR2	= (1 << I2C_CR2_NBYTES_Pos) | (0x1d << (I2C_CR2_SADD_Pos + 1)) |
				I2C_CR2_START;
		lsm303_state	= Acc_ReqValues;
}
void LSM303_Dma_ACC_ReadValues(void)
{
		I2C3->CR2	= (6 << I2C_CR2_NBYTES_Pos) | (0x1d << (I2C_CR2_SADD_Pos + 1)) |
				I2C_CR2_RD_WRN | I2C_CR2_START;
		lsm303_state	= Acc_ReadValues;
}
void LSM303_Dma_ACC_Values(void)
{
		for(int i=0;i<6;i++)
		{
			tabacc[i] = lsm303_tabr[i];
		}
		accx = (int16_t)(((uint16_t)tabacc[1] << 8) + ((uint16_t)tabacc[0] << 0)) / 88.89;
		accy = (int16_t)(((uint16_t)tabacc[3] << 8) + ((uint16_t)tabacc[2] << 0)) / 88.89;
		accz = (int16_t)(((uint16_t)tabacc[5] << 8) + ((uint16_t)tabacc[4] << 0)) / 88.89;
}
```

Uruchomienie całości jest analogiczne jak podczas poprzednich zajęć.

```
#include "Myfun.h"
int main(void)
{
		SysTick_Config(4000000 / 1000);
		Led_Conf();
		I2C3_ConfDMA();
		LSM303_Dma_ACC_ReqWhoAmI();
		while(1)
		{
			Led_OnOff(0, LedTog);
			delay_ms(100);
		}
}
```

Zadanie 1:

Proszę uruchomić i przetestować działanie powyższego programu odczytującego wartość przyspieszeń z akcelerometru wbudowanego w czujnik *LSM303C*. Do oceny działania wykorzystać oprogramowanie *STMStudio* lub wyświetlić wartość na wyświetlaczu LCD.

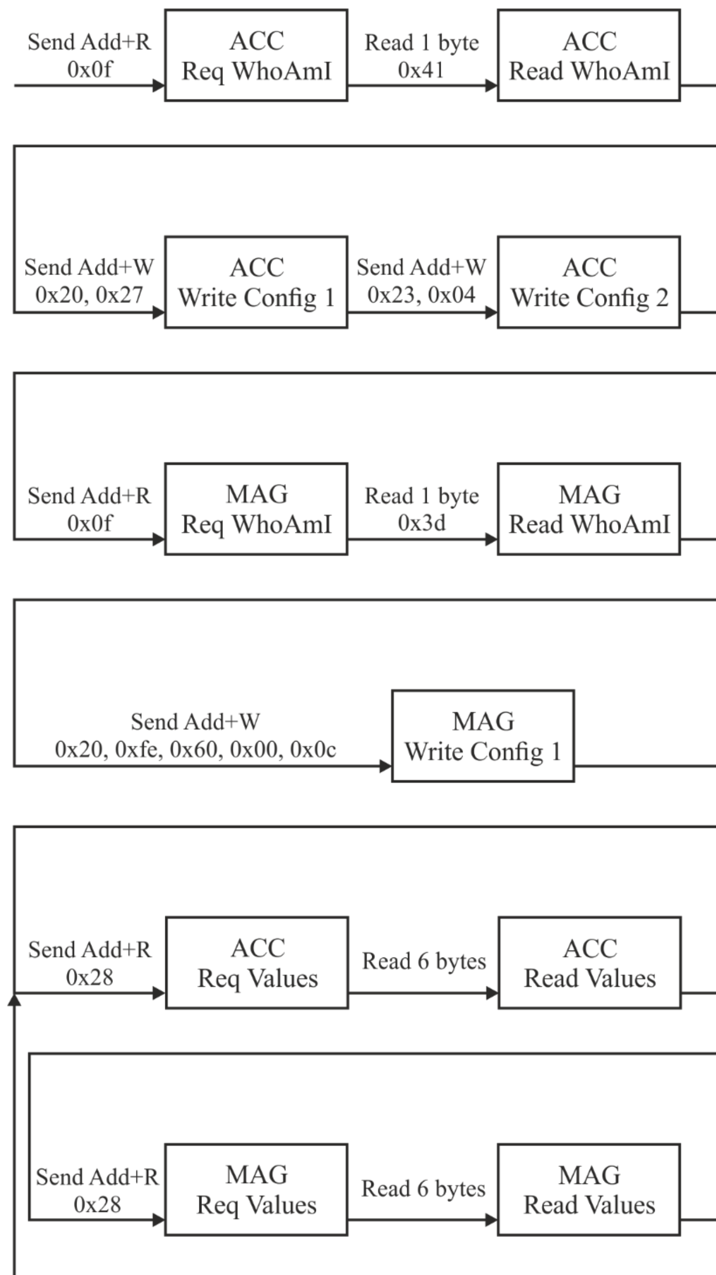
Zadanie 2:

Proszę przygotować zestaw nowych funkcji dla magnetometru. Proszę przetestować działanie przy odczycie tylko z magnetometru.. Do oceny działania wykorzystać oprogramowanie *STMStudio* lub wyświetlić wartość na wyświetlaczu *LCD*.

Zadanie 3:

Poniżej przedstawiono graf obrazujący procedurę pracy magistrali przy naprzemiennym odczycie danych z akcelerometru i magnetometru. Jest ona swego rodzaju połączeniem obsługi

akcelerometru i magnetometru. Nie wymaga tworzenia nowych funkcji. Wymaga natomiast odpowiedniej modyfikacji funkcji obsługi przerwania *TC*, tak aby zawarta tam logika działania obejmowała całą procedurę. Proszę przetestować działanie przy naprzemiennym odczycie z akcelerometru i magnetometru.



Zajęcia nr 11 Przetwornik analogowo – cyfrowy.

Wprowadzenie

Podczas tych zajęć student zapozna się z działaniem przetwornika analogowo – cyfrowego wbudowanego w mikrokontrolery *STM32L4*. Student dokona konfiguracji układu do pracy z jednym sygnałem analogowym oraz skonfiguruje przetwornik do pracy z wieloma sygnałami analogowymi i kontrolerem *DMA*.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków”.

Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu te trzy pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1”…* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie. Ponadto do folderu kopiujemy pliki z biblioteką do obsługi wyświetlacza HD44780 oraz dodajemy je do projektu. Wyświetlacz inicjujemy i sprawdzamy jego działanie wyświetlając dowolny tekst.

Pomiar jednego sygnału analogowego

Mikrokontroler *STM32L496ZGT* posiada trzy niezależne przetworniki analogowo – cyfrowe (*ADC*), z których każdy może dokonywać pomiaru 16 zewnętrznych sygnałów analogowych. Należy jednak pamiętać, że w danej chwili możliwy jest pomiar tylko jednego sygnału. Ponadto przetworniki mają możliwość pomiaru sygnału analogowego pochodzącego z wewnętrznego czujnika temperatury i wartości napięcia podłączonego do wyprowadzenia *VBAT* mikrokontrolera oraz pomiar napięcia referencyjnego *REFINT*. W pierwszej kolejności uruchomimy przetwornik do pracy z jednym sygnałem zewnętrznym z ręcznym wyzwalaniem pomiaru. W tym celu napiszemy funkcję konfigurującą:

```
void ADC_ConfBasic(void)
{
```

```
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOCEN | RCC_AHB2ENR_ADCEN;
	RCC->CCIPR		|= RCC_CCIPR_ADCSEL;
	GPIOC->MODER		|= GPIO_MODER_MODER2;

	ADC3->CR		= ADC_CR_ADEN | ADC_CR_ADVREGEN;
	ADC3->CFGR		|= ADC_CFGR_DISCEN;
	ADC3->SQR1		|= (3<< ADC_SQR1_SQ1_Pos);
	ADC3->SMPR1		|= ADC_SMPR1_SMP3;
	ADC3->ISR		|= ADC_ISR_EOC;
	ADC3->CR		|= ADC_CR_ADSTART;
}
```

Funkcja ta uruchamia taktowanie zegarowe dla *ADC* i dla portu *C*, wybiera odpowiednie źródło sygnału zegarowego dla *ADC* i konfiguruje *GPIOC.2* jako wyprowadzenie analogowe. Dalej włączane jest *ADC3* i włączany jest generator napięcia odniesienia dla niego. Następnie konfigurowany jest przetwornik *ADC3* do pracy pojedynczej (tylko jeden pomiar), ustawiany maksymalny czas pomiaru aby poprawić dokładność i wybierany jest odpowiedni kanał (w tym przypadku kanał 3, gdyż tam jest podłączony potencjometr). Na koniec uruchamiany jest pierwszy pomiar.

Aby odczytać wartość należy odczekać na koniec konwersji, co sygnalizuje flaga *EOC* w rejestrze statusowym *ISR*. Po jej ustawieniu można odczytać wartość z rejestru *DR*. Następnie należy wyczyścić flagę i ponownie uruchomić konwersję. Można to wykonać następującym fragmentem kodu:

```
#include "Myfun.h"
#include "HD44780.h"
#include <stdio.h>
uint16_t val;
char buf[20];
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	LCD_Init();
	ADC_ConfBasic();
	while(1)
	{
		if((ADC3->ISR & ADC_ISR_EOC) != RESET)
		{
			val = ADC3->DR;
			ADC3->CR |= ADC_CR_ADSTART;
		}
		sprintf(buf, "%d", val);
		LCD_Clear();
		LCD_WriteText(buf);
		delay_ms(100);
	}
}
```

Ciągłe oczekiwanie na koniec konwersji znacznie obciąża procesor, dlatego skonfigurujemy teraz nasz *ADC* do pracy z generowaniem przerwań po zakończeniu konwersji. W tym celu

zmienimy tryb pracy na ciągły (bit *CONT* zamiast *DISCEN* w rejestrze *CFGR*), włączymy przerwanie od końca konwersji (bit *EOCIE* w rejestrze *IER*) oraz włączymy *NVIC*.

```
void ADC_ConfInterrupt(void)
{
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOCEN | RCC_AHB2ENR_ADCEN;
	RCC->CCIPR		|= RCC_CCIPR_ADCSEL;
	GPIOC->MODER		|= GPIO_MODER_MODE2;

	ADC3->CR		= ADC_CR_ADEN | ADC_CR_ADVREGEN;
	ADC3->SQR1		|= (3<<ADC_SQR1_SQ1_Pos);
	ADC3->SMPR1		|= ADC_SMPR1_SMP3;
	ADC3->CFGR		|= ADC_CFGR_CONT;
	ADC3->IER		|= ADC_IER_EOCIE;
	NVIC_EnableIRQ(ADC3_IRQn);
	ADC3->CR		|= ADC_CR_ADSTART;}
```

Niezbędna będzie również funkcja obsługi przerwania oraz globalna zmienna do przechowywania wyniku pomiaru.

```
volatile uint16_t val = 0;
void ADC3_IRQHandler(void)
{
	if((ADC3->ISR & ADC_ISR_EOC) != RESET)
	{
		val = ADC3->DR;
	}
}
```

W trybie ciągłym po zakończeniu jednej konwersji natychmiast rozpoczyna się kolejna. Dlatego nie musimy ponownie ustawiać bitu *ADSTART*. Natomiast w pliku main.c wystarczy jedynie wyświetlić wynik, pamiętając przy tym o dodaniu deklaracji zmiennej opatrzonej słowem kluczowym *extern*.

Zadanie 1:

Mierzony sygnał jest w zakresie 0 – 3VDC. Proszę na wyświetlaczu wyświetlić wynik zarówno w jednostkach *ADC* jak i po przeliczeniu na jednostkę napięcia. Należy uwzględnić, że domyślna rozdzielczość *ADC* wynosi 12 bitów.

Pomiar wielu sygnałów analogowych z wykorzystaniem DMA

Możliwy jest pomiar wielu sygnałów analogowych przez jeden przetwornik *ADC*, przy czym należy rozumieć, że jest to pomiar sekwencyjny polegający na kolejnym przełączaniu się przetwornika pomiędzy kanałami. Ważne jest tutaj to, że przetwornik posiada w danej chwili tylko jedną wartość, będącą wynikiem ostatniego pomiaru. Dlatego bardzo pomocne jest tutaj skorzystanie z kontrolera *DMA*, którego zadaniem będzie kopiowanie bieżących wartości pomiarów w odpowiednie miejsca wcześniej przygotowanej tablicy. Napiszmy funkcję konfigurującą:

```
volatile uint16_t tab[2];
void ADC_Conf_DMA_2Channels(void)
{
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOCEN | RCC_AHB2ENR_ADCEN;
	RCC->AHB1ENR		|= RCC_AHB1ENR_DMA1EN;
	RCC->CCIPR		|= RCC_CCIPR_ADCSEL;
	GPIOC->MODER		|= GPIO_MODER_MODE2;

	DMA1_Channel3->CPAR		= (uint32_t)&ADC3->DR;
	DMA1_Channel3->CMAR		= (uint32_t)tab;
	DMA1_Channel3->CNDTR		= (uint16_t)2;
	DMA1_CSELR->CSELR		= (0 << DMA_CSELR_C3S_Pos);
	DMA1_Channel3->CCR		|= DMA_CCR_CIRC | DMA_CCR_MINC | DMA_CCR_PSIZE_0 |
					DMA_CCR_MSIZE_0 | DMA_CCR_EN;

	ADC123_COMMON->CCR		|= ADC_CCR_TSEN;
	ADC3->CR			= ADC_CR_ADEN | ADC_CR_ADVREGEN;
	ADC3->SQR1			= (1<<ADC_SQR1_L_Pos) | (3<<ADC_SQR1_SQ1_Pos) |
					(17<<ADC_SQR1_SQ2_Pos);
	ADC3->SMPR1			= ADC_SMPR1_SMP3;
	ADC3->SMPR2			= ADC_SMPR2_SMP17;
	ADC3->CFGR			|= ADC_CFGR_DMACFG | ADC_CFGR_DMAEN |
					ADC_CFGR_CONT;
	ADC3->CR			|= ADC_CR_ADSTART;
}
```

Nasz program będzie zajmował się pomiarem dwóch sygnałów: z potencjometru i z czujnika temperatury. Zmiany w stosunku do poprzednich funkcji polegają na włączeniu taktowania dla *DMA*, skonfigurowaniu odpowiedniego kanału i żądania *DMA* i ustawieniu ilości kopiowanych danych na 2. Ważne jest tutaj też to, że przetwornik ma rozdzielczość 12 bit, więc musimy kopiować liczby szesnastobitowe. W tej sytuacji w konfiguracji *DMA* ustawiamy bity *MSIZE* i *PSIZE*. Włączamy również tryb *CIRC* czyli ciągłą pracę *DMA*. W samym przetworniku włączamy pomiar temperatury, ustawiamy dwa kanały w rejestrze *SQR1*, dla obydwu ustawiamy najdłuższy czas pomiaru oraz ustawiamy bity *DMACFG* i *DMAEN*, w celu włączenia współpracy z *DMA*.

Przy tak napisanym programie pomiar z danego kanału znajduje się w odpowiadającym mu elemencie tablicy. Możemy je wyświetlić na przykład za pomocą takiego kodu:

```
#include "Myfun.h"
#include "HD44780.h"
#include <stdio.h>
extern uint16_t tab[2];
char buf[20];
int main(void)
{
	SysTick_Config(4000000 / 1000);
	LCD_Init();
	ADC_Conf_DMA_2Channels ();
	while(1)
	{
		sprintf(buf, "%d", tab[0]);
		LCD_Clear();
```

```
                    LCD_WriteText(buf);
                    sprintf(buf, "%d", tab[1]);
                    LCD_GoTo(0,1);
                    LCD_WriteText(buf);
                    delay_ms(100);
          }
}
```

Zadanie 2:

Proszę odszukać w dokumentacji wzór pozwalający przeliczyć odczytany wynik na temperaturę procesora w stopniach Celsjusza i wyświetlić ją na LCD.

Zadanie 3:

Proszę zmienić rozmiar tablicy na 2000 elementów. Następnie proszę tak zmodyfikować funkcję inicjującą pracę *DMA*, aby liczba pomiarów wynosiła 2000. Następnie proszę przed wyświetleniem wyników obliczyć dla każdego kanału średnią arytmetyczną z 1000 ostatnich pomiarów. Proszę pamiętać, że dane w tablicy umieszczane są przez *DMA* naprzemiennie (P1.1, P2.1, P1.2, P2.2, P1.3, P2.3, …..).

Zajęcia nr 12 Przerwania zewnętrzne.

Wprowadzenie

Podczas tych zajęć student zapozna się z działaniem wbudowanych w mikrokontroler układów licznikowych. Wykorzysta je do odmierzania czasu z dokładnością na poziomie pojedynczych mikrosekund. Następnie student zapozna się z obsługą przerwań zewnętrznych i wykorzysta te mechanizmy do pomiaru odległości za pomocą czujnika ultradźwiękowego.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków".

Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu te pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1"...* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie.

Przerwania zewnętrzne

Przygotowany przez nas podczas zajęć numer dwa program do obsługi przycisków, mimo wielu zalet takich jak łatwość obsługi i niewielki rozmiar nacechowany jest kilkoma wadami. Najważniejsze z nich to fakt, że pozwala on jedynie na wykrycie stanu w jakim znajduje się dane wejście cyfrowe, więc nie umożliwia łatwego wykrycia zdarzenia polegającego na zmianie stanu wejścia cyfrowego z niskiego na wysokie (zbocze narastające) lub z wysokiego na niskie (zbocze opadające). Drugą zasadniczą wadą jest fakt, że aby odczytać stan przycisku należy cyklicznie wywoływać funkcję *Joy_Read().* W przypadku szybkich zmian stanu na wyprowadzeniu taka obsługa wejść zewnętrznych sprowadza się do ciągłego skanowania linii za pomocą ww. funkcji.

Rozwiązaniem tego problemu jest wykorzystanie tzw. przerwań rozszerzonych (*extended interrupts, EXTI*). Mikrkontrolery z serii *STM32L49x* posiadają 41 linii *EXTI*, z których 16 to

linie obsługujące przerwania zewnętrzne (*external interrupts*), a więc takie, których źródłami są zmiany stanów na wyprowadzeniach mikrokontrolera. Przy czym w danej chwili do pojedynczej linii *EXTIx (x = 0 .. 15)* możemy podłączyć tylko jedno wyprowadzenie jednego portu z zachowaniem zasady *EXTIx – GPIOy.x*. To oznacza, że do linii *EXTI0* możemy podłączyć wyprowadzenie o numerze 0 z dowolnego portu (na przykład możemy podłączyć *GPIOA.0* lub *GPIOG.0*, ale nie możemy podłączyć *GPIOA.1*) tak jak zostało to zobrazowane na poniższy schemacie zaczerpniętym z *Reference manual*.

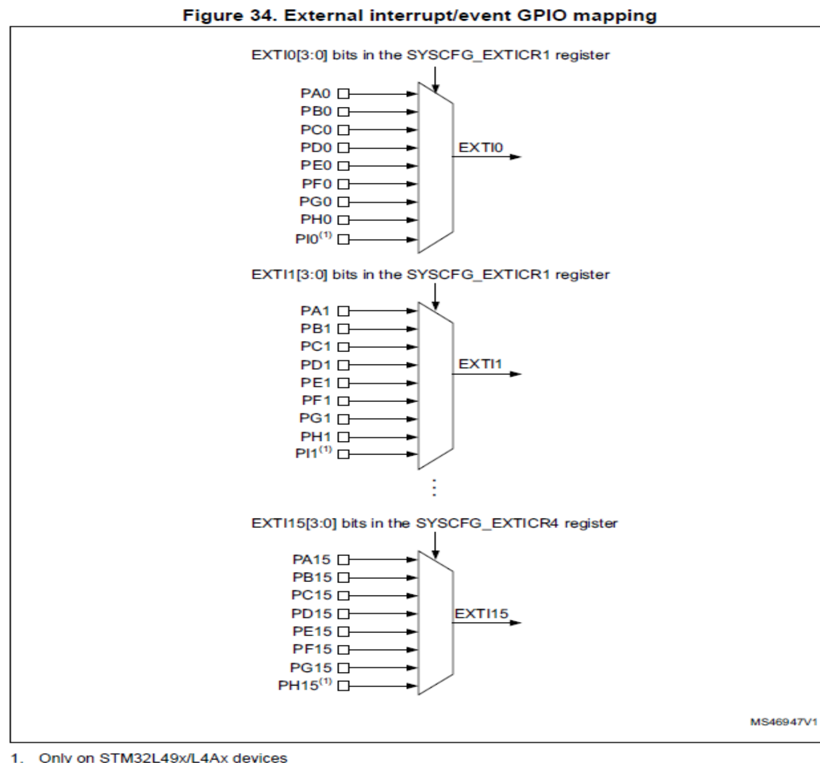


Figure 34. External interrupt/event GPIO mapping

1. Only on STM32L49x/L4Ax devices

Dane wyprowadzenie zewnętrzne mikrokontrolera, które ma być wykorzystane jako źródło przerwana zewnętrznego powinno być skonfigurowane jako cyfrowe wejście (*Input*), cyfrowe wyjście (*Output*) lub alternatywna funkcja (*Alternate Function*). Nie może być natomiast skonfigurowane w trybie analogowym. W celu uruchomienia danej linii *EXTI* jako przerwania zewnętrznego należy:

- włączyć taktowanie dla danego portu *GPIO*,
- skonfigurować dane wyprowadzenie portu *GPIO* do pracy w jednym z trzech powyższych trybów,
- włączyć taktowanie dla bloku *SYSCFG* (*System Configuration*),
- w odpowiednim rejestrze z grupy *SYSCFG->EXTICR[x]* wskazać, który port będzie źródłem przerwania dla danej linii *EXTI*,

- w bloku *EXTI* w rejestrze *IMR* ustawić bit odpowiadający danej linii *EXTI* celem odblokowania przerwania dla tej linii,
- w bloku *EXTI* w rejestrach *FTSRx* i/lub *RTSRx* skonfigurować czy przerwanie ma reagować na narastające zbocze, opadające zbocze lub obydwa,
- zezwolić w kontrolerze przerwań NVIC na przerwanie od danej linii *EXTI*.

Następnie należy napisać funkcję obsługi przerwania dla danej linii *EXTI*. Wszystkie 16 linii *EXTI* (od *EXTI0* do *EXTI15*) mają przypisane 7 wektorów przerwań (siedem funkcji obsługi przerwania). Linie *EXTI0* do *EXTI4* mają indywidualne funkcje o nazwach od *EXTI0_IRQHandler(void)* do *EXTI4_IRQHandler(void)*. Linie od *EXTI5* do *EXTI9* mają jedną wspólną funkcję obsługi przerwania o nazwie *EXTI9_5_IRQHandler(void)*. Linie od *EXTI10* do *EXTI15* mają jedną wspólną funkcję obsługi przerwania o nazwie *EXTI15_10_IRQHandler(void)*.

Zaawansowana obsługa przycisków.

Przygotujemy teraz funkcję konfiguracyjną pozwalającą na obsługę joystick'a (pięć wejść cyfrowych) z wykorzystaniem przerwań zewnętrznych. W ramce poniżej zamieszczono przykładowy kod konfigurujący wejście *GPIOE.0* z wykorzystaniem przerwań zewnętrznych i wykrywanie zbocza opadającego, co odpowiada akcji wciśnięcia przycisku. W ramce umieszczono też funkcję obsługi przerwania dla linii *EXTI0*.

```
void Joy_Interrupt_Conf(void)
{
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOEEN;
	RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
	GPIOE->MODER &= ~GPIO_MODER_MODER0;
	SYSCFG->EXTICR[0] |= SYSCFG_EXTICR1_EXTI0_PE;
	EXTI->IMR1 |= EXTI_IMR1_IM0;
	EXTI->FTSR1 |= EXTI_FTSR1_FT0;
	NVIC_EnableIRQ(EXTI0_IRQn);
}
void EXTI0_IRQHandler(void)
{
	Led_OnOff(0, LedTog);
	EXTI->PR1 |= EXTI_PR1_PIF0;
}
```

Zadanie 1:

Proszę przetestować działanie powyższych funkcji. Następnie proszę je zmodyfikować tak, aby na wyświetlaczy LED siedmiosegmentowym wyświetlała się ilość dotychczas wykrytych zboczy narastających.

Zadanie 2:

Proszę zmodyfikować funkcję konfiguracyjną tak, aby źródłem przerwania były obydwa zbocza (opadające i narastające). Proszę zmodyfikować funkcję obsługi przerwania dla *EXTI0* tak, aby w momencie wykrycia zbocza opadającego, dioda *LED0* została zapalona, a w momencie wykrycia zbocza narastającego, dioda *LED0* została zgaszona. Na wyświetlaczu *LED* powinna wyświetlać się ilość wystąpień wszystkich przerwań, niezależnie od rodzaju zbocza, które je wyzwoliło.

Zadanie 3:

Proszę zmodyfikować program tak, aby przyciski *GPIOE.1*, *GPIOE.2* i *GPIOE.3* również były obsługiwane z wykorzystaniem przerwań zewnętrznych.

Obsługa linii od EXTI5 do EXTI15 jest bardzo podobna. Różni się jedynie tym, że linie te nie posiadają własnych indywidualnych wektorów przerwań. Zatem aby obsłużyć przycisk środkowy joystick'a, który jest podłączony do wyprowadzenia *GPIOE.15* musimy odpowiednio skonfigurować linię *EXTI15*, a w jej funkcji obsługi przerwania wykryć która linia jest źródłem tego przerwania. W tym celu powinniśmy sprawdzić, czy w rejestrze *EXTI->PR1* jest ustawiony odpowiedna flaga przerwania. Przedstawia to poniższy fragment programu zawierający funkcję obsługi przerwania dla linii *EXTI10* do *EXTI15*.

```
void EXTI15_10_IRQHandler(void)
{
	if((EXTI->PR1 & EXTI_PR1_PIF15) != RESET)
	{
		//Ewentualny dodatkowy kod programu
		EXTI->PR1 |= EXTI_PR1_PIF15;
	}
}
```

Zadanie 4:

Proszę napisać ostateczną wersję funkcji konfiguracyjnej pozwalająca na obsługę wszystkich pięciu wyprowadzeń, do których podłączony jest joystick z wykorzystanie przerwań zewnętrznych.

Zadanie 5:

Proszę napisać program, który mierzy czas z dokładnością do 1ms pomiędzy wciśnięciem dwóch różnych przycisków. Czas powinien być wyświetlony na wyświetlaczu *LED* siedmiosegmentowym.

Zajęcia nr 13 Układy licznikowe podstawowe. Pomiar czasu. Ultradźwiękowy czujnik odległości.

Wprowadzenie

Podczas tych zajęć student zapozna się z działaniem wbudowanych w mikrokontroler układów licznikowych. Wykorzysta je do realizacji pomiaru czasu z dokładnością 1 mikrosekundy. Mechanizm ten zostanie połączony z poznanym podczas poprzednich zajęć mechanizmem przerwań zewnętrznych i wykorzystany do obsługi ultradźwiękowego czujnika odległości.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków".

Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu te trzy pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1"…* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie.

Ultradźwiękowy czujnik odległości

Celem dzisiejszych zajęć jest obsługa ultradźwiękowego czujnika odległości HC-SR04 marki Cytron Technologies. Urządzenie to wykorzystuje zjawisko odbicia fali dźwiękowej od przeszkody i pomiaru czasu w jakim fala dźwiękowa wyemitowana przez czujnik powróci do sensora. Urządzenie jest zasilane napięciem 5VDC i posiada zakres pomiarowy od 2 cm do 200cm.

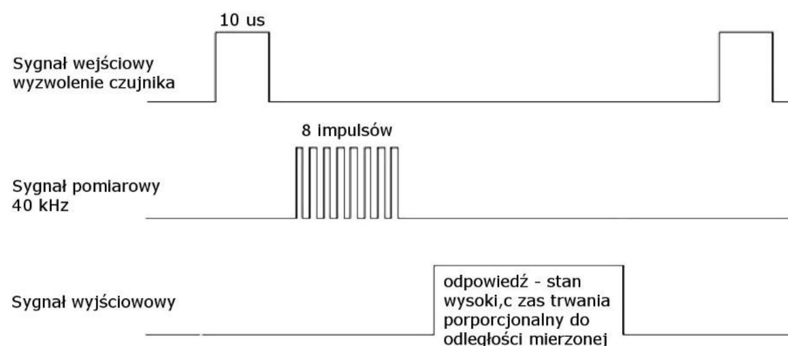
Na poniższym zdjęciu przedstawiano omawiany czujnik.

Rys. 1 Źródło - https://botland.com.pl/

Posiada on cztery wyprowadzenia oznaczone odpowiednio:

- VCC – potencjał dodatni zasilania – 5VDC,
- Trig – wejście cyfrowe czujnika wyzwalające pomiar,
- Echo – wyjście cyfrowe czujnika. Impuls o czasie trwania proporcjonalnym do zmierzonej odległości
- GND – potencjał ujemny zasilania.

Obsługa czujnika polega na podaniu na jego wejście *Trig* impulsu o czasie trwania co najmniej 10 mikrosekund. Następnie czujnik rozpoczyna generowanie fali ultradźwiękowej o częstotliwości około 40kHz. Fala dźwiękowa odbija się od przeszkody i powraca do czujnika. Po wykonaniu obliczeń, czujnik generuje na wyjściu *Echo* impuls, którego czas trwania jest równy czasowy jaki minął od wysłania fali dźwiękowej do jej powrotu.



Rys. 2 Źródło - https://botland.com.pl/

Przy założeniu, że znana jest prędkość rozchodzenia fali dźwiękowej (zazwyczaj przyjmuje się 340m/s) można wyznaczyć odległość czujnika od przeszkody według poniższego wzoru.

distance[mm] = time[us] * 0.340[mm/us] / 2

Podstawowa obsługa czujnika

Na początek przygotujemy mechanizm programowej obsługi czujnika w prostym trybie z wykorzystaniem licznika *SysTick*. W tym celu musimy skonfigurować wyprowadzenie *GPIOF.3* jako cyfrowe wyjście i podłączyć do niego sygnał *Trig*. Następnie konfigurujemy wyprowadzenie *GPIOF.4* jako cyfrowe wejście i podłączamy do niego sygnał *Echo*. Dalej konfigurujemy przerwanie zewnętrzne *EXTI4* tak aby źródłem przerwania były obydwa zbocza z wyprowadzenie *GPIOF.4*. Poniżej przedstawiono zarys programu. Na początek definiujemy dwie zmienne. Jedna do odmierzania czasu a druga do wyznaczania zmierzonej odległości. Zmienna odmierzająca czas powinna być inkrementowana w przerwaniu od *SysTick*. Następnie przygotowujemy funkcję konfiguracyjną, funkcję wyzwalającą czujnik oraz funkcję obsługi przerwania od EXTI4. W funkcji wyzwalającej czujnik ustawiamy odpowiednie wyprowadzenie w stan wysoki, czekamy jedną milisekundę i zmieniamy stan wyprowadzenia na niski. W ten sposób generujemy sygnał Trig o czasie co najmniej 10 mikrosekund. W przerwaniu sprawdzamy czy jego źródłem jest zbocze narastające (początek impulsu) czy zbocze opadające. Reakcja na zbocze narastające polega na wyzerowaniu licznika czasu. Reakcja na zbocze opadające polega na obliczeniu zmierzonego dystansu i wyświetleniu go na wyświetlaczu siedmiosegmentowym. Cały poniższy mechanizm wymaga cyklicznego uruchamiania funkcji wyzwalającej. Robimy to w pętli głównej *while(1)* co 100 milisekund.

```
uint32_t HCSR04_Time = 0;
double HCSR04_Distancemm = 0.0;
void HCSR04_Conf(void)
{
}
void HCSR04_SendTriger(void)
{
	GPIOF->ODR |= GPIO_ODR_OD3;
	delay_ms(1);
	GPIOF->ODR &= ~GPIO_ODR_OD3;
}
void EXTI4_IRQHandler(void)
{
	if((GPIOF->IDR & GPIO_IDR_ID4) != RESET)
	{
		HCSR04_Time = 0;
	}
	else
	{
		HCSR04_Distancemm = (double)HCSR04_Time / 2.0 * 340.0;
		Led7seg_value = HCSR04_Distancemm;
```

```
	}
	EXTI->PR1 |= EXTI_PR1_PIF4;
}
```

Zadanie 1:

Proszę uzupełnić powyższą funkcję konfiguracyjną *HCSR04_Conf()* i przetestować działanie programu.

Zaawansowana obsługa czujnika

Jak można zauważyć pomiar czasu z dokładnością do 1ms pozwolił nam na pomiar odległości z dokładnością do 170mm. Jest to dokładność, którą można nazwać wysoce niezadowalającą. Musimy zatem przygotować mechanizm, który pozwoli nam na odmierzanie czasu z wyższą precyzją, najlepiej z dokładnością do 1 mikrosekundy. Pierwsze rozwiązanie tego problemu, które można by zastosować to rekonfiguracja licznika *SysTick* tak aby generował przerwania co 1us. Niestety ciągłe przerwanie z częstotliwością 1MHz przy zegarze systemowym 4MHz zbyt mocno obciążałoby procesor lub prowadziło by do jego zawieszenia. Co więcej taka zmiana wymusiłaby na nas dokonywanie zmian w wielu dotychczas przygotowanych funkcjach. Dlatego wykorzystamy inny licznik do odmierzania czasu. Skonfigurujemy jeden z dwóch podstawowych liczników (*Basic Counters*) tak aby odmierzał czas z dokładnością do 1us. W tym celu zmodyfikujemy funkcję konfiguracyjną tak aby uruchamiała licznik *TIM7* w odpowiednim trybie pracy. Przygotujemy dwie pomocnicze funkcje: *HCSR04_SendTriger()* i *HCSR04_ReadEcho()* oraz funkcję obsługi przerwania od *TIM7*. Niezbędna będzie również modyfikacja funkcji obsługi przerwania od *EXTI4*. Cały omówiony kod przedstawiono poniżej.

```
void HCSR04_Conf(void)
{
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOFEN;
	RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
	RCC->APB1ENR1 |= RCC_APB1ENR1_TIM7EN;

	GPIOF->MODER &= ~GPIO_MODER_MODER3;
	GPIOF->MODER |= GPIO_MODER_MODER3_0;
	GPIOF->MODER &= ~GPIO_MODER_MODER4;
	GPIOF->PUPDR |= GPIO_PUPDR_PUPD4_1;
	SYSCFG->EXTICR[1] |= SYSCFG_EXTICR2_EXTI4_PF;
	EXTI->IMR1 |= EXTI_IMR1_IM4;
	EXTI->FTSR1 |= EXTI_FTSR1_FT4;
	EXTI->RTSR1 |= EXTI_RTSR1_RT4;
	NVIC_EnableIRQ(EXTI4_IRQn);

	HCSR04_SendTriger();
}
void HCSR04_SendTriger(void)
{
	TIM7->CR1 &= ~TIM_CR1_CEN;
	TIM7->PSC = 4-1;
```

```
	TIM7->ARR = 10-1;
	TIM7->DIER |= TIM_DIER_UIE;
	TIM7->CR1 |= TIM_CR1_CEN;
	NVIC_EnableIRQ(TIM7_IRQn);
	GPIOF->ODR |= GPIO_ODR_OD3;
}
void HCSR04_ReadEcho(void)
{
	TIM7->CR1 &= ~TIM_CR1_CEN;
	TIM7->CNT = 0;
	TIM7->PSC = 4-1;
	TIM7->ARR = 0xffff;
	TIM6->DIER &= ~TIM_DIER_UIE;
	TIM7->CR1 |= TIM_CR1_CEN;
}
void EXTI4_IRQHandler(void)
{
	if((GPIOF->IDR & GPIO_IDR_ID4) != RESET)
	{
		HCSR04_ReadEcho();
	}
	else
	{
		HCSR04_Time = TIM7->CNT;
		HCSR04_Distancemm = (double)HCSR04_Time / 2.0 * 0.340;
		led7seg_value = HCSR04_Distancemm;
		HCSR04_SendTriger();
	}
	EXTI->PR1 |= EXTI_PR1_PIF4;
}
void TIM7_IRQHandler(void)
{
	if((TIM7->SR & TIM_SR_UIF) != RESET)
	{
		TIM7->CR1 &= ~TIM_CR1_CEN;
		GPIOF->ODR &= ~GPIO_ODR_OD3;
		TIM7->SR &= ~TIM_SR_UIF;
	}
}
```

Powyższy program realizuje następujące funkcje. W pierwszej kolejności w funkcji konfiguracyjnej włączane jest taktowanie dla *TIM7* i uruchamiane jest generowanie sygnału *Trig* poprzez wywołanie funkcji *HCSR04_SendTriger()* , w której konfigurowany jest *TIM7* i ustawiane jest wyjście *Trig*. Zadaniem odliczania 10us zajmuje się *TIM7*. Jest on skonfigurowany tak aby zliczać impulsy z częstotliwością 1MHz (preskaler: *TIM7->PSC = 4-1*) i wygenerować przerwanie po 10us (rejestr *Auto Reload Register*: *TIM7->ARR = 10-1*). W przerwaniu od *TIM7* wyjście *Trig* jest czyszczone a *TIM7* jest wyłączany. Kasowana jest też flaga przerwania. Następnie czekamy na zbocze narastające na wejściu Echo. W przerwaniu od EXTI4 od zbocza narastającego wywoływana jest funkcja *HCSR04_ReadEcho(),* w której konfigurowany jest *TIM7* tak aby odliczał maksymalną wartość czasu i nie generował przerwań. Natomiast w przerwaniu od EXTI4 od zbocza opadającego przechwytywana jest wartość zliczona przez *TIM7* (rejestr *TIM7->CNT*), obliczana jest odległość i pomiar

uruchamiany jest ponownie. Dzięki temu czujnik jest obsługiwany bez dodatkowych opóźnień i bez udziału pętli głównej programu.

Zadanie 2:

Proszę przygotować alternatywną wersję programu tak aby możliwa byłą obsługa dwóch czujników. Drugi czujnik powinien być podłączony do wyprowadzeń *GPIOB.1* (*Trig*) i *GPIOF.5* (*Echo*).

Zajęcia nr 14 Układy licznikowe ogólnego zastosowania. Generowanie sygnału PWM.

Wprowadzenie

Podczas tych zajęć student zapozna się z działaniem wbudowanych w mikrokontroler układów licznikowych. Wykorzysta je do generowania sygnałów *PWM* w celu zmiany jasności i barwy diody LED RGB. Następnie student wykorzysta układ licznikowy do generowania sygnału *PWM* i sterowania z jego pomocą serwonapędem modelarskim.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków".

Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu te trzy pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1"…* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie.

Zmiana barwy diody RGB

Sygnał *PWM* to sygnał cyfrowy prostokątny o stałej amplitudzie, stałej częstotliwości jednak o zmiennym wypełnieniu. Poprzez wypełnienie rozumiem stosunek czasu trwania stanu wysokiego do całego okresu sygnału wyrażony w procentach. Ważny jest fakt iż średnia wartość sygnału jest wprost proporcjonalna do jego współczynnika wypełnienia. Jeżeli zatem taki sygnał wykorzystamy do sterowania diodą LED uzyskamy wrażenie zmiany intensywności takiej diody. Ponadto jeżeli wykorzystamy trzy diody LED umieszczone w jednej obudowie w kolorach: czerwonym, zielonym i niebieskim, co de facto tworzy nam diodę RGB, uzyskamy możliwość generowania różnych kolorów świecenia takiego układu. W naszym zestawie dydaktycznym znajduje się dioda RGB podłączona do wyprowadzeń *GPIOD.12*, *GPIOD.13* i *GPIOB.8*, będącymi jednocześnie wyjściami kanałów układu licznikowego *TIM4*

(*TIM4.CH1*, *TIM4.CH2, TIM4.CH3*). Napiszemy teraz funkcję konfigurującą układ *TIM4* do generowania sygnału *PWM* o częstotliwości ok 4kHz.

```
void LedRGB_PwmConf(void)
{
	//Red GPIOD.13, Green GPIOB.8, Blue GPIOD.12
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOBEN;
	GPIOB->MODER		&= ~GPIO_MODER_MODER8;
	GPIOB->MODER		|= GPIO_MODER_MODER8_1;
	GPIOB->PUPDR		|= GPIO_PUPDR_PUPD8_1;
	GPIOB->AFR[1]		|= 0x00000002;
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIODEN;
	GPIOD->MODER		&= ~GPIO_MODER_MODE12 & ~GPIO_MODER_MODE13;
	GPIOD->MODER		|= GPIO_MODER_MODE12_1 | GPIO_MODER_MODE13_1;
	GPIOD->PUPDR		|= GPIO_PUPDR_PUPD12_1 | GPIO_PUPDR_PUPD13_1;
	GPIOD->AFR[1]		|= 0x00220000;

	RCC->APB1ENR1		|= RCC_APB1ENR1_TIM4EN;
	TIM4->PSC		= 4-1;
	TIM4->ARR		= 256-1;
	TIM4->CCMR1		|= TIM_CCMR1_OC1M_1 | TIM_CCMR1_OC1M_2 |
				TIM_CCMR1_OC2M_1 | TIM_CCMR1_OC2M_2;
	TIM4->CCMR2		|= TIM_CCMR2_OC3M_1 | TIM_CCMR2_OC3M_2;
	TIM4->CCR1		= 0;
	TIM4->CCR2		= 0;
	TIM4->CCR3		= 0;
	TIM4->CCER		|= TIM_CCER_CC1E | TIM_CCER_CC2E | TIM_CCER_CC3E;
	TIM4->CR1		|= TIM_CR1_CEN;
}
```

Na początku konfigurujemy odpowiednie wyprowadzenia jako Alternate Function z wyborem odpowiedniej funkcji. Następnie uruchamiamy taktowanie dla *TIM4*, ustawiamy preskaler na wartość 4 (timer domyślnie taktowany jest zegarem systemowym, czyli sygnałem o częstotliwości 1MHz. Preskaler przez 4 zmniejsza częstotliwość do 1MHz). Następnie ustawiamy wartość rejestru *ARR* (Autoreload register) na 255, co pozwoli generować sygnał o częstotliwości ok 4kHz. Konfigurujemy trzy kanały timera jako wyjścia *PWM*, ustawiamy początkowa wartość wypełnienia na 0, włączamy wszystkie kanały i włączamy timer. Teraz, aby wygenerować na danym kanale sygnał o odpowiednim wypełnieniu wystarczy wpisać to rejestru *CCRx* liczbę z przedziału 0 – 255, co odpowiada pełnemu zakresowi jednej barwy w kodzie RGB. Program możemy przetestować w następujący sposób:

```
#include "Myfun.h"
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	LedRGB_PwmConf();
	while(1)
	{
		for(uint16_t i=0;i<256;i++)
		{
			TIM4->CCR1 = i;
```

```
				delay_ms(2);
			}
			TIM4->CCR1 = 0;
			for(uint16_t i=0;i<256;i++)
			{
				TIM4->CCR2 = i;
				delay_ms(2);
			}
			TIM4->CCR2 = 0;
			for(uint16_t i=0;i<256;i++)
			{
				TIM4->CCR3 = i;
				delay_ms(2);
			}
			TIM4->CCR3 = 0;
	}
}
```

Zadanie 1:

Proszę napisać funkcję o nazwie *LedRGB_SetColor(uint8_t red, uint8_t green, uint8_t blue)*, której zadaniem będzie ustawianie odpowiednich wartości wypełnienia dla danej barwy zakodowanej jako trzy liczby jednobajtowe. Proszę przetestować działanie funkcji.

Wykorzystamy teraz układ licznikowy do generowania sygnału *PWM* dla serwonapędu modelarskiego. W tym przypadku sygnał ten jest wykorzystywany do przenoszenia informacji o zadanej wartości pozycji kątowej jaką serwonapęd ma wypracować. Zazwyczaj należy wygenerować sygnał prostokątny o częstotliwości 50Hz i wypełnieniu w zakresie od 3% do 12% co powinno odpowiadać pozycji kątowej w zakresie od 0° do 180°.

Zadanie 2:

Proszę napisać funkcję, która skonfiguruje wyprowadzenie *GPIOA.5* jako alternatywna funkcja dla *TIM2*. Następnie skonfiguruje układ licznikowy *TIM2_CH1* jako źródło sygnału *PWM* o częstotliwości 50Hz. Proszę następnie wprowadzić do rejestru *TIM2->CCR1* wartość wypełnienia odpowiadającą wypełnieniu w zakresie od 3% do 12%. Proszę przetestować działanie programu na podłączonym uprzednio serwonapędzie modelarskim.

Zadanie 3:

Proszę przygotować program zawierający funkcję o nazwie *Serwodrive_SetPosition(double angle),* który będzie sterował serwonapędem modelarskim przyjmując wartość pozycji kątowej do wypracowania.

Zajęcia nr 15 Układy licznikowe ogólnego zastosowania – silnik DC z enkoderem.

Wprowadzenie

Podczas tych zajęć student zapozna się z możliwością wykorzystania mikrokontrolera generującego sygnał PWM do sterowania prędkością silnika prądu stałego. Następnie student pozna możliwości wykorzystania układów licznikowych do obsługi enkodera inkrementalnego.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków".

Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu te trzy pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1"…* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie.

Sterowanie silnikiem prądu stałego

Zestaw dydaktyczny wyposażony jest w mostek H *STSPIN250*. Posiada on trzy wejścia, na które należy podać odpowiednie stany logiczne. Na wejście *ENABLE* (*GPIOC.11*) należy podać na stałe stan wysoki, aby aktywować układ. Na wejście *PWM* (*GPIOA.8, TIM1_CH1*) podajemy sygnał *PWM*. Prędkość silnika będzie proporcjonalna do wartości tego sygnału. Natomiast na wejście *PHASE* (*GPIOC.10*) podajemy stan wysoki lub niski aby zmienić kierunek pracy silnika.

Zadanie 1:

Należy zatem napisać funkcję konfigurującą układ licznikowy *TIM1* do generowania sygnału *PWM* o częstotliwości 1kHz. Należy również skonfigurować dwa wyjścia cyfrowe niezbędne do załączania pracy układu i do sterowania kierunkiem obrotów silnika. Podpowiedź: *TIM1* jest timerem zawansowanym i jego konfiguracja do generowania sygnału *PWM* różni się

nieznacznie od innych timerów. Poniżej znajduje się szablon funkcji konfiguracyjnej z dopisanymi fragmentami kodu, które odróżniają ten timer od wcześniej używanych na zajęciach.

```
void DCMotor_Conf(void)
{
	//Tutaj proszę wpisać konfiguracje dwóch wyjsć cyfrowych
	//Tutaj proszę wpisać konfigurację wyprowadzenia do generowania PWM

	//Tutaj proszę wpisać konfigurację TIM1 CH1
	//PWM 1kHz
	//Poniższa linijka odróżnia ten timer od poprzednio używanych i jest niezbędna
	TIM1->BDTR |= TIM_BDTR_MOE;
}
```

Zadanie 2:

Proszę teraz przygotować funkcję o nazwie jak poniżej, która będzie przyjmować tylko jeden parametr: liczbę z przedziału od -999 do 999 określająca prędkość i kierunek pracy silnika. Proszę przetestować działanie tych funkcji.

```
void DCMotor_SetSpeed(int16_t speed)
{
	//Tutaj proszę wpisać kod sterujący prędkością i kierunkiem silnika
}
```

Odczyt wartości enkodera inkrementalnego

Mikrokontroler *STM32L496ZGT* posiada kilkanaście układów licznikowych, przy czym sześć najbardziej zaawansowanych (*TIM1* – *TIM5* i *TIM8*) posiada możliwość pracy w trybie *Encoder Mode*. W trybie tym licznik sprzętowo obsługuje enkoder inkrementalny odczytując jego pozycję przy pracy w obydwu kierunkach. Napiszemy teraz funkcję pozwalającą skonfigurować jeden z timerów do pracy w tym trybie. Enkoder należy zasilić napięciem 5VDC, a jego przewody sygnałowe podłączyć do wyprowadzeń *GPIOC.6* (*TIM3_CH1*) i *GPIOC.7* (*TIM3_CH2*).

```
void Encoder_Conf(void)
{
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOCEN;
	GPIOC->MODER		|= GPIO_MODER_MODE6_1 | GPIO_MODER_MODE7_1;
	GPIOC->OSPEEDR		|= GPIO_OSPEEDER_OSPEED6 | GPIO_OSPEEDER_OSPEED7;
	GPIOC->PUPDR		|= GPIO_PUPDR_PUPD6_0 | GPIO_PUPDR_PUPD7_0;
	GPIOC->AFR[0]		|= 0x22000000;
	RCC->APB1ENR1		|= RCC_APB1ENR1_TIM3EN;
	TIM3->ARR		= 0xffff;
	TIM3->SMCR		|= TIM_SMCR_SMS_0 | TIM_SMCR_SMS_1;
	TIM3->CCMR1		|= TIM_CCMR1_CC1S_0 | TIM_CCMR1_CC2S_0;
	TIM3->CR1		|= TIM_CR1_CEN;
}
```

Na początku włączamy taktowanie dla portu C i konfigurujemy wyprowadzenia jako *Alternate Function* z odpowiednim numerem funkcji. Następnie włączamy taktowanie dla *TIM3*, do rejestru *ARR* wpisujemy maksymalną wartość do jakiej licznik ma zliczać i po której ma się przepełnić. Licznik jest 16 – bitowy, więc najlepiej wpisać wartość szesnastkowo 0xffff. Następnie konfigurujemy odpowiednio kanał pierwszy i drugi i włączamy reakcję na obydwa zbocza. Na zakończenie włączamy licznik. Po uruchomieniu tej funkcji licznik zaczyna zliczać impulsy z enkodera. Możemy to sprawdzić odczytując rejestr *CNT* tego timera.

```
#include "Myfun.h"
int main(void)
{
	SysTick_Config(4000000 / 1000);
	Led_Conf();
	Encoder_Conf();
	while(1)
	{
		int16_t pos = TIM3->CNT;
		delay_ms(10);
	}
}
```

Podglądając zmienną *pos* np. w programie *STMStudio*, lub wyświetlając jej wartość na LCD można zauważyć, że układ pracuje poprawnie poza jednym szczegółem. Otóż licznik i zmienna są szesnastobitowe, więc bardzo łatwo można je przepełnić. Musimy zatem przygotować program, który nie będzie posiadał takich ograniczeń i ponadto będzie zwracał pozycję kątową enkodera w stopniach. W tym celu będziemy musieli wykryć moment przepełnienia licznika i będziemy potrzebować globalnych zmiennych do przechowywania liczby przepełnień.

Zadanie 3:

Proszę uzupełnić poniższą funkcję o kod pozwalający na odczyt pozycji kątowej w stopniach w pełnym zakresie. Proszę przetestować działanie.

```
volatile int32_t fullturn = 0; // liczba pełnych obrotów enkodera
volatile int32_t imp = 0; //aktualna pozycja w impulsach w zakresie jednego obrotu
volatile int32_t impprev = 0; //poprzednia pozycja w impulsach w zakresie jednego obrotu
double Encoder_ReadPos(void)
{
	double resolution = 64.0; //rozdzielczość enkodera – liczba impulsów na obrót
	//Tutaj należy uzupełnić kodem obliczającym pozycję kątową
	double angle; // pozycją kątowa po wyliczeniu
	return angle;
}
```

Zajęcia nr 16 Serwonapęd DC

Wprowadzenie

Podczas tych zajęć student nabędzie umiejętności projektowania i wykonywania układów regulacji z wykorzystaniem systemów mikroprocesorowych. Student wykona serwonapęd utrzymujący pozycję kątową silnika prądu stałego.

Przed przystąpieniem do pracy student proszony jest o utworzenie, na znajdującym się w pracowni komputerowej komputerze, i jednoznaczne nazwanie, katalogu, w którym będzie on umieszczał wszystkie swoje projekty. Ponadto, student proszony jest o tworzenie dla każdego nowego projektu osobnego podkatalogu. Ważne jest aby nazwy katalogów nie zawierały „polskich znaków”.

Tworzenie projektu w środowisku Keil uVision

Pracę zaczynamy od utworzenia nowego projektu według instrukcji do pierwszych zajęć. Następnie kopiujemy trzy pliki z poprzednich zajęć (dwa pliki z kodem źródłowym: *main.c* i *Myfun.c* oraz jeden plik nagłówkowy *Myfun.h*.) do naszego aktualnego katalogu. Dodajemy do projektu te trzy pliki klikając PPM na napis *Source Group 1* i wybierając opcję *Add Existing File to Group „Source Group 1”…* Następnie utworzony projekt modyfikujemy tak aby praca mikrokontrolera polegała na zmienianiu stanu jednej z diod LED z częstotliwością ok 10Hz. Tak przygotowany program wgrywamy do mikrokontrolera i testujemy działanie.

Synteza serwonapędu

Serwonapęd zbudowany jest z trzech elementów: elementu wykonawczego, który w naszym przypadku stanowi silnik prądu stałego wraz z układem wzmacniającym, sprzężenie zwrotne, które w naszym przypadku stanowi enkoder inkrementalny oraz układ regulacji, którym w naszym przypadku będzie mikrokontroler. Podczas poprzednich zajęć student zapoznał się z wykorzystaniem mikrokontrolera do sterowania prędkością silnika prądu stałego oraz do odczytu pozycji kątowej za enkodera inkrementalnego. Obydwa zagadnienia będą przydatne podczas dzisiejszych zajęć. Poniżej przedstawiono jeszcze raz niezbędne fragmenty programu.

Funkcja konfigurująca podzespoły sterujące prędkością silnika:

```
void DCMotor_Conf(void)
{
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOCEN;
	GPIOC->MODER		&= ~GPIO_MODER_MODE10 & ~GPIO_MODER_MODE11;
	GPIOC->MODER		|= GPIO_MODER_MODE10_0 | GPIO_MODER_MODE11_0;
	GPIOC->PUPDR		|= GPIO_PUPDR_PUPD10_0 | GPIO_PUPDR_PUPD11_0;
```

```
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOAEN;
	GPIOA->MODER		&= ~GPIO_MODER_MODE8;
	GPIOA->MODER		|= GPIO_MODER_MODE8_1;
	GPIOA->PUPDR		|= GPIO_PUPDR_PUPD8_0;
	GPIOA->AFR[1]		|= 0x00000001;

	RCC->APB2ENR		|= RCC_APB2ENR_TIM1EN;
	TIM1->BDTR		|= TIM_BDTR_MOE;
	TIM1->PSC		= 4-1;
	TIM1->ARR		= 1000-1;
	TIM1->CCMR1		|= TIM_CCMR1_OC1M_1 | TIM_CCMR1_OC1M_2;
	TIM1->CCR1		= 0;
	TIM1->CCER		|= TIM_CCER_CC1E;
	TIM1->CR1		|= TIM_CR1_CEN;
}
```

Funkcja konfigurująca podzespoły odpowiedzialne z odczyt pozycji z enkodera inkrementalnego.

```
void Encoder_Conf(void)
{
	RCC->AHB2ENR		|= RCC_AHB2ENR_GPIOCEN;
	GPIOC->MODER		|= GPIO_MODER_MODE6_1 | GPIO_MODER_MODE7_1;
	GPIOC->OSPEEDR		|= GPIO_OSPEEDER_OSPEED6 | GPIO_OSPEEDER_OSPEED7;
	GPIOC->PUPDR		|= GPIO_PUPDR_PUPD6_0 | GPIO_PUPDR_PUPD7_0;
	GPIOC->AFR[0]		|= 0x22000000;
	RCC->APB1ENR1		|= RCC_APB1ENR1_TIM3EN;
	TIM3->ARR		= 0xffff;
	TIM3->SMCR		|= TIM_SMCR_SMS_0 | TIM_SMCR_SMS_1;
	TIM3->CCMR1		|= TIM_CCMR1_CC1S_0 | TIM_CCMR1_CC2S_0;
	TIM3->CR1		|= TIM_CR1_CEN;
}
```

Funkcja ustawiająca zadaną prędkość silnika. Przyjmuje ona wartość niecałkowite w przedziale od -1 do 1, co stanowi pełny zakres prędkości silnika.

```
void DCMotor_SetSpeed(double speed)
{
	if(speed > -1.0 && speed < 1.0)
	{
		TIM1->CCR1	= (uint16_t)(999.0 * fabs(speed));
		if(speed > 0)	GPIOC->ODR |= GPIO_ODR_OD10;
		else		GPIOC->ODR &= ~GPIO_ODR_OD10;
	}
}
```

Funkcja zwracająca pozycję kątową silnika w stopniach.

```
volatile int32_t fullturn = 0;
volatile int32_t imp = 0;
volatile int32_t impprev = 0;
double Encoder_ReadPos(void)
{
	double resolution = 64.0 * 19.0; //rozdzielczosc enkodera po uwzglednieniu przelozenia przekladni
	int32_t r = 0;
	impprev = imp;
	imp = TIM3->CNT;
	r = imp - impprev;
	if(r < (-0xffff/2))		fullturn++;
```

```
        else if(r > (0xffff/2))         fullturn--;
        int32_t pos = fullturn * 0xffff + imp;
        double angle = (double)pos / resolution * 360.0;
        return angle;
}
```

Powyższe fragment programu realizują zadania sterowania elementem wykonawczym i obsługi sprzężenia zwrotnego. Zadaniem studenta jest przygotowanie programu realizującego funkcję układu regulacji.

Zadanie 1:

Napisać program realizujący funkcje regulatora trzystanowego z histerezą o wartości $\pm 20°$. Wartość zadaną wprowadzać z poziomu oprogramowania *STMStudio*. Przetestować działanie układu. Pracę układu oceniać poprzez podgląd zmiennych w programie *STMStudio*, ze szczególnym uwzględnieniem zmiennej zawierającej uchyb regulacji.

Zadanie 2:

Napisać program realizujący funkcje regulatora typu P. Wartość zadaną wprowadzać z poziomu oprogramowania *STMStudio*. Wartość wzmocnienia k_p wprowadzać z poziomu oprogramowania *STMStudio*. Przetestować działanie układu. Pracę układu oceniać poprzez podgląd zmiennych w programie *STMStudio*, ze szczególnym uwzględnieniem zmiennej zawierającej uchyb regulacji.

Zadanie 2:

Napisać program realizujący funkcje regulatora typu PID. Wartość zadaną wprowadzać z poziomu oprogramowania *STMStudio*. Wartości nastaw regulatora k_p, k_i, k_d wprowadzać z poziomu oprogramowania *STMStudio*. Przetestować działanie układu. Pracę układu oceniać poprzez podgląd zmiennych w programie *STMStudio*, ze szczególnym uwzględnieniem zmiennej zawierającej uchyb regulacji. **Podpowiedź**: działanie tego regulatora zależne jest od kroku czasowego. Ważne jest zatem synchroniczne uruchamianie wszelkich obliczeń związanych z regulatorem. Zaleca się początkowo wykonywanie ich z krokiem czasowym równym 10ms.