

Warszawa, 2016

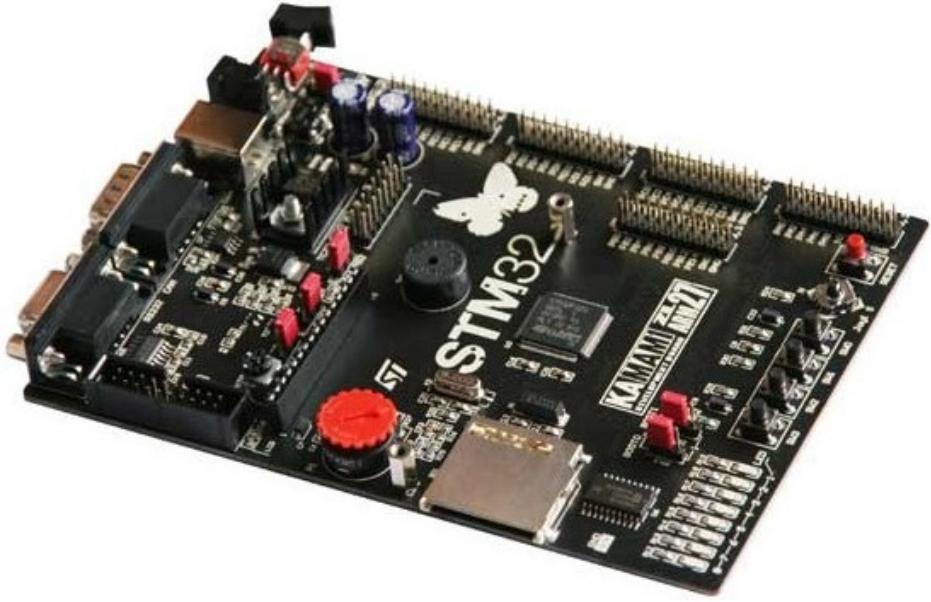
Systemy mikroprocesorowe w sterowaniu

Ćwiczenia laboratoryjne

Patryk Chaber

Spis treści

1 Praca z zestawem uruchomieniowym, praca krokowa, debugowanie, przygotowywanie i uruchomienie prostych programów: obsługa portów wejścia-wyjścia, obsługa wyświetlacza tekstowego LCD, sterowanie szerokością impulsu, przetwornik analogowo-cyfrowy	2
1.1 Cel	2
1.2 Przebieg laboratorium (prowadzenie za rączkę)	2
1.3 Przebieg laboratorium (samodzielnie wykonywane zadanie)	17
1.4 Wejściówka:	34
2 Obsługa prostych czujników i urządzeń wykonawczych mikroprocesorowego systemu automatyki przy wykorzystaniu systemu przerwań	35
2.1 Cel	35
2.2 Wykorzystane mechanizmy	35
2.3 Przebieg laboratorium	42
2.4 Wejściówka:	56
3 Obsługa złożonych czujników i urządzeń wykonawczych mikroprocesorowego systemu automatyki	57
3.1 Przebieg laboratorium	69
4 Obsługa wyświetlacza graficznego LCD (panelu dotykowego), wykorzystanie jednostki zmiennopozycyjnej do przetwarzania sygnałów	71
4.1 Przebieg laboratorium	74
5 Implementacja algorytmów regulacji PID i DMC prostego procesu dynamicznego, interfejs użytkownika, archiwizacja pomiarów, dobór nastaw algorytmów, badania porównawcze	77
5.1 Zadanie do wykonania	96
5.2 Sugerowany przebieg projektu	97
6 Identyfikacja modeli (typu odpowiedzi skokowej) procesu laboratoryjnego, implementacja algorytmu regulacji DMC, dobór nastaw algorytmu, badania porównawcze	98
6.1 Przebieg ćwiczenia	100
7 System operacyjny czasu rzeczywistego	102
7.1 Implementacja	103
7.2 Zadanie do wykonania	105



Rysunek 1: Zestaw uruchomieniowy ZL27ARM

1 Praca z zestawem uruchomieniowym, praca krokowa, debugowanie, przygotowywanie i uruchomienie prostych programów: obsługa portów wejścia-wyjścia, obsługa wyświetlacza tekstowego LCD, sterowanie szerokością impulsu, przetwornik analogowo-cyfrowy

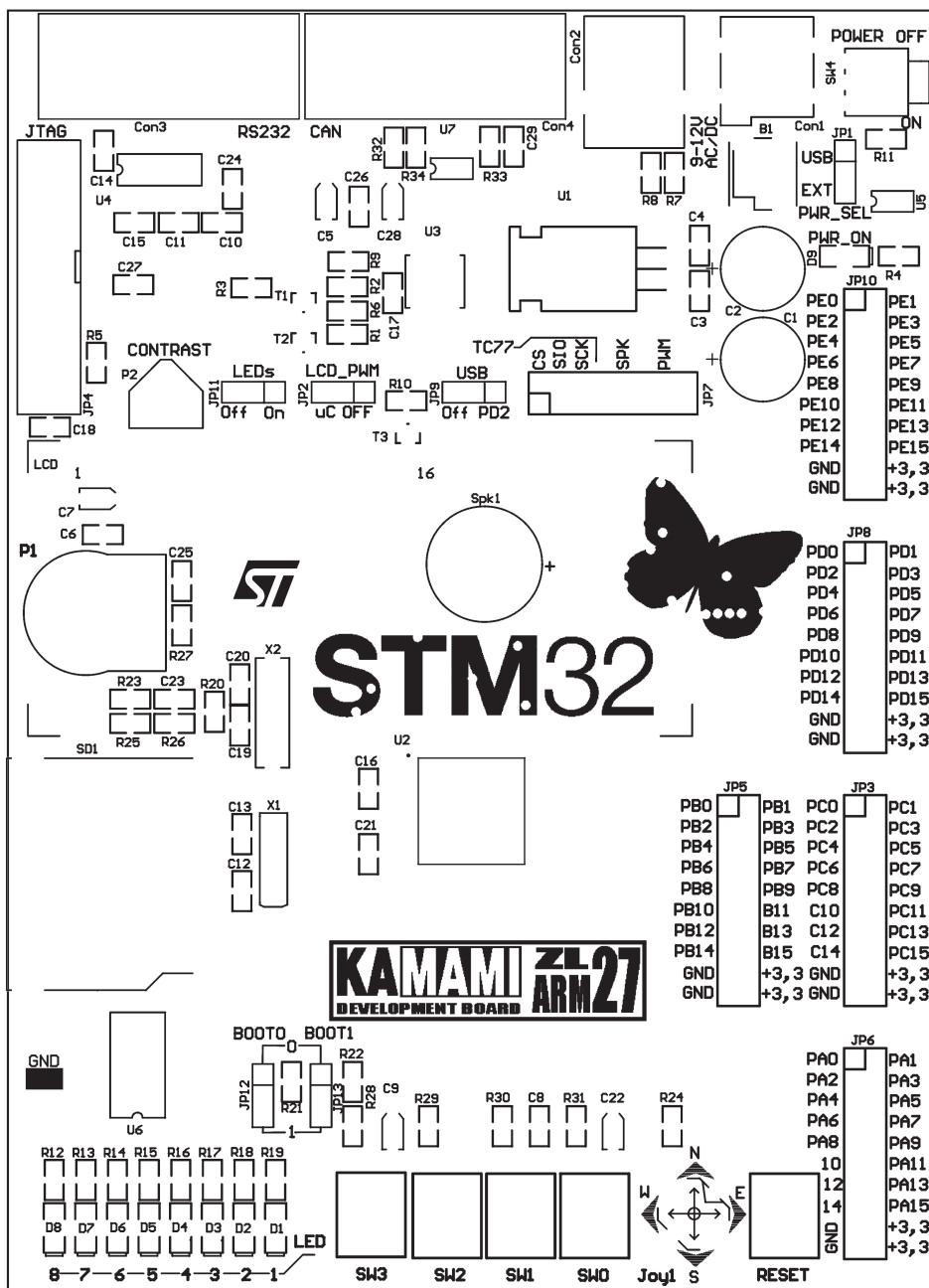
1.1 Cel

Celem tego ćwiczenia jest zapoznanie studenta z obsługą środowiska programistycznego Keil µVision 5 oraz nauka podstawowej obsługi mikrokontrolera. Dotyczy to zarówno symulowanej postaci mikrokontrolera STM32F103, jak i jego fizycznej wersji zamontowanej na płycie rozwojowej ZL27ARM.

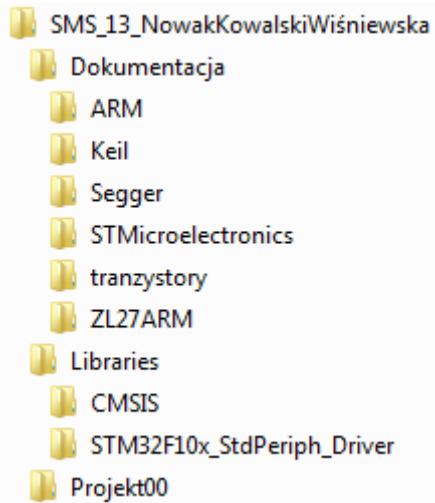
1.2 Przebieg laboratorium (prowadzenie za rączkę)

Instalacja środowiska Keil µVision 5: Do realizacji poniższych ćwiczeń wymagane jest środowisko Keil µVision 5 wraz z oprogramowaniem pozwalającym na programowanie i symulowanie mikrokontrolerów z rodziny STM32. Instalator można pobrać ze strony <https://www.keil.com/demo/eval/arm.htm>, gdzie należy się zarejestrować (bez ponoszenia jakichkolwiek opłat i narażania się na niechciane wiadomości). Po zarejestrowaniu otwiera się strona z linkiem do MDK521A.EXE (nazwa na dzień 04.10.2016r.), który należyściągnąć i uruchomić.

Po instalacji otworzy się okno Pack Installer'a, gdzie należy zaczekać aż skończy on aktualizować listę swoich paczek. Gdy już tak się stanie, z drzewa po lewej stronie należy wybrać nazwę mikrokontrolera



Rysunek 2: Schemat zestawu uruchomieniowy ZL27ARM



Rysunek 3: Struktura katalogów dla projektu 00

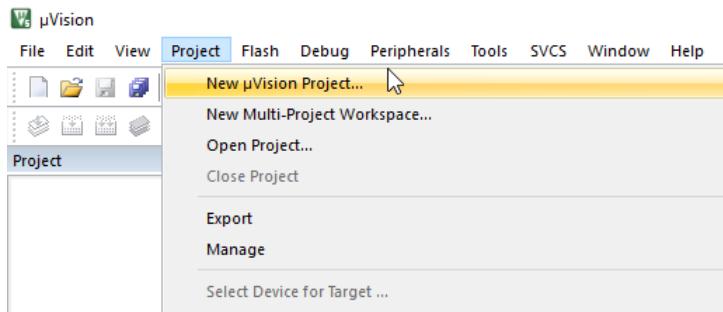
STM32F103VB (All Devices → STMicroelectronics → STM32F1 Series → STM32F103 → STM32F103VB). Po prawej stronie zaktualizuje się lista paczek Zainstalować paczkę Keil::STM32F1xx_DFP i zaktualizować paczkę ARM::CMSIS. Tak przygotowane środowisko pozwoli na symulację wyżej wspomnianego mikrokontrolera, a co za tym idzie – zapoznanie się ze środowiskiem programistycznym na przykładzie symulowanego programu.

Stworzenie pierwszego projektu (symulacja): Zapoznanie się ze środowiskiem Keil µVision 5 warto rozpocząć od praktyki – pierwszego projektu. Projekt ten będzie systematycznie rozwijany, a następnie powielany w celu zachowania poprzednich wersji. Pierwszy projekt nie wymaga posiadania mikrokontrolera ani programatora – wszystkie aspekty sprzętowe są symulowane dzięki środowisku Keil µVision 5, natomiast późniejsze przejście z symulowanego środowiska do uruchomienia programu na mikrokontrolerze jest wyjątkowo łatwe – nie jest wymagana żadna modyfikacja kodu programu, a konfiguracja zmienia się jedynie nieznacznie.

Utworzenie pierwszego projektu należy zacząć od założenia katalogu roboczego. Jego nazwa powinna być unikalna, stąd proponowana jest postać: SMS_{1}_{2}, gdzie w miejsce {1} należy wpisać numer grupy, {2} nazwiska członków grupy (wielką literą, bez odstępów, bez polskich znaków). Przykładową nazwą spełniającą te kryteria jest np. SMS_13_NowakKowalskiWisniewska.

Do utworzonego katalogu należy skopiować biblioteki (katalog **Libraries**) oraz dokumentacje (katalog **Dokumentacja**) ze wskazanego przez prowadzącego źródła. Na koniec należy utworzyć katalog z pierwszym projektem – **Projekt00** i skopiować do niego pliki:

- stm32f10x_conf.h
- stm32f10x_it.c
- stm32f10x_it.h



Rysunek 4: Tworzenie nowego projektu

ze wskazanego przez prowadzącego źródła. Po zakończeniu struktura katalogów powinna być taka jak na Rys. 3.

Aby utworzyć nowy projekt należy uruchomić środowisko Keil μVision 5, a następnie wybrać menu *Project → New μVision Project...* (Rys. 4). Plik projektu należy zapisać w przygotowanym katalogu **Projekt00**, pod nazwą **projekt00** (aby odróżnić ją od nazwy katalogu). Jako platformę docelową należy z drzewa dostępnych platform wybrać *STMicroelectronics → STM32 F1 Series → STM32 F103 → STM32F103VB* (Rys. 5), zatwierdzając przyciskiem *OK*. Ponieważ w tym projekcie nie będą dodawane żadne biblioteki, pojawiające się okno *Manage Run-Time Environment* należy zamknąć przyciskiem *OK*.

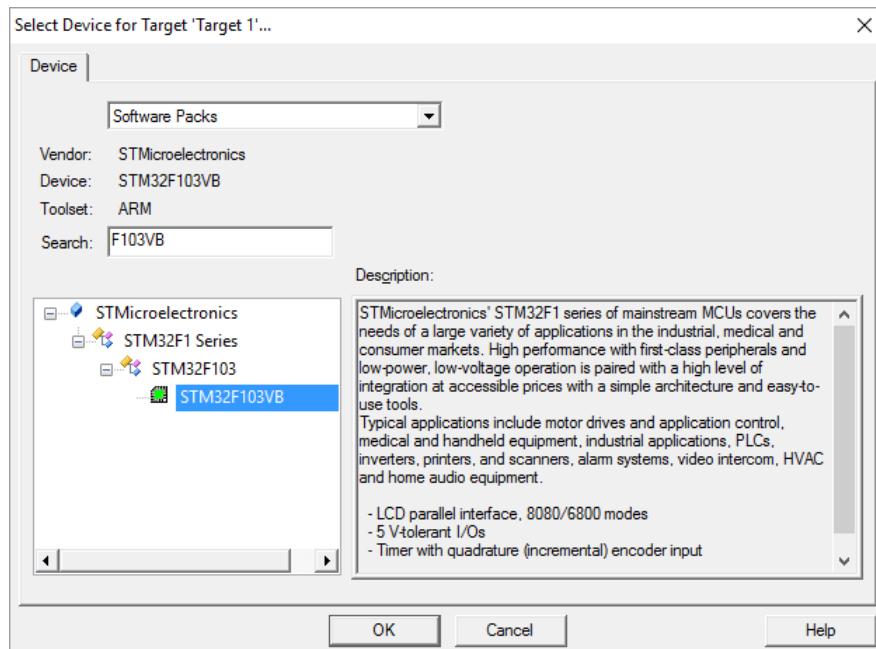
Inicjalizację projektu należy rozpocząć od stosownego podziału plików na katalogi. W tym celu w katalogu głównym **Target 1** należy utworzyć katalogi (kliknąć prawym przyciskiem myszy na **Target 1** i wybrać opcję *Add Group...*): **UserCode** (na kod użytkownika), **StdPeriphDrv** (na biblioteki do obsługi periferii), **CMSIS** (na biblioteki do obsługi rdzenia) oraz **RVMDK** (na plik startowy mikrokontrolera). Katalog, który domyślnie zostaje utworzony w nowym projekcie **Source Group 1** można usunąć lub zmienić mu nazwę na jedną z wyżej wymienionych.

Następnie trzeba uzupełnić utworzone katalogi odpowiednimi plikami. Warto utworzyć najpierw plik **main.c** poprzez kliknięcie prawym przyciskiem myszy katalogu **UserCode**, a następnie wybranie opcji *Add New Item to Group 'UserCode'*. Z okna, które się pojawiło należy wybrać plik typu **C File (.c)**, nazwać go **main.c** i zatwierdzić przyciskiem *OK*. Plik ten automatycznie zostanie otwarty w edytorze. Należy do niego zapisać minimalny działający kod, np.:

```
#include "stm32f10x.h"
int main(void){
    return 0;
}
```

Następnie trzeba dodać kolejne pliki (dwukrotnie kliknąć lewym przyciskiem myszy na katalog) do następujących katalogów:

- do **UserCode** dodać:
 - .\stm32f10x_it.c
- do **StdPeriphDrv** dodać:

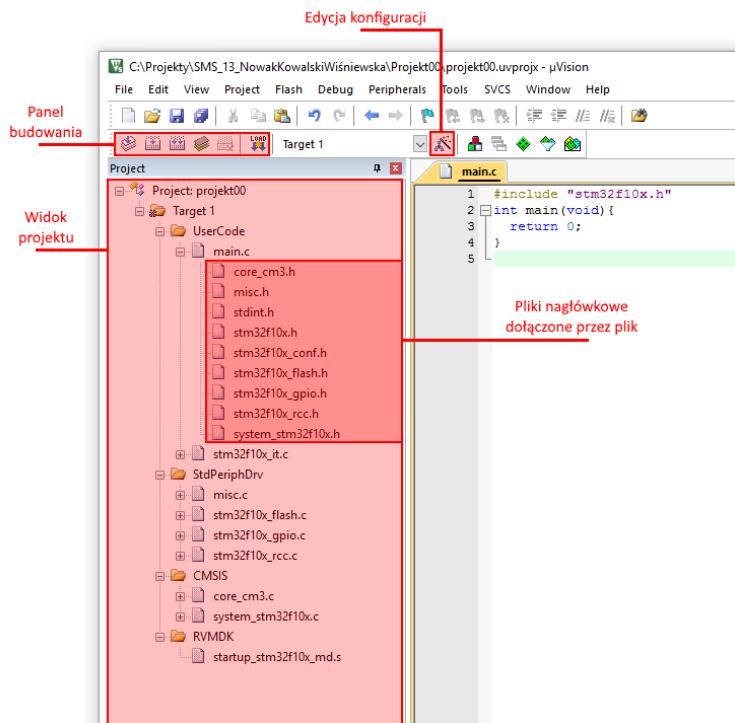


Rysunek 5: Wybór mikrokontrolera

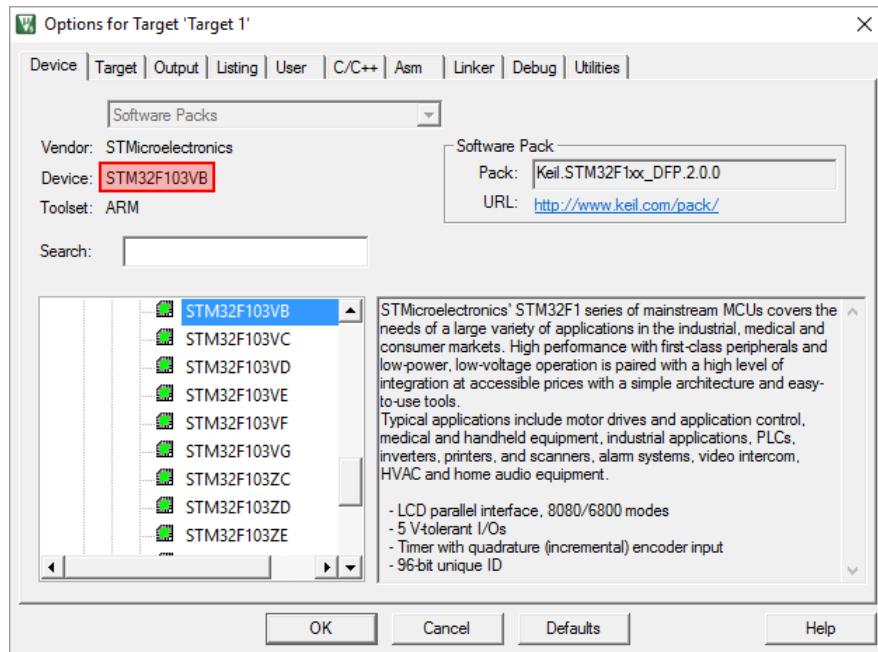
- ..\Libraries\STM32F10x_StdPeriph_Driver\src\misc.c
- ..\Libraries\STM32F10x_StdPeriph_Driver\src\stm32f10x_flash.c
- ..\Libraries\STM32F10x_StdPeriph_Driver\src\stm32f10x_gpio.c
- ..\Libraries\STM32F10x_StdPeriph_Driver\src\stm32f10x_rcc.c
- do CMSIS dodać:
 - ..\Libraries\CMSIS\CM3\CoreSupport\core_cm3.c
 - ..\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\system_stm32f10x.c
- do RVMDK dodać:
 - ..\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm←
 → \startup_stm32f10x_md.s

Wszelkie powyższe ścieżki są względne. Zakłada się, że użytkownik znajduje się w głównym katalogu swojego projektu – w tym przypadku *Projekt00*. Po dodaniu wszystkich potrzebnych plików, struktura projektu powinna być taka jak na Rys. 6.

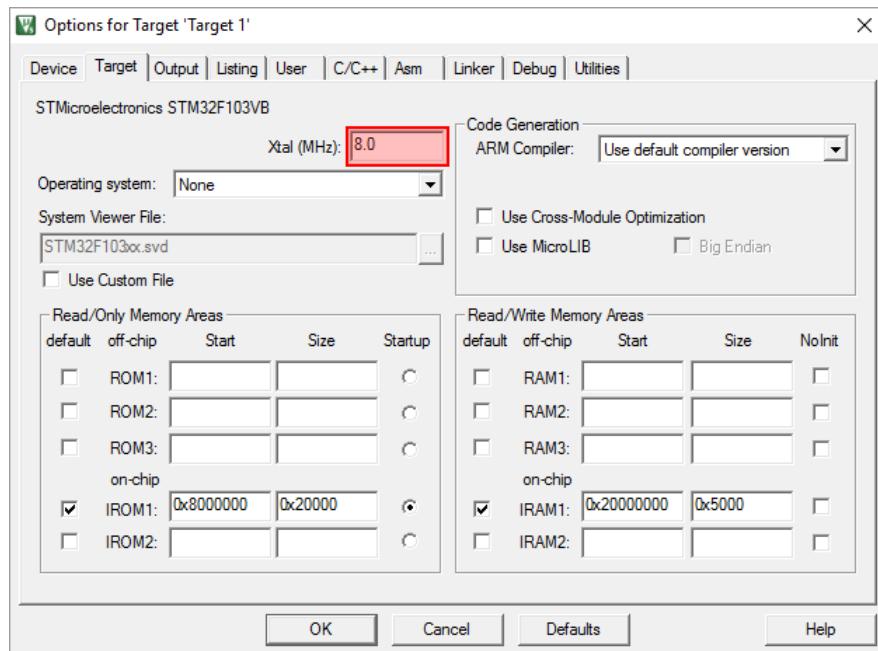
Gdy już wszystkie pliki są dodane do projektu można przejść do jego konfiguracji. W tym celu należy dwukrotnie kliknąć *Target 1* z drzewa projektu, a następnie wybrać menu *Project → Options for Target 'Target 1'*.... Konfiguracja powinna uwzględniać:



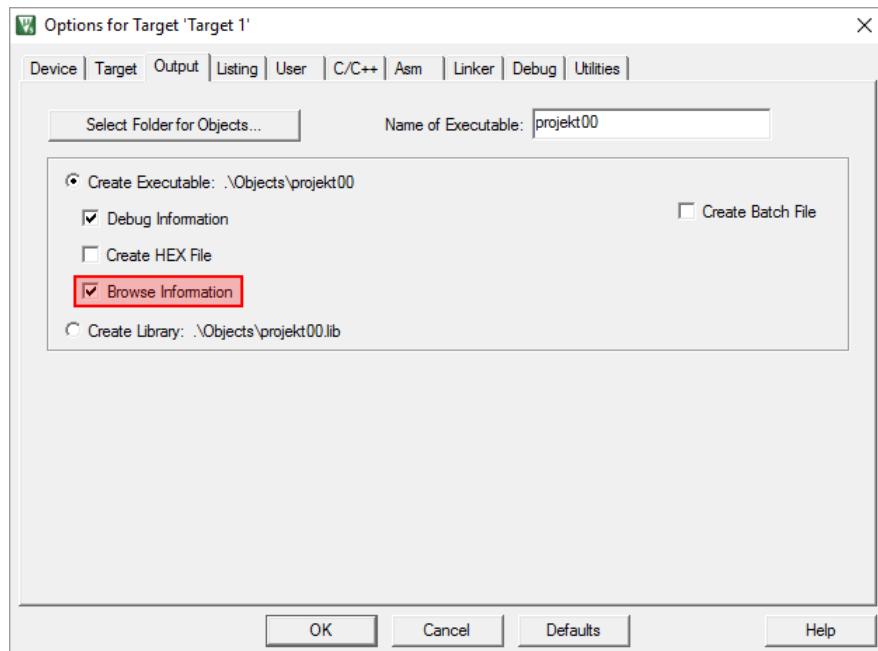
Rysunek 6: Widok projektu z gotową strukturą plików



Rysunek 7: Konfiguracja projektu – zakładka Device

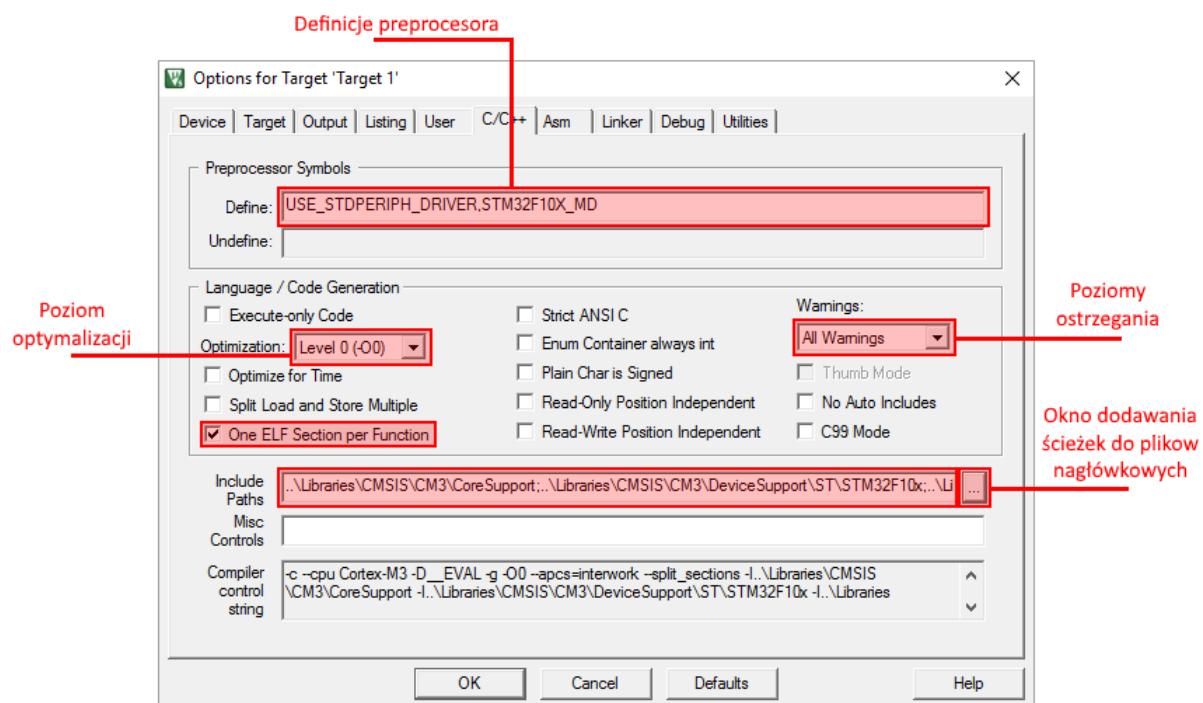


Rysunek 8: Konfiguracja projektu – zakładka Target

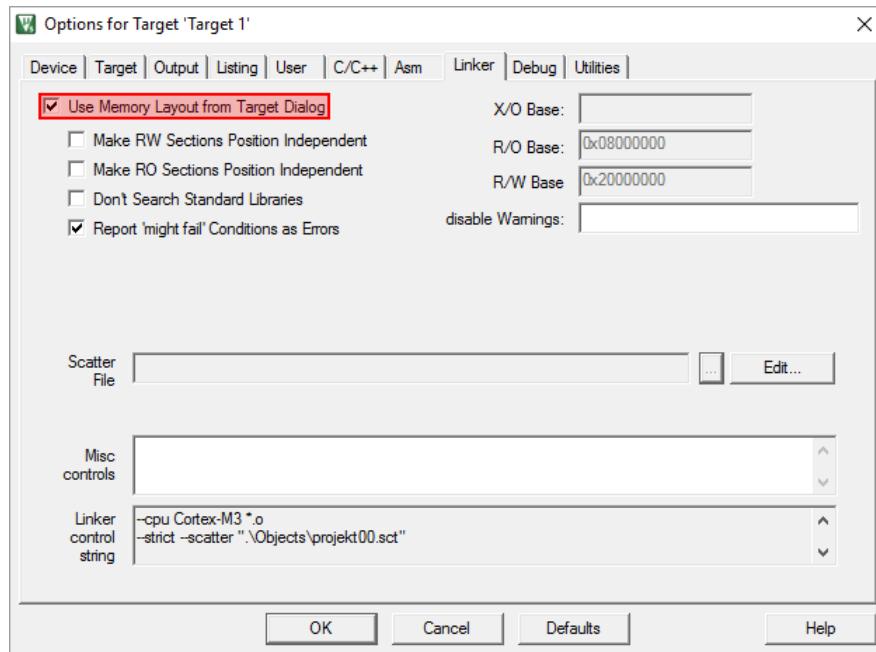


Rysunek 9: Konfiguracja projektu – zakładka Output

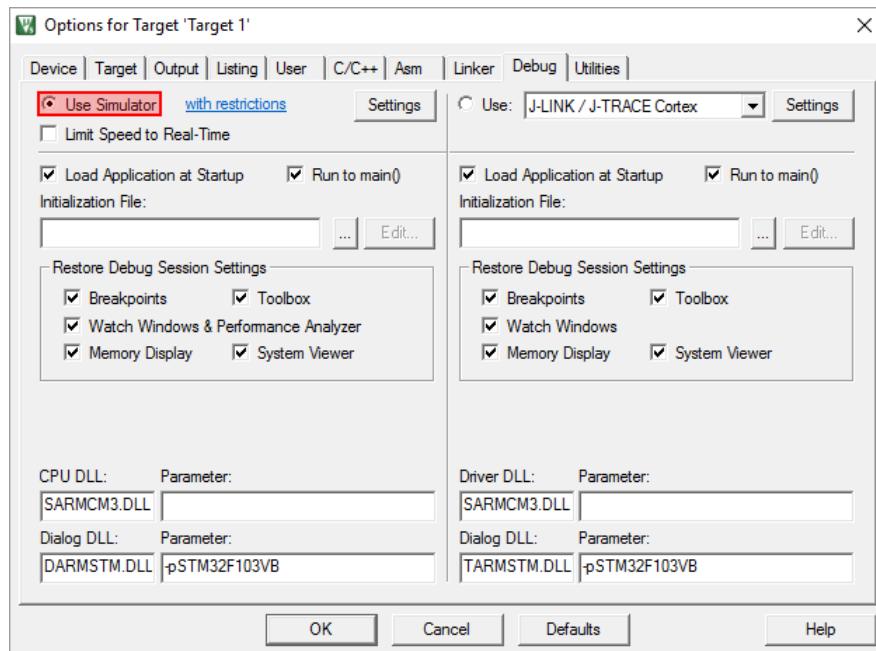
- w zakładce Device (Rys. 7)
 - wybór odpowiedniego mikrokontrolera, tj. STM32F103VB,
- w zakładce Target (Rys. 8)
 - ustawienie odpowiedniej częstotliwości kwarca *Xtal (MHz)*, tj. na 8,0 (taki właśnie jest zamontowany w rozważanym zestawie uruchomieniowym),
- w zakładce Output (Rys. 9)
 - zaznaczenie opcji *Browse Information*, w celu umożliwienia wyszukiwania funkcji i zmiennych w plikach źródłowych,
- w zakładce C/C++ (Rys. 10)
 - uzupełnienie pola *Define* o definicje USE_STDPERIPH_DRIVER, pozwalające na wykorzystanie standardowej biblioteki do obsługi peryferialnych oraz STM32F10X_MD umożliwiające warunkową komplikację dla urządzenia typu *medium-density* – definicje należy rozdzielić przecinkiem,
 - wybór poziomu optymalizacji na zerowy, tj. *Level 0 (-O0)*,
 - zaznaczenie opcji *One ELF Section per Function*,
 - wybór wyświetlania wszystkich ostrzeżeń, tj. *All Warnings*,



Rysunek 10: Konfiguracja projektu – zakładka C/C++



Rysunek 11: Konfiguracja projektu – zakładka Linker



Rysunek 12: Konfiguracja projektu – zakładka Debug

- dodanie ścieżek do plików nagłówkowych (kliknąć lewym przyciskiem myszy na trzy kropki, na prawo od *Include Paths*, utworzyć rekord klikając klawisz *Insert*, wpisać ścieżkę):

```
..\\Libraries\\CMSIS\\CM3\\CoreSupport
..\\Libraries\\CMSIS\\CM3\\DeviceSupport\\ST\\STM32F10x
..\\Libraries\\STM32F10x_StdPeriph_Driver\\inc
..\\Projekt00
```

- w zakładce Linker (Rys. 11)
 - zaznaczenie opcji *Use Memory Layout from Target Dialog*,
- w zakładce Debug (Rys. 12)
 - zaznaczenie opcji *Use Simulator*.

Tak sporządzona konfiguracja pozwala na wstępna komplikację projektu – wybrać menu *Project → Build Target* (lub wcisnąć klawisz F7). Komplikacja nie powinna zgłosić żadnego błędu ani ostrzeżenia, pozwalając na napisanie pierwszego programu.

Aby program ubogacić w treść należy przepisać przykładowy kod do pliku `main.c`:

```
*****
 * projekt00: symulacja komputerowa
 ****
#include "stm32f10x.h"

char strDst[32] = "\0";

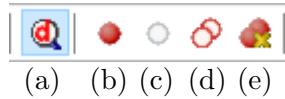
int copyStr(char *, char *);

int main(void){
    copyStr(strDst, "Source String: 0123456789");
    while(1);
}

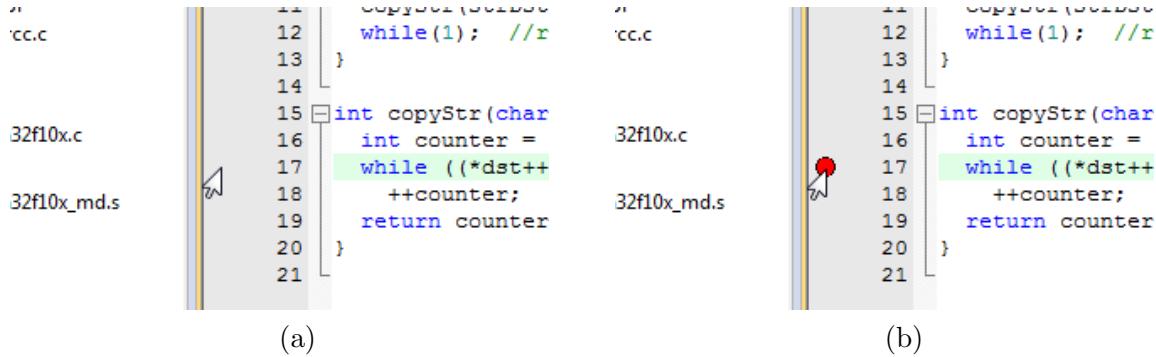
int copyStr(char *dst, char *src){
    int counter = 0;
    while( src[counter] != '\0' ){
        dst[counter] = src[counter];
        ++counter;
    }
    return counter;
}
```

a następnie ponownie skompilować cały projekt.

Program skompilowany w trybie symulacyjnym z założenia jest uruchamiany na żądanie. Co więcej uruchamiany jest on zawsze w trybie debugowania. Debugowanie to, w skrócie, proces detekcji i eliminacji błędów polegający na kontroli wykonywanych operacji programu oraz kontroli zawartości poszczególnych fragmentów pamięci i rejestrów. Wykonywane jest to poprzez ustawianie *pulapek programowych* (ang. *Breakpoint*), które oznaczają linię kodu w języku C lub instrukcję asemblerową, przed której wykonaniem



Rysunek 13: Narzędzia do debugowania: a) włączenie/wyłączenie trybu debugowania, b) wstawienie/usunięcie pułapki programowej, c) aktywowanie/dezaktywowanie pułapki programowej, d) dezaktywowanie wszystkich pułapek programowych, e) usunięcie wszystkich pułapek programowych

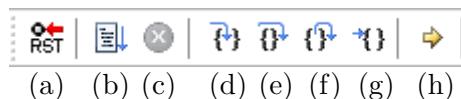


Rysunek 14: Pułapka programowa w linii 17: a) usunięta, b) ustawiona

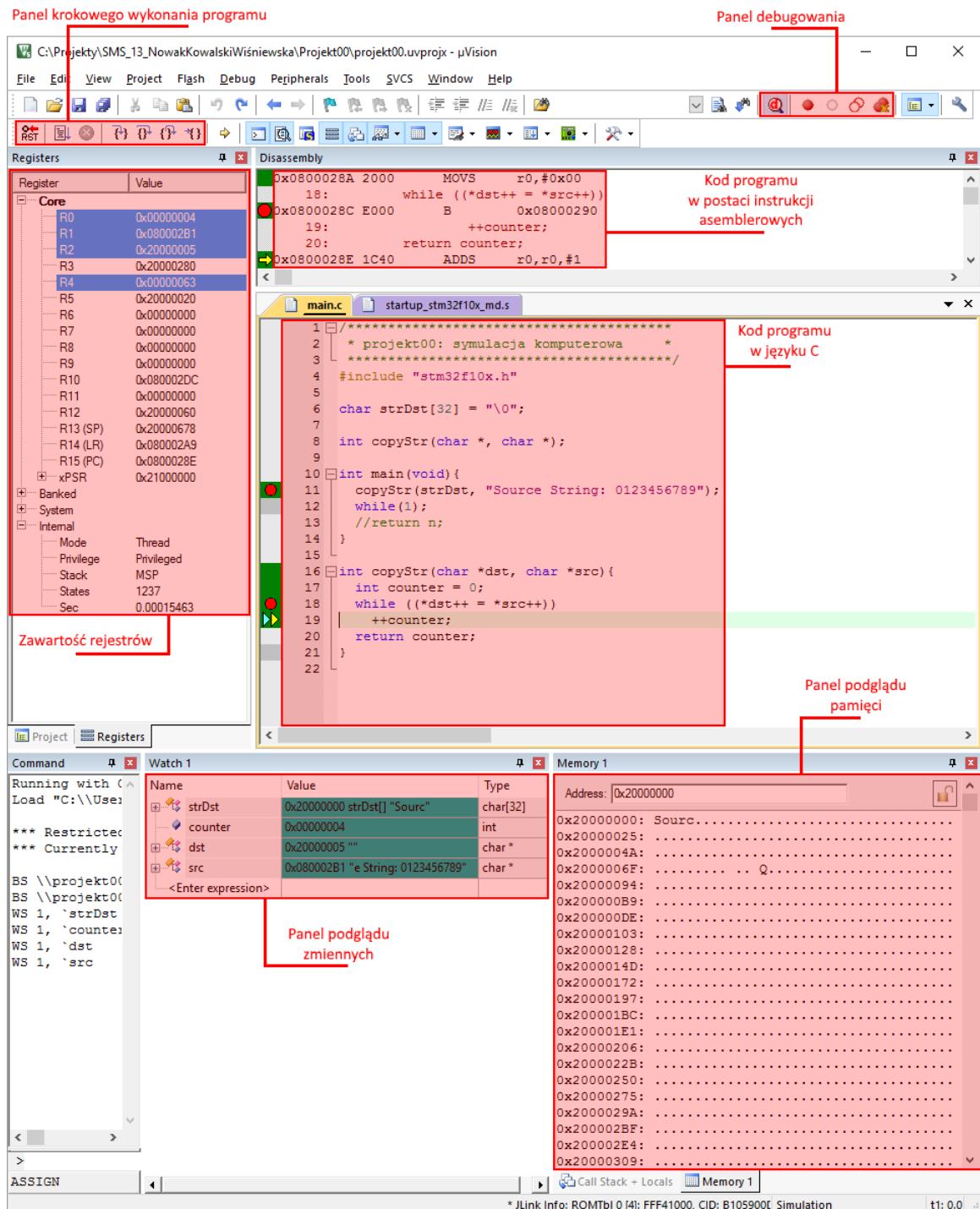
użytkownik chce wstrzymać działanie programu. Należy jednak pamiętać, że pojedyncza linia kodu może zostać skompilowana do kilku instrukcji asemblerowych, a w przypadku włączenia optymalizacji mogą być widoczne uproszczenia kodu utrudniające porównanie kodu w języku C oraz kod asemblerowy.

Aby wejść w tryb debugowania należy wcisnąć przycisk widoczny na Rys. 13a) lub kliknąć menu *Debug → Start/Stop Debug Session* (kombinacja klawiszy **CTRL+F5**). Pułapki programowe można stawiać zarówno w trybie debugowania jak i poza – dodaje się je przy użyciu przycisku widocznego na Rys. 13b) lub (wygodniej) poprzez kliknięcie na lewo od linii, w której chce się postawić pułapkę (Rys. 14). Kliknięcie na postawioną pułapkę usuwa ją. Aby aktywować/dezaktywować pułapkę należy kliknąć na nią prawym przyciskiem myszy i wybrać opcję *Enable/Disable Breakpoint*.

Po wejściu w tryb debugowania program zatrzymuje się przed pierwszą linią funkcji **main**. Jest to dobry moment na ustawienie potrzebnych pułapek, szczególnie jeśli mają się one znaleźć przy poszczególnych instrukcjach asemblerowych. Znaczenie przycisków służących do odpowiedniego wykonywania instrukcji uruchomionego programu jest przedstawione na Rys. 15.



Rysunek 15: Narzędzia do kontrolowania wykonania programu: a) reset CPU, b) rozpoczęcie wykonania kodu c) przerwanie wykonania kodu, d) wykonanie jednej linii (z ewentualnym zagłębianiem się w funkcję), e) wykonanie jednej linii (bez zagłębiania się w funkcję), f) wykonanie funkcji do końca i przejście do funkcji nadzędnej, g) wykonanie kodu do miejsca, w którym znajduje się kursor, h) pokazanie następnej instrukcji



Rysunek 16: Widok debugowania



Rysunek 17: Programator firmy Segger, *j-link EDU*

Narzędzie do debugowania (Rys. 16) oferuje ogromne możliwości: pozwala podejrzeć pamięć, sterować odpowiednimi rejestrami i peryferiami, można dokonywać analizy stanów logicznych oraz wiele innych. Do najbardziej przydatnych należą podgląd pamięci i podgląd zmiennych w programie. Podgląd pamięci jest widoczny w panelu *Memory 1*, który jest domyślnie umiejscowiony w prawym dolnym rogu ekranu, razem z panelem *Call Stack + Locals*. Rozważając program napisany wcześniej warto spojrzeć na blok pamięci rozpoczynając od adresu 0x20000000 – tutaj kopiowany jest ciąg znaków, a więc jest to adres pod którym znajduje się tablica znaków `strDst[32]`. Aby zmienić zapis decymalny na znaki ASCII należy kliknąć prawym przyciskiem na panel *Memory 1* i zaznaczyć opcję *ASCII*.

Podgląd zmiennych w programie domyślnie nie jest włączony – należy go otworzyć klikając menu *View* → *Watch Windows* → *Watch 1*. W panelu, który się pojawił w polu z napisem <Enter expression> należy wpisać nazwę zmiennej, której podgląd chce się uzyskać. W rozważanym programie warto podejrzeć zmienną `strDst`. W kolumnie *Value* pojawił się adres obserwowanej zmiennej, oraz jej zawartość (na początku programu będąca pustym ciągiem znaków), a w kolumnie *Type* widoczny jest zadeklarowany typ tej zmiennej. Wraz z wykonywaniem się programu zmieniona ta jest uzupełniana kolejnymi znakami, co jednocześnie jest odzwierciedlane w panelu *Watch 1*. Warto zauważyć, że wartość tej zmiennej można w trakcie działania programu modyfikować (w tym celu należy rozwinąć poddział tej zmiennej i edytować poszczególne jej elementy).

Stworzenie pierwszego projektu (zestaw uruchomieniowy ZL27ARM): Następnym krokiem po zapoznaniu się z procesem tworzenia oraz testowania projektu w trybie symulacyjnym jest wykonanie analogicznego zadania z wykorzystaniem prawdziwego mikrokontrolera oraz programatora.

Jako mikrokontroler użyty zostanie STM32F103VB, zawierający się w zestawie uruchomieniowym

ZL27ARM. Ogólne informacje dotyczące tego zestawu znajdują się w pierwszych rozdziałach, natomiast szczegółowe informacje na temat samego mikrokontrolera STM32F103VB można znaleźć w dokumentacji znajdującej się na stronie producenta i w katalogu *Dokumentacja* (wartymi uwagi są pliki Datasheet oraz RM0008). Programowanie zestawu ZL27ARM (Rys. 1) odbywa się będzie za pośrednictwem programatora *j-link* (firmy *Segger*) w wersji edukacyjnej (*EDU*) – Rys. 17. Jest on podłączany poprzez złącze JTAG (*Joint Test Action Group*), które pozwala na testowanie (w tym debugowanie i śledzenie wykonania programu) procesora wlutowanego w zmontowaną płytę drukowaną. Połączenie między zestawem ZL27ARM a programatorem *j-link EDU* następuje przy użyciu 20-żyłowego kabla, który z jednej strony jest wpięty w programator (złącze opisane etykietą *Target*), a z drugiej wpięte w złącze o etykiecie *JTAG* znajdujące się na mikrokontrolerze. Specjalnie umieszczone wypustki złączy znajdujących się na kablu skutecznie uniemożliwiają wpięcie go w innej pozycji niż poprawna. Połączenie programatora z komputerem następuje poprzez kabel USB. Od strony programatora jest to wtyczka USB typu B, natomiast od strony komputera wtyczka USB typu A. Poprawne podłączenie programatora powinno być sygnalizowane przez świecenie się (z okresowym chwilowym przygasaniem) zielonej diody znajdującej się na jego obudowie, nad logo producenta.

Zestaw uruchomieniowy ZL27ARM można uruchomić w różnych konfiguracjach. W tym ćwiczeniu oczekiwana konfiguracją jest:

- zestaw zasilany jest z portu USB,
- program uruchamiany jest zewnętrznej pamięci Flash,
- diody LED, sterowanie podświetleniem wyświetlacza LCD oraz komunikacja po USB wyłączone.

Przekłada się to na następujące ustawienia zworek:

Nazwa	Pozycja
<i>PWR_SEL</i>	USB
<i>BOOT0</i>	0
<i>BOOT1</i>	0
<i>LEDs</i>	OFF
<i>LCD_PWM</i>	OFF
<i>USB</i>	OFF

Ponieważ mikrokontroler nie jest zasilany przez programator, należy go podłączyć kablem USB do źródła zasilania (np. komputera). W tym celu (przy przełączniku zasilania *POWER* ustawionym na *OFF*) do złącza opisanego etykietą *Con2* należy podłączyć wtyczkę typu B, natomiast do komputera wtyczkę typu A. Po podłączeniu wszystkich elementów można ustawić przełącznik zasilania *POWER* w położenie *ON*. Jeśli wszystko zostało zrealizowane poprawnie powinna zapalić się zielona LED o nazwie *PWR_ON*.

Gdy sprzęt już jest podłączony i włączony można przejść do zmiany konfiguracji programowej, tj. do modyfikacji ustawień projektu w Keil µVision. W tym celu ponownie klikamy *Project → Options for Target 'Target 1'...*, a w otwartym oknie dokonujemy zmian w zakładce *Debug*:

- zaznaczenie opcji *Use:*,
- wybranie z listy *J-LINK / J-TRACE Cortex*.

Następnie w tej samej zakładce należy kliknąć przycisk *Settings*, aby skonfigurować programator. Chwilę po otwarciu się nowego okna wpisana zostanie automatycznie domyślna konfiguracja wykrytego programatora (należy zaakceptować *Terms of Use* na cały dzień jeśli będzie taka możliwość). Aby upewnić się, czy jest ona poprawna warto sprawdzić czy zgadzają się numery seryjne: widoczny w oknie (pole *SN*) oraz znajdujący się na spodzie programatora (pole *S/N*). Przydatną opcją jest możliwość wymuszenia restartu mikrokontrolera i uruchomienia nowego programu tuż po jego załadowaniu. Służy do tego opcja *Reset and Run* w zakładce *Flash Download*. Pozostałe opcje należy pozostawić bez zmian i zamknąć widoczne okna poprzez wciśnięcie dwóch kolejnych przycisków *OK*.

Gdy konfiguracja jest już gotowa można wykonać wgranie programu (tego co wcześniej) na mikrokontroler poprzez wciśnięcie przycisku *Download* (klawisz F8) lub poprzez menu *Flash → Download*. W panelu *Build Output* powinny pojawić się linie:

```
Erase Done.  
Programming Done.  
Verify OK.  
Application running ...  
Flash Load finished at HH:MM:SS
```

gdzie na końcu ostatniej linii, zamiast *HH:MM:SS*, wstawiona jest godzina, minuta oraz sekunda w której zakończone zostało ładowanie programu do pamięci mikrokontrolera. Oznacza to zarazem, że proces ładowania programu zakończył się sukcesem. Warto pamiętać, że w przeciwieństwie do trybu symulacyjnego, teraz program uruchamiany jest natychmiastowo po wgraniu go na mikrokontroler (dzięki wcześniej zaznaczonej opcji *Reset and Run*). Jednocześnie aby uruchomić program od samego początku należy zrestartować mikrokontroler klikając przycisk *Reset* znajdujący się na płytce uruchomieniowej lub wejść w tryb debugowania i z jego poziomu rozpoczęć wykonanie programu od początku (przycisk widoczny na Rys. 15a).

Od tej pory możliwe jest uruchomienie trybu debugowania w ten sam sposób co w przypadku wcześniej przeprowadzanej symulacji. Różnica jest taka, że teraz wszystkie operacje wykonywane są na mikrokontrolerze. Warto powtórzyć ćwiczenie z krokową analizą wykonania przykładowego programu kopiącego ciągi znaków aby zaobserwować, że nie ma żadnej różnicy w kodzie, a w wykonaniu jest ona znikoma.

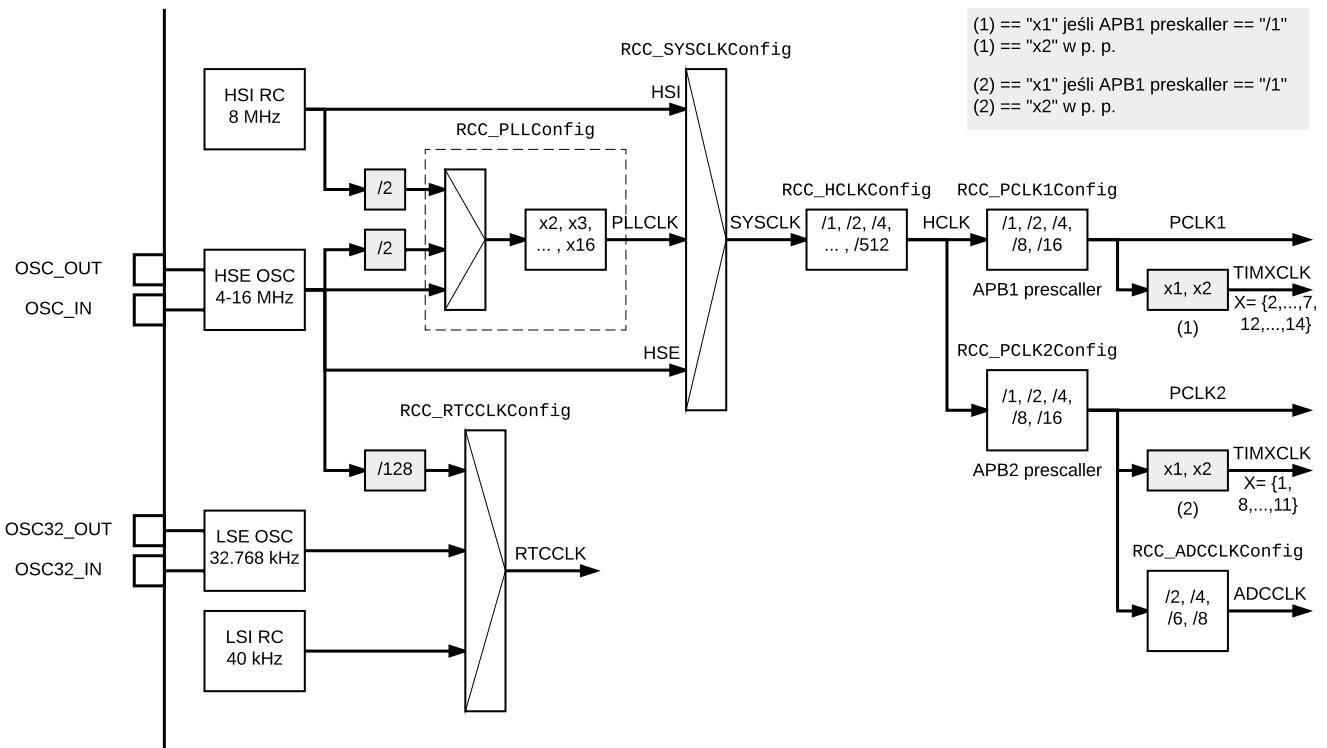
1.3 Przebieg laboratorium (samodzielnie wykonywane zadanie)

Mikrokontroler składa się z wielu (tysięcy) bramek logicznych, które pracują w trybie synchronicznym. Oznacza to, że są one taktowane zegarem i wraz z nim zmienia się wartość na wyjściu bramek. Dzięki temu unika się takich problemów jak zjawisko hazardu lub wyścigu, a więc problemów wynikających z niezerowego czasu propagacji sygnału logicznego.

Wprowadzenie sygnału zegarowego do układu mikrokontrolera pociąga za sobą także inne zjawisko – wszystkie bramki przełączają się w tej samej chwili, a co za tym idzie cały układ pobiera impulsowo duży prąd. Z drugiej strony w pozostałych chwilach mikrokontroler nie pobiera praktycznie energii.

Za dostarczenie do wszystkich układów odpowiedniego sygnału zegarowego odpowiada moduł *Reset and Clock Control* (RCC). Źródłem sygnałów taktujących mogą być:

- *Low-Speed Internal (LSI)* – wewnętrzny oscylator RC 40 kHz,
- *High-Speed Internal (HSI)* – wewnętrzny oscylator RC 8 MHz,



Rysunek 18: Uproszczony schemat układu taktowania procesora i peryferiali – pełna wersja znajduje się w dokumencie RM0008 (Rys. 8)

- *Low-Speed External* (LSE) – zewnętrzny rezonator kwarcowy 32,768 kHz,
- *High-Speed External* (HSE) – zewnętrzny rezonator kwarcowy 8 MHz.

Domyślnie (tj. tuż po resecie mikrokontrolera) wykorzystywany jest sygnał HSI oraz LSI. Są to sygnały o niewielkiej dokładności (tj. około 1%) i mimo, że mogą być w wielu aplikacjach z powodzeniem wykorzystywane, warto rozważyć użycie tanich, znacznie dokładniejszych rezonatorów kwarcowych. Ponadto moduł RCC jest wyposażony w konfigurowalne dzielniki częstotliwości i pętlę *Phase Locked Loop* (PLL) – można ją utożsamiać z „mnożnikiem częstotliwości”. Uproszczony schemat układu taktowania procesora i peryferiali widoczny jest na Rys. 18. Na schemacie są dodatkowo umieszczone nazwy funkcji ze standardowej biblioteki do obsługi peryferiali, które pozwalają na konfigurację poszczególnych elementów i sygnałów taktujących (nazwy te są zapisane czcionką stałoszerokościową).

Kolejne ćwiczenie to konfiguracja sygnałów HCLK, PCLK1 oraz PCLK2 – warto wykonać jako osobny projekt. W tym celu najłatwiej jest otworzyć ostatni projekt, wyczyścić go poprzez menu *Projekt* → *Clean Targets*, a następnie go zamknąć. Dalej należy skopiować ten projekt (tj. cały katalog Projekt00), zmieniając mu nazwę na Projekt01 i odpowiednio modyfikując jego zawartość. Przede wszystkim należy z nowo utworzonego katalogu usunąć zawartość podkatalogów *Listings* i *Objects*, w tym wszelkie pliki o rozszerzeniach *.crf oraz plik projekt00.sct. Nazwy wszystkich pozostałych plików w katalogu z projektem, które zaczynają się od projekt00, zmieniamy na projekt01, tj.:

- projekt00.uvoptx zmieniamy na projekt01.uvoptx,
- projekt00.uvguix.Student na projekt01.uvguix.Student (ostatni człon jest zależny od nazwy konta na którym jest zalogowany użytkownik),
- projekt00.uvprojx na projekt01.uvprojx.

Następnie należy uruchomić plik projektu o nowej nazwie, w którym (po otwarciu) w *Options for Target 'Target 1'*, w zakładce *Output* zmienić trzeba wartość pola *Name of Executable* z projekt00 na projekt01. Na koniec nie można zapomnieć o zmianie pola *Include Paths* z zakładki *C/C++*, gdzie oczywiście uaktualniamy wszelkie pozycje zawierające nazwę Projekt00 zmieniając te fragmenty na Projekt01 (nazwa katalogu zaczyna się wielką literą).

W tym projekcie będą wykorzystywane moduły do obsługi RCC, pamięci Flash oraz GPIO, w związku z czym należy dodać do projektu odpowiednie pliki. Te akurat już zostały dodane w ramach poprzedniego projektu, są to:

- ..\Libraries\STM32F10x_StdPeriph_Driver\src\stm32f10x_flash.c
- ..\Libraries\STM32F10x_StdPeriph_Driver\src\stm32f10x_gpio.c
- ..\Libraries\STM32F10x_StdPeriph_Driver\src\stm32f10x_rcc.c

Aby jednak zostały one faktycznie dołączone do projektu należy się upewnić, że w pliku **stm32f10x_conf.h** odkomentowane są następujące linijki:

```
#include "stm32f10x_flash.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
```

Tak skompilowany projekt jest punktem wyjścia do konfiguracji zegarów – częstotliwości sygnałów podane są przez prowadzącego. Należy również pamiętać o stosownej konfiguracji opóźnień odczytu z pamięci Flash. Wymaga to zastosowania prostych reguł zdefiniowanych w pliku PM0075:

- **FLASH_Latency_0** jeśli $0 < \text{SYSCLK} \leq 24 \text{ MHz}$
- **FLASH_Latency_1** jeśli $24 < \text{SYSCLK} \leq 48 \text{ MHz}$
- **FLASH_Latency_2** jeśli $48 < \text{SYSCLK} \leq 72 \text{ MHz}$

Jak widać sygnał SYSCLK nie może przekraczać 72 MHz – naruszenie tego ograniczenia może powodować krytyczny błąd wykonania programu. Jako jeden z dowodów na poprawną konfigurację zegarów (dokładniej zegara HCLK) należy w pętli zapalać diodę na sekundę i gasić na sekundę (co daje pojedynczy cykl o długości 2 s) zgodnie z poniższym kodem (**main.c**):

```
*****  
* projekt01: konfiguracja zegarow          *  
*****  
#include "stm32f10x.h"  
#include <stdbool.h>    // true, false  
  
#define DELAY_TIME 8000000  
  
bool RCC_Config(void);  
void GPIO_Config(void);  
void LEDOn(void);  
void LEDOff(void);  
void Delay(unsigned int);  
  
int main(void) {  
    RCC_Config();           // konfiguracja RCC  
    GPIO_Config();          // konfiguracja GPIO  
  
    while(1) {             // pętla główna programu  
        LEDOn();            // włączenie diody  
        Delay(DELAY_TIME);   // oczekanie 1s  
        LEDOff();            // wyłączenie diody  
        Delay(DELAY_TIME);   // oczekanie 1s  
    }  
}
```

W powyższym kodzie należy zmodyfikować wartość stałej **DELAY_TIME** zgodnie z poniższą tabelą

Oczekiwane HCLK	DELAY_TIME
1 MHz	143000
5 MHz	715000
13 MHz	1850000
14 MHz	2000000
15 MHz	2150000
30 MHz	3325000
72 MHz	8000000

Funkcja `main` nie jest zadeklarowana jako niezwracająca żadnej wartości (`void`) aby uniknąć ostrzeżeń kompilatora. Z tego samego powodu na końcu tej funkcji nie znajduje się `return 0;` – gdyby się tam znajdowało, to kompilator by zwrócił uwagę, że linijka ta może nigdy nie zostać wykonana z powodu poprzedzającej jej nieskończonej pętli `while(1)`. Mimo więc tej niekonsekwencji w kodzie, schemat ten będzie powtarzany w dalszych ćwiczeniach aby nie generować łatwych do wyeliminowania ostrzeżeń.

Diody w poprzednich ćwiczeniach były wyłączone poprzez ustawienie zworki JP11 o nazwie LEDs na Off. Aby można było je kontrolować należy wyłączyć mikrokontroler, przestawić zworkę na pozycję On i ponowniełączyć mikrokontroler. Diody najprawdopodobniej rozświetlą się z czasem mimo braku jakiejkolwiek interakcji ze strony użytkownika. Jest to ciekawe zjawisko wynikające z niepodciągnięcia wyjść prowadzących do diod, które niestety nie zostanie tutaj szczegółowo omówione. Należy jednak pamiętać, że zjawisko to ma wpływ wyłącznie na piny, które nie są skonfigurowane jako wyjścia – na tę chwilę nie należy się tym przejmować.

Poniżej została przedstawiona przykładowa funkcja konfiguruująca zegary na ich maksymalne dozwolone wartości (dla mikrokontrolera STM32F103VB są to: HCLK = 72 MHz, PCLK1 = 36 MHz, PCLK2 = 72 MHz) z wykorzystaniem HSE jako źródłowego sygnału SYSCLK (patrz Rys. 18).

```
bool RCC_Config(void) {
    ErrorStatus HSEStartUpStatus;                                // zmienienna opisujaca rezultat
                                                               // uruchomienia HSE
    // konfigurowanie sygnalow taktujacych
    RCC_DeInit();                                              // reset ustawien RCC
    RCC_HSEConfig(RCC_HSE_ON);                                 // wlacz HSE
    HSEStartUpStatus = RCC_WaitForHSEstartUp();                // czekaj na gotowosc HSE
    if(HSEStartUpStatus == SUCCESS) {
        FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable); // zwloka Flasha: 2 takty
        FLASH_SetLatency(FLASH_Latency_2);                      // zwloka Flasha: 2 takty

        RCC_HCLKConfig(RCC_SYSCLK_Div1);                         // HCLK=SYSCLK/1
        RCC_PCLK2Config(RCC_HCLK_Div1);                          // PCLK2=HCLK/1
        RCC_PCLK1Config(RCC_HCLK_Div2);                          // PCLK1=HCLK/2
        RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);   // PLLCLK = (HSE/1)*9
                                                               // czyli 8MHz * 9 = 72 MHz
        RCC_PLLCmd(ENABLE);                                     // wlacz PLL
        while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET);    // czekaj na uruchomienie PLL
        RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);             // ustaw PLL jako zrodlo
                                                               // sygnalu zegarowego
        while(RCC_GetSYSCLKSource() != 0x08);                  // czekaj az PLL bedzie
                                                               // sygnalem zegarowym systemu
    }
}
```

```

    // konfiguracja sygnałów taktujących używanych peryferii
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); // włącz taktowanie portu GPIO B
    return true;
}
return false;
}

```

Nazwy funkcji są wyjątkowo długie, lecz doskonale oddają ich funkcjonalność. Znaczenie ich zostało skrótnie opisane w komentarzach. Szerszy opis można znaleźć w komentarzu nad definicją funkcji (należy prawym przyciskiem myszy kliknąć na nazwę funkcji i wybrać Go To Definition Of <nazwa funkcji>). Niestety standardowa biblioteka peryferialna nie została opisana w formie dokumentacji – takową można jedynie wygenerować za pomocą narzędzia Doxygen, co jest jednak równoznaczne z czytaniem komentarzy nad definicji funkcji.

Należy pamiętać, że mikrokontroler rozpoczyna pracę ustawiając jako źródło zegara generator RC HSI. Oznacza to, że konieczna jest pełna konfiguracja modułu RCC zanim zostanie zmienione źródło sygnału zegarowego aby działał on poprawnie.

Na koniec inicjalizacji warto także włączyć taktowanie peryferialnego – w tym ćwiczeniu potrzebna jest wyłącznie jedna dioda znajdująca się na płycie uruchomieniowej, podłączona do pinu 8 portu B. Konfiguracja tego pinu znajduje się w osobnej funkcji i przebiega następująco:

```

void GPIO_Config(void) {
    // konfigurowanie portów GPIO
    GPIO_InitTypeDef GPIO_InitStructure;

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;           // pin 8
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz; // częstotliwość zmiany 2MHz
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; // wyjście w trybie push-pull
    GPIO_Init(GPIOB, &GPIO_InitStructure);           // inicjalizacja portu B
}

```

Częstotliwość zmiany określa prędkość narastania sygnału wraz z jego zmianą – w przypadkach gdy nie jest to niezbędne, warto wybierać najniższą dozwoloną wartość. Wyjście w trybie *push-pull* oznacza, że sygnał wyjściowy przyjmuje wyłącznie dwie wartości – logiczne 0 i logiczne 1. Jak okaże się w późniejszych ćwiczeniach nie jest to jedyny wybór do dyspozycji – na potrzeby tego ćwiczenia jest on jednak najrozsądzniejszy.

Funkcje służące do obsługi diody LED są zdefiniowane następująco:

```

void LEDOn(void) {
    // włączenie diody LED podłączonej do pinu 8 portu B
    GPIO_WriteBit(GPIOB, GPIO_Pin_8, Bit_SET);
}

void LEDOff(void) {
    // wyłączenie diody LED podłączonej do pinu 8 portu B
    GPIO_WriteBit(GPIOB, GPIO_Pin_8, Bit_RESET);
}

```

Widoczna funkcja `GPIO_WriteBit` służy do nadawania wartości poszczególnym bitom portów wyjściowych. W tym przypadku korzystamy z portu B, na którym modyfikujemy wartość bitu 8, któremu odpowiada

dioda o numerze 1. Bit_SET oraz Bit_RESET oznaczają odpowiednio ustawienie 1 i 0 logicznego (tj. odpowiednio zapalenie i zgaszenie diody).

Ostatnią funkcją jest programowe opóźnienie:

```
void Delay(unsigned int counter){  
    // opoznienie programowe  
    while (counter--){ // sprawdzenie warunku  
        __NOP(); // No Operation  
        __NOP(); // No Operation  
    }  
}
```

wykonuje ono w pętli: sprawdzenie warunku, dekrementację zmiennej oraz dwie puste instrukcje mikroprocesora *No Operation* (NOP). Otrzymane w ten sposób opóźnienie nie jest dokładne i wymaga wyłączenia optymalizacji kompilatora (inaczej może pominąć wykonanie takiego „bezużytecznego” kodu), lecz jest ono wystarczające do wstępnych testów. Wartość argumentu dająca opóźnienie równe 1 s jest wyznaczana eksperymentalnie i jest ona zależna od wartości HCLK.

Odczyt wartości cyfrowej Po poprawnym skonfigurowaniu sygnałów zegarowych oraz uzyskaniu odpowiedniej częstotliwości przełączania diody świecącej **należy wrócić do ustawień maksymalnych sygnałów zegarowych**, tj. ponownie skopiować definicję funkcji RCC_Config do pliku main.c.

Rozszerzeniem poprzedniego programu będzie dodanie obsługi przycisku znajdującego się na płycie rozwojowej. Konfiguracja takiego przycisku wykorzystuje ten sam mechanizm co konfiguracja pinu sterującego świeceniem diody LED. Płyta rozwojowa zawiera serię przycisków, które są podpięte pod piny od 0 (SW0) do 3 (SW3) portu A – wykorzystany zostanie pin 0. W tym momencie warto dodać kod odpowiedzialny za aktywowanie portu A (w funkcji RCC_Config):

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); // wlacz taktowanie portu GPIO A
```

Wciśnięcie tego przycisku będzie powodowało zapalenie diody LED podłączonej do pinu 9 portu B (sąsiednia dioda w stosunku do poprzednio używanej). Rozwinięcie konfiguracji pinu 9 portu B wymaga modyfikacji jednej linijki z kodu funkcji GPIO_Config:

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8 | GPIO_Pin_9; // pin 8 i 9
```

pozostała część konfiguracji pinów wyjściowych pozostaje bez zmian. Na koniec tej funkcji należy jednak dodać kod odpowiedzialny za konfigurację pinu wejściowego:

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU; // wejscie w trybie pull-up  
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

natomiast odczyt wartości cyfrowej przy użyciu tego pinu realizowany jest funkcją:

```
GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0)
```

Zwraca ona wartość 0 jeśli na podanym pinie jest napięcie równe masie, a 1 jeśli to napięcie jest równe napięciu zasilania. W tym przypadku, przycisk jest podłączony tak, aby zwierał podany pin do masy w momencie jego wciśnięcia. Gdy przycisk nie jest wciśnięty, zwiera on podany pin przez rezystor do

zasilania (3,3 V). Warto jednak zauważyć, że takie rozwiązanie w naszym przypadku jest redundantne. Dokładnie ten sam mechanizm został zrealizowany na płytce rozwojowej, co powoduje, że niejako użyte zostały dwa pociągnięcia w górę, co nie daje absolutnie żadnego zysku w stosunku do jednokrotnego podciągnięcia. Wynika z tego, że możemy wyłączyć podciągnięcie w górę w mikrokontrolerze stosując zamiast `GPIO_Mode_IPU` wartość `GPIO_Mode_IN_FLOATING`, co oznacza wyłączenie zarówno podciągania w górę jak i w dół.

Program ma działać tak, aby tak jak do tej pory – jedna dioda LED włączała się i wyłączała z okresem 2 s i wypełnieniem 50% oraz aby wcisnięcie przycisku powodowało zapalenie sąsiedniej diody LED. Ćwiczenie to ma za zadanie pokazać jakie problemy mogą wyniknąć z programowo realizowanego opóźnienia, które zostanie poprawione w jednym z dalszych projektów.

Obsługa alfanumerycznego wyświetlacza LCD: Nastepnym ćwiczeniem jest ożywienie wyświetlacza znakowego 2×16 znaków. Mimo, że implementacja obsługi tego wyświetlacza nie jest problematyczna, wykorzystana zostanie w tym celu gotowa biblioteka. Składa się ona z dwóch plików: `lcd_hd44780.c` oraz `lcd_hd44780.h`, która zawierają odpowiednio definicje i deklaracje funkcji do obsługi wyświetlacza. Znaleźć można je w katalogu `Drivers\LCD1602`, który natomiast znajduje się w miejscu podanym przez prowadzącego. Dla wygody i zachowania struktury katalogów, warto dodać ten katalog do katalogu w którym znajdują się `Projekt00` oraz `Projekt01`.

Aby dołączyć wspomniane pliki do projektu należy uzupełnić listę plików nagłówkowych dołączanych do projektu (*Include Paths* z zakładki *C/C++*) o katalog ..\Drivers\LCD1602 i dodać nową grupę do drzewa projektu (np. o nazwie `Drivers`) uzupełniając ją plikiem `lcd_hd44780.c`. Ponieważ są to pliki nie należące do standardowej biblioteki peryferialnej, nagłówek należy osobno dołączyć do pliku `main.c`, przy użyciu stosownej dyrektywy.

Omawiany wyświetlacz alfanumeryczny wyposażony jest w sterownik HD44780, który łączy się z rozważanym mikrokontrolerem poprzez 4 linie danych (transmisja dwukierunkowa), oraz dwie linie określające znaczenie przesyłanych danych (transmisja jednokierunkowa – mikrokontroler nadaje). Dodatkowo zastosowana jest linia taktująca wyświetlacz (sygnał generowany jest przez mikrokontroler). Służy ona do wyznaczania chwil, w których wyświetlacz może odebrać/wysłać dane. Poniżej przedstawiona jest tabela opisująca podłączenie wyświetlacza do mikrokontrolera:

nazwa pinu			
LCD	STM32	we/wy	opis
RS	PC12	wy	<i>Register Select</i> , wybór rejestrów: 0 – rejestr instrukcji, 1 – rejestr danych
R/W	PC11	wy	<i>Read/Write</i> , kierunek transferu 0 – zapis, 1 – odczyt
E	PC10	wy	<i>Enable</i> , sygnał zapisu/odczytu – aktywne zbocze opadające
DB7	PC0	we/wy	
DB6	PC1	we/wy	
DB5	PC2	we/wy	
DB4	PC3	we/wy	
DB3	—	—	
DB2	—	—	<i>Data Bits: b4-7</i> , trój-stanowe wejścia/wyjścia danych. W trybie z transferem 4-bitowym te cztery bity wykorzystywane są do przesyłu dwóch połówek (<i>nibble</i>) bajtu danych
DB1	—	—	
DB0	—	—	

Korzystanie z wyświetlacza należy rozpocząć od wywołania funkcji `LCD_Initialize`. Należy mieć świadomość, że funkcja ta zawiera konfigurację pinów potrzebnych przez wyświetlacz (zgodnie z powyższą tabelą), co powoduje, że konfiguracja tych samych pinów po inicjalizacji wyświetlacza może spowodować błędy w komunikacji z wyświetlaczem. Poza tym, aby wyświetlacz poprawnie został zainicjalizowany, należy przed konfiguracją jego pinów włączyć taktowanie portu C.

Najważniejszymi funkcjami dostępnymi w ramach biblioteki są:

- `LCD_Initialize` – funkcja odpowiedzialna za inicjalizację pinów połączonych z wyświetlaczem oraz przeprowadzenie poprawnej sekwencji inicjalizującej wyświetlacz,
- `LCD_WriteCommand` – funkcja służąca do wysłania do wyświetlacza komendy o podanym znaczeniu,
- `LCD_WriteText` – funkcja służąca do wysłania do wyświetlenia na wyświetlaczu całego napisu (zakończonego znakiem '\0'),
- `LCD_GoTo` – funkcja służąca do ustawienia kurSORA na zadaną pozycję.

Dokumentacja (plik `Dokumentacja/LCD_44780/HD44780.pdf`) do wyświetlacza opisuje dokładnie poszczególne komendy, które są obsługiwane przez jego sterownik (strona 26). Naśladowując procedurę inicjalizacji, można wywołać przykładowo komendę przesunięcia **kursora w prawą stronę** o jedno miejsce:

```
LCD_WriteCommand(HD44780_DISPLAY_CURSOR_SHIFT |
                  HD44780_SHIFT_CURSOR |
                  HD44780_SHIFT_RIGHT);
```

Pierwsza stała określa komendę, którą przesyła się do wyświetlacza (w tym przypadku jest to *Cursor or display shift*), a następnie „argumenty” tej komendy. W powyższym przykładzie są to: przesunięcie kursora, przesunięcie w prawą stronę.

Zadaniem studenta jest dopisanie do poprzedniego kodu możliwości przesuwania napisu "Hello,\nWorld" (zapisanego w dwóch linijkach), na podstawie stanu przycisku SW0 podłączonego do pinu PA0. Dla usprawnienia testowania można zmniejszyć zastosowane opóźnienie 10-krotnie.

Odczyt i wykorzystanie wejścia analogowego: Ostatnim ćwiczeniem jest podłączenie zewnętrznej płytki zawierającej:

- LED,
- przycisk,
- potencjometr.

Z wykorzystaniem tej płytki należy napisać program, który będzie:

- wyświetlał na wyświetlaczu wartość napięcia na potencjometrze (warto wykorzystać funkcje `sprintf` z biblioteki `stdio`),
- na podstawie odczytanej wartości analogowej będzie zmieniana jasność świecenia LED (przy użyciu sygnału PWM),
- na podstawie przycisku wyłączane/włączane jest sterowanie LED (wciśnięty – LED zgaszony, wycisnieto – LED zapalony).

Ćwiczenie należy rozpocząć od uzupełnienia plików źródłowych projektu oraz odkomentowania kolejnych plików nagłówkowych. Ponieważ w tym ćwiczeniu wykorzystane zostaną timery i przetwornik ADC, to właśnie nich będą dotyczyć wspomniane pliki. Tak jak wcześniej dodane zostały pliki o nazwach `stm32f10x_flash.c`, `stm32f10x_gpio.c` oraz `stm32f10x_rcc.c`, tak teraz należy dodać pliki o nazwach `stm32f10x_tim.c`, `stm32f10x_adc.c` (znajdujące się w tym samym katalogu). Dalej, analogicznie jak przy wspomnianych plikach, należy odkomentować linijki załączające pliki nagłówkowe

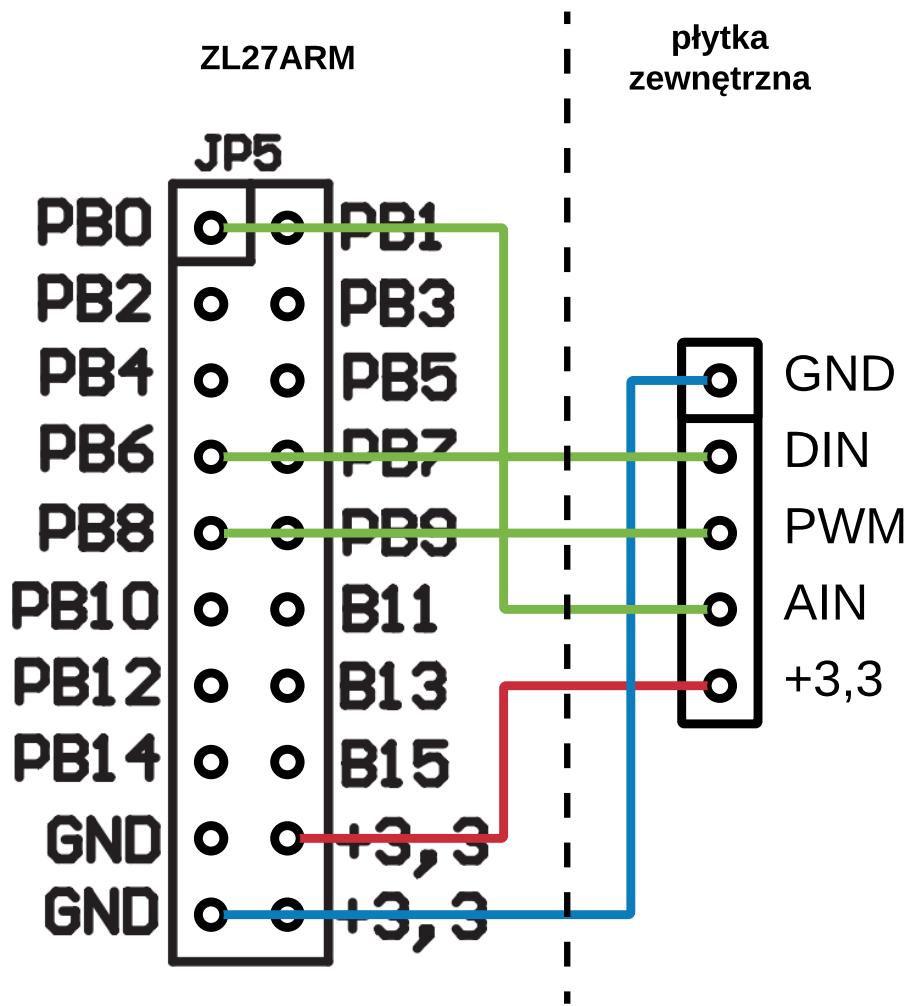
```
#include "stm32f10x_tim.h"
#include "stm32f10x_adc.h"
```

znajdujące się w pliku `stm32f10x_conf.h`. W ten sposób zostały dodane pliki do obsługi timerów i przetworników ADC, co pozwala przejść do części sprzętowej. Do płytki uruchomieniowej należy podłączyć zewnętrzną płytę zgodnie ze schematem widocznym na rysunku 19. Płytkę ta zawiera LED, potencjometr działający jako dzielnik napięcia oraz przełącznik monstabilny ze zworkami służącymi do „konfiguracji” podciągania i wartości wyjścia po wciśnięciu. Elementy te zostaną podłączone następująco:

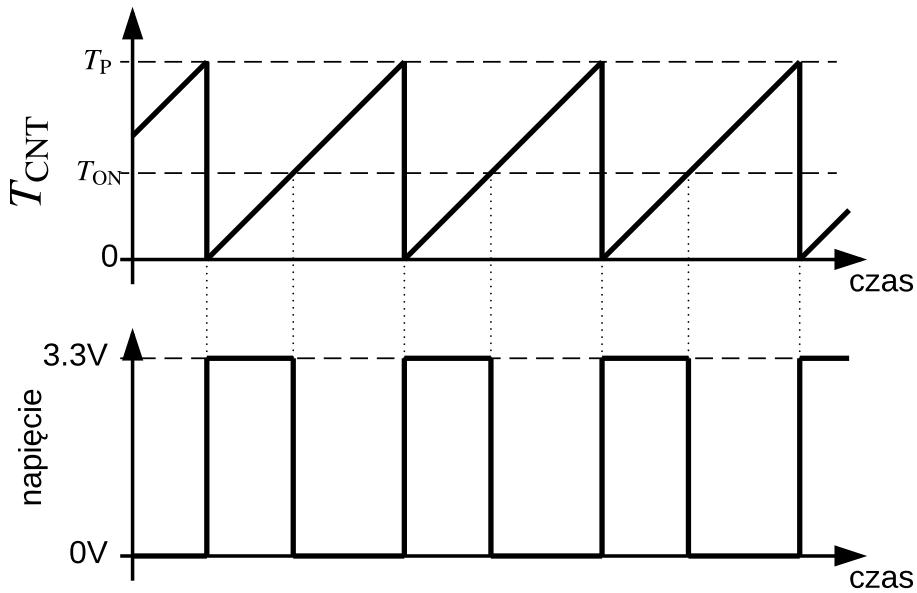
- do PB0 – potencjometr, pozwalający na odczyt napięcia z zakresu od 0 V do 3,3 V,
- do PB6 – przycisk pozwalający na odczyt napięcia 0 V lub 3,3 V (lub innej wartości jeśli przycisk nie będzie podciągnięty),
- do PB8 – LED, którego jasnością świecenia będzie można sterować poprzez zmianę wypełnienia fali PWM.

Ponieważ mikrokontrolery z rodziny STM32 są wysoce konfigurowalne, poniższy opis ograniczy się do omówienia wyłącznie potrzebnych do tego zadania opcji. Mechanizm wystawiania sygnału PWM wymaga użycia timerów, lecz, z powodu ich ogromnych możliwości, temat ten nie zostanie omówiony szczegółowo w ramach tego ćwiczenia.

Sygnał PWM (*Pulse-width modulation*) jest to sygnał cyfrowy, dzięki któremu w prosty i tani sposób można sterować jasnością świecenia diody LED lub prędkością obrotową silnika prądu stałego poprzez



Rysunek 19: Schemat podłączenia płytki zewnętrznej do płytki uruchomieniowej



Rysunek 20: Uproszczony wykres zależności między zawartością licznika T_{CNT} , a postacią wygenerowanej fali PWM

sterowanie szerokością impulsu. Realizowane jest to poprzez okresową zmianę wartości logicznej na wyjściu jednego z pinów, w taki sposób, że przez T_{ON} czasu utrzymywany jest stan wysoki, a przez T_{OFF} utrzymywany jest stan niski. $T_{\text{ON}} + T_{\text{OFF}} = T_p$, gdzie T_p to czas trwania pojedynczego okresu. Szerokość wspomnianego impulsu może być wyrażona w procentach jako stosunek trwania sygnału wysokiego do okresu, tj. $\frac{T_{\text{ON}}}{T_p} \cdot 100\%$. W mikrokontrolerach osiągane jest to przy użyciu timera, który zlicza kolejne takty zegara (źródło zegara można skonfigurować stosownie do potrzeb) i porównuje wartość licznika T_{CNT} z wartością T_{ON} . Jeśli wartość licznika jest mniejsza, to na wyjściu jest stan wysoki, jeśli jest większa, to stan niski. Przekroczenie wartości T_p powoduje automatyczne zresetowanie licznika (zakładamy zliczanie w górę). Na rys. 20 widoczne jest (w uproszczeniu) jak generowana jest fala PWM. W dalszej części poszczególne wartości będą wynosić: $T_{\text{ON}} = 1024$, $T_{\text{OFF}} = 3071$, $T_p = 4095$.

Warto zauważyć, że jeśli sygnał zegarowy, którego takty zliczane są przez timer będzie sygnałem o niskiej częstotliwości (tj. rzędu kilku Hz), to wyraźnie widoczne będą momenty w których sterowana takim sygnałem dioda LED świeci i gaśnie. Aby sterować jasnością takiej diody należy użyć sygnału o wysokiej częstotliwości. Dokładniej, okres sygnału PWM powinien być krótszy niż około 20 ms – teoretycznie przełączenia z częstotliwością 50 Hz (tj. $\frac{1}{20\text{ms}}$) nie są widzialne dla oka ludzkiego, co w rezultacie da efekt diody świecącej z intensywnością zależną (nieliniowo) od szerokości impulsu.

Konfiguracja pinu w trybie PWM została przedstawiona poniżej i zrealizowana na pinie PB8, do którego podłączony jest kanał 3 timera TIM_4 (zgodnie z tabelą 5: *Medium-density STM32F103xx pin definition*, z dokumentacji technicznej używanego mikrokontrolera – Rys. 21).

Pins				Pin name	Type ⁽¹⁾	I / O level ⁽²⁾	Main function ⁽³⁾ (after reset)	Alternate functions ⁽³⁾⁽⁴⁾	
LQFP100	LQFP64	TFBGA64	LQFP48					Default	Remap
...									
95	61	B3	45	PB8	I/O	FT	PB8	TIM4_CH3 ⁽¹¹⁾⁽¹²⁾ / TIM16_CH1 ⁽¹²⁾ / CEC ⁽¹²⁾	I2C1_SCL
								TIM4_CH4 ⁽¹¹⁾⁽¹²⁾ /	

Rysunek 21: Fragment noty katalogowej mikrokontrolera STM32F100, tabela 4. – rozpiska funkcji pinów

Ponieważ pin PB8 będzie wykorzystany w inny sposób niż poprzednio, należy usunąć jego poprzednią konfigurację, aby nie stały z nową w sprzeczności. Nie skutkowałoby to błędem, lecz zastosowaniem ostatniej konfiguracji – nie ma jednak potrzeby aby obniżać na siłę czytelności kodu. Konfiguracja wejść i wyjść cyfrowych (`GPIO_Config`) powinna teraz zawierać:

- inicjalizację pinu PB6 jako wejścia typu `GPIO_Mode_IN_FLOATING` (aby bez przeszkodek zastosować podciąganie na płytce zewnętrznej),
- inicjalizację pinów związanych z LED-ami – powinna się tu znaleźć co najmniej inicjalizacja pinu PB9 (taka jak poprzednio), lecz warto rozważyć konfigurację od PB9 do PB15 aby zgasić nieużywane LED-y na początku programu.

Konfiguracja PWM:

```

void GPIO_PWM_Config(void) {
    //konfigurowanie portow GPIO
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef timerInitStructure;
    TIM_OCInitTypeDef outputChannelInit;

    // konfiguracja pinu
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;           // pin 8
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; // szybkosc 50MHz
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;     // wyjscie w trybie alt. push-pull
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    // konfiguracja timera
    timerInitStructure.TIM_Prescaler = 0;                // prescaler = 0
    timerInitStructure.TIM_CounterMode = TIM_CounterMode_Up; // zliczanie w gore
    timerInitStructure.TIM_Period = 4095;                 // okres dlugosci 4095+1
    timerInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;   // dzielnik czestotliwosci = 1
    timerInitStructure.TIM_RepetitionCounter = 0;         // brak powtorzen
    TIM_TimeBaseInit(TIM4, &timerInitStructure);        // inicjalizacja timera TIM4
    TIM_Cmd(TIM4, ENABLE);                             // aktywacja timera TIM4

    // konfiguracja kanalu timera
    outputChannelInit.TIM_OCMode = TIM_OCMode_PWM1;    // tryb PWM1
}

```

```

outputChannelInit.TIM_Pulse = 1024; // wypełnienie 1024/4095*100% = 25%
outputChannelInit.TIM_OutputState = TIM_OutputState_Enable; // stan Enable
outputChannelInit.TIM_OCPolarity = TIM_OCPolarity_High; // polaryzacja Active High
TIM_OC3Init(TIM4, &outputChannelInit); // inicjalizacja kanalu 3 timera TIM4
TIM_OC3PreloadConfig(TIM4, TIM_OCPreload_Enable); // konfiguracja preload register
}

```

Ponieważ generacja sygnału PWM wymaga użycia timera *TIM4*, należy do funkcji konfigurującej zegary dodać linijkę odpowiedzialną włączenie taktowania dla tego timera:

```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE); // włącz taktowanie timera TIM4
```

Warto zwrócić uwagę na fakt, że timer ten jest podłączony do szyny *APB1* w przeciwnieństwie do portów GPIO, które są podłączone do szyny *APB2*. Aby sygnał PWM można było „przekierować” do wyjścia PB8 należy jeszcze uruchomić moduł zarządzający funkcjami alternatywnymi:

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE); // włącz taktowanie AFIO
```

Tak przeprowadzona konfiguracja pozwala na regulację jasności świecenia diody LED podłączonej pod pin PB8. Zmiana szerokości impulsu fali PWM w trakcie działania programu odbywa się poprzez zapisanie nowej wartości T_{ON} do odpowiedniego rejestru mikrokontrolera:

```
unsigned int val = 1024; // liczba 16-bitowa
TIM4->CCR3 = val;
```

TIM4 jest strukturą, która zawiera wskaźniki na poszczególne adresy w pamięci mikrokontrolera związane z timerem *TIM4*. W szczególności znajduje się tam pole o nazwie *CCR3* (*Compare/Capture 3 value*), któremu odpowiada wartość T_{ON} . Taki sposób modyfikacji zawartości rejestrów mikrokontrolera jest często szybszy w stosunku do użycia odpowiednich funkcji standardowej biblioteki do obsługi peryferiali, lecz jest zazwyczaj bardziej skomplikowany i trudniejszy w czytaniu – na szczęście w tym przypadku jest to pojedynczy zapis. Jak widać wykorzystanie standardowej biblioteki peryferiali równolegle z pisaniem do rejestrów mikrokontrolera jest możliwe i nierzadko stosowane. Dla tych, którzy wolą konsekwentnie trzymać się jednego rozwiązania: w standardowej bibliotece peryferiali znajduje się funkcja która robi dokładnie to co powyżej (z dodatkową opcjonalną weryfikacją argumentu tej funkcji):

```
TIM_SetCompare3(TIM4, val); // TIM4->CCR3 = val;
```

Mikrokontrolery bardzo często wyposażone są w przetworniki analogowo-cyfrowe. Dzięki nim napięcie przyłożone do pinu wejściowego może zostać odczytane jako wartość cyfrowa. W przypadku mikrokontrolera zawartego na płytce uruchomieniowej ZL27ARM do dyspozycji są 2 12-bitowe przetworniki analogowo-cyfrowe (do 16 kanałów każdy). Przetworniki te mierzą napięcia w zakresie od 0 V do 3,3 V. Oznacza to jednocześnie, że sygnał o maksymalnej wartości napięcia (3,3 V) zostanie zinterpretowany jako wartość 0xFFFF, natomiast wartość minimalna (0 V) jako 0x000. Zapis składający się z trzech znaków wynika z faktu, iż przetwornik jest 12-bitowy. Ponieważ jednak rejesty są 16-bitowe, należy podjąć decyzję, do której strony wyrównana zostanie odczytana wartość. Najbardziej intuicyjnie będzie wyrównać do prawej strony, tak aby nieużywane 4 bity (będące zerami) były jednocześnie najbardziej znaczącymi bitami. Dodatkowymi założeniami przyjętymi w poniższym kodzie konfiguruującym przetwornik analogowo-cyfrowy są:

- niezależne działanie przetworników ADC1 oraz ADC2,

- pomiar wyłącznie jednego kanału (nr 8) przetwornika ADC1,
- start pomiaru rozpoczyna się na programowe żądanie użytkownika,
- pomiar trwać będzie możliwie krótko (tutaj 1,5 cyklu + stały czas przetwarzania 12,5 cyklu – szcze-góły w RM0008, rozdział 11.6)

Po włączeniu taktowania modułu ADC (w tym przypadku dokładniej ADC1):

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE); // włącz taktowanie ADC1
```

można przejść do implementacji opisanej konfiguracji:

```
void ADC_Config(void) {
    ADC_InitTypeDef    ADC_InitStructure;
    GPIO_InitTypeDef   GPIO_InitStructure;

    ADC_DeInit(ADC1);                                // reset ustawien ADC1

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;           // pin 0
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;  // szybkosc 50MHz
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING; // wyjscie w floating
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent; // niezalezne dzialanie ADC 1 i 2
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;       // pomiar pojedynczego kanalu
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE; // pomiar na zadanie
    ADC_InitStructure.ADC_ExternalTrigConv=ADC_ExternalTrigConv_None; // programowy start
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; // pomiar wyrownany do prawej
    ADC_InitStructure.ADC_NbrOfChannel = 1;              // jeden kanal
    ADC_Init(ADC1, &ADC_InitStructure);                // inicjalizacja ADC1
    ADC-RegularChannelConfig(ADC1, 8, 1, ADC_SampleTime_1Cycles5); // ADC1, kanal 8,
                                                                // 1.5 cyklu
    ADC_Cmd(ADC1, ENABLE);                            // aktywacja ADC1

    ADC_ResetCalibration(ADC1);                      // reset rejestrów kalibracji ADC1
    while(ADC_GetResetCalibrationStatus(ADC1));      // oczekiwanie na koniec resetu
    ADC_StartCalibration(ADC1);                      // start kalibracji ADC1
    while(ADC_GetCalibrationStatus(ADC1));            // czekaj na koniec kalibracji
}
```

Jak widać pin służący do pomiaru analogowej wartości napięcia został skonfigurowany jako niepodciagnięty pin wejściowy (GPIO_Mode_IN_FLOATING). Podciagnięcie takiego pinu w którymkolwiek kierunku skutkowałoby błędnymi odczytami. Warto zadać sobie jednocześnie pytanie „gdzie jest zapisana informacja, że właśnie pin PB0 będzie podłączony do kanału 8 przetwornika analogowo-cyfrowego ADC1?”. Odpowiedź na to pytanie wymaga przestudiowania noty katalogowej mikrokontrolera STM32F100, a dokładniej tabeli 4, gdzie można znaleźć wpis widoczny na Rys. 22. Należy zauważyć, że mimo, że podłączenie do ADC jest funkcją alternatywną, sam pin jest skonfigurowany jako pin wejściowy – nie jest to reguła koniecznie stosowana w innych mikrokontrolerach, nawet tych z rodziny STM32.

Na końcu przedstawionego kodu widoczna jest procedura kalibracji przetwornika. Należy (choć nie jest to konieczne) ją przeprowadzić w celu osiągnięcia dokładniejszych pomiarów. Przed uruchomieniem

Pins				Pin name	Type ⁽¹⁾	I / O level ⁽²⁾	Main function ⁽³⁾ (after reset)	Alternate functions ⁽³⁾⁽⁴⁾	
LQFP100	LQFP64	TFBGA64	LQFP48					Default	Remap
33	24	H5	-	PC4	I/O	-	PC4	ADC1_IN14	-
34	25	H6	-	PC5	I/O	-	PC5	ADC1_IN15	-
35	26	F5	18	PB0	I/O	-	PB0	ADC1_IN8/TIM3_CH3 ⁽¹²⁾	TIM1_CH2N
22	27	G5	10	PD1	I/O	-	PD1	ADC1_IN10/TIM3_CH4 ⁽¹²⁾	TIM1_CH1N

Rysunek 22: Fragment noty katalogowej mikrokontrolera STM32F100, tabela 4. – rozpiska funkcji pinów

przetwornika należy pamiętać o włączeniu jego zegara, poprzedzając to odpowiednią konfiguracją prescalera ADC. Zgodnie z dokumentacją (RM0008, rozdział 11.1) częstotliwość tego zegara nie może przekraczać 14 MHz. Stąd wynika, że z dostępnych wartości prescalera (/2, /4, /6, /8), należy wybrać co najmniej /6 (72 MHz / 6 = 12 MHz) – tak też konfigurujemy ten zegar. W tym celu dodajemy do funkcji RCC_Config następującą linijkę:

```
RCC_ADCCLKConfig(RCC_PCLK2_Div6); // ADCCLK = PCLK2/6 = 12 MHz
```

Od tego momentu przetwornik analogowo-cyfrowy będzie oczekiwany na sygnał do rozpoczęcia pomiaru, po którym będzie można odczytać przygotowaną przez niego wartość. Wykonuje się to w trzech krokach:

```
unsigned int readADC(void){
    ADC_SoftwareStartConvCmd(ADC1, ENABLE);           // start pomiaru
    while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET); // czekaj na koniec pomiaru
    return ADC_GetConversionValue(ADC1);               // odczyt pomiaru (12 bit)
}
```

Jako pierwszy należy wysłać rozkaz rozpoczęcia pomiaru, następnie należy odczekać na ustawienie flagi EOC (*End Of Conversion*), a na koniec można odczytać gotową 12-bitową wartość pomiaru z przetwornika ADC1. W powyższej implementacji odczytana wartość zwracana jest jako `unsigned int`, choć należy pamiętać, że zawierać się ona będzie w przedziale od 0 do 4095.

Wnioski: Powyższy projekt pozwolił na zaprogramowanie mikrokontrolera w skuteczny, lecz mało efektywny sposób. Pojawiło się wiele problemów z obsługą oprogramowanego już mikrokontrolera, między innymi:

- przyciski nie reagowały natychmiastowo na zmiany stanu,
- wszelkie opóźnienia wstrzymywały działanie całego programu,
- wykorzystanie przetwornika analogowo-cyfrowego powodowało wstrzymanie programu na czas oczekiwania na wyniki konwersji – w tym przypadku wykorzystany został jeden kanał przetwornika dzięki czemu było to niemal niezauważalne, lecz ich zwiększenie powodowały wyraźne problemy.

Podsumowując, wszystkie instrukcje wykonywane były jedna po drugiej, co powodowało utrudnioną interakcję z mikrokontrolerem. Oczekiwany rezultatem byłoby aby wciśnięcie przycisku powodowało natychmiastową reakcję mikrokontrolera (co jak się okaże również nie jest tak skuteczne jak mogłoby się zdawać), natomiast część akcji powinna być wykonywana „w tle” (np. wykorzystanie przetwornika analogowo-cyfrowego). To właśnie zostanie zaimplementowane poprzez wykorzystanie liczników i timerów.

1.4 Wejściówka:

W ramach wejściówki student ma za zadanie przesłać prowadzącemu zawartość zmiennej `bufor` na koniec 10, 15 i 20 wywołania poniższej funkcji (numer wywołania jest równy wartości `i`):

```
void foo(){
    static char text[175] = "<text>";
    static unsigned int i = 1;
    static char* buforp[4] = {0};
    char bufor[4] = "";
    if(i == 1){
        buforp[0] = &text[13];
        buforp[1] = &text[01];
        buforp[2] = &text[19];
        buforp[3] = &text[90];
    } else {
        buforp[0] += 1;
        buforp[1] += 2;
        buforp[2] += 3;
        buforp[3] += 4;
    }
    bufor[0] = *buforp[0];
    bufor[1] = *buforp[1];
    bufor[2] = *buforp[2];
    bufor[3] = *buforp[3];
    ++i; // tutaj warto postawić pulapkę
}
```

Rozwiązaniem zadania jest **spakowany projekt stworzony w programie Keil µVision 5**, który doprowadził do uzyskania żądanego treści oraz sama treść (zapisana jako jawni tekst w mailu), tj. trzy cztero-elementowe ciągi znaków. Poza zawartością zmiennej `text`, definicji funkcji `foo` nie wolno modyfikować. Każdy student otrzymuje własny zestaw znaków zawartych w zmiennej `text`, pozostałe wartości pozostają takie same dla wszystkich. Przykładowo dla zmiennej `text` wypełnionej następującymi znakami:

```
static char text[175] = "XFZDYKTLUHUQGBJDVCVBEJOEOUYUSLOFRCUVETHLBTEOUYSKRYQEXCELRCGMBJDMOTRSTQRDUQEWFUVFEHLLWDDTBPFDGEFSXCSMVHMEWMQMZSJZYKOVEVQMQUEOIJLDKKHMUFLVZTEUOJBFGSCGVUYWWIEBDWAXGMKEKRNORGGACWDO";
```

oczekiwanaą zawartością zmiennej `bufor` są następujące ciągi znaków: ODSL, ULJG, RLWØ kolejno dla 10, 15 i 20 wywołania funkcji `foo`.

2 Obsługa prostych czujników i urządzeń wykonawczych mikroprocessorowego systemu automatyki przy wykorzystaniu systemu przerwań

2.1 Cel

Celem ćwiczenia jest zapoznanie studentów z metodami obsługi najprostszych czujników i urządzeń wykonawczych z poziomu mikrokontrolera wykorzystując do tego mechanizm przerwań. Sterowanie elementami wykonawczymi odbywać się będzie przy użyciu fali PWM oraz „włączania” i „wyłączania” elementów, natomiast pomiary będą dokonywane poprzez pomiar napięcia na pinach mikrokontrolera. Uwaga skupiona będzie jednak na wykorzystaniu mechanizmu przerwań do „jednoczesnego”, regularnego wykonywania zadań oraz na zastąpieniu mechanizmu aktywnego oczekiwania (odpytywania) na rzecz obsługi przerwań.

2.2 Wykorzystane mechanizmy

Przerwania: Mechanizm przerwań, jak sama nazwa wskazuje, pozwala na natychmiastowe przerwanie pracy mikrokontrolera w celu obsługi zdarzenia, które wymaga niezwłocznej reakcji. Wstrzymane może być zarówno wykonywanie głównej pętli programu (tj. funkcja `main`) jak też i samych funkcji obsługujących przerwania. O tym, które z przerwań spowoduje wstrzymanie wykonania kodu na rzecz swojej funkcji obsługi przerwania, które trafi do kolejki obsługiwanych, a które nie zostanie obsłużone wcale decyduje kontroler przerwań NVIC (*Nested Vectored Interrupt Controller*). W dalszej części pojawiać się będzie pojęcie funkcji lub programu obsługi przerwań (*Interrupt Service Routine*), tj. funkcji, która wywoływana jest w wyniku zgłoszenia przerwania, jest to odpowiedź mikrokontrolera na zdarzenie zewnętrzne. Warto mieć na uwadze, że przeciwieństwem przerwań jest odpytywanie. Przykładem odpytywania jest poniższy kod:

```
while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET); // czekaj na koniec pomiaru
```

gdzie mikrokontroler nieustannie odpytuje przetwornik, czy zakończył pomiar. Czas, który poświęcany (wręcz marnowany) jest na oczekiwanie na odpowiedź, można by wykorzystać znacznie lepiej, np. wykonując dalsze operacje, które nie wymagają informacji otrzymanych z przetwornika. Efekt ten osiągniemy właśnie dzięki przerwaniom.

Tabela 63 z pliku RM0008 zawiera rozpiszaną tablicę wektorów przerwań – podane są: pozycja w tablicy, priorytet, możliwość zmiany priorytetu, akronim, opis oraz adres programu obsługi tego przerwania. Kilka z nich jest wyjątkowo interesujących z punktu widzenia późniejszej wykonywanego ćwiczenia: SysTick, EXTI0, ADC1_2, EXTI9_5 oraz TIM2, TIM3, TIM4. Z tabeli tej można odczytać pod jakimi adresami w pamięci znajdują się poszczególne programy obsługi przerwań, a więc pod jakimi adresami powinny znaleźć się funkcje, które mają zostać wywołane w momencie nastąpienia odpowiedniego zdarzenia zewnętrznego. Implementacja funkcji służącej do obsługi przerwania będzie realizowana poprzez implementację funkcji o odpowiedniej deklaracji (nazwa wraz z argumentami i typem zwracanym), ponieważ w trakcie inicjalizacji pamięci mikrokontrolera przypisane zostały im już odpowiednie adresy w pamięci.

Bezpośrednio przed rozpoczęciem obsługi przerwania, procesor chroni środowisko przerwanego programu, wysyłając zawartości odpowiednich rejestrów na stos – dzięki temu po zakończeniu wykonania kodu odpowiedzialnego za obsługę przerwania może powrócić do przerwanych zadań. Stąd wynika, że o ile nie zostaną zmodyfikowane zawartości wspomnianych rejestrów, mikrokontroler będzie kontynuował pracę

wcześniej przerwanego programu od miejsca, w którym nastąpiło przerwanie. Świadomość tego jest niezmiernie ważna szczególnie w sytuacji, gdy zarówno w funkcji obsługującej przerwanie jak i w pętli głównej operujemy na tych samych zmiennych w pamięci. Dla przykładu, jeśli w trakcie wykonywania następującej pętli głównej:

```
unsigned char x = 10;
while(1){
    if (x > 0) {
        // (1)
        --x;
    } else {
        break;
}
}
```

nie nastąpi przerwanie, to po odpowiednio dużej liczbie iteracji, zmienna `x` będzie równa 0. Jeśli natomiast nastąpi przerwanie, które np. wyzeruje zmienną `x`, wtedy (w zależności od momentu w którym przerwanie nastąpi) pętla główna wykona mniejszą lub większą liczbę iteracji niż gdyby przerwanie się nie pojawiło. Jeśli przerwanie nastąpiło przed sprawdzeniem warunku, lub po dekrementacji zmiennej `x` – liczba iteracji zmniejszy się lub pozostanie taka sama, a więc wykonane zostaną następujące operacje:

- sprawdzenie warunku `if (x > 0){ ...}`, założmy, że `x = 5`, a więc warunek spełniony,
- dekrementacja zmiennej `--x;`, po którym zmienna `x` ma wartość `5-1`, a więc `x = 4`,

!! tutaj następuje przerwanie, a więc wejście do funkcji obsługi przerwania:

1. wyzerowanie zmiennej `x`,
 2. zakończenie funkcji obsługi przerwania,
- sprawdzenie warunku `if (x > 0){ ...}`, który nie jest spełniony, bo `x = 0` – wykonanie pętli jest przerwywane (`break;`).

W przeciwnym razie wykonane zostaną następujące operacje:

- sprawdzenie warunku `if (x > 0){ ...}`, założmy, że `x = 5`, a więc warunek spełniony,

!! tutaj następuje przerwanie, a więc wejście do funkcji obsługi przerwania:

1. wyzerowanie zmiennej `x`,
 2. zakończenie funkcji obsługi przerwania,
- dekrementacja zmiennej `--x;`, po którym zmienna `x` ma wartość `0-1`, a więc ze względu na użyty typ `unsigned char`, będzie to wartość 255,
 - sprawdzenie warunku `if (x > 0){ ...}`, który jest spełniony, bo `x = 255` – pętla jest kontynuowana.

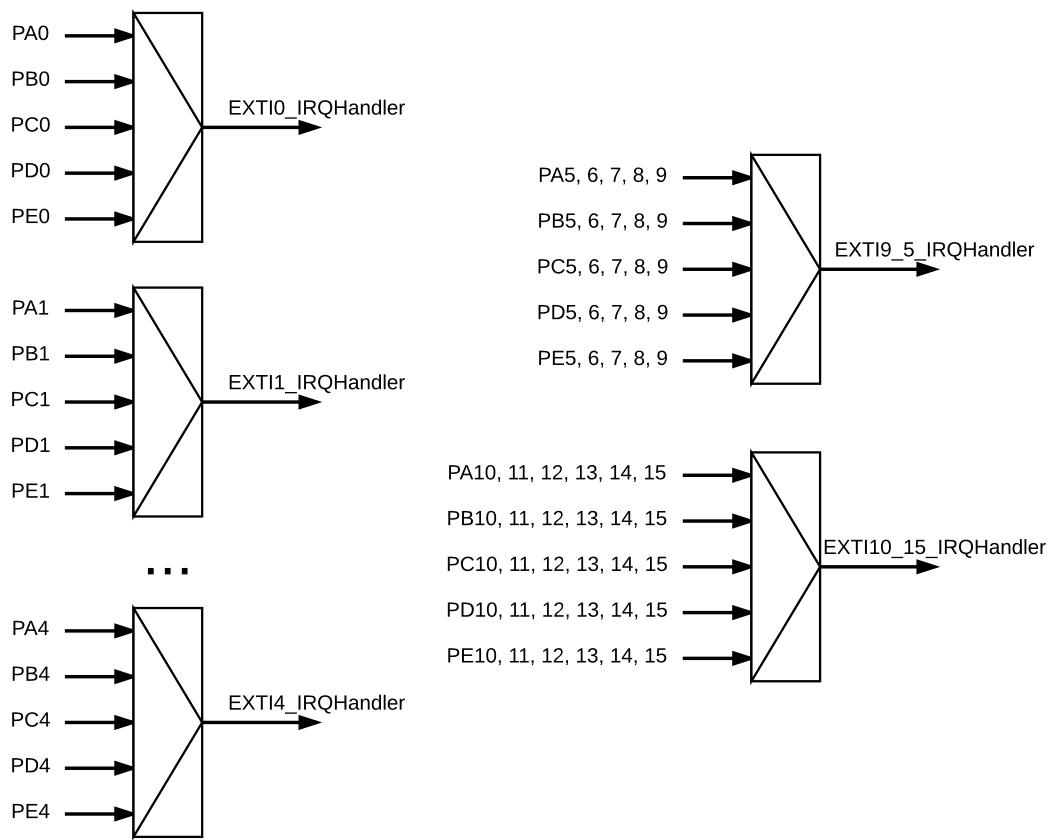
Jak widać działanie takiego programu jest zależne od momentu, w którym nastąpiło przerwanie. Aby zapewnić sobie, że zmienna w trakcie wykonania pewnego bloku programu nie zostanie zmodyfikowana przez wystąpienie przerwania, można posłużyć się mechanizmem monitorów, semaforów lub nawet na ten czas wyłączyć przerwania, które mogą nam przeszkodzić.

Poza tym, że przerwania powodują zatrzymanie wykonania głównej pętli programu, mogą także powodować zatrzymanie wykonania funkcji obsługującej inne przerwanie. Z tego powodu zostały wprowadzone priorytety przerwań – im niższy priorytet przerwania tym jest ono ważniejsze. Warto tutaj zwrócić uwagę na wspomnianą tabelę 63 z pliku RM0008, gdzie widać, że najważniejszym przerwaniem jest przerwanie związane z resetowaniem mikrokontrolera, a niedaleko za nim znajduje się przerwanie związane z błędami. Co więcej – priorytetu tych przerwań nie można zmienić – zawsze będą ważniejsze od przerwań o dodatniej wartości priorytetu.

Przerwania mogą być obsługiwane natychmiast lub zaraz po obsłudze ważniejszych przerwań. W mikrokontrolerach rodziny STM32 decyzja zapada na podstawie numeru przerwania, a dokładniej poszczególnych jego bitów. Numer przerwania tworzą w rzeczywistości dwa pola, których długość można zmieniać: priorytet wywłaszczenia oraz pod-priorytet w obrębie priorytetu wywłaszczenia. Oba pola w sumie zapisane są na 4 bitach tworząc razem priorytet przerwania. Możliwości podziału priorytetu na wspomniane dwa pola jest 5: 0+4, 1+3, 2+2, 3+1, 4+0. Ostatni podział pozwala na wyłączenie całkowicie pod-priorytetów, co prowadzi do wywłaszczenia wyłącznie na podstawie wartości priorytetu (jeśli wyższy priorytet, to następuje wywłaszczenie), który z resztą zajmowany jest w całości przez pole priorytet wywłaszczenia. Pierwszy podział powoduje, że wszystkie przerwania mają jednakowy priorytet wywłaszczenia, a co za tym idzie, w przypadku występowania wielu przerwań jednocześnie, wykonywane są kolejno, od tego z najniższą wartością pod-priorytetu, do tego z największą jego wartością. Tryb mieszany, czyli np. 2+2, pozwala na wyznaczenie przerwań, które muszą zostać obsłużone natychmiastowo (o mniejszym niż inne priorytecie wywłaszczenia), oraz takie, które mogą zostać obsłużone po innych przerwaniach o tym samym priorytecie (ustawiając odpowiednio pod-priorytet).

Odpowiednie i rozważne ustawianie wartości priorytetów jest kluczowe w każdym projekcie, który wykorzystuje mechanizm przerwań. W przeciwnym razie mogą nastąpić bardzo nieprzyjemne sytuacje takie jak zagłodzenie lub zakleszczenie. Jednym z przypadków, gdzie następuje zakleszczenie jest zle ustawienie priorytetu przerwania SysTick, który ma za zadanie odmierzanie czasu opóźnienia. Jeśli wystąpi przerwanie, które ma wyższy priorytet, a jednocześnie wywołuje funkcję opóźnienia, następuje natychmiastowe wstrzymanie działania programu. Wynika to z faktu, że ponieważ opóźnienie bazuje na przerwaniu SysTick, które ma niższy priorytet niż obecnie obsługiwane, to nie ma możliwości, aby wywłączyło ono przerwanie o wyższym priorytecie. Z drugiej strony obsługiwane przerwanie oczekuje na zakończenie funkcji opóźnienia, co powoduje, że program zatrzymuje wykonywanie oczekując w nieskończoność w pętli.

Przerwania zewnętrzne: Układ EXTI (*External interrupt / event controller*) obsługuje zewnętrzne źródła przerwań – może on zgłosić przerwanie lub zdarzenie. Zdarzenia w przeciwnieństwie do przerwań nie muszą być obsługiwane poprzez wywołanie funkcji obsługi – mogą one bezpośrednio wywoływać pewną reakcję, np. wybudzić procesor, rozpoczęć przetwarzanie sygnału analogowego na cyfrowy, itp. Co więcej niektóre układy mogą generować wiele zdarzeń w ramach jednego przerwania. Przykładem takiego układu jest przetwornik analogowo-cyfrowy. Może on generować jedno przerwanie, którego funkcja obsługi musi zawierać kod sprawdzający jakie zdarzenie je wywołało – tych jest kilka: *End of conversion*, *End of injection*.



Rysunek 23: Schemat przypisywania linii do zgłoszanego przerwania

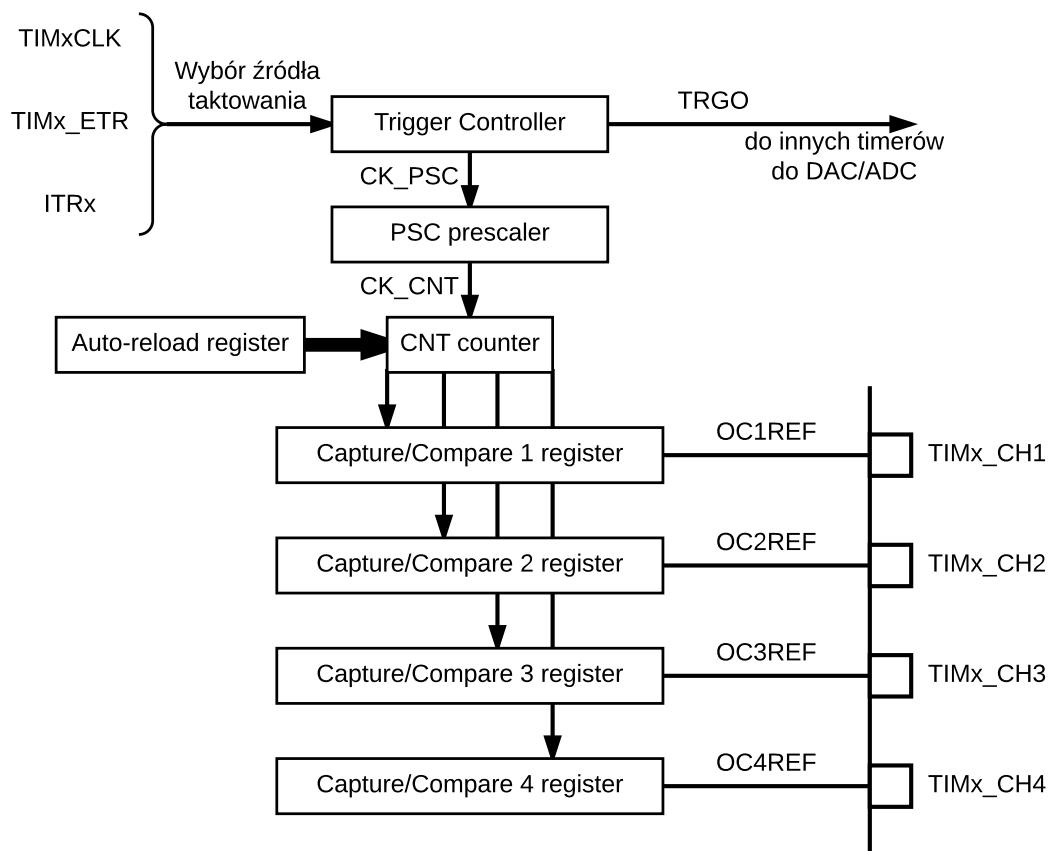
tion, Analog watchdog event. Obsługa zewnętrznych źródeł przerwań obejmuje między innymi wykrywanie zboczy sygnału wejściowego. Mogą być wykrywane zarówno zbocza narastające, opadające jak i jednocześnie oba wymienione. Aby wybrane zbocze generowało przerwanie, należy przypisać odpowiednią linię do zgłoszanego przerwania – przedstawia to rys. 23. Każda z linii od 0 do 4 zgłasza osobne przerwanie, są one jednak wspólne dla wszystkich portów, np. PA0, PB0, PC0, PD0, PE0, PF0 zgłaszają jedno przerwanie EXTI0_IRQ. Linie od 5 do 9 zgłaszą wspólne przerwanie EXTI9_5_IRQ (wspólne dla wszystkich portów). Analogicznie linie od 10 do 15 zgłaszą przerwanie EXTI10_15_IRQ.

Timery: Timery, są to liczniki, których sygnałem wejściowym jest sygnał zegarowy. W mikrokontrolerach rodziny STM32 nie ma mowy o licznikach, lecz używa się właśnie pojęcia timer. W szczególnym przypadku timery służą do zliczania zboczy sygnału, który nie jest zegarowym (a więc działają jak zwykłe liczniki), lecz ponieważ ich głównym zastosowaniem jest odmierzanie czasu – także i w tym opracowaniu będzie używane pojęcie timer.

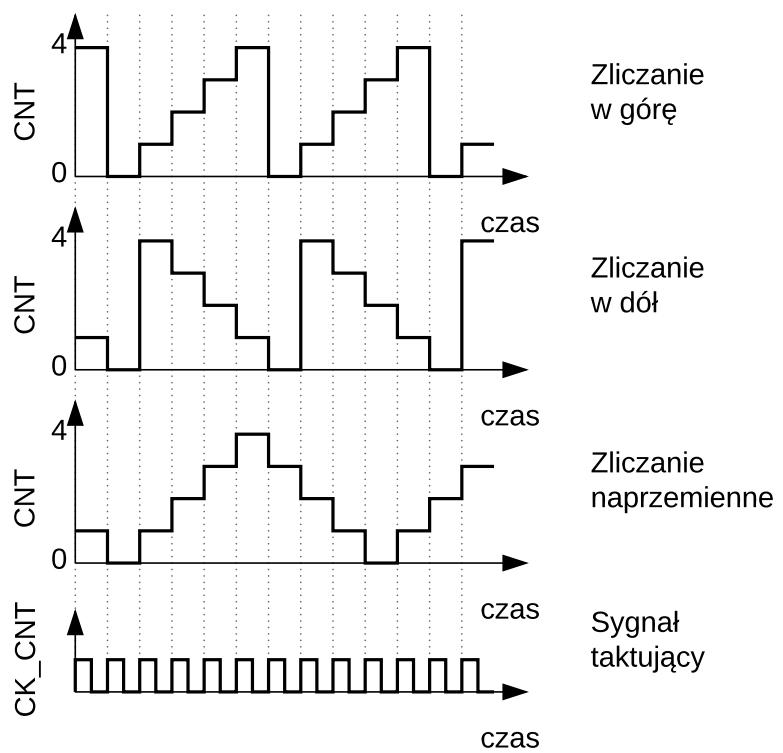
W mikrokontrolerach rodziny STM32 znaleźć można wiele rodzajów timerów. Począwszy od bardzo prostych (mających wyłącznie dwa sygnały zegarowe do wyboru), po niezmiernie skomplikowane (zazwyczaj TIM1 i TIM8, które oznaczane są jako *Advanced-control timers* i poświęcony jest im najczęściej osobny rozdział w dokumentacji).

W tym ćwiczeniu w centrum uwagi będą dwa typy timerów: *Cortex System Timer* (nazywany często *SysTick*) oraz *General-purpose timer*. Pierwszy z nich należy do najprostszych w obsłudze (co wynika z niewielkich jego możliwości) timerów – zlicza on w dół takty sygnału zegarowego HCLK lub sygnału HCLK/8 (tj. sygnał zegarowy HCLK spowolniony ośmiokrotnie). Timer ten zlicza od zadanej 24-bitowej wartości do zera, po czym przeładowuje swój licznik ponownieadaną wcześniej wartością. Osiągnięcie zera przez ten timer powoduje wygenerowanie przerwania. Warto zwrócić uwagę, że timer ten doskonale się nadaje do wykorzystania w systemach czasu rzeczywistego, gdzie konieczny jest przydział kwantów czasu poszczególnym zadaniom – takie kwenty czasu mogą być wyznaczane właśnie przez ten bardzo prosty, lecz jakże przydatny timer.

Timery z kategorii *General-purpose timer* (TIM2, TIM3, TIM4) mają znacznie więcej możliwości niż wspomniany *Cortex System Timer*. Schemat działania timerów ogólnego przeznaczenia w trybie odmierzania czasu (na tym trybie skupiać się będzie dalszy opis) widoczny jest na Rys. 24. Na schemacie widać, że źródłem taktowania timera może być sygnał TIMxCLK (patrz Rys. 18), zewnętrzny sygnał TIMx_ETR lub wyjście innego licznika ITRx. Wybrany sygnał trafia na wejście prescaler (*PSC prescaler*), gdzie może zostać podzielony przez dowolną 16-bitową wartość. Poszczególne takty takiego sygnału dopiero zliczane są przez licznik (*CNT counter*). Timer może pracować w kilku trybach: zliczania w góre, zliczania w dół, zliczania naprzemiennie w góre i w dół. Rejestr *Auto-reload* zawiera 16-bitową wartość, od której odlicza lub do której zlicza licznik (zależnie od trybu zliczania). Tryby zliczania pokazane zostały na Rys. 25. Przedstawiona została na nim zawartość licznika CNT w zależności od wybranego trybu zliczania, przy założeniu, że rejestr *Auto-reload* ma wartość 4. W trybie zliczania w góre licznik po osiągnięciu wartości 4, zeruje zawartość licznika i kontynuuje zliczanie. W trybie zliczania w dół zawartość licznika jest inicjalizowana wartością 4 za każdym razem jak licznik osiągnie 0. W przypadku zliczania naprzemiennego, gdy zawartość licznika osiągnie 0 (przy zliczaniu w dół), licznik zaczyna zliczać w góre. W przypadku gdy licznik osiągnie wartość 4 (przy zliczaniu w góre), następuje rozpoczęcie zliczania w dół. Ważną cechą tych timerów jest, że gdy do licznika wpisywane jest zero (w przypadku zliczania w góre) lub zawartość rejestru



Rysunek 24: Schemat działania timera ogólnego przeznaczenia w trybie odmierzania czasu



Rysunek 25: Zawartość licznika CNT w zależności od wybranego trybu zliczania

Auto-reload (w przypadku zliczania w dół) generowane jest zdarzenie *Update Event*. W przypadku zliczania naprzemiennego nie jest dokonywane wpisywanie wartości do licznika – można jednak skonfigurować timer tak, aby generował zdarzenie *Update Event* za każdym razem gdy nastąpi zdarzenie przepełnienia *Overflow* lub niedomiaru *Underflow*.

Każdy z timerów ogólnego przeznaczenia wyposażony jest w 4 kanały. Każdy z kanałów może służyć do przechwytywania zawartości licznika lub do porównywania z zawartością licznika. W tym ćwiczeniu uwaga została skupiona na tej drugiej funkcji, dzięki temu będzie można okresowo wykonywać pewne operacje oraz generować falę PWM (tak jak zostało to wykonane w poprzednim ćwiczeniu).

Zawartość licznika jest porównywana w każdym taktie ze wszystkimi rejestrami *Capture/Compare* (CCR) – w zależności od trybu mogą zostać wykonane różne operacje na wyjściu kanału OCxREF. Dostępne tryby to:

- *Timing* – wyjście OCxREF nie zmienia wartości,
- *Active* – OCxREF ustawiane w stan wysoki gdy zawartości rejestrów CNT i CCR są równe
- *Inactive* – OCxREF ustawiane w stan niski gdy zawartości rejestrów CNT i CCR są równe
- *Toggle* – gdy CNT i CCR są równe, OCxREF jest ustawiane na stan przeciwny
- *PWM1* – *Pulse Width Modulation*(tryb 1)
- *PWM2* – *Pulse Width Modulation*(tryb 2)

Prostą operację, jaką jest przełączenie bitu rejestru wyjściowego (TIMx_CHy), można wykonać więc na wiele sposobów: odpowiednią implementację obsługi przerwania (tryb *Timing*), wykorzystanie trybu *Toggle* lub jednego z trybów PWM. W poniższym ćwiczeniu będzie wykorzystany głównie tryb *Timing* – pozwoli to na późniejsze wykorzystanie wiedzy o implementacji obsługi przerwań pod kątem bardziej zaawansowanych operacji niż proste przełączanie stanu diody. Same diody należy tutaj traktować bardziej jako prymitywne narzędzie do diagnostyki (na zasadzie „jeśli migą to zazwyczaj znaczy, że działa”).

Tryb PWM został wyjaśniony w poprzednim ćwiczeniu, jednak nie została omówiona różnica między trybem PWM1 a PWM2. Jest ona jednak wyjątkowo prosta – sygnał powstały w trybie PWM2 jest negacją sygnału powstałego w trybie PWM1.

2.3 Przebieg laboratorium

Ćwiczenie obejmuje przełączanie stanu diod z odpowiednimi wymaganiami na momenty włączenia i wyłączenia. Dodatkowo, dla uproszczenia (tj. ograniczenia zgłębiania dokumentacji mikrokontrolera), odpowiednie diody mają wyznaczone timery, którymi będą sterowane. Wynikiem pracy w trakcie ćwiczenia powinno być:

- Przełączanie diody:
 - LED1 z częstotliwością 0,5 Hz z wypełnieniem 50 % (tj. przez 1 s świeci, przez 1 s nie świeci) z wykorzystaniem funkcji *Delay* i timera SysTick,

- LED2 z częstotliwością 1 Hz z wypełnieniem 20 % (tj. przez 0,2 s świeci, przez 0,8 s nie świeci) z wykorzystaniem timera TIM4,
 - LED3 z częstotliwością 1 Hz z wypełnieniem 70 % (tj. przez 0,7 s świeci, przez 0,3 s nie świeci) z wykorzystaniem timera TIM4,
 - LED4 w wyniku wykrycia zbocza opadającego na przycisku SW0 (PA0),
 - LED5 w wyniku wykrycia zbocza opadającego na przycisku SW0 (PA0) z zaimplementowanym algorytmem niwelacji styków,
 - LED6 wraz z zakończeniem przez przetwornik ADC konwersji sygnału wejściowego.
- Wyzwalanie przetwarzania sygnału wejściowego przez ADC1 (kanał 16 – wewnętrzny termometr mikrokontrolera) z okresem 1 s,
 - Okresowe odświeżanie wyświetlacza LCD (tj. zmiana wyświetlonej treści na aktualną) – okres dobrany dowolnie (np. co 5 s),
 - Obsługa klawiatury numerycznej – co zostanie wciśnięte na klawiaturze powinno zostać wypisane na wyświetlaczu (wystarczy jeden znak do wyświetlania ostatniego wciśniętego klawisz).

Dla skrócenia listingów wprowadzona została funkcja do obsługi LED-ów. Plik `main.h` uzupełniony zostanie następującym kodem:

```
#include "stm32f10x.h" // definicja typu uint16_t i stałych GPIO_Pin_X

#define LED1 GPIO_Pin_8
#define LED2 GPIO_Pin_9
#define LED3 GPIO_Pin_10
#define LED4 GPIO_Pin_11
#define LED5 GPIO_Pin_12
#define LED6 GPIO_Pin_13
#define LED7 GPIO_Pin_14
#define LED8 GPIO_Pin_15
#define LEDALL (LED1|LED2|LED3|LED4|LED5|LED6|LED7|LED8)
enum LED_ACTION { LED_ON, LED_OFF, LED_TOGGLE };

void LED(uint16_t led, enum LED_ACTION act);
```

natomiast do pliku `main.c` dodana zostanie definicja funkcji obsługi LED (należy oczywiście pamiętać o uzupełnieniu nagłówka):

```
void LED(uint16_t led, enum LED_ACTION act) {
    switch(act){
        case LED_ON: GPIO_SetBits(GPIOB, led); break;
        case LED_OFF: GPIO_ResetBits(GPIOB, led); break;
        case LED_TOGGLE: GPIO_WriteBit(GPIOB, led,
            (GPIO_ReadOutputDataBit(GPIOB, led) == Bit_SET?Bit_RESET:Bit_SET));
    }
}
```

Opóźnienie: Realizacja opóźnienia przy użyciu timera SysTick jest wyjątkowo użyteczna a jednocześnie niewymagająca dużego nakładu implementacyjnego. Koncepcja tego typu rozwiązania jest następująca: timer nieustannie odmierza pewien kwant czasu (dla ustalenia uwagi niech to będzie 1 ms), za każdym razem dekrementując zawartość pewnego licznika (zrealizowanego jako zwykła zmienna w pamięci mikrokontrolera). W momencie kiedy licznik ten osiąga zero – odmierzanie czasu oczekiwania kończy się. Powoduje to, że potrzebujemy kilku elementów: licznika (zmiennej), funkcji dekrementującej licznik wywoływanej ze stałą częstotliwością, funkcji ustawiającej i testującej zawartość licznika.

Najpierw zajmijmy się implementacją odmierzania kwantu czasu, a więc 1 ms. W tym celu należy skonfigurować timer SysTick, a wykonuje się to przy użyciu funkcji:

```
SysTick_Config(Ticks);
```

gdzie **Ticks** oznacza liczbę taktów sygnału wejściowego, po których odliczeniu (timer SysTick zlicza w dół) następuje wyzwolenie przerwania oraz reset licznika timera SysTick. Drugą przydatną funkcją jest:

```
SysTick_CLKSourceConfig(SysTick_CLKSource_X);
```

gdzie **SysTick_CLKSource_X** definiuje sygnał wejściowy dla timera SysTick. Do wyboru są dwie opcje: sygnał HCLK – **SysTick_CLKSource_HCLK** lub sygnał HCLK/8 – **SysTick_CLKSource_HCLK_Div8**. Wybór odpowiedniego sygnału taktującego jest bardzo ważny, gdyż licznik timera SysTick jest 24-bitowy, więc przy sygnale wejściowym HCLK przepelnić się on będzie z częstotliwością od HCLK/ 2^{24} do HCLK, czyli dla HCLK=72 MHz jest to zakres od około 4,29 Hz (okres około 0,23 s) do 72 MHz (okres około 13,89 ns). W przypadku jednak sygnału wejściowego HCLK/8 zakres dostępnych częstotliwości wynosi od HCLK/ 2^{27} do HCLK/8, czyli dla HCLK=72 MHz, od około 0,54 Hz (okres około 1,86 s) do 9 MHz (okres około 111,11 ns). Tak więc aby zliczać pojedyncze sekundy należy koniecznie użyć sygnału wejściowego HCLK/8. W tym ćwiczeniu proponowane jest zliczanie kwantów 1 ms, a więc wybór zegara wejściowego nie jest krytyczny (na obu można zrealizować to zadanie) – dla przykładu zostanie użyty sygnał HCLK/8. Tak więc **konfiguracja timera SysTick**, który zgłasza przerwanie co 1 ms wygląda następująco:

```
SysTick_Config(9000); // (72MHz/8) / 9000 = 1KHz (1/1KHz = 1ms)  
SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8);
```

Obsługa przerwania generowanego przez SysTick sprawdza się do implementacji funkcji:

```
void SysTick_Handler(void);
```

znajdującej się w pliku **stm32f10x_it.c**.

Deklaracje i definicje funkcji związanych z opóźnieniem warto pisać w osobnych plikach, o nazwach odpowiednio np. **delay.h** oraz **delay.c**. W pliku nagłówkowym należy stworzyć zmienną będącą licznikiem milisekund o przykładowej nazwie **msc** typu **static unsigned int**. Słowo kluczowe **static** w tym przypadku służy do ograniczenia widoczności zmiennej **msc** do pojedynczej jednostki kompilacji (w uproszczeniu, jednostką kompilacji jest pojedynczy plik z nagłówkami). Oznacza to, że kompilator nie będzie zgłaszał błędów dotyczących wielokrotnej deklaracji tej samej zmiennej. Oczywiście zmienną tą należy zainicjalizować zerem.

W pliku źródłowym **delay.c** należy zdefiniować potrzebne funkcje: funkcja do dekrementacji licznika (o ile jest większy od 0) oraz funkcja do zmiany wartości licznika i testowania czy jest on większy od 0. **Definicje tych funkcji pozostawia się czytelnikowi do uzupełnienia:**

```

void DelayTick(void){
    // dekrementacja licznika (o ile jest wiekszy od 0)
}

void Delay(unsigned int ms){
    // zmiana wartosci licznika
    // testowanie czy jest on wiekszy od 0
}

```

Funkcja `DelayTick` powinna być wywoływana co 1 ms, a więc **należy ją dodać do ciała obsługi przerwania timera SysTick**, natomiast funkcja `Delay` będzie od tej pory wykorzystywana do wprowadzania opóźnień poprzez jej wywołanie w postaci `Delay(time)`, gdzie `time` jest to liczba milisekund jakie chcemy odczekać. Oczywiście, o ile to już nie zostało zrobione, **należy pozbyć się poprzedniej – programowej – implementacji opóźnienia** lub co najmniej zmienić jej nazwę.

Na koniec warto zauważyć, że domyślnie SysTick nie ma zbyt wysokiego priorytetu – zaleca się ustalenie jego priorytetu na dość wysokim poziomie (tj. należy obniżyć jego wartość). Rozsądna z punktu widzenia tego ćwiczenia jest wartość 0. **Zmianę priorytetu przerwania generowanego przez timer SysTick realizuje się przy użyciu**

```
NVIC_SetPriority(SysTick_IRQn, 0);
```

uprzednio konfiguruując timer SysTick. Kolejność wynika z zawartości definicji funkcji `SysTick_Config`.

Jak widać obsługa timera SysTick jest wyjątkowo prosta, lecz doskonała do wielu zadań związanych z odliczaniem stałych kwantów czasu – stąd jego wielka użyteczność w kontekście systemów operacyjnych.

Ustawienie podziału priorytetów przerwań: Przed rozpoczęciem pracy nad przerwaniami warto **ustalić w jaki sposób wartość priorytetu przerwania ma być dzielona** na priorytet wywłaszczenia i podpriorytet. Służy do tego funkcja:

```
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_X);
```

gdzie podział zmienia się w zależności od wartości `NVIC_PriorityGroup_X`, a dokładniej od wartości X zgodnie z poniższym:

- 0 – 0 bitów priorytetu wywłaszczenia, 4 bity podpriorytetu,
- 1 – 1 bit priorytetu wywłaszczenia, 3 bity podpriorytetu,
- 2 – 2 bity priorytetu wywłaszczenia, 2 bity podpriorytetu,
- 3 – 3 bity priorytetu wywłaszczenia, 1 bit podpriorytetu,
- 4 – 4 bity priorytetu wywłaszczenia, 0 bitów podpriorytetu.

Odmierzanie czasu przy użyciu timera ogólnego przeznaczenia: Poza timerem SysTick, do odmierzania stałych odcinków czasu można wykorzystać także zwykłe timery ogólnego przeznaczenia. Metod realizacji tego zadania jest wiele – jedna z nich wymaga następującej konfiguracji timera (odmierzanie odcinków czasowych o długości 1 s):

- prescaler = 7200,
- zawartość rejestru *auto-reload* = 10000,
- tryb zliczania w góre,
- włączona obsługa przerwania zgłoszanego przy zerowaniu licznika timera.

Tak uruchomiony timer będzie zliczał 10000 taktów wejściowego sygnału zegarowego o częstotliwości HCLK/7200, czyli 10 kHz. Oznacza to, że licznik będzie się przepętliał równo co sekundę, co będzie powodowało wyzerowanie licznika timera, a za razem zgłoszenie stosownego przerwania. Funkcja, która służy do obsługi tego przerwania będzie więc wywoływana dokładnie co sekundę.

Dodatkowe skonfigurowanie odpowiednich kanałów tego timera może być przydatne do odmierzania również odcinków czasu o długości 1 s, lecz tym razem np. przesuniętych w czasie o 0,2 s względem wcześniej skonfigurowanego timera. Aby tego dokonać należy skonfigurować kanał tego timera z następującymi właściwościami:

- niezmienna wartość rejestru wyjściowego kanału OCxREF,
- włączona obsługa przerwania zgłoszanego gdy zawartość licznika timera jest równa 2000.

Pozwoli to na uzyskanie wspomnianego przesunięcia wywołania okresowego przerwania kanału względem przerwania związanego z samym timerem. Poniżej znajduje się stosowna **implementacja powyższego odmierzania czasu** w kodzie wykorzystującym standardową bibliotekę peryferialną:

```

TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
TIM_OCInitTypeDef TIM_OCInitStructure;

TIM_TimeBaseStructure.TIM_Prescaler = 7200-1;           // 72MHz/7200=10kHz
TIM_TimeBaseStructure.TIM_Period = 10000;                // 10kHz/10000=1Hz (1s)
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; // zliczanie w gore
TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;         // brak powtorzen
TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure);          // inicjalizacja TIM4
TIM_ITConfig ( TIM4, TIM_IT_CC2 | TIM_IT_Update, ENABLE ); // wlaczenie przerwan
TIM_Cmd(TIM4, ENABLE);                                  // aktywacja timera TIM4

// konfiguracja kanału 2 timera
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Timing;      // brak zmian OCxREF
TIM_OCInitStructure.TIM_Pulse = 2000;                      // wartosc do porownania
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; // wlaczenie kanału
TIM_OC2Init(TIM4, &TIM_OCInitStructure);                 // inicjalizacja CC2

```

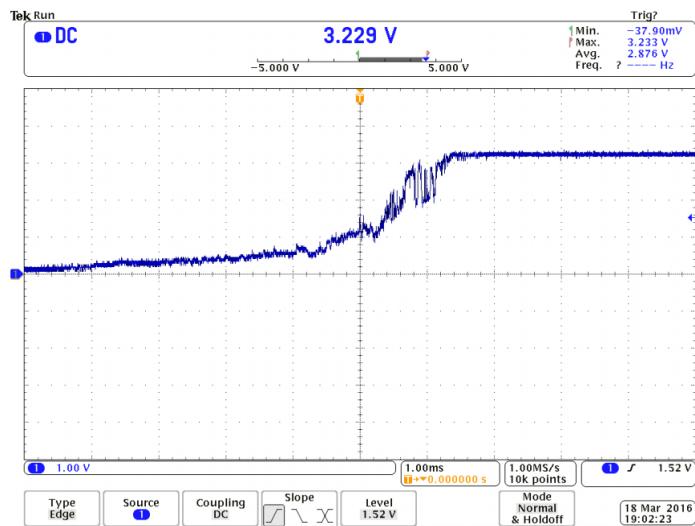
Konfiguracja przerwania związanego z timerem następuje poprzez wykorzystanie modułu NVIC:

```

NVIC_InitTypeDef NVIC_InitStructure;

NVIC_ClearPendingIRQ(TIM4_IRQn);                         // wyczyszczenie bitu przerwania
NVIC_EnableIRQ(TIM4_IRQn);                             // wlaczenie obsługi przerwania
NVIC_InitStructure.NVIC IRQChannel = TIM4_IRQn;        // nazwa przerwania
NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 2; // priorytet wywłaszczenia
NVIC_InitStructure.NVIC IRQChannelSubPriority = 1;       // podpriorytet

```



Rysunek 26: Oscylogram zjawiska drgań styków (wciśnięcie przełącznika)

```
NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;           // włączenie
NVIC_Init(&NVIC_InitStructure);                           // inicjalizacja struktury
```

Po wprowadzeniu powyższych konfiguracji można napisać funkcję obsługującą przerwanie:

```
void TIM4_IRQHandler(void){
    if(TIM_GetITStatus(TIM4, TIM_IT_CC2) != RESET){
        LED(LED2, LED_TOGGLE);
        TIM_ClearITPendingBit(TIM4, TIM_IT_CC2);
    } else if(TIM_GetITStatus(TIM4, TIM_IT_Update) != RESET){
        LED(LED3, LED_TOGGLE);
        TIM_ClearITPendingBit(TIM4, TIM_IT_Update);
    }
}
```

Warto zauważyć, że jedna funkcja obsługuje oba przerwania, dlatego na początku funkcji ważne jest aby sprawdzić, które przerwanie zostało zgłoszone. W przypadku gdy jest to przerwanie wynikające z przepełnienia licznika timera – przełączana jest dioda LED3. Jeśli jest to przerwanie wynikające z nastąpienia równości między zawartością licznika timera a rejestrze kanału 2 – przełączana jest dioda LED2. Powoduje to, że obie diody będą świecić z takim samym okresem, lecz będzie między nimi przesunięcie w fazie o 0,2 s. Na koniec każdego programu do obsługi przerwań koniecznie trzeba wyczyścić bit świadczący o oczekiwaniu na usługę tego przerwania. Wykonuje się to (w przypadku timerów) przy użyciu funkcji `TIM_ClearITPendingBit`. W przypadku niewykonania tej operacji przerwanie to będzie nieustannie fałszywie zgłasiane jako nieobsłużone.

Niwelowanie drgań styków: Zjawisko drgań styków jest powszechnie znanym problemem związanym głównie z przełącznikami mechanicznymi i enkoderami. Szczegóły powstawania tego zjawiska nie będą omawiane – warto jednak mieć świadomość jakie są jego skutki. Na Rys. 26 widoczny jest przykład pomiaru wartości napięcia na przełączniku w momencie jego wcisnięcia. Zamiast zaobserwować pojedynczą zmianę wartości napięcia, widać wyraźnie wiele krótkich impulsów. Właśnie te krótkie piki powodują, że prosty odczyt stanu przełącznika może być wyjątkowo uciążliwy w implementacji. Dwie podstawowe metody odczytu stanu przełącznika to nieustanne odpytywanie lub wykorzystanie przerwań. Nieustanne odpytywanie implementuje się poprzez odczytywanie w nieskończonej pętli stanu przycisku:

```
while(1){  
    // ...  
    stan_przelacznika = GPIO_ReadInputDataBit(GPIOx, GPIO_Pin_y);  
    // ...  
}
```

Powyższy kod jest często uważany za błędny lub co najmniej niewłaściwy. Wynika to z zasady jego działania – zamiast reagować wyłącznie na zmiany poziomu napięcia na odpowiedniej linii wybranego portu, nieustannie testowany jest jego stan. Dodatkowym problemem jest czas odpytywania – jeśli mamy mało do zrobienia to będzie on krótki, lecz jeśli nagle postanowimy wykonać jakąś dłuższą, konieczną operację jesteśmy pozbawieni możliwości monitorowania stanu przełącznika.

Drugim podejściem jest wykorzystanie przerwań. Jest ono znacznie wygodniejsze od poprzedniego, ponieważ bez względu na obciążenie zadaniami wciąż jesteśmy w stanie zareagować na zmianę stanu przełącznika. W przypadku odpytywania jeśli wykonywana operacja jest czasochłonna, to dopiero po jej zakończeniu można na nowo sprawdzić stan przełącznika. Przekłada się to bezpośrednio na wygodę w obsłudze mikrokontrolera przez użytkownika końcowego. W przypadku nieustannego odpytywania przełącznika, użytkownik musi trzymać go tak długo wcisniętym, aż mikrokontroler zdąży go odpytać o obecny stan. Sytuacja ta przypomina „zawieszenie się” systemu operacyjnego – mikrokontroler nie reaguje na interakcję. W sytuacji wykorzystania mechanizmu przerwań, mikrokontroler otrzymuje natychmiastową informację o zmianie stanu przełącznika, co może skutkować anulowaniem obecnie wykonywanego zadania lub choćby prośbą o oczekивание na zakończenie działania. Jest to niemal konieczne w przypadku interakcji użytkownik-mikrokontroler, aby ten pierwszy miał świadomość, że mikrokontroler nie przestał działać, a jest po prostu bardzo zajęty realizacją niezmiernie ważnych zadań.

Wykorzystanie przerwań prowadzi jednak do innego problemu – reakcje na zmiany stanu przełącznika są niemal natychmiastowe. A więc widoczny na Rys. 26 stan przełącznika jest widziany często właśnie z taką dokładnością – każda jego zmiana jest rejestrowana i zgłasza za pomocą mechanizmu przerwań. Zamiast więc uzyskać jedną pewną informację o wcisnięciu przełącznika, otrzymujemy ich wiele o różnym poziomie zaufania.

Tak jak w wielu innych aspektach programowania mikrokontrolerów, tak i tutaj sposobów na radzenie sobie z drganiem styków jest mnóstwo. Jednym z najpopularniejszych jest sprzętowa realizacja filtra dolnoprzepustowego (niwelującego sygnał o wysokiej częstotliwości). Tutaj poruszony jednak zostanie mechanizm programowy, tj. opóźnienie odczytu. Zasada działania jest następująca:

1. jeśli zmieniony został stan przełącznika, np. z wysokiego na niski – zgłoś przerwanie,
2. jeśli przerwanie zgłoszone, to oczekaj chwilę, aby zniknęły drgania,

3. odczytaj ustalony stan przełącznika.

Pomysł ten opiera się na założeniu, że istnieje pewien maksymalny czas występowania drgań styków – czasem taką informację można znaleźć w dokumentacji przełączników. W pozostałych przypadkach warto rozważyć czas od 20 do 50 ms – jest to na tyle krótki czas, aby opóźnienie nie było zauważalne, a na tyle długi, żeby skuteczność takiego mechanizmu niwelacji drgań styków była satysfakcyjną.

Aby zrealizować powyższą koncepcję należy **skonfigurować przerwanie zewnętrzne** (wykrycie zbozca opadającego na wybranej linii) oraz **timer odpowiadający za realizację opóźnienia**. Oczywiście należy to **poprzedzić dodaniem odpowiednich plików** standardowej biblioteki peryferialnej do projektu oraz stosowną konfiguracją pinu, do którego podpięty jest rozważany przełącznik. Konfiguracja przerwania wygląda następująco:

```

GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0); // PA0 -> EXTI0_IRQHandler
EXTI_InitStructure.EXTI_Line = EXTI_Line0; // linia : 0
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; // tryb : przerwanie
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; // zbozce: opadajace
EXTI_InitStructure.EXTI_LineCmd = ENABLE; // aktywowanie konfig.
EXTI_Init(&EXTI_InitStructure); // inicjalizacja

NVIC_ClearPendingIRQ(EXTI0_IRQHandler); // czyszcz. bitu przerw.
NVIC_EnableIRQ(EXTI0_IRQHandler); // wlaczenie przerwania
NVIC_InitStructure.NVIC IRQChannel = EXTI0_IRQHandler; // przerwanie EXTI0_IRQHandler
NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0; // prior. wywlaszczania
NVIC_InitStructure.NVIC IRQChannelSubPriority = 0; // podpriorytet
NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE; // aktywowanie konfig.
NVIC_Init(&NVIC_InitStructure); // inicjalizacja

```

Konfiguracja timera jest analogiczna jak w przypadku okresowego wyzwalania przerwania, lecz tym razem sam **timer zostaje skonfigurowany jako wyłączony** (wraz z przerwaniami przez niego generowanymi):

```

TIM_TimeBaseStructure.TIM_Prescaler = 7200-1; // 72MHz/7200=10kHz
TIM_TimeBaseStructure.TIM_Period = 350; // 10kHz/350~=29Hz(35ms)
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; // zliczanie w gore
TIM_TimeBaseStructure.TIM_RepetitionCounter = 0; // brak powtorzen
TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); // inicjalizacja TIM3
TIM_ITConfig ( TIM3, TIM_IT_Update, DISABLE ); // wylaczenie przerwan
TIM_Cmd(TIM3, DISABLE); // wylaczenie timera

NVIC_ClearPendingIRQ(TIM3_IRQHandler); // czyszcz. bitu przerw.
NVIC_EnableIRQ(TIM3_IRQHandler); // wlaczenie przerwania
NVIC_InitStructure.NVIC IRQChannel = TIM3_IRQHandler; // nazwa przerwania
NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 1; // prior. wywlaszczania
NVIC_InitStructure.NVIC IRQChannelSubPriority = 1; // podpriorytet
NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE; // aktywowanie konfig.
NVIC_Init(&NVIC_InitStructure); // inicjalizacja

```

Taka konfiguracja pozwala nam na wstrzymanie uruchomienia timera do momentu kiedy będzie on potrzebny. A potrzebny będzie w momencie gdy zgłoszone zostanie przerwanie wynikające z wykrycia zbozca opadającego na linii podłączonej do przełącznika. Z tego powodu należy następująco **zaimplementować obsługę tego przerwania**:

```

void EXTI_IRQHandler(void){
    if(EXTI_GetITStatus(EXTI_Line0) != RESET){           // sprawdzenie przyczyny
        EXTI_ClearITPendingBit(EXTI_Line0);              // wyczyszczenie bitu przerwania

        TIM_SetCounter(TIM3, 0);                         // reset licznika timera
        TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE );      // aktywacja przerwania
        TIM_Cmd(TIM3, ENABLE);                          // aktywacja timera TIM3
    }
}

```

Powyższy kod spowoduje, że w momencie wykrycia przerwania na linii 0, wyzerowany i uruchomiony zostanie timer. Gdy timer się przepełni wyzwolone zostanie jego przerwanie, które należy obsłużyć:

```

void TIM3_IRQHandler(void){
    if(TIM_GetITStatus(TIM3 ,TIM_IT_Update) != RESET){ // sprawdzenie przyczyny
        TIM_ClearITPendingBit(TIM3 , TIM_IT_Update);    // wyczyszczenie bitu przerw.
        if(GPIO_ReadInputDataBit(GPIOA , GPIO_Pin_0) == Bit_RESET) // jesli wcisniety
            LED(LED5 ,LED_TOGGLE);                      // zrob cos (przelacz LED5)
        TIM_ITConfig (TIM3 , TIM_IT_Update , DISABLE ); // deaktywacja przerwania
        TIM_Cmd(TIM3 , DISABLE);                       // deaktywacja timera TIM3
    }
}

```

Jest to oczywiście jedna z mnóstwa możliwości niwelacji drgań styków. Eliminacja tego zjawiska jest bardzo trudna i nie istnieje niestety metoda, która dobrze by sprawdzała się dla wszelkiego rodzaju przełączników i enkoderów – rozsądek projektanta i dostosowanie do potrzeb jest tutaj kluczową kwestią.

Obsługa klawiatury numerycznej Do zestawu ZL27ARM można bardzo łatwo podpiąć klawiaturę o 4 kolumnach i 4 wierszach (Rys. 27). Wciśnięcie poszczególnych klawiszy na tej klawiaturze powoduje zwarcie linii kolumny oraz wiersza, w których znajduje się dany klawisz. A więc jeśli wciśnięty zostanie klawisz „1”, to linia kolumny pierwszej i linia wiersza pierwszego zostaną ze sobą zwarte, jeśli zostanie wciśnięty klawisz „2” to zwarta zostanie linia wiersza 1 i kolumny 2, itd. Wykrywanie wciśniętego klawisza wymaga drobnych ulepszeń sprzętowych (podłączenia kilku rezystorów, a najlepiej również kondensatorów), które zostały wprowadzone do omawianej klawiatury w postaci osobnej płytki łączącej zestaw uruchomieniowy i klawiaturę. Schemat usprawnionej klawiatury jest przedstawiony na Rys. 28. Na schemacie widoczne są rezystory $10\text{ k}\Omega$ podciągające linie ROW1, ROW2, ROW3 i ROW4 do napięcia zasilania, rezystory $2,2\text{ k}\Omega$ podciągające linie COL1, COL2, COL3, COL4 do napięcia zasilania oraz pary rezistor-kondensator realizujące filtr dolnoprzepustowy. W każdej z par rezistor ma wartość $220\text{ k}\Omega$, a kondensator $47\text{ }\mu\text{F}$.

Klawiatura jest w całości podłączona do portu D. Piny od 10 do 13 należy skonfigurować w trybie **otwartego drenu**, natomiast pozostałe (od 6 do 9) w trybie **floating**. Zasada testowania, który klawisz jest wciśnięty jest następująca: pinom od 10 do 13 przypisana jest logiczna jedynka, co oznacza, że są one zawieszone w powietrzu, jednak dzięki rezystorom podciągającym ustala się na nich napięcie zasilania ($3,3\text{ V}$). Piny od 6 do 9 są również podciągnięte do zasilania, co powoduje, że domyślnie występuje na nich właściwe napięcie zasilania. Ponieważ wszystkie piny mają ten sam poziom napięcia, procedura sprawdzenia, który klawisz jest wciśnięty wymaga wprowadzenia dodatkowego sygnału testującego. Kolejno więc należy ustawić zero logiczne na pinie 10 (napięcie na tym pinie jest teraz równe napięciu masy), co powoduje, że



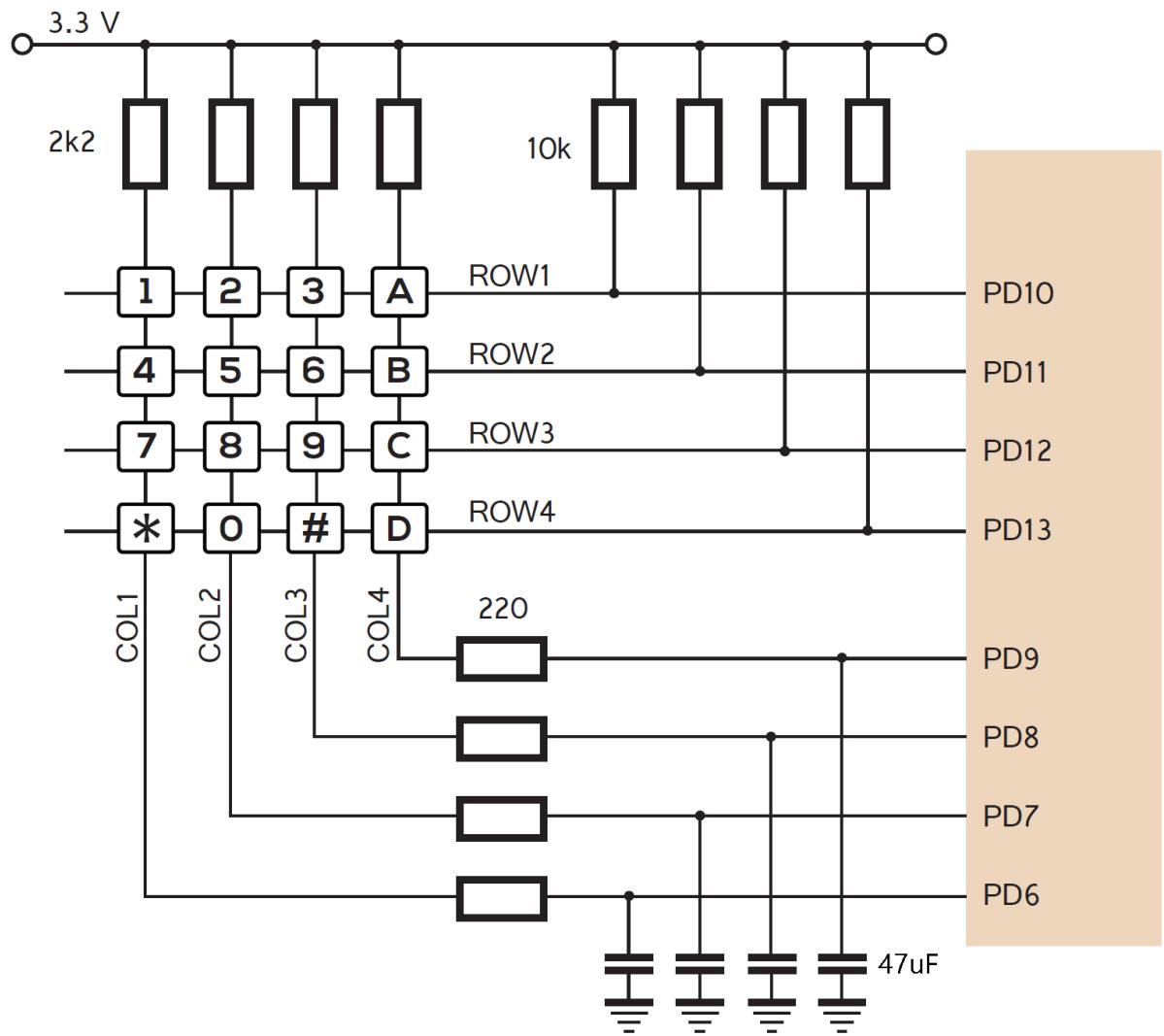
Rysunek 27: Klawiatura 4×4

jeśli któryś z klawiszy w pierwszym rzędzie jest wcisnięty, to na jednym z pinów od 6 do 9 pojawi się logiczne 0. Analogiczny zabieg należy przeprowadzić dla pinów 11, 12 i 13 ustawiając logiczną 1 na pozostałych pinach. Po przeskanowaniu wszystkich wierszy można odczytać informację o tym, dla których rzędów, w których kolumnach występowały zera logiczne. Oczywiście podejście to nie gwarantuje wykrycia wszystkich klawiszy, które są wcisnięte – możliwości tej nie daje jednak już sama budowa klawiatury. Odpowiednia kombinacja klawiszy może spowodować błędne wykrycie klawiszy niewcisniętych.

Wybrane do tego zadania tryby linii portu D są kluczowe dla bezpieczeństwa mikrokontrolera. Dla takiej konfiguracji wcisnięcie kilku klawiszy klawiatury jednocześnie nie spowoduje zwarcia – nie można wywołać sytuacji kiedy masy jest zwierana z zasilaniem. Inaczej byłoby gdyby zamiast trybu otwartego drenu zastosować wyjście *push-pull*. Wtedy dla logicznego 0 na pinie np. 10 występowałoby tam napięcie masy, a na pinach 11, 12 i 13 występowałoby napięcie zasilania. Wcisnięcie w tym momencie klawiszy z rzędów pierwszego i któregokolwiek innego spowodowałoby zwarcie masy z zasilaniem, a zarazem prawdopodobnie uszkodzenie płyty uruchomieniowej.

Poniżej znajduje się **kod służący do skanowania i odczytu klawisza** wcisniętego na klawiaturze:

```
char KB2char(void){
    unsigned int GPIO_Pin_row, GPIO_Pin_col, i, j;
    const unsigned char KBkody[16] = {'1','2','3','A', \
        '4','5','6','B', \
        '7','8','9','C', \
        '*','0','#','D'};
    GPIO_SetBits(GPIO_D, GPIO_Pin_10|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13);
    GPIO_Pin_row = GPIO_Pin_10;
    for(i=0;i<4;++i){
```



Rysunek 28: Schemat podłączenia klawiatury 4×4 (źródło: *Systemy Mikroprocesorowe w Sterowaniu. Część I: ARM Cortex-M3*)

```

    GPIO_ResetBits(GPIOB, GPIO_Pin_row);
    Delay(5);
    GPIO_Pin_col = GPIO_Pin_6;
    for(j=0;j<4;++j){
        if(GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_col) == 0){
            GPIO_ResetBits(GPIOB, GPIO_Pin_10|GPIO_Pin_11|
                           GPIO_Pin_12|GPIO_Pin_13);
            return KBkody[4*i+j];
        }
        GPIO_Pin_col = GPIO_Pin_col << 1;
    }
    GPIO_SetBits(GPIOB, GPIO_Pin_row);
    GPIO_Pin_row = GPIO_Pin_row << 1;
}
GPIO_ResetBits(GPIOB, GPIO_Pin_10|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13);
return 0;
}

```

Powyższy kod może być wykorzystany w pętli głównej programu do odpytywania (tj. nieustannego skanowania) klawiatury w celu ustalenia, które klawisze są wcisnięte (przydatne do testu podłączenia klawiatury do płyty uruchomieniowej). Podejście to jest jednak niewygodne i mało wydajne – lepiej wykorzystać przerwania.

Wybór pinów, do których zostały podpięte piny kolumn nie był przypadkowy – wykrycie na nich zbońca opadającego powoduje wyzwolenie wspólnego przerwania EXTI_Line9_5. Oznacza to, że bez względu na to, który klawisz zostanie wcisnięty, zostanie wywołana ta sama funkcja obsługująca przerwanie, gdzie będzie można wykonać powyższą funkcję do skanowania klawiatury. **Implementacja przerwania oraz jego obsługa jest analogiczna jak w przypadku poprzednich przykładów, nie będzie więc przytaczana.** Warto jednak zauważyć, że ponieważ zastosowane zostały filtry dolnoprzepustowe na wejściach mikrokontrolera można założyć, że zbocze opadające nie jest w żaden sposób zakłócone. Oznacza to, że wykryte zbocze jest jednoznacznie związane z pojedynczym wcisnięciem klawisza na klawiaturze. Nie należy implementować tutaj mechanizmu niwelacji drgań styków.

Okresowe wykonywanie pomiarów: Do tej pory wykonywanie przetwarzania analogowo-cyfrowego rozpoczynane było poprzez programowe jego uruchomienie. Jest to mało wydajna metoda, gdyż w trakcie gdy wykonywany jest pomiar można by wykonać inne potrzebne operacje, zamiast oczekiwania na otrzymanie wyniku. Poza tym regularne próbkowanie sygnału mierzonego jest kluczowe z punktu widzenia późniejszego zadania regulacji. Dlatego warto by było aby przetwarzanie rozpoczęło się ze stałą częstotliwością. Dodatkowo w trakcie wykonywania przetwarzania warto zwolnić procesor, aby nie oczekiwając bezsensownie na cyfrową postać pomiaru – zamiast tego niech zgłoszone przerwanie będzie sygnałem, że procedura przetwarzania została zakończona.

Aby regularnie wyzwolić przerwanie, które może zostać potem wykryte przez przetwornik, należy skonfigurować timer i jego kanał w trybie PWM:

```

TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
TIM_OCInitTypeDef TIM_OCInitStructure;

TIM_TimeBaseStructure.TIM_Prescaler = 7200-1;

```

```

TIM_TimeBaseStructure.TIM_Period = 5000; // 2Hz -> 0.5s
TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);

// konfiguracja kanału timera
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
TIM_OCInitStructure.TIM_Pulse = 1; // minimalne przesunięcie
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
TIM_OC2Init(TIM2, &TIM_OCInitStructure);
TIM_ITConfig ( TIM2, TIM_IT_CC2 , ENABLE );
TIM_Cmd(TIM2, ENABLE);

```

Warto zwrócić uwagę na wartość rejestru kanału – jest ona możliwie mała, aby nie wprowadzać zbędnego przesunięcia w fazie między przeładowaniem licznika timera, a wyzwoleniem przerwania przez jeden z kanałów tego timera. Dodatkowo, w przeciwieństwie do wcześniejszych przykładów, tym razem **nie implementujemy funkcji obsługującej przerwanie TIM_IT_CC2**.

Konfiguracja przetwornika jest bardzo podobna jak poprzednio:

```

void ADC_Config(void) {
    ADC_InitTypeDef ADC_InitStructure;

    ADC_DeInit(ADC1);                                // reset ustawien ADC1
    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent; // niezalezne dzialanie ADC 1 i 2
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;      // pomiar pojedynczego kanału
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE; // pomiar automatyczny
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T2_CC2; // T2CC2->ADC
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; // pomiar wyrownany do prawej
    ADC_InitStructure.ADC_NbrOfChannel = 1;             // jeden kanał
    ADC_Init(ADC1, &ADC_InitStructure);                // inicjalizacja ADC1

    ADC-RegularChannelConfig(ADC1, ADC_Channel_16, 1, ADC_SampleTime_41Cycles5); // konf.
    ADC_ExternalTrigConvCmd(ADC1, ENABLE);
    ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);

    ADC_Cmd(ADC1, ENABLE);                            // aktywacja ADC1

    ADC_ResetCalibration(ADC1);                      // reset rejestrów kalibracji ADC1
    while(ADC_GetResetCalibrationStatus(ADC1));       // oczekiwanie na koniec resetu
    ADC_StartCalibration(ADC1);                      // start kalibracji ADC1
    while(ADC_GetCalibrationStatus(ADC1));            // czekaj na koniec kalibracji

    ADC_TempSensorVrefintCmd(ENABLE);                // włączenie czujnika temperatury
}

```

Do głównych różnic należy **wybór przerwania TIM_IT_CC2 jako sygnału rozpoczynającego przetwarzanie**, zmiana kanału, który będzie wykorzystany do pomiarów i co za tym idzie dodatkowa linijka uruchamiająca czujnik temperatury znajdujący się w samym mikrokontrolerze. Oczywiście należy również pamiętać o aktywowaniu i konfiguracji przerwania wyzwalanego na zakończenie konwersji ADC_IT_EOC. **W obsłudze tego przerwania należy, tak jak zawsze, sprawdzić**, co wywo-

łało uruchomienie funkcji obsługi przerwania, **wyczyścić** bity oczekujących przerwań i **pobrać** wartość przetworzoną przez przetwornik:

```
void ADC1_2_IRQHandler(void){  
    if(ADC_GetITStatus(ADC1, ADC_IT_EOC) != RESET){  
        ADC_ClearITPendingBit(ADC1, ADC_IT_EOC);  
        sprintf((char*)bufor,"%2d*C",((V25-ADC_GetConversionValue(ADC1))/Avg_Slope+25));  
    }  
}
```

Wartości V25 i Avg_Slope są zdefiniowane jako:

```
const uint16_t V25 = 1750; // gdy V25=1.41V dla napięcia odniesienia 3.3V  
const uint16_t Avg_Slope = 5; // gdy Avg_Slope=4.3mV/C dla napięcia odniesienia 3.3V
```

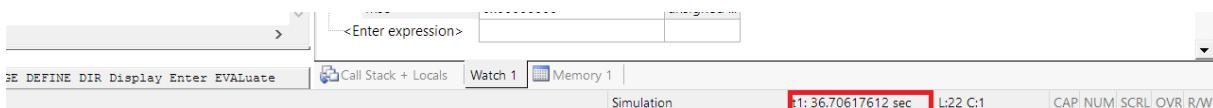
Obie te wartości wynikają z dokumentacji, tak jak sam wzór na przeliczenie wartości odczytanej z przetwornika na faktyczną wartość temperatury (rozdział 11.10 w dokumencie RM0008).

Na koniec należy **pamiętać o konfiguracji NVIC** (ADC1_2_IRQHandler) – kod ten nie różni się niemal niczym od wyżej prezentowanych fragmentów.

Temperatura mikrokontrolera jest przeważnie stała – aby zaobserwować jej zmianę warto zresetować mikrokontroler (tuż po uruchomieniu jest odrobinę chłodniejszy). Dodatkowo temperatura mikrokontrolera jest zależna od częstotliwości taktowania zegarów – spowolnienie zegara HSE powoduje zmniejszenie temperatury. Zarówno informacja o obecnej temperaturze jak i o sposobach na jej obniżenie pozwala na wykorzystanie mikrokontrolerów w wymagającym środowisku, np. małej zamkniętej obudowie telefonu, gdzie głównym źródłem ciepła jest właśnie sam mikrokontroler.

Dodatkowe informacje: Warto pamiętać o kolejności konfiguracji kolejnych peryferiali:

1. włączenie taktowania poszczególnych elementów – RCC_APBxPeriphClockCmd,
2. konfiguracja pinów GPIO – wypełnienie struktury GPIO_InitTypeDef,
3. konfiguracja docelowej funkcjonalności (timer, przetwornik ADC) – wypełnienie dodatkowych struktur np. TIM_TimeBaseInitTypeDef lub ADC_InitTypeDef,
4. włączenie generowania przerwań przez poszczególne moduły – np. TIM_ITConfig lub ADC_ITConfig,
5. konfiguracja przerwań – uzupełnienie struktury NVIC_InitTypeDef i wywołanie NVIC_EnableIRQ,
6. implementacja obsługi przerwania – wypełnienie funkcji z pliku stm32f10x_it.c.



Rysunek 29: Czas symulacji widoczny w momencie debugowania programu

2.4 Wejściówka:

Aby zaliczyć wejściówkę student musi zrealizować przed zajęciami laboratoryjnymi projekt w środowisku Keil µVision 5, w którym pewna flaga (zmienna dwustanowa) będzie przełączana co 5 s. Należy to zrealizować poprzez implementację funkcji `Delay` oraz `DelayTick`, które to są omówione w opisie do tego ćwiczenia, a ich szablon jest umieszczony w pliku `delay.c`. Dodatkowo należy w ramach tego ćwiczenia poprawnie skonfigurować zegar SysTick (oraz uzupełnić funkcję obsługi przerwania z nim związaną) w oparciu o **istniejącą** konfigurację zegara HCLK widoczną w definicji funkcji `RCC_Config` i dotychczas przyswojone informacje.

Test poprawności takiej konfiguracji jest realizowany poprzez uruchomienie programu w trybie debugowania z użyciem symulacji mikrokontrolera. Flaga `flag` powinna zmieniać swoją wartość raz na 5 s, przy czym należy mieć na uwadze, że czas symulowany nie musi pokrywać się z czasem symulacji. Oznacza to, że 5 s odmierzone przez symulator nie musi trwać 5 s obserwowanych z punktu widzenia użytkownika. Aby dowiedzieć się ile czasu zostało już zasymulowanego, należy zwrócić wzrok na prawy dolny róg okna programu Keil µVision 5, gdzie można zobaczyć pole oznaczone jako `t1` (Rys. 29) – jest to czas symulacji. Czas symulacji może zostać spowolniony do czasu symulowanego poprzez zaznaczenie opcji *Limit Speed to Real-Time* z zakładki *Debug* okna *Options for Target 'Target 1'*....

Pliki projektu, z brakującą implementacją funkcji do obsługi opóźnienia, z brakującą implementacją obsługi przerwania związanego z timerem SysTick oraz pozbawione konfiguracji timera SysTick, znajdują się w miejscu podanym przez prowadzącego. Dowodem realizacji wejściówki są zmodyfikowane pliki (tj. `delay.c`, `main.c` oraz `stm32f10x_it.c`) – te należy przesłać do prowadzącego laboratoria.

3 Obsługa złożonych czujników i urządzeń wykonawczych mikroprocesorowego systemu automatyki

Celem tego ćwiczenia jest zapoznanie studenta z popularnymi standardami przekazywania informacji w przemyśle. Jako klasyczny standard do analogowej transmisji danych użyty zostanie standard 4-20 mA, natomiast przykładem transmisji cyfrowej będzie MODBUS RTU. Jest to oczywiście znikomy podzbiór standardów komunikacyjnych, lecz pozwala on uświadomić, że nawet wyjątkowo proste rozwiązania mogą być i są skutecznie implementowane w przemyśle.

Pętla prądowa 4-20 mA. Pomiar z wykorzystaniem standardu 4-20 mA jest wyjątkowo łatwy w realizacji o ile opanowana została umiejętność pomiaru napięcia na jednym z pinów mikrokontrolera. Ponieważ rozważany zestaw rozwojowy ZL27ARM nie posiada możliwości bezpośredniego pomiaru prądu, należy wykorzystać znajomość prawa Ohma. Założymy, że posiadamy opornik o oporze R , przez który płynie prąd I . Napięcie na tym oporniku U wyrażone jest wzorem:

$$U = I \cdot R$$

Ponieważ rezystancja opornika jest stała (nie zależy od napięcia ani prądu), stąd wynika, że napięcie na tym oporniku jest wprost proporcjonalne do płynącego przez niego prądu. Natężenie prądu, które chcemy zmierzyć przyjmuje wartości od 4 do 20 mA. Mikrokontroler, którym się posługujemy jest w stanie mierzyć napięcie z zakresu 0 do 3,3 V. Aby więc prądowi 20 mA odpowiadało napięcie 3,3 V należy zastosować opornik o wartości:

$$R = \frac{U^{\max}}{I^{\max}} = \frac{3,3 \text{ V}}{0,02 \text{ A}} = 165 \Omega$$

Dla prądu o wartości 4 mA uzyskane zostanie napięcie:

$$U = 0,004 \text{ A} \cdot 165 \Omega = 0,66 \text{ V}$$

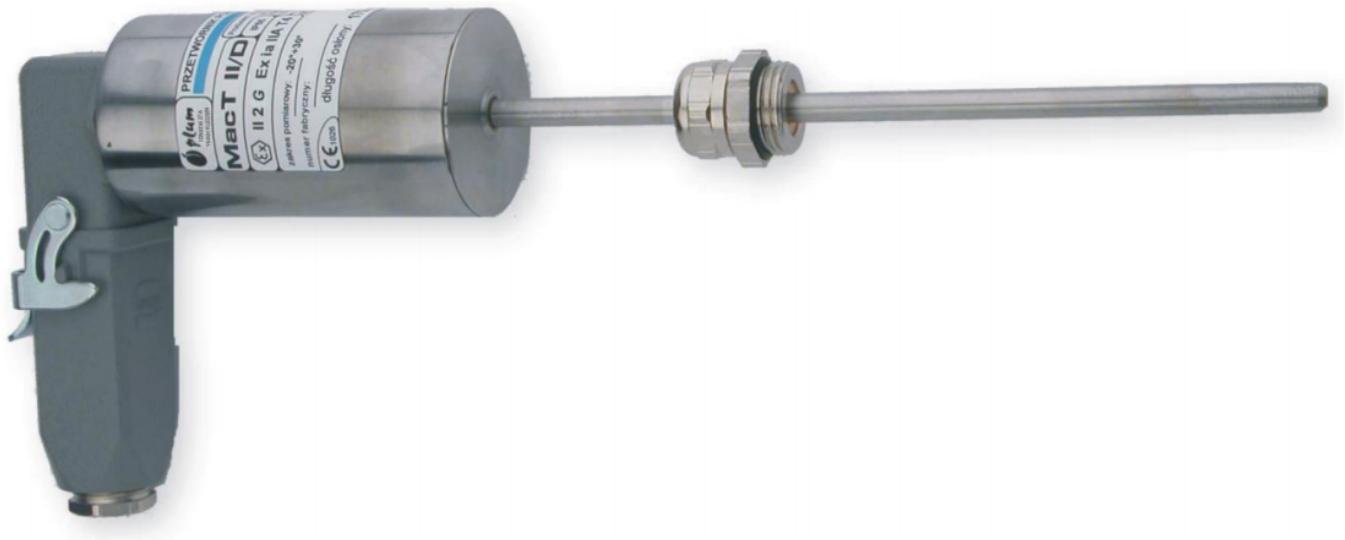
Tak dobrany rezystor posłuży do „zamiany” wartości prądu na napięcie. W razie braku opornika o stosownej wartości należy dobrać rezystor o **mniejszej** wartości, aby nie przekroczyć napięcia zasilania mikrokontrolera. W przypadku tego ćwiczenia wykorzystany zostanie rezystor o wartości 160Ω .

Zastosowanie prądu do reprezentacji pomiaru posiada wiele pozytywnych (szczególnie w przemyśle) cech. Między innymi:

- odczyt wartości prądu poniżej 4 mA oznacza uszkodzenie czujnika lub instalacji,
- podłączenie czujnika jest tanie i łatwe w realizacji,
- wpływ rezystancji linii pomiarowej na odczyt jest niewielki,
- zasilanie i odczyt pomiaru realizowany może być przy użyciu 2 kabli,
- takie podłączenie cechuje się bardzo wysoką odpornością na zakłócenia.

W związku z powyższym przemysłowe czujniki mogą być podłączone bardzo długimi kablami do urządzeń odczytujących pomiary. Pomiar z wykorzystaniem napięcia w takiej sytuacji byłby obarczony znaczącym błędem.

Implementacja pomiaru z użyciem pętli prądowej 4-20 mA nie zostanie przytoczona, gdyż jest ona identyczna jak pomiar napięcia.

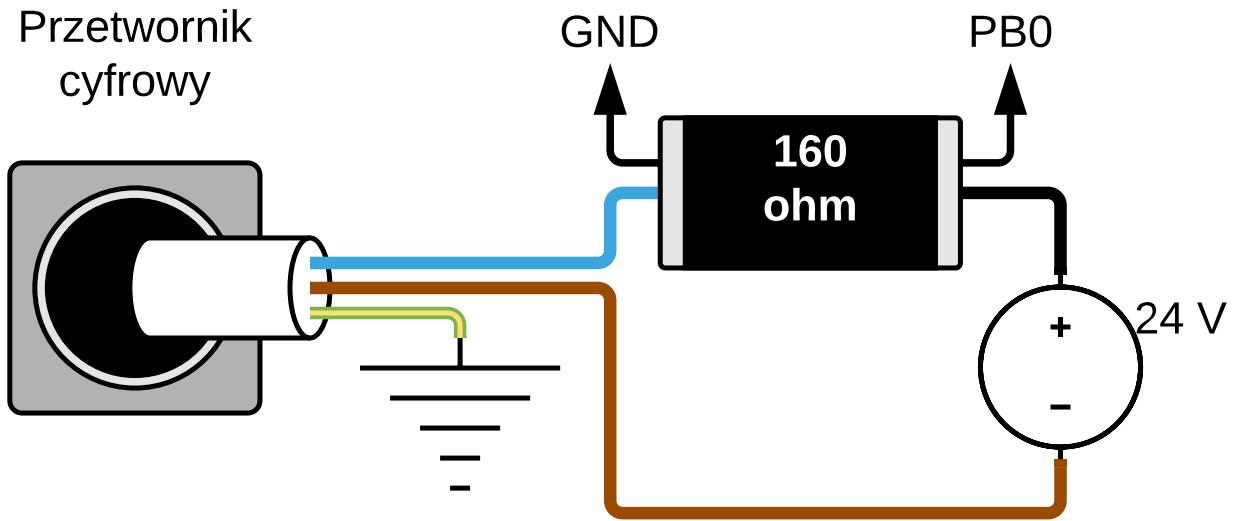


Rysunek 30: Cyfrowy przetwornik temperatury typu MacT II firmy Plum.

Cyfrowy przetwornik temperatury z termometrem rezystancyjnym. W tym ćwiczeniu jako przykład urządzenia komunikującego przy użyciu pętli prądowej 4-20 mA posłuży cyfrowy przetwornik temperatury typu MacT II firmy Plum, z termometrem rezystancyjnym Pt100 (Rys. 30). Jest on wyposażony w mikrokontroler, który steruje pomiarami oraz kompensuje charakterystyki układu pomiarowego oraz pętli prądowej, dzięki temu została osiągnięta wysoka dokładność przetwornika. Jest to sprzęt wykorzystywany w przemyśle, przeznaczony do pomiaru temperatury dowolnych mediów, na które odporna jest obudowa przetwornika wykonana ze stali kwasoodpornej. Został wykonany jako urządzenie iskrobezpieczne i może pracować w strefach zagrożenia wybuchem 1 i 2.

Powyższy przetwornik należy zasilić stałym napięciem 9-30 V, przy czym należy mieć na uwadze, iż w zależności od napięcia zasilania dopuszczalne są różne wartości rezystancji w linii zasilającej. Oznacza to, że w szeregu z termometrem można wstawić tylko odpowiednio małe rezystory, które następnie mogą być wykorzystane do pomiarów. Wcześniej ustalone zostało, że użyty zostanie opornik o wartości 160 Ω , co (zgodnie z dokumentacją) oznacza, że przetwornik należy zasilić napięciem trochę ponad 13 V. Z tego powodu do zasilania zostanie wykorzystany zasilacz o napięciu 24 V, mogący dostarczyć maksymalnie 0,6 A. Połączenie przetwornika do zasilacza przedstawione jest na Rys. 31. Należy zwrócić szczególną uwagę na podłączenie do mikrokontrolera – zamiana masy płytki z pinem wejściowym może spowodować uszkodzenie płytki rozwojowej. Dla ułatwienia podłączeń została zrealizowana płytka ze złączami śrubowymi, które tutaj posłużyły do połączenia kabli od przetwornika do płytki i od płytki do zasilacza, przy czym jedno z połączeń posiada w szeregu wlutowany opornik widoczny na schemacie.

Ostatnim ważnym aspektem jest kwestia translacji wartości zmierzonego prądu/napięcia na faktyczną temperaturę. Termometr potrafi zmierzyć wartość temperatury od -30°C do 60°C . Minimalnej wartości prądu wytwarzanej przez przetwornik (4 mA) odpowiadać będzie więc -30°C , natomiast maksymalnej



Rysunek 31: Schemat podłączenia przetwornika do zasilacza w celu pomiaru temperatury.

(20 mA) 60 °C. Wartość prądu w funkcji temperatury (t) przedstawia się więc następującym wzorem:

$$I(t) = \frac{t^{\circ}\text{C} - (-30^{\circ}\text{C})}{60^{\circ}\text{C} - (-30^{\circ}\text{C})} (20 \text{ mA} - 4 \text{ mA}) + 4 \text{ mA} = \frac{t + 30}{90} 16 \text{ mA} + 4 \text{ mA}$$

Oczywiście natychmiastowo można wyznaczyć również wzór na napięcie odłożone na oporniku w funkcji temperatury:

$$U(t) = \left(\frac{t + 30}{90} 16 \text{ mA} + 4 \text{ mA} \right) 160 \Omega = \frac{t + 30}{90} 2,56 \text{ V} + 0,64 \text{ V}$$

Warto jednak zauważyc, że w związku z użyciem mniejszego opornika niż było oryginalnie planowane, wartość napięcia dla maksymalnej mierzonej temperatury nie będzie równa 3,3 V, lecz $U(60) = 2,56 \text{ V} + 0,64 \text{ V} = 3,2 \text{ V}$.

Na koniec pozostaje kwestia reprezentacji cyfrowej takiego pomiaru. Ponieważ w rozważanym mikrokontrolerze wykorzystany jest 12-bitowy przetwornik analogowo-cyfrowy, pomiar napięcia może być reprezentowany przez wartości od 0 (0 V) do 4095 (3,3 V). A więc wartość otrzymana na mikrokontrolerze w funkcji temperatury mierzonej wygląda następująco:

$$U^c(t) = \left(\frac{t + 30}{90} 2,56 \text{ V} + 0,64 \text{ V} \right) \frac{4095}{3,3 \text{ V}} = \left(\frac{t + 30}{90} 2,56 + 0,64 \right) \frac{4095}{3,3}$$

Ponieważ jednak wartość U^c (oznaczenie c podkreśla cyfrową reprezentację napięcia) jest z założenia liczbą całkowitą, należy wynik zaokrąglić. Wynik uzyskany na mikrokontrolerze może się nieznacznie różnić w stosunku do uzyskanego przy użyciu powyższego wyliczenia, lecz jest to związane z odpornością układu na zakłócenia, który to temat nie będzie poruszany.

Wszystkie powyższe wyprowadzenia służą do tego, aby zamienić temperaturę na 12 bitową wartość cyfrową. W praktyce przyda się jednak funkcja odwrotna, pozwalająca na podstawie odczytanej wartości 12 bitowej określić jaką to reprezentuje temperaturę.

$$t(U^c) = \left(U^c \frac{3,3}{4095} - 0,64 \right) \frac{90}{2,56} - 30$$

Oczywiście przed implementacją warto uproszczyć tę funkcję, tak aby była wygodniejsza w implementacji i nie wymagała tak dużej liczby obliczeń.

Transmisja szeregową. Transmisja szeregowa (w przeciwnieństwie do transmisji równoległej) polega na sekwencyjnym przesyłaniu kolejnych bitów danych. Oznacza to, że bity nadchodzą jeden za drugim w ustalonej kolejności przy użyciu jednego połączenia. W przypadku transmisji równoległej, jednocześnie przesyłanych jest wiele bitów poprzez wykorzystanie wielu połączeń (tak jest w przypadku komunikacji z wyświetlaczem LCD znajdującym się na płytce ZL27ARM).

Oczywiście poprzez „przesył danych” należy rozumieć, że urządzenie nadawcze ustala napięcie na linii służącej do transmisji, a urządzenie odbiorcze dokonuje pomiaru tego napięcia. Transmisja cyfrowa oznacza, że dane mogą być reprezentowane wyłącznie jako 0 lub 1 logiczne (tj. napięcie poniżej lub powyżej pewnego, wcześniej ustalonego poziomu napięcia). Niewielkie zakłócenia nie powodują problemów z taką transmisją, gdyż wspomniany próg napięcia rozróżniający 0 i 1 logiczną często znajduje się w połowie przedziału dozwolonego napięcia.

Transmisja szeregowa może być realizowana w trybie synchronicznym lub asynchronicznym. W pierwszym przypadku, wykorzystując dodatkowe połączenie, przesyłany jest sygnał zegarowy. Sygnał ten służy do wyznaczania chwil, w których transmisja danych jest w stanie gotowości do odczytu/zapisu. Odbiorca i nadawca są zobowiązani do synchronizacji zegarów tak, aby odbiorca odbierał dane wyłącznie wtedy gdy nadawca te dane wysyła.

W dalszej części jednak skupienie padnie na komunikację asynchroniczną, tj. pozbawioną dodatkowego zegara taktującego. Komunikacja w tym przypadku odbywa się na podstawie założenia, że odbiorca i nadawca mają tak samo skonfigurowane zegary, na podstawie których będą wyznaczane chwile służące do nadawania/odbierania kolejnych bitów. Do określenia częstotliwości tych zegarów określa się wartość *baudrate*, która oznacza „liczbę zmian medium transmisyjnego na sekundę”. W przypadku przesyłu binarnych wartości (bitów) można tę wartość utożsamiać z bitami na sekundę. Popularne wartości, które przyjmuje się jako *baudrate* są następujące: 1200, 2400, 4800, 9600, 19200, 38400, 57600 i 115200.

Istnieją trzy możliwości zestawienia transmisji szeregowej przy użyciu trybu synchronicznego: *simplex*, *half-duplex* oraz *full-duplex*. Pierwszy z nich oznacza, że transmisja odbywa się przy użyciu jednego połączenia i jest jednokierunkowa (jedno z urządzeń zawsze wyłącznie nadaje). Transmisja half-duplex oznacza transmisję dwukierunkową przy użyciu jednego, dzielnego połączenia. Stąd też jednoczesne nadawanie i odbieranie jest niemożliwe. Ostatni tryb pozwala na jednoczesne nadawanie i odbieranie danych ze względu na wykorzystanie dwóch osobnych połączeń przeznaczonych na komunikację w każdą stronę. W dalszej części skupimy się na komunikacji *full-duplex*.

Kolejnymi parametrami transmisji szeregowej są długość porcji danych, konfiguracja bitów parzystości i liczba bitów stopu. Długość porcji danych może być równa od 5 do 8 bitów. Przesył znaków ASCII często realizuje się na 7 bitach, gdyż właśnie zajmuje jeden taki znak.

Test parzystości	Liczba bitów stopu				Długość danych							
brak	0	11110010	1	1	0	11110010	1					
parzystość	0	11110010	1	1	2	0	11110010	11	7	0	1111001	1
nieparzystość	0	11110010	0	1					6	0	111100	1
									5	0	11110	1

Rysunek 32: Możliwości konfiguracji wiadomości w transmisji szeregowej. Oznaczenia kolorystyczne: pomarańczowy – bit startu, zielony – dane, szary – bit parzystości, niebieski – bity stopu.

Bit parzystości służy jako prosty mechanizm sprawdzania poprawności danych. Są trzy (podstawowe) możliwości konfiguracji tego bitu:

- brak – do danych nie będzie dodawana informacja o ich poprawności,
- parzystość – do danych będzie dodawana informacja o tym, czy liczba jedynek w części danych jest parzysta (0 jeśli jest, 1 w przeciwnym wypadku),
- nieparzystość – do danych będzie dodawana informacja o tym, czy liczba jedynek w części danych jest nieparzysta (0 jeśli jest, 1 w przeciwnym wypadku).

Ostatnim parametrem jest liczba bitów stopu. Tutaj są tylko dwie opcje: może ich być 1 lub 2.

Transmisja szeregowa rozpoczyna się od bitu startu – zera logicznego. Następnie przesyłane są kolejne bity danych, ewentualny bit parzystości i bit(y) stopu – jedynka(-i) logiczne. Dla przykładu rozważmy konfigurację 8N1 (najpopularniejsza konfiguracja, tj. 8 bitów na dane, brak testu parzystości, 1 bit stopu). Pojedyncza wiadomość będzie składała się z:

- 1 bitu startu,
- 8 bitów danych,
- 0 bitów parzystości,
- 1 bitu stopu.

W sumie będzie to wiadomość o długości 10 bitów. Przykładowa wiadomość wygląda jak na Rys. 32 (pierwsza wiadomość w każdej kolumnie). Zakładając, że przesyłamy dane z prędkością 9600 (*baudrate*), możemy stwierdzić, że pojedynczy bajt wiadomości przesyłamy z prędkością $9600/10 = 960$ bajtów na sekundę, a więc jeden bajt wysyłany jest co $1/960 \approx 1,04$ ms.

Implementacja na ZL27ARM Implementację transmisji szeregowej z użyciem modułu USART (*Universal Synchronous and Asynchronous Receiver and Transmitter*) należy rozpocząć od wyboru wolnych pinów, które posiadają możliwość pracy jako pin nadawczy i odbiorczy modułu USART. Do takich pinów należą między innymi piny PA9 (nadawczy) i PA10 (odbiorczy) – są one częścią USART1. Ponieważ komunikacja szeregowa jest funkcją alternatywną tych pinów należy je odpowiednio skonfigurować. Przedtem

jednak należy zadbać o dołączenie do projektu plików do obsługi USART (tj. `stm32f10x_usart.*`) oraz włączenie odpowiednich modułów, tj.:

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE); // włącz taktowanie AFIO
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); // włącz taktowanie GPIOA
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE); // włącz taktowanie USART1
```

Następnie należy przejść do właściwej konfiguracji pinów do komunikacji z użyciem USART1:

```
GPIO_InitTypeDef GPIO_InitStruct;

// Pin nadawczy należy skonfigurować jako "alternative function, push-pull"
GPIO_InitStruct.GPIO_Pin = GPIO_Pin_9;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOA, &GPIO_InitStruct);

// Pin odbiorczy należy skonfigurować jako wejście "pływające"
GPIO_InitStruct.GPIO_Pin = GPIO_Pin_10;
GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOA, &GPIO_InitStruct);
```

Dalej następuje konfiguracja samej transmisji szeregowej:

```
USART_InitTypeDef USART_InitStruct;
USART_InitStructUSART_BaudRate = 19200;
USART_InitStructUSART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructUSART_WordLength = USART_WordLength_9b;
USART_InitStructUSART_Parity = USART_Parity_Even;
USART_InitStructUSART_StopBits = USART_StopBits_1;
USART_InitStructUSART_Mode = USART_Mode_Tx | USART_Mode_Rx;

USART_Init(USART1, &USART_InitStruct);
USART_ITConfig(USART1, USART_IT_RXNE, DISABLE);
USART_ITConfig(USART1, USART_IT_TXE, DISABLE);
```

Kolejne linie oznaczają prędkość transmisji, tj. *baudrate* 19200, brak sprzętowej kontroli przepływu danych, 8 bitów na dane, test parzystości, 1 bit stopu, transmisja w obie strony. Następnie następuje przypisanie powyższej konfiguracji do USART1 i wyłączenie przerwań związanych z odbiorem (*RXNE – RX buffer Not Empty*) i nadawaniem (*TXE – TX buffer Empty*). Przerwania te są wyzwalane odpowiednio w chwili gdy bufor odbiorczy przestanie być pusty, bufor nadawczy zostanie opróżniony (tj. wszystkie dane zostaną wysłane). Warto zwrócić uwagę na konfigurację długości danych – wartość ta w mikrokontrolerach rodziny STM32 oznacza długość danych wraz z bitem parzystości. A więc jeśli bit parzystości jest wykorzystywany, należy pamiętać o dodaniu tego bitu do długości słowa (jak to jest nazwane w standardowej bibliotece peryferialni).

W dalszej implementacji wykorzystane zostaną oczywiście przerwania związane z komunikacją, lecz muszą one zostać użyte w taki sposób, aby nie wysyłać gdy nie ma nic do wysłania i nie odbierać gdy niczego się nie spodziewamy. Ponieważ mowa o przerwaniach to konieczne jest również nadanie priorytetu przerwaniu:

```
NVIC_InitStructure.NVIC IRQChannel = USART1 IRQn;
NVIC_InitStructure.NVIC IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
```

Dopiero po tym etapie należy włączyć USART:

```
USART_Cmd(USART1, ENABLE);
```

Transmisja szeregowa – standard MODBUS RTU. Implementacja protokołu MODBUS RTU została opracowana na podstawie dokumentów: *MODBUS over Serial Line: Specification & Implementation guide V1.0* (obecnie dostępna jest nowsza wersja) oraz *MODBUS Application Protocol Specification V1.1b3*. Oba dokumenty są dostępne na stronie www.modbus.org. Dodatkową inspiracją przy opracowywaniu użytej niżej implementacji była implementacja o nazwie FreeMODBUS (www.freemodbus.org) – nie zawierała ona jednak wygodnego mechanizmu wysyłania żądań (tj. pracy w trybie *master*). Omawiana implementacja jest jednocześnie uproszczona na tyle, aby można było bez większych trudności zrozumieć zasadę działania tego protokołu. Warto dodatkowo zwrócić uwagę na fakt, że implementacja FreeMODBUS nie uwzględnia wykorzystania przerwy między poszczególnymi znakami oznaczonej w dokumentacji jako t1,5. Protokół MODBUS pozwala na dużą elastyczność w implementacji, dlatego niestety należy mieć na uwadze, że dwie różne implementacje tego protokołu mogą nie współpracować ze sobą w pełni. Wciąż jednak jest to jeden z najbardziej popularnych protokołów w przemyśle.

Implementacja protokołu MODBUS RTU (nie została zaimplementowana wersja ASCII ani TCP/IP) jest tak naprawdę implementacją maszyny stanów opisanej w dokumentacji tego protokołu (Rys. 14 z *MODBUS over serial line...*). W zależności od stanu i kontekstu, wykonywane są poszczególne operacje przejścia między stanami, co pozwala na wygodne i niemal jednoznaczne ustalenie przyczyny błędu w razie jego występowania. Aby aktualizować stan wspomnianej maszyny stanów należy regularnie wywoływać **void MB(void)**. Wywoływanie tej funkcji powinno odbywać się nie rzadziej niż odświeżany jest timer odpowiedzialny za wyznaczanie czasu t1,5 oraz t3,5 (omówione zostaną za chwilę). Z poziomu urządzenia typu *master* (gdyż taki będzie nas interesował) nie można wysyłać jednocześnie dwóch wiadomości. Co więcej należy uważać, aby nie podjąć takiej próby, gdyż obecna implementacja nie jest na to odporna – obowiązkiem użytkownika i programisty jest zachowanie ostrożności lub zapewnienie sobie takiego środowiska, w którym nie dojdzie do wysłania dwóch osobnych zapytań w jednym momencie (dokładniej: nie kolejno). Wynika to z prostego mechanizmu tworzenia wiadomości – istnieje bufor, w którym przechowywane są zarówno bajty wychodzące jak i przychodzące, w zależności od stanu maszyny stanów. Próba wielokrotnego wysłania danych (lub wysłania danych w trakcie odbierania odpowiedzi) może spowodować nadpisanie niewysłanej lub nieprzetworzonej ramki danych. W związku z tym należy pamiętać, że protokół MODBUS jest protokołem typu *master-slave*, a co za tym idzie, *master* wysyła zapytanie i oczekuje na odpowiedź od *slave-a*. Dopiero taka para zdarzeń powoduje, że można ponownie wysłać zapytanie.

Wspomniane zostało, że funkcja aktualizująca maszynę stanów powinna być wywoływana nie rzadziej niż odświeżany jest pewien timer. Wymieniony timer służy od wyznaczania czasów t1,5 oraz t3,5, które (zgodnie z dokumentacją) oznaczają czas potrzebny na przesłanie pojedynczego znaku przemnożony przez

kolejno 1,5 oraz 3,5. Czasy te wyznaczają moment, kiedy zakończona została transmisja pojedynczej wiadomości (t1,5) oraz czas między poszczególnymi wiadomościami (t3,5). Mimo bardzo zbliżonego znaczenia (dlatego prawdopodobnie FreeMODBUS nie wykorzystuje t1,5), ich rozróżnienie jest wyraźne w dokumentacji. Aby odmierzyć ten czas można wykorzystać albo dwa timery (dla każdego z czasów osobny) lub jednego, który będzie zliczał małe kwanty czasu, co pozwoli na ustalenie kiedy minęło t1,5 oraz t3,5. Drugie rozwiązanie ma dwie kluczowe zalety – oszczędność timerów oraz elastyczność. Ponieważ MODBUS może pracować przy różnych prędkościach przesyłu danych, także i czasy t1,5 i t3,5 są zmienne. Zamiast komplikować proceduręinicjalizacji timerów można modyfikować jedynie wartość liczników. Stąd bierze się zalecenie dotyczące częstotliwości wywoływania funkcji MB() – jeśli będzie ona wywoływana rzadziej niż pojedynczy kwant zliczany przez timer, możemy odczekać więcej czasu niż było w założeniu. Może to nie być zauważalne, lecz dla zwiększenia niezawodności implementacji warto przestrzegać możliwie dokładnie limitów czasowych. W tej implementacji wykorzystany zostanie timer, który odliczać będzie kwanty 50 μ s – taka sama lub wyższa częstotliwość jest zalecana do wywoływania funkcji MB().

Konstrukcja wiadomości w protokole MODBUS RTU przedstawiona jest w pliku *MODBUS Application Protocol Specification...* w rozdziale *6 Function codes descriptions*. Opis jest wyjątkowo prosty, co pozwala na bezproblemową ręczną konstrukcję wiadomości.

Do wysyłania wiadomości w trybie *master* wykorzystana jest funkcja:

```
void MB_SendRequest(uint8_t addr, MB_FUNCTION f, uint8_t* datain, uint16_t lenin)
```

która jako kolejne parametry przyjmuje:

- adres urządzenia typu *slave*,
- identyfikator funkcji protokołu MODBUS,
- adres tablicy zawierającej treść wiadomości,
- długość treści wiadomości.

Bajty CRC są dodawane automatycznie. Identyfikatory funkcji zostały zaimplementowane w postaci zmiennej wyliczeniowej (*enum*) w pliku *main.h*. Podobnie wygląda funkcja służąca do odebrania odpowiedzi:

```
MB_RESPONSE_STATE MB_GetResponse(uint8_t addr, MB_FUNCTION f,
                                    uint8_t** dataout, uint16_t* lenout, uint32_t timeout)
```

która także przyjmuje jako pierwsze parametry adres oraz identyfikator funkcji protokołu MODBUS – wykorzystane jest to do sprawdzenia poprawności odpowiedzi. Pozostałymi parametrami są:

- adres na zmienną, do której ma być wpisany adres tablicy, w której znajduje się treść odpowiedzi,
- adres na zmienną, do której ma być wpisana długość treści odpowiedzi,
- wartość w milisekundach określająca ile czasu mikrokontroler ma oczekiwania na odpowiedź od urządzenia typu *slave*.

Funkcja ta zwraca stan odpowiedzi MB_RESPONSE_STATE, który może przyjmować jedną z następujących wartości:

- RESPONSE_OK – odpowiedź poprawna,

- RESPONSE_TIMEOUT – czas oczekiwania na odpowiedź został przekroczony,
- RESPONSE_WRONG_ADDRESS – odpowiedź zawiera inny adres urządzenia typu *slave* niż oczekiwany,
- RESPONSE_WRONG_FUNCTION – odpowiedź zawiera inny identyfikator funkcji niż oczekiwany,
- RESPONSE_ERROR – urządzenie typu *slave* zgłosiło błąd (typ błędu zawarty jest w treści wiadomości).

Wartość argumentu określającego czas oczekiwania na wiadomość powinna wynosić około 1 s, lecz dokładny czas oczekiwania nie został zdefiniowany (w dokumentacji można znaleźć jedynie sugestie – rozdział 2.4.1 dokumentu *MODBUS over serial line...*)

Dla przykładu, gdyby chcieć z urządzenia *master* wysłać do urządzenia *slave* o adresie 103 wiadomość ustawienia wartości cewki 3. na 1, należałoby wywołać następujący kod:

```
MB_SendRequest(103, FUN_WRITE_SINGLE_COIL, write_single_coil_3, 4);
```

gdzie `write_single_coil_3` zdefiniowane jest jako:

```
uint8_t write_single_coil_3[] = {0x00, 0x03, 0xFF, 0x00};
```

W odpowiedzi otrzymane zostanie echo wiadomości nadanej:

```
uint8_t *resp;
uint16_t resplen;
MB_RESPONSE_STATE respstate;
respstate = MB_GetResponse(103, FUN_WRITE_SINGLE_COIL, &resp, &resplen, 1000);
```

a więc wartości znajdujące się w tablicy `resp` powinny pokrywać się z zawartością tablicy `write_single_coil_3`.

Z drugiej strony, gdyby chcieć odczytać z tego urządzenia *slave* wartości przez niego zmierzzone (np. dyskretne wejście 4), należałoby wysłać następującą wiadomość:

```
MB_SendRequest(103, FUN_READ_DISCRETE_INPUTS, read_discrete_input_4, 4);
```

gdzie

```
uint8_t read_discrete_input_4[] = {0x00, 0x04, 0x00, 0x01};
```

Odpowiedź urządzenia *slave* odbierana jest przy użyciu:

```
uint8_t *resp;
uint16_t resplen;
MB_RESPONSE_STATE respstate;
respstate = MB_GetResponse(103, FUN_READ_DISCRETE_INPUTS, &resp, &resplen, 1000);
```

gdzie wartość cewki znajduje się na najmniej znaczącym bicie w tablicy spod adresu `resp`. Analogicznie można przeprowadzić zapis i odczyt wartości 16-bitowych – informacje o strukturze wiadomości znajdują się w dokumencie *MODBUS Application Protocol Specification....* Z powyższego wynika pewna niedogodność – to użytkownik jest odpowiedzialny za uzupełnienie ramki zgodnej z protokołem MODBUS RTU w treść odpowiadającą wykorzystywanej funkcji protokołu, gdyż dostarczona jest jedynie funkcja do przesyłania ramek, a nie uzupełniania ich treści.

Propozycja implementacji protokołu MODBUS RTU zrealizowana została w postaci szablonu, który wymaga od użytkownika implementacji kilku funkcji. Funkcje te mają następujące deklaracje:

- `void __attribute__((weak)) Disable50usTimer(void)` – funkcja służąca do wyłączenia działania timera odliczającego kwanty 50 μ s,
- `void __attribute__((weak)) Enable50usTimer(void)` – funkcja służąca do włączenia działania timera odliczającego kwanty 50 μ s,
- `uint8_t __attribute__((weak)) Communication_Get(void)` – funkcja służąca do odczytania pojedynczego znaku,
- `void __attribute__((weak)) Communication_Mode(bool rx, bool tx)` – funkcja służąca do przełączania modułu wykorzystanego do komunikacji w tryb oczekiwania na znak (tj. tryb nasłuchiwanie), wysyłania danych (tj. tryb transmisji) lub oczekiwania (wyłączenia) – jednokrotne włączenie transmisji i nasłuchiwanie nie jest wykorzystane,
- `void __attribute__((weak)) Communication_Put(uint8_t c)` – funkcja służąca do wysłania pojedynczego znaku.

Nadanie funkcji atrybutu `weak` pozwala na zdefiniowanie funkcji, której domyślna postać jest pusta, natomiast użytkownik może ją „nadpisać”. Jest to podobnie działający mechanizm jak przeciążanie znane z języka C++.

Dodatkowo użytkownik jest zobowiązany do wywołania kilku funkcji mających na celu sygnalizację pewnych zdarzeń lub odświeżenie wartości związanych z implementacją protokołu MODBUS RTU. Przykładowa implementacja podanych funkcji i wywołanie prezentowane są poniżej. Przyjęte zostały pewne założenia:

- zegar SysTick zgłasza przerwanie co 10 μ s,
- za odliczanie kwantów 50 μ s odpowiedzialny jest timer TIM4,
- do komunikacji zostanie wykorzystany USART1.

Konfiguracja zegara SysTick była już prezentowana – zostanie tutaj pominięta. Należy jednak zwrócić uwagę na fakt, iż do tej pory w obsłudze przerwania wynikającej z działania SysTick-a znajdowało się wywołanie funkcji `DelayTick()`, które powodowało dekrementację licznika milisekund. Ponieważ częstotliwość pracy zegara SysTick zmieniła się – należy wprowadzić odpowiednie zmiany także i w wywołaniu/definicji funkcji związanych z opóźnieniem.

Zgodnie z wcześniejszym opisem, w funkcji obsługi przerwania SysTick musi znaleźć się wywołanie funkcji `MB()`. Dodatkowo jednak umieszczone tutaj zostanie wywołanie funkcji `TimeoutTick()`, której zadaniem jest dekrementacja licznika tak, jak ma to miejsce przy funkcji opóźnienia. Pozwala ona za to na realizację nie opóźnienia, a odmierzenia pewnej ilości czasu jednocześnie nie blokując działania programu. Można by to oczywiście zrealizować z użyciem timera, lecz ponieważ dokładność pomiaru tutaj nie odgrywa kluczowej roli można zastosować właśnie tak prosty mechanizm. Na koniec warto pamiętać o nadaniu przerwaniu zgłoszanemu przez SysTick jeden z wyższych priorytetów, aby nie doprowadzić do zakleszczenia programu.

Ponieważ wykorzystany zostanie USART1 do komunikacji, potrzeba go skonfigurować. Podstawową konfiguracją protokołu MODBUS RTU jest 8E1, z prędkością 19200 baudrate – wykorzystana zostanie

jednak prędkość 115200. Dodatkowo USART1 będzie pracował z użyciem przerwań TXE, RXNE, których funkcja obsługi zdefiniowana jest jako:

```
void USART1_IRQHandler(void){
    if( USART_GetITStatus(USART1, USART_IT_RXNE) ){
        USART_ClearITPendingBit(USART1, USART_IT_RXNE);
        SetCharacterReceived(true);
    }
    if( USART_GetITStatus(USART1, USART_IT_TXE) ){
        USART_ClearITPendingBit(USART1, USART_IT_TXE);
        SetCharacterReadyToTransmit();
    }
}
```

Widoczne tutaj wywołania funkcji `SetCharacterReceived(true)` oraz `SetCharacterReadyToTransmit()` mają na celu poinformowanie funkcji obsługującej protokół MODBUS o odpowiednio otrzymaniu znaku i osiągnięciu gotowości do transmisji znaku. Faktyczne wysłanie oraz odebranie znaku realizowane jest w osobnych funkcjach, które wywoływane są z poziomu funkcji `MB()`. Ich definicje wymagają zaimplementowania przez użytkownika – przykładowo w następujący sposób:

```
void Communication_Put(uint8_t ch){
    USART_SendData(USART1, ch);
}

uint8_t Communication_Get(void){
    uint8_t tmp = USART_ReceiveData(USART1);
    SetCharacterReceived(false);
    return tmp;
}
```

gdzie `SetCharacterRecieved` jest funkcją ustawiającą flagę świadczącą o gotowości do odczytu kolejnego znaku – po odbiorze té gotowości należy oczywiście odwołać, co jest zrealizowane wyżej.

Funkcja służąca do przełączania modułu komunikacyjnego w tryb nasłuchiwanego i transmisji jest również wyjątkowo prosta i może zostać zrealizowana jako funkcja włączająca i wyłączająca stosowne przerwania w module USART1:

```
void Communication_Mode(bool rx, bool tx){
    USART_ITConfig(USART1, USART_IT_RXNE, rx?ENABLE:DISABLE);
    USART_ITConfig(USART1, USART_IT_TXE, tx?ENABLE:DISABLE);
}
```

Kolejnymi funkcjami wymagającymi implementacji są:

```
void Enable50usTimer(void){
    TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
}

void Disable50usTimer(void){
    TIM_ITConfig(TIM4, TIM_IT_Update, DISABLE);
}
```

które są wykorzystane do zatrzymywania i uruchamiania timera odmierzającego kwanty $50\mu\text{s}$. Aby na liczanie tych kwantów miało faktycznie miejsce należy dodać do obsługi przerwania generowanego przez TIM4 wywołanie funkcji `Timer50usTick`, która nie przyjmuje argumentów. Konfiguracja samego timera nie będzie przytaczana gdyż była ona omawiana w poprzednim ćwiczeniu.

Procedura inicjalizacji komunikacji z użyciem protokołu MODBUS składa się z następujących etapów:

1. Konfiguracja USART – pinów PA9 i PA10, modułu USART1, przerwań,
2. Konfiguracja timera – modułu TIM4, przerwań,
3. Konfiguracja protokołu MODBUS – tj. wywołanie `MB_Config`, w argumencie podając prędkość komunikacji (`baudrate`),
4. Wyłączenie przerwań USART-a – tj. stosowne wywołanie `Communication_Mode`,
5. Konfiguracja SysTick-a (wyzwalając od tej pory regularnie `DelayTick`, `TimeoutTick` i `MB`),-
6. Nadanie SysTick-owi wysokiego priorytetu.

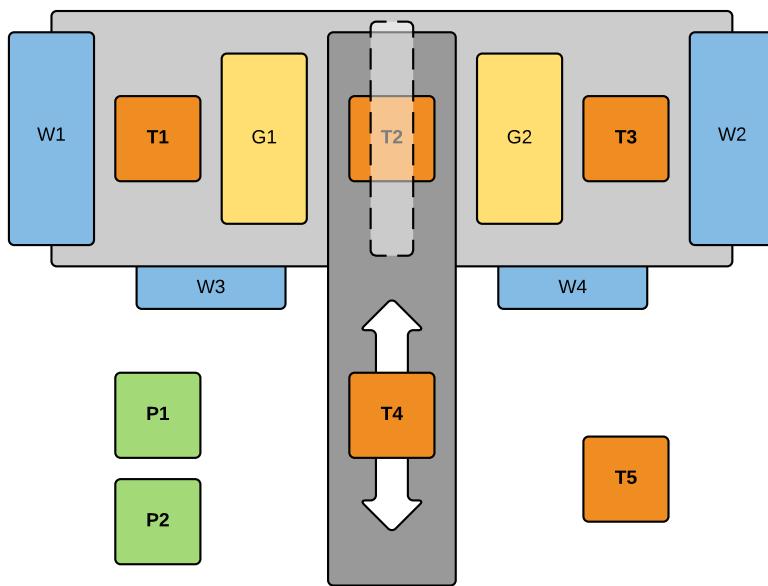
Ponieważ omawiane niżej stanowisko, będące adresatem wszelkich wiadomości wysyłanych z mikrokontrolera STM32, posiada interfejs komunikacyjny RS-485, a nie TTL czy RS-232 jak na rozważanej płycie rozwojowej – wykorzystany zostanie stosowny konwerter. Jego schemat jest nieistotny z punktu widzenia realizacji ćwiczenia, tak więc zostanie on pominięty.

Stanowisko grzejaco-chłodzące. W tym ćwiczeniu (oraz następnych) wykorzystane zostanie stanowisko grzejaco-chłodzące zrealizowane w Instytucie Automatyki i Informatyki Stosowanej Politechniki Warszawskiej. Pełna dokumentacja stanowiska znajduje się w podanym przez prowadzącego katalogu, natomiast poniżej przedstawione zostaną wyłącznie istotne, z punktu widzenia tego ćwiczenia, cechy.

Stanowisko składa się z 6 elementów wykonawczych: 4 wentylatorów (W1-W4) i 2 grzałek (G1, G2) oraz 7 czujników: 5 czujników temperatury (T1-T5), pomiar prądu (P1) i pomiar napięcia (P2). Rozmieszczenie elementów widoczne jest na Rys. 33. Czujnik T5 służy do pomiaru temperatury otoczenia – aby spełniał swoją rolę właściwie, nie powinien się znajdować w okolicy strumienia powietrza wytwarzanego przez wentylatory ani nie powinien znajdować się w pobliżu nagrzewających się elementów. W ramach tego ćwiczenia skupienie pada na elementy W1, T1, G1.

Komunikacja ze stanowiskiem może odbywać się na trzy sposoby: przy użyciu opracowanego dla niego protokołu komunikacyjnego (z pośrednictwem przejściówki USB-UART), przy użyciu standardu napięciowego 0-10 V lub korzystając z protokołu MODBUS RTU i standardu RS-485. Ta ostatnia opcja będzie głównie wykorzystana w tym i następnych ćwiczeniach. Konfiguracja protokołu MODBUS RTU dla stanowiska zakłada występowanie bitu parzystości (sprawdzanie czy liczba jedynek jest parzysta), danych o długości 8 bitów i 1 bitu stopu. Prędkość transmisji jest konfigurowalna, lecz na rzecz tego ćwiczenia wykorzystana będzie prędkość 115200. Adres stanowiska jest również konfigurowalny – każde stanowisko przygotowane zostało tak, aby mieć unikalny adres.

Stanowisko to odświeża wartości pomiarów i sterowania z częstotliwością 1 Hz. Oznacza to, że zmiana sygnału sterującego może zostać zauważona maksymalnie po 1 s. Zmiana sterowania (tj. mocy z jaką pracują grzałki lub wentylatory) następuje poprzez zapisanie nowej wartości do rejestru typu *Holding Register*, o



Rysunek 33: Schemat rozmieszczenia elementów wykonawczych i pomiarowych w stanowisku grzejaco-chłodzącym

adresie od 0 do 6, natomiast odczyt pomiaru dokonywany jest poprzez odczyt zawartości rejestru typu *Input Register* o adresie od 0 do 7. Wspomniane elementy, które są interesujące z punktu widzenia tego ćwiczenia mają adresy: W1 – 0, T1 – 0, G1 – 4. Wartości sterowania wyrażane są w promilach pełnej mocy elementu wykonawczego. Oznacza to, że zapisanie wartości 200 do odpowiedniego rejestru typu *Holding Register* spowoduje, że powiązany z tym rejestrem element będzie pracował z mocą równą 20,0% pełnej mocy tego elementu. Podobnie należy traktować pomiary temperatury – te wyrażone są w setnych stopniach Celsjusza. Wartość 2500 znajdująca się w rejestrze typu *Input Register* oznacza, że związanego z nim czujnik temperatury zmierzył 25,00 °C.

Aby zapewnić poprawność komunikacji ze stanowiskiem należy uważnie śledzić odpowiedzi na wysłane wiadomości. Brak odpowiedzi może sugerować problemy z komunikacją (niepoprawny adres, prędkość). Odpowiedzi zawierające kody błędu należy przeanalizować w oparciu o dokumentację protokołu MODBUS RTU.

3.1 Przebieg laboratorium

Celem ćwiczenia jest wykorzystanie komunikacji przy użyciu protokołu MODBUS RTU do sterowania mikrokontrolerem oraz pętli prądowej 4-20 mA do odczytu wartości temperatury. Aby to osiągnąć należy:

- Skonfigurować piny PA9 oraz PA10 pod kątem komunikacji USART, odpowiednio jako pin nadawczy i odbiorczy. Prędkość komunikacji musi zgadzać się z ustawieniami w stanowisku, tj. baudrate 115200, 8 bitów na dane oraz bit parzystości.

- Skonfigurować timer tak, aby odmierzał $50\mu\text{s}$ kwanty, inkrementując w ten sposób licznik.
- Korzystając z dostarczonych funkcji należy zrealizować prosty regulator, który będzie regulował temperaturę T1 stanowiska grzejaco-chłodzącego zgodnie z następującymi regułami:
 - temperatura zbyt wysoka – włącza się wentylator W1,
 - temperatura za niska – włącza się grzałka G1,
 - temperatura w okolicach (należy je rozsądnie zdefiniować) temperatury zadanej – oba elementy są wyłączone.
- Obecną oraz zadaną temperaturę należy wyświetlić na wyświetlaczu LCD,
- Należy regularnie wykonywać pomiary temperatury z wykorzystaniem pętli prądowej 4-20 mA i wynik pomiaru przedstawić na wyświetlaczu w $^{\circ}\text{C}$ (jako symbol $^{\circ}$ można wykorzystać *).

Diody można skonfigurować dowolnie – w szczególności mogą służyć jako doskonałe narzędzie do zgrubnego badania stanu wykonania programu lub nawet jako prosty mechanizm debugowania.

Prowadzący może zostać zaproponowane zadanie dodatkowe, np: zezwolić użytkownikowi na wprowadzanie wartości zadanej z klawiatury. Wymaga to obsługi kolejnego przerwania (związanego z klawiaturą), interpretacji klawiszy wcisniętych przez użytkownika oraz aplikacja nowej wartości zadanej do powyższego prostego regulatora. Aplikacja powinna nastąpić w momencie wcisnięcia klawisz ' * '. Wpisywane znaki powinny być wyświetlane na bieżąco, aby użytkownik miał kontrolę nad tym co pisze.



Rysunek 34: Zdjęcie płytki rozwojowej STM32F746G-DISCO

4 Obsługa wyświetlacza graficznego LCD (panelu dotykowego), wykorzystanie jednostki zmiennopozycyjnej do przetwarzania sygnałów

Celem tego ćwiczenia jest zapoznanie się z nową platformą z rodziną STM32. Zadaniem studenta jest zapoznać się z obsługą nowych elementów służących do komunikacji z użytkownikiem, takich jak kolorowy wyświetlacz graficzny, ekran dotykowy. Jednocześnie student jest zobowiązany rozsądnie zarządzić czasem procesora, a co za tym idzie priorytetami przerwań. Efektem końcowym tego ćwiczenia powinien być program, który jest wygodny w użyciu (tj. taki, który będzie działał szybko, nie będzie zatrzymywał pracy i będzie stale responsywny).

STM32F746G-DISCO Przejście z płytka rozwojowej ZL27ARM na płytę STM32F746G-DISCO (Rys. 34) nie jest skomplikowanym procesem. Jednak należy pamiętać, że rodzina mikrokontrolerów STM32F7 i wyższe nie posiadają już dostosowanych do nich standardowych bibliotek periferiali. Wyparta ona została przez bibliotekę HAL (*Hardware Abstraction Layer*), która w połączeniu z generatorem kodu CubeMX ma za zadanie usprawnić projektowanie i implementację programów na te mikrokontrolery. W związku z tym, że nauka i oswojenie się z nową biblioteką byłoby czasochłonne, dalsze ćwiczenia będą w dużej mierze oparte o przygotowany wcześniej szablon, tak, aby skupić się na poszerzaniu umiejętności, a nie ich pogłębianiu – ostatecznym celem jest przecież implementacja regulatora, a nie nauka programowania mikrokontrolera.

Wspomniana płytka wyposażona jest w wyświetlacz LCD-TFT o przekątnej 4,3 cala, rozdzielcości 480x272 z pojemnościowym ekranem dotykowym. Złącze zgodne z Arduino Uno V3 zamieszczone po drugiej stronie przydatne będzie do zamontowania na następnych ćwiczeniach dodatkowych przetworników cyfrowo-analogowych. Oryginalnie mikrokontroler zamontowany na tej płytce wyposażony jest w dwa takie przetworniki, lecz nie są one przygotowane do użytku ogólnego – są wykorzystywane do generacji

dźwięku stereo. Posiada on jednak przetwornik analogowo-cyfrowy, który jest jednocześnie podłączony do pinów złącza Arduino.

Do ciekawszych właściwości tego mikrokontrolera, a właściwie jego rdzenia – Cortex®-M7 – należy możliwość obliczeń w arytmetyce zmiennopozycyjnej (pojedynczej precyzji). Jest to znaczące ułatwienie i usprawnienie programów, które tę arytmetykę wykorzystują. W poprzednim mikrokontrolerze korzystanie z dobrodziejstwa arytmetyki na liczbach o zmiennym przecinku odbywało się poprzez programową implementację takich operacji (zajmował się tym dyskretnie kompilator). Teraz do tego posiadamy sprzętowe narzędzie, co pozwoli na sprawniejsze wykonywanie przyszłego kodu regulatora.

Kolejnym przydaniem tej płytka jest dołączony na płytce programator ST-Link. Zmniejsza to wyraźnie ilość sprzętu potrzebnego do programowania takiego mikrokontrolera. Dodatkowo ST-Link jest również wykorzystany w roli debuggera, co powinno być dla użytkownika końcowego (niemal) identyczne w działaniu w stosunku do np. programatora/debuggera J-Link.

Biblioteka HAL W dotychczasowych projektach używana była Standardowa Biblioteka Peryferiali. Dla mikrokontrolerów z rodziną STM32F7 nie została ona przygotowana. W jej miejsce pojawia się biblioteka HAL (*Hardware Abstraction Layer*). Przyczyną tej zmiany jest próba dalszego ułatwiania użytkownikom (programistom) generacji kodu. Słowo „generacja” nie zostało użyte przypadkowo – wraz z biblioteką HAL ST wytworzył oprogramowanie służące do automatycznej generacji kodu z pełną konfiguracją wszystkich potrzebnych peryferiali z odpowiednimi ustawieniami. Narzędziem tym jest STM32CubeMX. W tym ćwiczeniu zostanie wykorzystany już wygenerowany kod, dodatkowo jeszcze oczyszczony z powtarzających się fragmentów.

Warto mieć na uwadze, że wraz ze wprowadzaniem kolejnych uproszczeń i ujednoliceniem w kodzie, twórcy i użytkownicy takiego kodu narażają się na pojawiające się redundancje. Tak jak biblioteka SPL nie jednorazowo powtarza te same operacje na rejestrach, tak i biblioteka HAL generuje powtórzenia wywołań funkcji SPL. Tutaj należy zaznaczyć, że wiele funkcji ze Standardowej Biblioteki Peryferiali wchodzi w skład biblioteki HAL. Co więcej nieraz biblioteka HAL wprowadza jedynie nową nazwę funkcji znaną z SPL. Najważniejszą jednak różnicą między wykorzystaniem SPL a HAL jest struktura projektu. W dotychczasowych projektach wszystko było konfigurowane wprost – HAL ma na celu oddzielenie warstwy sprzętowej od aplikacji, bibliotek i stosów. Pozwala to na łatwiejszy rozwój aplikacji bez uzależnienia od konkretnej płytki wykorzystanej w projekcie.

Biblioteka HAL jak i SPL posiadają swoich zwolenników i przeciwników – tutaj nie będą omawiane szczegółowo cechy obu podejść. Warto jednak mieć na uwadze, że o ile projekt jest w fazie testów – użycie biblioteki HAL jest jak najbardziej wskazane. W przypadku jednak gdy produkt jest gotowy i zaczyna się optymalizacja wydajności – wtedy warto posprzątać trochę kod, który został wygenerowany automatycznie.

Omówienie funkcji biblioteki HAL zostanie ograniczone do minimum. W tym oraz kolejnych ćwiczeniach oczekuje się od studenta umiejętności dobierania parametrów konfiguracji (o ile to konieczne) na podstawie komentarzy w kodzie źródłowym bibliotek oraz własnej domyślności. Wiele nazw wciąż jest intuicyjnych (w szczególności stałych), a modyfikacja parametrów konfiguracji i uzupełnianie jej pojedynczych pól jest zadaniem, które było realizowane na każdym z dotychczasowych ćwiczeń – zakłada się więc, że umiejętność ta została opanowana przez studenta.

Wyświetlacz LCD z ekranem dotykowym W ramach tego ćwiczenia wykorzystana zostanie biblioteka do obsługi wyświetlacza LCD oraz ekranu dotykowego dostarczona wraz z jednym z przykładów przez ST. Zagłębienie się w kod tej biblioteki nie jest celem tego ćwiczenia – należy skupić się na jej jak najlepszym wykorzystaniu. W tym ćwiczeniu będzie wykorzystanych kilka kluczowych funkcji do rysowania na wyświetlaczu:

- `BSP_LCD_SetTextColor` – funkcja do ustawienia koloru tekstu i koloru rysowanych pikseli,
- `BSP_LCD_SetBackColor` – funkcja do ustawienia koloru tła tekstu,
- `BSP_LCD_DisplayStringAt` – funkcja do wyświetlania tekstu w podanym miejscu na wyświetlaczu,
- `BSP_LCD_DrawLine` – funkcja do rysowania linii,
- `BSP_LCD_DrawPixel` – funkcja do rysowania pojedynczego piksela,
- `BSP_LCD_FillRect` – funkcja do rysowania wypełnionego prostokąta (zniechęca się do korzystania z tej funkcji, gdyż bez zapewnienia mechanizmu synchronizacji może nie generować spodziewanych efektów),
- `BSP_LCD_GetXSize` – funkcja zwracająca szerokość ekranu,
- `BSP_LCD_GetYSize` – funkcja zwracająca wysokość ekranu.

Argumenty przyjmowane przez te funkcje są oczywiste – nie będą tutaj objaśniane.

Do obsługi ekranu dotykowego wymagane jest utworzenie specjalnej struktury:

```
TS_StateTypeDef TS_State;
```

której zawartość będzie odświeżana za każdym razem, gdy sobie tego zażyczy użytkownik. Do odświeżenia zawartości użyć należy funkcji `BSP_TS_GetState`, której argumentem jest wskaźnik na instancję wspomnianej struktury. Do przydatnych atrybutów tej struktury należą:

- `touchDetected` – atrybut przechowujący liczbę wykrytych jednocześnie dotknięć ekranu,
- `touchX` – tablica przechowująca współrzedną poziomą dla każdego z wykrytych jednocześnie dotknięć ekranu (początek osi znajduje się z lewej strony ekranu),
- `touchY` – tablica przechowująca współrzedną pionową dla każdego z wykrytych jednocześnie dotknięć ekranu (początek osi znajduje się u góry ekranu),

Ekran dotykowy komunikuje się z użyciem protokołu I²C i posiada wiele dodatkowych możliwości (jak np. wykrywanie gestów), lecz ewentualne zagłębienie tego tematu pozostawia się zainteresowanym. W związku z uproszczoną obsługą ekranu dotykowego, zamiast oczekiwania na przerwanie sygnalizujące zmianę stanu ekranu, będzie on regularnie odpłytywany o stan. Wystarczająco duża częstotliwość sprawdzania powinna pozwolić na równie wygodną obsługę programu, co wykorzystanie przerwań.

Przy obsłudze ekranu należy zwrócić uwagę na współdzielenie zasobów – w tym przypadku będzie to wyświetlacz. Wystąpienie przerwania w momencie gdy wyświetlany jest obraz może spowodować, że zostanie wyświetlone nie to co powinno lub przynajmniej nie tak jak powinno. Warto użyć w tym przypadku semafora lub monitora, które pozwolą uzyskać wyłączny dostęp do wyświetlacza przez tylko jedną funkcję.

Jednostka do obliczeń zmiennoprzecinkowych Jednostka do obliczeń zmiennoprzecinkowych wykorzystywana jest automatycznie przez kompilator, dzięki czemu kod programu korzystający z niej nie różni się niczym od tego, który jej nie używa. W ramach tego ćwiczenia przeprowadzony zostanie test w jaki sposób działa program z użyciem i bez użycia tej jednostki – powinien zostać zaobserwowany wyraźny spadek wydajności mikrokontrolera.

4.1 Przebieg laboratorium

Celem ćwiczenia jest zapoznanie się z nową platformą z rodziny STM32. Student ma za zadanie przede wszystkim zapoznać się z obsługą wyświetlacza i ekranu dotykowego, z jednoczesnym przećwiczeniem umiejętności organizacji priorytetów przerwań i zadań tak, aby zaimplementowany program był wygodny w użyciu. W tym celu należy zaprojektować i wykonać aplikację w taki sposób aby:

- podzielić wyświetlacz na dwie części: lewą, która będzie związana z rysowaniem fraktalu Mandelbrota oraz prawą, gdzie będzie rysowany wykres wartości napięcia na jednym z pinów mikrokontrolera w funkcji czasu,
- każda z części powinna posiadać na górze nagłówek (np. MANDELBROT i INPUT PLOT),
- każda z części powinna posiadać na dole przycisk modyfikujący obraz wyświetlany w odpowiadającej mu części:
 - przycisk pod fraktałem powinien być bistabilny (tj. jego stan przełącza się z każdym jego naciśnięciem) – jego przełączenie powinno powodować narysowanie fraktalu w kolorach odpowiednio od czarnego do zielonego lub od białego do czerwonego,
 - przycisk pod wykresem powinien być monostabilny i jego wcisnięcie powinno powodować wy czyszczenie wykresu i rozpoczęcie ponownego zbierania danych tak, jak po starcie programu,
- przerysowanie fraktalu nie może uniemożliwiać interakcji użytkownika – w szczególności powinno być możliwe jednoczesne przerysowywanie fraktalu i czyszczenie wykresu,
- główną cechą tego interfejsu ma być jego funkcjonalność, a nie wygląd – jest to sprawa drugorzędna z powodu ograniczonego czasu ćwiczenia.

Większość implementacji związanej z konfiguracją peryferiów jest gotowa – do studenta należy implementacja „logiki” programu. Zakłada się, że przy skończonej ilości czasu student potrafiłby odtworzyć własnoręcznie kod służący do konfiguracji. Zostało to wykonane wcześniej, aby studentowi oszczędzić pisania tej (nudnej) części programu.

Fraktal W tym ćwiczeniu jako przykład wykorzystania obliczeń w arytmetyce zmiennoprzecinkowej użyty zostanie fraktal – zbiór Mandelbrota. Algorytm jego rysowania znajduje się poniżej:

```
void mandelbrot(){  
    static int Px = 0;  
    static int Py = 0;  
    static float x = 0;  
    static float xtemp = 0;  
    static float y = 0;
```

```

static float x0 = 0;
static float y0 = 0;
static uint8_t iteration = 0;

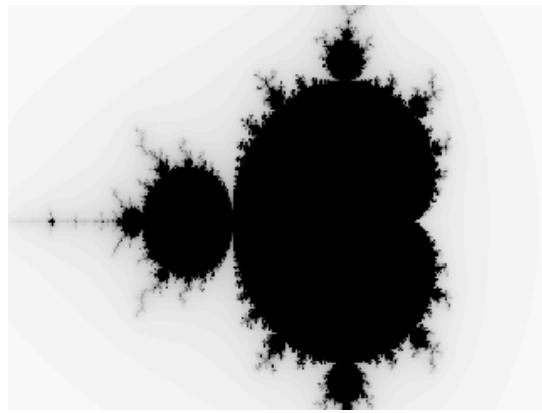
// obszar rysowany zbioru Mandelbrota
static float xmin = -0.1011-0.01;
static float xmax = -0.1011+0.01;
static float ymin = 0.9563-0.01;
static float ymax = 0.9563+0.01;

// for(Px=nr pierwszej kolumny pixeli; Px<=nr ostatniej kolumny pixeli; ++Px)
for(; ;++Px){ // TODO
    // for(Py=nr pierwszego wiersza pixeli; Px<=nr ostatniego wiersza pixeli; ++Py)
    for(; ;++Py){ // TODO
        // wyznaczyc x0 i y0 tak, aby:
        // x0 = xmin <=> Px = indeks pierwszej kolumny pikseli;
        // x0 = xmax <=> Px = indeks ostatniej kolumny pikseli;
        // y0 = ymin <=> Py = indeks pierwszego wiersza pikseli;
        // y0 = ymax <=> Py = indeks ostatniego wiersza pikseli;

        x0 = 0.0; // TODO
        y0 = 0.0; // TODO
        // algorytm rysujacy zbior Mandelbrota
        x = 0.0;
        y = 0.0;
        iteration = 0;
        while ((x*x + y*y < 2*2) && (iteration < 0xFF)) {
            xtemp = x*x - y*y + x0;
            y = 2*x*y + y0;
            x = xtemp;
            iteration = iteration + 1;
        }
        // rysowanie piksela w punkcie Px, Py,
        // o kolorze zaleznym od wartosci iteration (0x00-0xFF)
        // TODO
    }
}
}

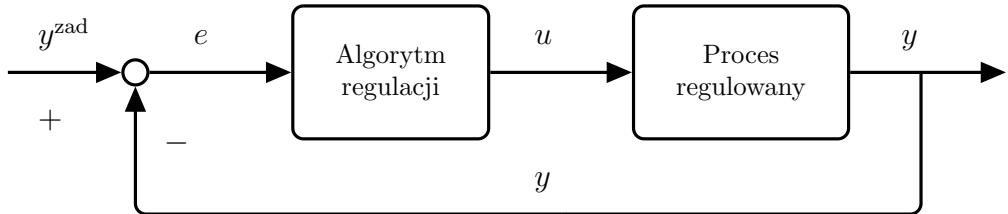
```

Ponieważ umiejscowienie rysunku nie jest jednoznacznie zdefiniowane – do programisty należy zadanie odpowiedniego przeskalowania obszaru rysowania (wyznaczenie wartości `x0` i `y0`) i wyznaczenia jego granic (wypełnienie pętli `for`). Oczywiście na koniec należy zaimplementować również rysowanie poszczególnych pikseli w zależności od zmiennej `iteration`. Wynik rysowania z użyciem powyższego kodu powinien być identyczny (z dokładnością do koloru) do widocznego na Rys. 35.



Rysunek 35: Zbiór Mandelbrota w skali szarości

Algorytm ten wykorzystany został jako test szybkości obliczeń wykonywanych przez mikrokontroler. Wielokrotne operacje na liczbach zmiennoprzecinkowych, wykonywane dla każdego z pikseli wyświetlacza, generują znaczne opóźnienie między rozpoczęciem a zakończeniem obliczeń. Opóźnienie to jest widoczne gołym okiem, co powoduje, że konieczna jest implementacja, która pozwoli na jednoczesne rysowanie czasochłonnego obrazu i obsługę pozostałej części interfejsu graficznego przez użytkownika. Jednak jedną z najważniejszych cech tego testu jest to, że bardzo ładnie on wygląda na kolorowych wyświetlaczach.



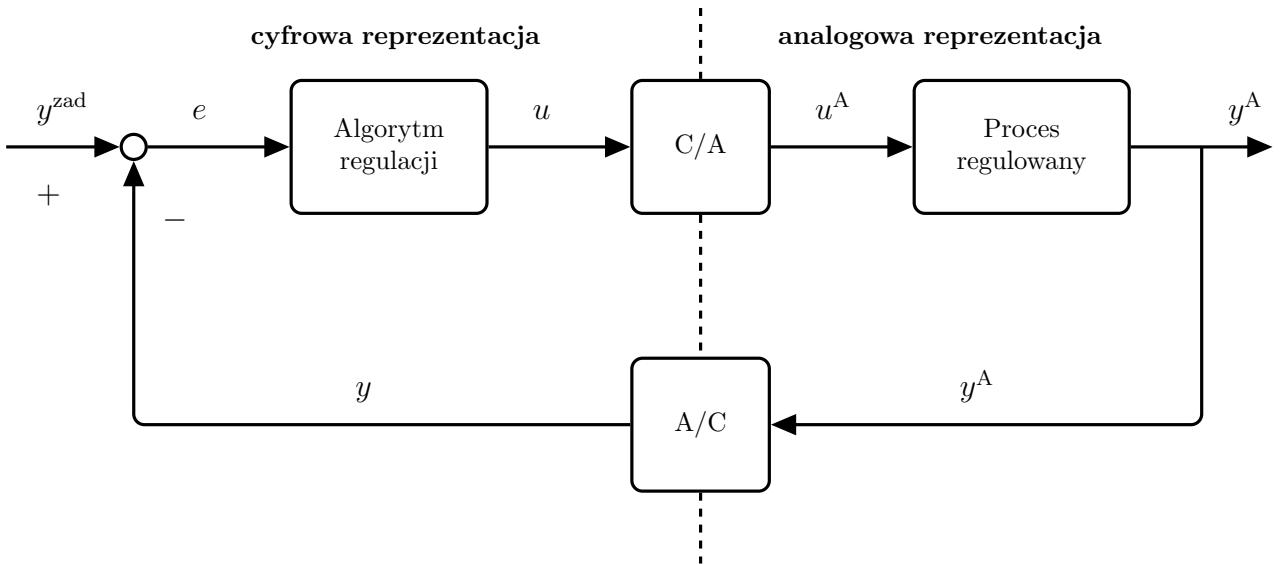
Rysunek 36: Schemat ideowy układu regulacji

5 Implementacja algorytmów regulacji PID i DMC prostego procesu dynamicznego, interfejs użytkownika, archiwizacja pomiarów, dobór nastaw algorytmów, badania porównawcze

Cel Celem tego projektu jest implementacja dwóch algorytmów regulacji: PID oraz DMC, oraz porównanie jakości ich działania przy użyciu symulowanego obiektu. Do realizacji tego zadania konieczna jest umiejętność implementacji podanych algorytmów, umiejętność gromadzenia i analizy danych oraz rozsądne zarządzanie przerwaniami i ich priorytetami. Regulator PID jest obecny powszechnie w przemyśle, dlatego jego poprawna implementacja i strojenie są ważnymi praktycznymi umiejętnościami. Regulator DMC jest jednym z najprostszych algorytmów regulacji predykcyjnej, który wymaga wyjątkowo niewiele obliczeń oraz posiada analityczne rozwiązanie pod warunkiem nieuwzględniania ograniczeń. Ponieważ warto zdawać sobie sprawę z zalet i ograniczeń obu – celem ostatecznym tego projektu jest sprawozdanie podsumowujące ich różnice w działaniu.

Zadanie regulacji Zadaniem regulacji jest taka manipulacja sygnałem wejściowym procesu (manipulowanym) u , aby wartość sygnału wyjściowego procesu (regulowanego) y była możliwie bliska wartości zadanej y^{zad} . Często wartość uchybu $y^{zad} - y$ oznacza się symbolem e (Rys. 36). Jest to podstawowa realizacja sprzężenia zwrotnego. Algorytmy regulacji mogą być bardzo zróżnicowane – począwszy od najprostszego regulatora typu „włącz/wyłącz” (przykładowy piekarnik), aż po algorytmy wykorzystujące skomplikowane modele nielinowe, aby na ich podstawie wyznaczać nowe sygnały sterowania. Rosnący poziom skomplikowania algorytmu przekłada się często zarówno na jakość regulacji (tj. jej dokładność, szybkość) jak i na wymagania związane z realizacją tego algorytmu. Stąd wciąż jednym z najpopularniejszych algorytmów regulacji jest prosty regulator PID, którego wymagania są minimalne a jakość regulacji w wielu przypadkach satysfakcyjająca.

Realizując to ćwiczenie warto mieć świadomość istnienia przetworników analogowo-cyfrowych i cyfrowo-analogowych, które występują pomiędzy procesem regulowanym a regulatorem. Dodatkowo koniecznie należy pamiętać, że czas wyznaczenia obliczeń nie zawsze może być pomijalny. W przypadku, gdy nowa wartość sygnału sterującego wyznaczana jest w połowie czasu między kolejnymi iteracjami algorytmu regulacji (chwilami próbkowania) warto rozważyć opóźnienie aplikacji do procesu nowej wartości sterowania do momentu rozpoczęcia nowej iteracji. W ten sposób sztucznie wprowadzone opóźnienie pozwoli na utrzymanie regularnego przekazywania nowej wartości sterowania do obiektu regulacji. Takie podejście jest szczególnie ważne w sytuacjach, gdy czas pojedynczej iteracji algorytmu regulacji nie jest stały (np. z powodu użycia



Rysunek 37: Schemat ideowy układu regulacji z uwzględnieniem przetworników analogowo-cyfrowych i cyfrowo-analogowych

funkcji do rozwiązywania zadań programowania kwadratowego).

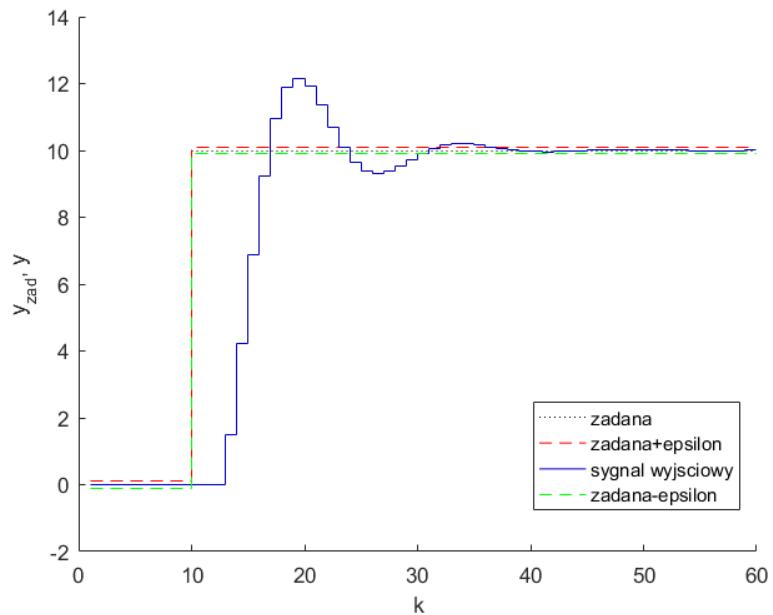
Zanim omówione zostaną algorytmy regulacji potrzebne jest wprowadzenie dwóch pojęć, które służą do oceny jakości regulacji. Są to „przesterowanie” oraz „czas ustalenia”. Obie te wartości są związane z odpowiedzią obiektu na skok wartości zadanej. Aby te wartości można było ze sobą porównać, a co za tym idzie porównać jakość regulacji różnych parametrów algorytmu regulacji, należy pamiętać o zachowaniu identycznych warunków eksperymentów. Oznacza to, że eksperymenty służące do wyznaczenia przesterowania oraz czasu ustalenia muszą rozpoczynać się zawsze w takim samym stanie początkowym, skok wartości zadanej musi być identyczny w każdym przebiegu eksperymentu a przesterowanie i czas ustalenia muszą być liczone w ten sam sposób.

Są różne definicje wartości przesterowania – jedną z najczęstszych jest

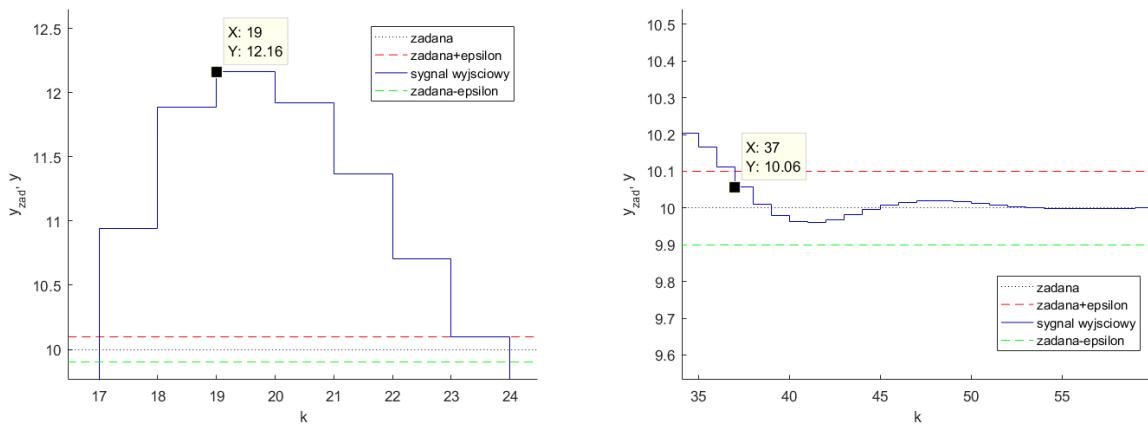
$$K_o = \frac{y_m - y^{\text{zad}}}{y^{\text{zad}}} \cdot 100\%$$

gdzie y_m jest maksymalną osiągniętą w trakcie eksperymentu wartością sygnału wyjściowego. Sygnał wyjściowy będzie w najwyższym położeniu chwilę po zmianie wartości zadanej – kolejne jego wartości powinny być od tej wyłącznie niższe. W przeciwnym razie układ może być niestabilny.

Wartość czasu ustalenia T_{ust} określa się jako czas od zmiany wartości sygnału zadanej, do chwili gdy wartość bezwzględna uchybu przekracza pewną przyjętą niewielką wartość ε . Dla przykładu, jeśli założyć, że wartość zadana y^{zad} zmieniana jest w chwili $k = 10$ z 0 na 10, a $\varepsilon = 0,1$, to czasem ustalenia jest czas od zmiany wartości zadanej (10), do momentu, w którym wartość wyjściowa wchodzi w zakres wartości $y^{\text{zad}} - \varepsilon$ do $y^{\text{zad}} + \varepsilon$ i więcej go nie opuszcza. Na Rys. 38 przedstawiony został wykres wartości wyjściowej obiektu regulacji, na podstawie którego wyznaczone mogą zostać przesterowanie oraz



Rysunek 38: Wykres będący efektem eksperymentu, mający na celu wyznaczenie wartości przesterowania oraz czasu ustalenia



Rysunek 39: Odczyt maksymalnej wartości sygnału wyjściowego obiektu regulacji będący podstawą do wyznaczenia przesterowania (lewy wykres) oraz odczyt czasu ustalenia (prawy wykres)

czas ustalenia. Rys. 39 (lewa strona) przedstawia fragment poprzedniego wykresu, gdzie widać wyraźnie wartość maksymalną sygnału wyjściowego obiektu regulacji, tj. $y_m = 12,16$. Na tej podstawie można wyznaczyć wartość przesterowania $K_o = \frac{12,16-10}{10} \cdot 100\% = 21,6\%$. Czas ustalenia wyznaczyć można na podstawie Rys. 39 (prawy wykres) – widać, że uchyb przestaje przekraczać wartość ε od chwili $k = 37$. Zakładając, że czas próbkowania jest równy 0,1 s, to czas ustalenia jest równy $T_{ust} = (37 - 10) \cdot 0,1 = 2,7$ s.

Algorytm PID PID jest to algorytm regulacji składający się z trzech czynników: proporcjonalnego (*Proportional*), całkującego (*Integral*) i różniczkującego (*Derivative*). Każdy z tych członów reaguje na zmianę sygnału wyjściowego procesu regulowanego o innym charakterze. Człon proporcjonalny powoduje wzrost wartości sterowania wraz ze wzrostem uchybu (tj. różnicy między wartością zadaną a wyjściową). Człon całkujący zwiększa wartość sygnału sterującego wraz z akumulowanym uchybem (tj. sumą uchybów z przeszłości) – jest to bardzo wygodny mechanizm pozwalający na niwelację uchybu ustalonego (zostanie on omówiony niżej). Ostatnim członem jest człon różniczkujący – im szybciej wzrasta uchyb, tym większa jest wartość sygnału sterującego.

Waga każdego z tych członów może być dowolnie modyfikowana, w szczególności każdy z tych członów może zostać wyłączony poprzez odpowiednią manipulację związaną z nim parametrem. Pozwala to na łatwe i jednocześnie dostosowanie właściwości regulatora do potrzeb użytkownika. W dalszej części omawiany będzie wyłącznie regulator PID w wersji dyskretnej – implementacja algorytmu PID w wersji ciągłej wymaga innego podejścia.

Implementacja algorytmu PID sprowadza się do wyznaczenia w każdej iteracji aktualnego uchybu i na tej podstawie wyznaczenia nowej wartości sygnału sterującego. Prawo regulacji (tj. wzór umożliwiający obliczenie wartości sygnału sterującego w aktualnej chwili dyskretnej k) algorytmu PID przedstawia się następująco:

$$u(k) = u_P(k) + u_I(k) + u_D(k) \quad (1)$$

gdzie

$$\begin{aligned} u_P(k) &= Ke(k) \\ u_I(k) &= u_I(k-1) + \frac{K}{T_I} T \frac{e(k-1) + e(k)}{2} \\ u_D(k) &= KT_D \frac{e(k) - e(k-1)}{T} \end{aligned} \quad (2)$$

Powyższe powstało poprzez zastosowanie metody Eulera oraz całkowania metodą trapezów do wzoru na ciągły w czasie regulator PID. Jak zostało wcześniej zaznaczone: $u(k)$, $y(k)$, $e(k)$ oznaczają kolejno wartości sygnału sterującego, wyjściowego procesu regulowanego i uchybu (różnice między wyjściem procesu regulowanego a wartością zadaną) w dyskretnej chwili k . $u_P(k)$, $u_I(k)$, $u_D(k)$ oznaczają kolejno wartość sterowania wyznaczoną na podstawie członu proporcjonalnego, całkującego i różniczkującego. Suma sygnałów wyjściowych trzech członów składowych regulatora jest faktycznym sygnałem sterującym, który należy zaaplikować następnie do procesu regulowanego. Zmienna T oznacza tutaj czas próbkowania (tj. czas pomiędzy kolejnymi dwiema iteracjami algorytmu regulacji) wyrażony w sekundach. Każdy z wymienionych czynników jest opisany pewnym parametrem. W przypadku członu proporcjonalnego jest to parametr K – wzmacnienie. Dla członu całkującego jest to czas zdwojenia T_I , natomiast dla członu różniczkującego tym

parametrem jest czas wyprzedzenia T_D . Wartym uwagi jest fakt, iż dla $K = 0$ wyłączony jest nie tylko człon proporcjonalny, ale także i pozostałe. Oczywiście algorytm PID ma wiele wersji i postaci – ta jednak posiada parametry, które reprezentują pewne rzeczywiste właściwości. Jeśli założyć, że uchyb $e(k) = 0$ dla $k < 0$ i $e(k) = \bar{e} \neq 0$ w pozostałych przypadkach, gdzie \bar{e} jest pewną niezerową stałą (tj. $e(k)$ jest stałe w czasie) i człon różniczkujący jest wyłączony (tj. $T_I = 0$), to T_I jest czasem (w sekundach), który musi minąć, aby $u_I(k)$ osiągnęło wartość równą $u_P(k)$ (które w tym przypadku jest, tak jak uchyb, stałe w czasie). Zakładając jednak, że wyłączony został człon całkujący (T_I jest nieskończenie duży), a uchyb $e(k) = 0$ dla $k < 0$ i $e(k) = \bar{e}k \neq 0$ w pozostałych przypadkach, gdzie \bar{e} jest pewną niezerową stałą (tj. $e(k)$ wzrasta liniowo w czasie), to T_D jest czasem (również w sekundach), po jakim człon różniczkujący $u_D(k)$ osiągnie wartość równą członowi proporcjonalnemu $u_P(k)$. Czasem wartości czasów wyprzedzenia i zdwojenia zamienia się na współczynniki wzmacniania poszczególnych członów zgodnie z poniższym:

$$K_I = \frac{K}{T_I}$$

$$K_D = KT_D$$

Tabelka z parametrami wyznaczonymi metodą Zieglera-Nicholsa operuje najczęściej takimi właśnie zmiennymi – zarówno tabela jak i metoda podane są w dalszej części.

We wzorze (2) można zobaczyć występowanie rekurencji. Aby uniknąć konieczności zapisywania dodatkowej zmiennej w pamięci mikrokontrolera, na którym realizowany będzie ten algorytm warto posłużyć się postacią przyrostową algorytmu PID

$$u(k) = r_2 e(k-2) + r_1 e(k-1) + r_0 e(k) + u(k-1) \quad (3)$$

gdzie

$$r_2 = \frac{KT_D}{T}$$

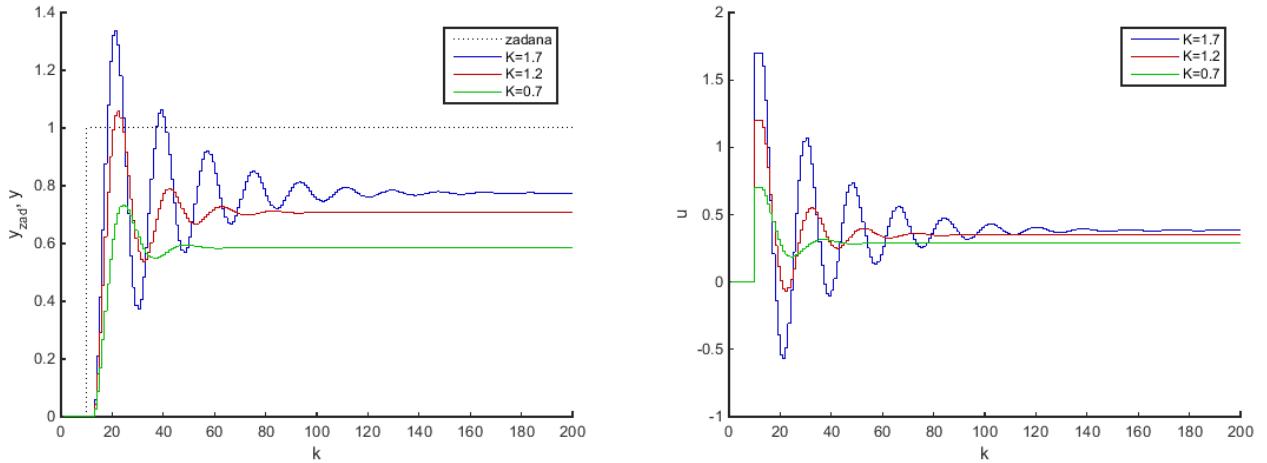
$$r_1 = K \left(\frac{T}{2T_I} - 2 \frac{T_D}{T} - 1 \right)$$

$$r_0 = K \left(1 + \frac{T}{2T_I} + \frac{T_D}{T} \right)$$

Jak widać rekurencja została zastąpiona wykorzystaniem dodatkowego uchybu z chwili $k-2$ oraz sterowania z chwili $k-1$. Wzory (1) oraz (3) są sobie równoważne, tj. można z jednego przejść do drugiego wyłącznie przy użyciu prostych przekształceń (wystellarzy od każdej strony równania (1) odjąć $u(k-1)$). Wszelkie oznaczenia są identyczne jak przy omawianiu poprzednich wzorów. Wartości r_2, r_1, r_0 nie posiadają jednak ciekawszego znaczenia niż „zmienne, które stoją przy kolejnych uchybach”. Do realizacji ćwiczenia warto posłużyć się wzorem (1), gdyż jest bardziej intuicyjny.

Skoro już wyznaczone zostały wzory, warto zastanowić się jak działa regulator PID w praktyce. Oczywiście wynika to wprost ze wzorów. Poniżej omawiane wykresy są efektem regulacji obiektu o następującym równaniu różnicowym:

$$y(k) = 0,043\ 209u(k-1) + 0,030\ 415u(k-2) + 1,309\ 644y(k-1) - 0,346\ 456y(k-2)$$



Rysunek 40: Wyjście oraz wejście procesu regulowanego – regulator P

gdzie k oznacza obecną chwilę dyskretną. Natomiast parametry algorytmu PID zostały dobrane arbitralnie jako następujące:

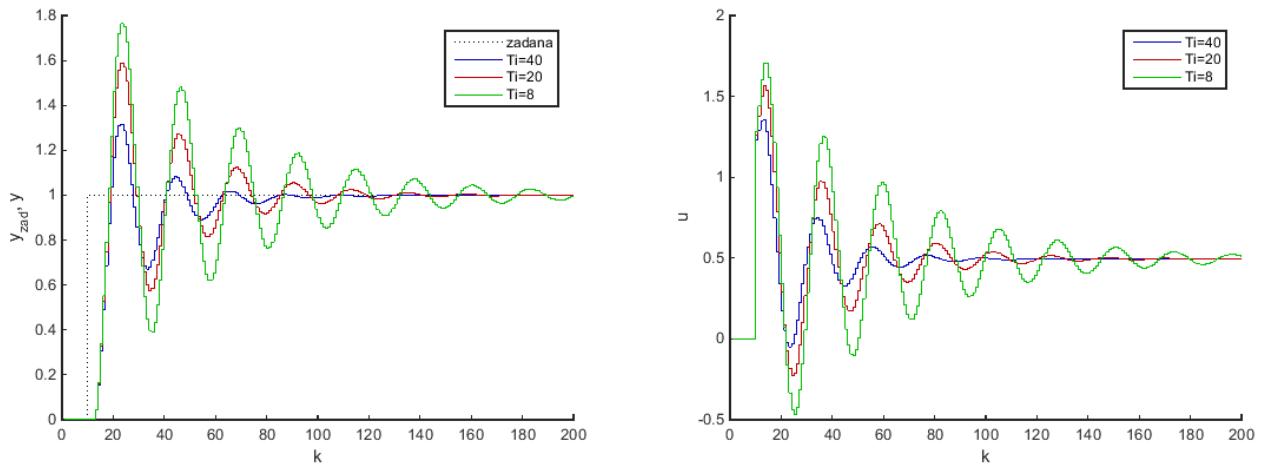
$$T = 2 \quad K = 1,2 \quad T_I = 20 \quad T_D = 5;$$

uwzględniając ewentualne wyłączanie poszczególnych członów.

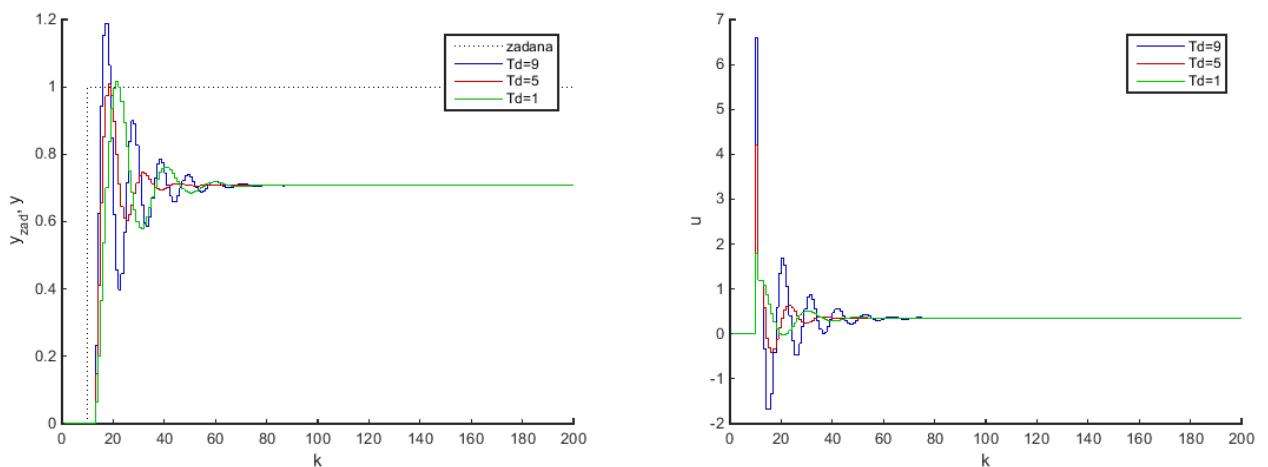
Regulator P (wyłączone człony całkujący i różniczkujący) generuje sygnał sterujący proporcjonalny do różnicy między wartością zadaną a wyjściem procesu regulowanego. Na Rys. 40 widać, że, nawet po bardzo długim czasie, wyjście procesu regulowanego nie jest równe wartości zadanej. Wręcz przeciwnie – różnica między nimi jest stała. Zjawisko takie nazywane jest uchybem ustalonym. Aby zrozumieć to zjawisko warto zastanowić się co działooby się w sytuacji gdyby jednak uchyb był zerowy. Wtedy sterowanie również byłoby zerowe, a co za tym idzie wartość wyjściowa procesu by spadła. To spowodowałoby narastanie uchybu, który powodowałoby narastanie sterowania i hamowanie opadania sygnału wyjściowego. W pewnym momencie nastąpi osiągnięcie równowagi i będzie można ponownie zaobserwować uchyb ustalony. Inną ciekawą cechą jest malejąca wartość uchybu ustalonego wraz ze wzrostem wzmacnienia (nie ma tak dużego wzmacnienia, które wyeliminuje uchyb ustalony!) – niestety jednocześnie wzrasta czas i amplituda oscylacji.

Aby pozbyć się tego zjawiska warto wprowadzić człon całkujący, który spowoduje, że kolejne błędy będą się akumulować i wraz ze wzrostem tego zsumowanego błędu regulator będzie zwiększał wartość sterowania przybliżając się do wartości zadanej. Działanie regulatora PI można zaobserwować na Rys. 41. Mimo zwiększonego czasu oscylowania sygnału wyjściowego – uchyb ustalony został wyeliminowany.

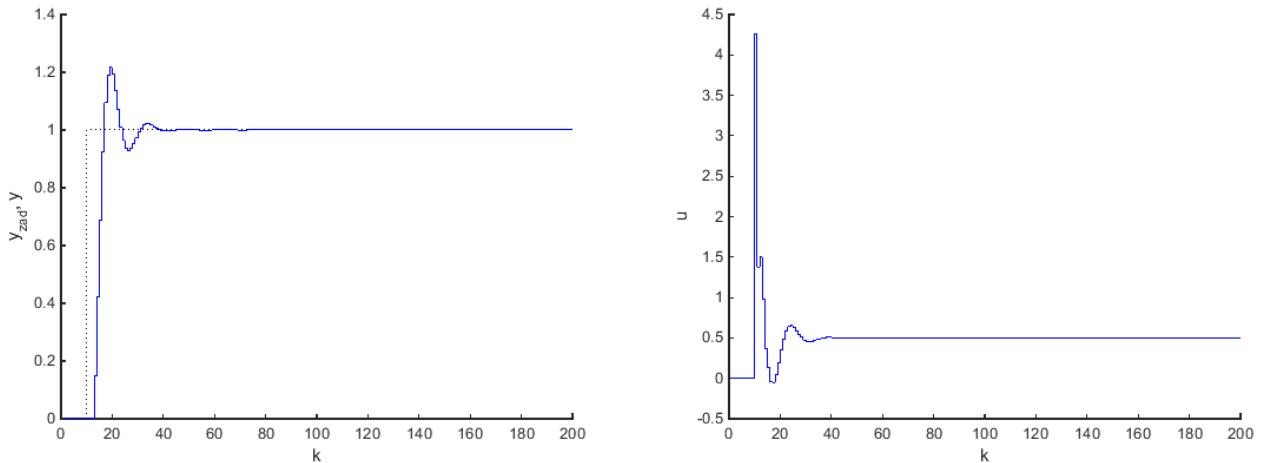
Człon różniczkujący ma za zadanie reagować na zmianę uchybu. Wraz ze wzrostem prędkości jego narastania, wzrasta wartość sterowania. Szczególnie widoczne jest to w przypadku zmiany wartości zadanej, która na Rys. 42 ma miejsce w chwili 20. W tym miejscu na wykresie sygnału sterującego zauważać można bardzo wysoki skok wartości sterowania – jest on spowodowany właśnie przez człon różniczkujący, ponieważ różnica między uchybem obecnym, a uchybem z poprzedniej chwili jest bardzo wysoka. Ponieważ rozważanym reguatorem jest regulator PD, ponownie można zaobserwować uchyb ustalony.



Rysunek 41: Wyjście oraz wejście procesu regulowanego – regulator PI



Rysunek 42: Wyjście oraz wejście procesu regulowanego – regulator PD



Rysunek 43: Wyjście oraz wejście procesu regulowanego – regulator PID będący punktem odniesienia

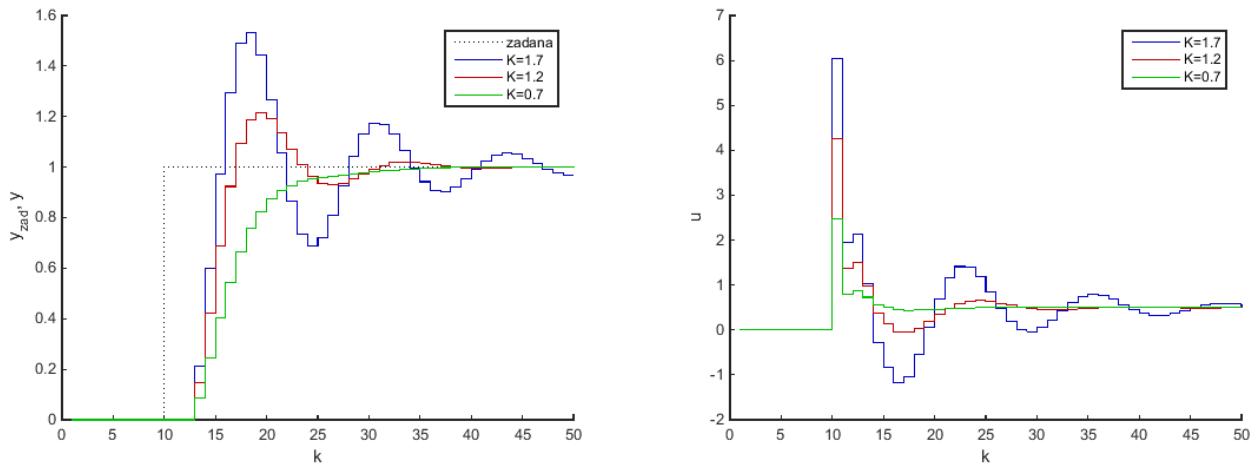
Regulator PID jest złączeniem wszystkich powyższych członów, a więc posiada wszystkie ich cechy. Najważniejszymi są brak uchybu ustalonego oraz chwilowy skok wartości sterowania w momencie zmiany wartości zadanej. Wykres pełnego regulatora PID widoczny jest na Rys. 43.

Na Rys. 44, 45 oraz 46 pokazane są przebiegi sygnałów wejściowego i wyjściowego w zależności od zmiany jednego z parametrów (nastaw) regulatora PID. Dla Rys. 45 pokazany został dłuższy przebieg aby można było zaobserwować powolne dążenie do wartości zadanej takiego algorytmu. Dla pozostałych przebiegów pokazane zostało tylko 100 pierwszych sekund, ponieważ po osiągnięciu wartości zadanej przebiegi szybko się stabilizowały i były do siebie zbliżone na przestrzeni pozostałoego czasu.

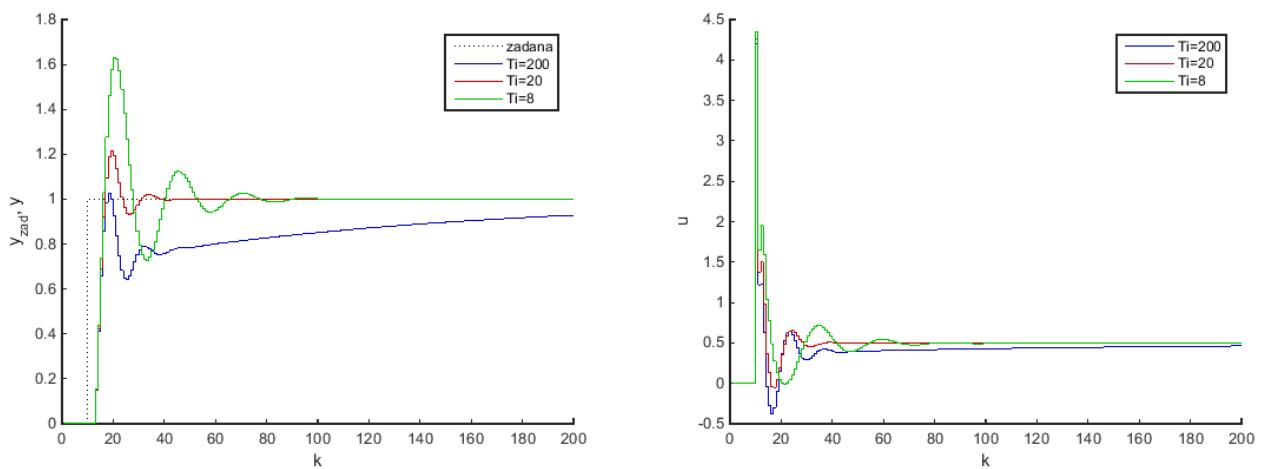
Ponieważ w prawie sterowania algorytmu PID nie ma miejsca na uwzględnienie ograniczeń na wartości sterowania, należy poradzić sobie z tym problemem osobno. Brak uwzględnienia ograniczeń powoduje, że gdy zostanie ono osiągnięte, a błąd będzie niezerowy, realizowane będzie niepotrzebne całkowanie, tj. wyznaczona wartość sygnału sterującego będzie rosła mimo, że faktyczna maksymalna wartość sygnału sterującego już została osiągnięta. Efekty tego zjawiska widoczne są szczególnie w momencie, gdy sygnał wyjściowy przekroczy wartość zadaną – wtedy przez długi czas dokonywane jest „odcałkowywanie”, tj. sygnał sterujący jest nieustannie pomniejszany (przez składową całkującą), aż jego wartości będą mniejsze niż ograniczenie i regulator będzie ponownie pracować zgodnie z oczekiwaniami (pod warunkiem, że w tym momencie jeszcze obiekt nie wpadł w oscylacje). Zjawisko przesadnego całkowania jest nazywane nawijaniem (*windup*) – poniżej omówiony zostanie prosty algorytm pozwalający na niwelację jego wpływu (algorytm *anti-windup*).

Jednym z prostszych algorytmów przeciwdziałających nawijaniu jest pomniejszanie składowej całkującej wartości sterowania o pewną stałą przemnożoną przez różnicę między nasyconą wartością sygnału sterującego, a wartością wyznaczoną przez regulator.

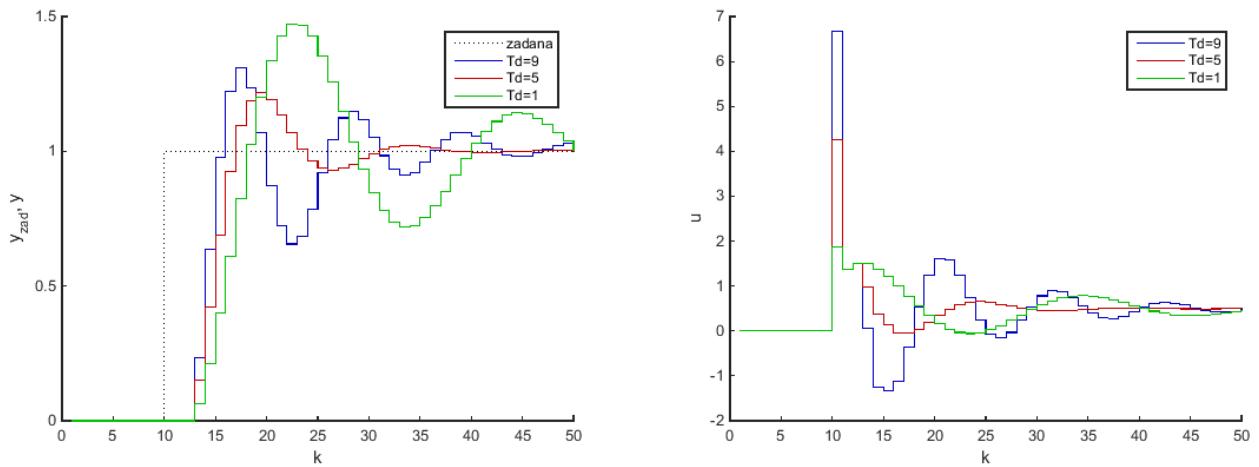
Realizowane jest to poprzez wprowadzenie do członu całkującego dodatkowego elementu proporcjonal-



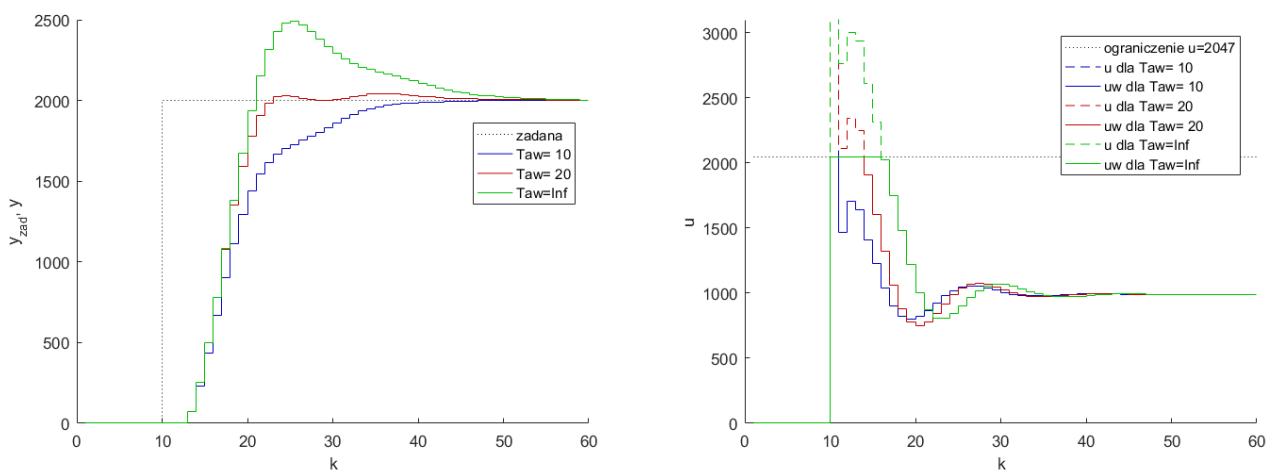
Rysunek 44: Wyjście oraz wejście procesu regulowanego w zależności od wzmocnienia – regulator PID



Rysunek 45: Wyjście oraz wejście procesu regulowanego w zależności od czasu zdwojenia – regulator PID



Rysunek 46: Wyjście oraz wejście procesu regulowanego w zależności od czasu wyprzedzenia – regulator PID



Rysunek 47: Wyjście oraz wejście procesu regulowanego – algorytm PID z oraz bez algorytmu *anti-windup* (pokazane wartości sygnału sterującego ograniczone są do 3100 dla zwiększenia czytelności)

nego (2) do różnicy między faktycznym sygnałem sterującym a oczekiwany

$$u_I(k) = u_I(k-1) + \frac{K}{T_I} T \frac{e(k-1) + e(k)}{2} + \frac{T}{T_v} (u_w(k-1) - u(k-1))$$

gdzie T_v jest parametrem algorytmu *anti-windup*. Sterowanie $u_w(k-1)$ jest to sterowanie, które zostało faktycznie zaaplikowane do procesu, a więc sygnał, który jest ograniczony najczęściej fizycznymi właściwościami obiektu regulowanego. Natomiast $u(k-1)$ oznacza wartość sygnału sterowania, która została wyznaczona poprzez zastosowanie prawa regulacji algorytmu PID. Na Rys. 47 przedstawione zostało porównanie przebiegu sygnału wyjściowego i sterowania przy wykorzystaniu ($T_v = 10$ i $T_v = 20$) i bez zastosowania algorytmu *anti-windup* ($T_v = \infty$). Na wykresach widoczne jest ucięcie sygnału sterującego wynikające z ograniczeń procesu ($u = 2000$) – dla wyłączonego algorytmu *anti-windup* występuje długie przesterowanie wynikające z przekroczenia przez sygnał sterujący wartości ograniczenia. Następuje wtedy ($k = 11$) akumulacja członu całkującego, a następnie dopiero w chwili $k = 13$ następuje odcałkowywanie, które sprowadza sygnał sterujący do dozwolonych wartości po kolejnych dwóch iteracjach. Korzystając z mechanizmu *anti-windup* można skrócić ten czas (w zależności od parametru T_v), co oznacza jednocześnie szybszą reakcję algorytmu regulacji w przypadku osiągnięcia ograniczeń i najczęściej również ograniczenie przesterowania.

Reguły Zieglera-Nicholsa Jedną z metod wyznaczenia parametrów algorytmu PID jest zastosowanie reguł Zieglera-Nicholsa. Ta metoda wyznaczania parametrów regulatora PID nie gwarantuje optymalności rozwiązania, lecz dla klasycznych obiektów regulacji przemysłowej sprawdza się bardzo dobrze jako punkt startowy.

Zastosowanie reguł Zieglera-Nicholsa wymaga przeprowadzenia eksperymentu. Należy zaimplementować regulator typu P dla rozważanego obiektu regulacji. Następnie należy tak dobrać wartość wzmacnienia regulatora K , aby sygnał wyjściowy obiektu regulacji miał charakter oscylacyjny. Trzeba tak dobrać wzmacnienie, aby amplituda oscylacji nie malała i nie rosła – taki układ będzie więc na skraju stabilności. K , które pozwala uzyskać niegasnące i nierosnące oscylacje nazwane zostanie wzmacnieniem krytycznym K_u . Drugim parametrem jaki jest potrzebny do dalszej pracy jest okres drgań w stanie skrajnej stabilności – parametr ten będzie oznaczany jako T_u . Posiadając takie dwie wartości można wyznaczyć parametry dla algorytmów P, PI oraz PID – wystarczy skorzystać z poniższej tabelki

Regulator	K	T_I	T_D
P	$0,5K_u$	–	–
PI	$0,45K_u$	$T_u/1,2$	–
PID	$0,6K_u$	$T_u/2,0$	$T_u/8$

Tak wyznaczone parametry powinny dawać rozsądную jakość regulacji. Niestety reguły te nie gwarantują ani najlepszych wyników, ani nawet poprawnych. Jest to jednak jedna z najstarszych metod wyznaczania nastaw regulatorów PID i jest to metoda przede wszystkim wygodna – nie ma potrzeby tworzenia żadnego modelu procesu. Dodatkowo nawet jeśli te parametry będą niesatysfakcjonujące – jest to dobry zestaw parametrów, aby od niego rozpocząć poszukiwanie takich nastaw, które będą spełniały wszystkie założone wymagania.

Z drugiej strony wprowadzanie obiektu regulacji w stan skrajnej stabilności nie brzmi jak najlepszy pomysł, szczególnie jeśli rozważanym obiektem regulacji jest np. reaktor jądrowy. Oczywiście taki eksperyment może być nie tylko niebezpieczny, ale również szkodliwy dla urządzeń wykonawczych – może nastąpić ich szybsze zużycie lub przesunięcie ich punktu pracy. W ramach projektu wykorzystany został obiekt symulacyjny, w związku z czym użycie tej metody jest jak najbardziej bezpieczne.

Metoda „inżynierska” Inną metodą jest metoda przypominająca zachowaniem metodę minimalizacji gradientowej. Metoda ta będzie podzielona na trzy etapy – dobór parametru K , dobór parametru T_I oraz dobór parametru T_D . Rozpocząć należy od wprowadzenia obiektu w stan skrajnej stabilności przy użyciu regulatora typu P – tak jak w metodzie Zieglera-Nicholsa. Tak wyznaczony parametr K zostanie nazwany wzmacnieniem krytycznym K_u . Należy przejść do doboru nastaw regulatora PI, gdzie $K = 0,5K_u$ natomiast parametr T_I należy dobrać metodą prób i błędów tak, aby uzyskać jak najlepsze rezultaty – oczywiście należy wcześniej przyjąć pewne kryterium oceny, jak np. minimalny czas ustalenia sygnału wyjściowego procesu. Gdy osiągnięty zostanie już satysfakcyjny czas zdwojenia, należy włączyć ostatni człon – różniczkujący. Tak samo jak dla członu całkującego należy metodą prób i błędów dobierać wartości T_D tak, aby zminimalizować wybrany wskaźnik jakości. Osiągnięcie satysfakcyjnych wyników kończy procedurę strojenia.

Algorytm DMC Algorytm DMC (*Dynamic Matrix Control*) jest jednym z najpopularniejszych algorytmów regulacji predykcyjnej. Omówiony zostanie algorytm wyłącznie dla procesu regulacji o jednym wejściu i jednym wyjściu, a do opisu równań będzie głównie wykorzystany zapis macierzowo-wektorowy.

W algorytmie DMC, w każdej kolejnej dyskretnej chwili (iteracji algorytmu) wyznaczany jest wektor przyszłych przyrostów wartości sterowania

$$\Delta U(k) = \begin{bmatrix} \Delta u(k|k) \\ \vdots \\ \Delta u(k + N_u - 1|k) \end{bmatrix} \quad (4)$$

Wektor ten jest długości N_u . Zakłada się, że $\Delta u(k + p|k) = 0$ dla $p \geq N_u$, gdzie N_u jest horyzontem sterowania. Celem działania algorytmu jest minimalizacja różnic między trajektorią zadaną $y^{\text{zad}}(k + p|k)$ (tj. wektorem kolejnych zadanego wartości sygnału wyjściowego) a predykowaną trajektorią sygnału wyjściowego $\hat{y}(k + p|k)$ (tj. wektorem kolejnych prognozowanych wartości) na horyzoncie predykcji N (tj. długość predykowanej trajektorii jest równa N). Rozwiązywane jest zatem następujące zadanie minimalizacji

$$\min_{\Delta U(k)} \left\{ \sum_{p=1}^N (y^{\text{zad}}(k + p|k) - \hat{y}(k + p|k))^2 + \sum_{p=0}^{N_u-1} \lambda_p (\Delta u(k + p|k))^2 \right\}$$

co korzystając z zapisu wektorowo-macierzowego można zapisać jako

$$\min_{\Delta U(k)} \left\{ \|Y^{\text{zad}}(k) - \hat{Y}(k)\|_I^2 + \|\Delta U(k)\|_\Lambda^2 \right\} \quad (5)$$

gdzie $\Lambda > 0$, a $\|x\|_Y^2 = x^T Y x$. Macierz \mathbf{I} jest macierzą identycznościową (tj. diagonalną z samymi jedynkami na diagonali i zerami w pozostałych miejscach) Macierz Λ jest macierzą diagonalną

$$\Lambda = \begin{bmatrix} \lambda_0 & 0 & \cdots & 0 \\ 0 & \lambda_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_{N_u-1} \end{bmatrix}$$

i służy do parametryzowania wpływu poszczególnych czynników na wartość minimalizowanej funkcji. W praktyce często ustala się elementy diagonali macierzy Λ , jako równe sobie i oznacza jako λ (tj. $\lambda_0 = \lambda_1 = \dots = \lambda_{N_u-1} = \lambda$), a więc $\Lambda = \lambda \mathbf{I}$. Wektory $Y^{\text{zad}}(k)$ oraz $Y(k)$ mają postać

$$Y^{\text{zad}}(k) = \begin{bmatrix} y^{\text{zad}}(k) \\ \vdots \\ y^{\text{zad}}(k) \end{bmatrix}, \quad Y(k) = \begin{bmatrix} y(k) \\ \vdots \\ y(k) \end{bmatrix}$$

Wektory $Y^0(k)$, $Y^{\text{zad}}(k)$ oraz $Y(k)$ są długości N .

Mimo, że wyznaczony wektor minimalizujący wartość wyżej przedstawionej funkcji zawiera przyrosty sterowania na iteracje od 0 do $N_u - 1$ w przód, to aplikowany do procesu jest jedynie przyrost wartości sygnału wyznaczony na chwilę obecną, tj. $\Delta u(k|k)$. Ponieważ wyznaczone zostały przyrosty, należy wyznaczoną wartość obecnego sterowania dodać do ostatniej wartości sygnału sterowania, tj. $u(k) = \Delta u(k|k) + u(k-1)$. W następnej iteracji algorytmu regulacji, aktualniane są pomiary, a następnie ponownie wyznaczany jest cały optymalny wektor $\Delta U(k)$ i aplikowany jest jedynie pierwszy element.

Prognozy wartości sygnału wyjściowego $\hat{y}(k+p|k)$ na horyzoncie predykcji N obliczane są na podstawie modelu w postaci odpowiedzi skokowej procesu. Odpowiedź skokowa, jest to seria kolejnych wartości sygnału wyjściowego procesu będących reakcją na nagłą, jednostkową zmianę wartości sterowania. A więc jeśli zmiana sygnału sterującego nastąpiła w chwili k , to odpowiedź skokowa będzie się składała z pomiarów $\{s_1, s_2, s_3, \dots\} = \{\Delta y(k+1), \Delta y(k+2), \Delta y(k+3), \dots\}$, gdzie $\Delta y(k+p)$ dla $p = 1, 2, \dots$ oznacza przyrost wartości sygnału wejściowego w stosunku do stanu sprzed zmiany wartości sterowania. Ponieważ algorytm DMC jest przeznaczony dla procesów asymptotycznie stabilnych, można zapisać tylko pewną liczbę początkowych wartości odpowiedzi skokowej przyjmując założenie, że dalsze elementy miałyby wartość tą samą lub zbliżoną co ostatni element odpowiedzi skokowej. Wprowadzić więc wartość wielkość D – horyzont dynamiki, taką, że $s_l = s_D$ dla $l \geq D$.

Jak widać eksperyment służący do pozyskania odpowiedzi skokowej jest wyjątkowo prosty, a jego linowość pozwala na wyznaczenie analitycznego prawa regulacji (pod warunkiem nie uwzględniania ograniczeń). Uwzględnienie ograniczeń w takim wypadku może być realizowane dopiero po wyznaczeniu nowych wartości sterowania poprzez ich rzutowanie na zbiór dozwolonych wartości. Szczegóły zostaną omówione niżej.

Wektor będący rozwiązaniem zadania (5) można wyznaczyć wprost jako

$$\Delta U(k) = \mathbf{K} [Y^{\text{zad}}(k) - Y^0(k)] \quad (6)$$

Macierz \mathbf{K} (o wymiarach $N_u \times N$) zdefiniowana jest jako

$$\mathbf{K} = [\mathbf{M}^T \mathbf{M} + \lambda \mathbf{I}]^{-1} \mathbf{M}^T = \begin{bmatrix} \overline{\mathbf{K}}_1 \\ \overline{\mathbf{K}}_2 \\ \vdots \\ \overline{\mathbf{K}}_{N_u} \end{bmatrix} = \begin{bmatrix} \mathbf{K}_{1,1} & \mathbf{K}_{1,2} & \dots & \mathbf{K}_{1,N} \\ \mathbf{K}_{2,1} & \mathbf{K}_{2,2} & \dots & \mathbf{K}_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{K}_{N_u,1} & \mathbf{K}_{N_u,2} & \dots & \mathbf{K}_{N_u,N} \end{bmatrix}$$

gdzie trajektoria swobodna $Y^0(k)$ obliczana jest jako $Y^0(k) = Y(k) + \mathbf{M}^P \Delta U^P(k)$. Do wyznaczenia macierzy \mathbf{K} wymagana jest znajomość macierzy \mathbf{M} (zwanej macierzą współczynników odpowiedzi skokowej)

$$\mathbf{M} = \begin{bmatrix} s_1 & 0 & \dots & 0 \\ s_2 & s_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ s_N & s_{N-1} & \dots & s_{N-N_u+1} \end{bmatrix}$$

Macierz \mathbf{M} ma wymiary $N \times N_u$.

Macierze służące do wyznaczenia trajektorii swobodnej $Y^0(k)$ określone są jako

$$\mathbf{M}^P = \begin{bmatrix} s_2 - s_1 & \dots & s_D - s_{D-1} \\ s_3 - s_1 & \dots & s_{D+1} - s_{D-1} \\ \vdots & \ddots & \vdots \\ s_{N+1} - s_1 & \dots & s_{N+D-1} - s_{D-1} \end{bmatrix}, \quad \Delta U^P(k) = \begin{bmatrix} \Delta u(k-1) \\ \vdots \\ \Delta u(k-(D-1)) \end{bmatrix}$$

Macierz \mathbf{M}^P ma wymiary $N \times (D-1)$, natomiast wektor $\Delta U^P(k)$ jest długości D-1.

Na podstawie powyższych wzorów można wyznaczyć prawo regulacji, tj. wyznaczony zostanie wyłącznie najbliższy przyrost sterowania (tj. $\Delta u(k|k)$), który od razu posłuży do wyznaczenia wartości przyszłego sterowania $u(k|k)$

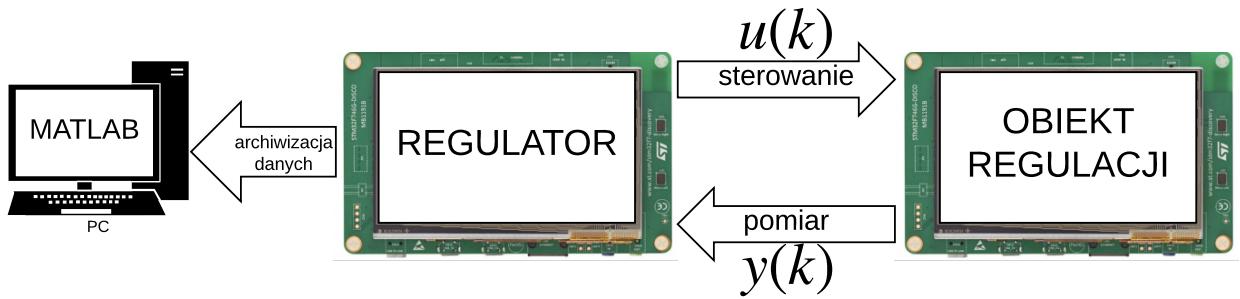
$$\begin{aligned} u(k|k) &= u(k-1) + \Delta u(k|k) = u(k-1) + \overline{\mathbf{K}}_1 [Y^{\text{zad}}(k) - Y^0(k)] \\ &= u(k-1) + \overline{\mathbf{K}}_1 [Y^{\text{zad}}(k) - Y(k) - \mathbf{M}^P \Delta U^P(k)] \\ &= u(k-1) + \overline{\mathbf{K}}_1 [Y^{\text{zad}}(k) - Y(k)] - \overline{\mathbf{K}}_1 \mathbf{M}^P \Delta U^P(k) \\ &= u(k-1) + \sum_{i=1}^N \mathbf{K}_{1,i} (y^{\text{zad}}(k) - y(k)) - \overline{\mathbf{K}}_1 \mathbf{M}^P \Delta U^P(k) \end{aligned}$$

gdzie po zdefiniowaniu symboli $e(k) = y^{\text{zad}}(k) - y(k)$, $K_e = \sum_{i=1}^N \mathbf{K}_{1,i}$ i wektora o długości $D-1$ $\mathbf{K}_u = \overline{\mathbf{K}}_1 \mathbf{M}^P$ można zapisać

$$u(k|k) = u(k-1) + K_e e(k) - \mathbf{K}_u \Delta U^P(k)$$

W tym momencie można dokonać rzutowania analitycznie wyznaczonego sterowania na zbiór dopuszczalnych rozwiązań. Realizowane jest to poprzez implementację poniższych warunków

$$\begin{aligned} &\text{jeżeli } u(k|k) < u^{\min} \text{ wtedy } u(k|k) = u^{\min} \\ &\text{jeżeli } u(k|k) > u^{\max} \text{ wtedy } u(k|k) = u^{\max} \\ &u(k) = u(k|k) \end{aligned}$$



Rysunek 48: Schemat połączenia obiektu symulowanego i regulatora

Symulowany obiekt Obiekt regulacji jest symulowany przy użyciu płytka rozwojowej STM32F746G-DISCOVERY. Jest ona tak samo wyposażona jak płytka działająca jako regulator, którą programować będzie student. Schemat połączenia jest przedstawiony na Rys. 48. Obiekt ten jest liniowy, asymptotycznie stabilny i posiada dwie incercje. Dokładne stałe czasowe oraz wzmacnienie, które opisują ten obiekt nie będą podawane, gdyż inaczej zadanie identyfikacji (tj. pozyskiwanie modelu w postaci odpowiedzi skokowej) miałoby się z celem (znając równania obiektu można by taką odpowiedź wygenerować).

Sterowanie obiektem odbywa się poprzez zmianę wartości napięcia (w zakresie od 0 do 3,3 V) na jednym z jego pinów. Wyjściem obiektu jest również wartość napięcia (także w zakresie od 0 do 3,3 V), które można zmierzyć na jednym z jego pinów. Na wyświetlaczu obiektu rysowane są na bieżąco wartości sygnału wyjściowego obiektu (Y_1) oraz sygnału wejściowego obiektu, tj. sterowania (U_1). Wartości te są skalowane do przedziału od -1 do 1, gdzie -1 odpowiada napięciu równemu 0 V (na wejściu lub wyjściu obiektu), natomiast 1 odpowiada napięciu 3,3 V (zarówno na wejściu jak i wyjściu obiektu). Połączenie obiektu z regulatorem (tj. dwóch płyt STM32F746G-DISCOVERY) realizowane jest przez prowadzącego.

Płytki z przetwornikami Do sterowania obiektem regulacji wykorzystana została płytka zawierająca przetworniki cyfrowo-analogowe (DAC – *Digital-Analog Converter*). Mikrokontroler STM32F746G-DISCOVERY wyposażony jest we własne DAC, lecz zostały już one wykorzystane do generacji dźwięku stereo (są bezpośrednio podłączone do odpowiednich elementów na płytce rozwojowej), co powoduje niemożność ich wykorzystania do innych zadań. Wspomniane przetworniki są dołączane do płytki rozwojowej poprzez płytkę wyposażoną w złącza zgodne ze złączem Arduino Uno.

Przetworniki te, o nazwie MCP4725A0T-E/CH, są 12-bitowe (tak jak te co są na płytce rozwojowej) a komunikacja z nimi odbywa się przy użyciu protokołu I²C. Funkcja służąca do wysłania nowej wartości cyfrowej do przetwornika wygląda następująco

```
void updateControlSignalValue(uint16_t out){
static uint8_t buffertx[] = {0x0F, 0xFF};
buffertx[0] = (out>>8)&0x0F;
buffertx[1] = (out>>0)&0xFF;
HAL_I2C_Master_Transmit(&hi2c1, 0xC2, (uint8_t*)buffertx, 2,1000);
}
```

Kolejno wartość `out` zapisywana jest na dwóch osobnych bajtach, po czym przesyłana jest do przetwornika za pomocą funkcji z biblioteki HAL. Nie zostało wykorzystane DMA ani przerwania, lecz jednokierunkowa komunikacja oraz znakomy czas wykonania są doskonałymi argumentami, aby dopuścić się takiego zabiegu.

Komunikacja na przykładzie pozyskiwania odpowiedzi skokowej Komunikacja z komputerem następuje przy użyciu kabla USB, który równocześnie służy do programowania oraz zasilania mikrokontrolera. Jest to możliwe dzięki oprogramowaniu zawartym na programatorze ST-LINK/V2-1 (zaimplementowanym na mikrokontrolerze STM32F103CBT6), który po zakończeniu programowania mikrokontrolera może być wykorzystany jako *Virtual Com Port*, z czego właśnie korzystamy. Do komunikacji została wykorzystana prędkość 115200, 8 bitów na treść, brak bitów parzystości oraz 1 bit stopu. Dodatkowo należy pamiętać o tym, aby nie korzystać ze sprzętowej kontroli przepływu.

Poniżej znajduje się **przykładowy** kod w języku MATLAB, który pozwala na wyświetlanie tego, co wysłał mikrokontroler:

```
delete(instrfindall); % zamkniecie wszystkich polaczen szeregowych
clear all;
close all;
s = serial('COM9'); % COM9 to jest port utworzony przez mikrokontroler
set(s,'BaudRate',115200);
set(s,'StopBits',1);
set(s,'Parity','none');
set(s,'DataBits',8);
set(s,'Timeout',1);
set(s,'InputBufferSize',1000);
set(s,'Terminator',13);
fopen(s); % otwarcie kanalu komunikacyjnego

y(1:100) = -1; % mikrokontroler zwraca tylko dodatnie wartosci, wiec kazde -1
% oznacza, ze nie otrzymalismy jeszcze wartosci o tym indeksie
while true
    txt = fread(s,16); % odczytanie z portu szeregowego
    eval(char(txt'));% wykonajmy to co otrzymalismy
    if(y(1) ~= -1)
        break;
    end
end
y(2:100) = -1; % ignorujemy wszystko co odczytalismy poza pierwszym elementem
while true
    txt = fread(s,16); % odczytanie z portu szeregowego
    eval(char(txt'));% wykonajmy to co otrzymalismy
    if(min(y) ~= -1) % jesli najmniejszym elementem nie jest -1, to znaczy ze
        break;          % nie ma brakujacych elementow, a wiec dane sa gotowe
```

```

    end
end
figure;
plot(0:length(y),y); % w tym momencie mozna juz wyrysowac dane

```

W kodzie tym zakładamy, że mikrokontroler przesyła wiadomości o treści "y(%3d) = %4d;\n\r", gdzie pierwszą wartością w tym formacie jest numer chwili począwszy od zmiany wartości sterowania, której dotyczy pomiar, natomiast druga wartość to właśnie pomiar bezpośrednio odczytany z ADC (a więc o wartościach od 0 do 4095). Założone zostało, że po uruchomieniu mikrokontroler wysyła do DAC wartość 1500, następnie po oczekaniu 1s (aby wyjście obiektu zdążyło się ustabilizować) do DAC wysyłana jest wartość 2500 a kolejne pomiary przesyłane są do komputera zgodnie z powyższym formatem. Po przesłaniu 100 kolejnych wartości program rozpoczyna działanie od początku. W ten sposób udało się uzyskać serię pomiarów o długości 100.

Tutaj warto zwrócić uwagę na kwestię komunikacji mikrokontrolera z komputerem – czy uruchomić najpierw skrypt MATLAB-a czy mikrokontroler jest bardzo ważnym pytaniem, które należy sobie zadać. Jeśli mikrokontroler chodzi w pętli, bez przerwy wykonując pewne operacje i wysyłający wiadomości, i uruchomiony zostanie skrypt je odbierający, pojawią się pewne ryzyko. Otóż odczytana może zostać nie pierwsza wiadomość, którą planowaliśmy odebrać, a jedna „ze środka”. W powyższym skrypcie przyjęte zostało, że właśnie taka sytuacja ma miejsce. W skrypcie tym najpierw oczekujemy na konkretną wiadomość (dokładniej, tę, która zmieni wartość elementowi $y(1)$, a następnie wykonywane jest odbieranie kolejnych wiadomości. Wynika to z faktu, iż skrypt z programem mikrokontrolera zostały zsynchronizowane, tj. od tej pory wiemy jakie wiadomości kolejno będą wysyłane.

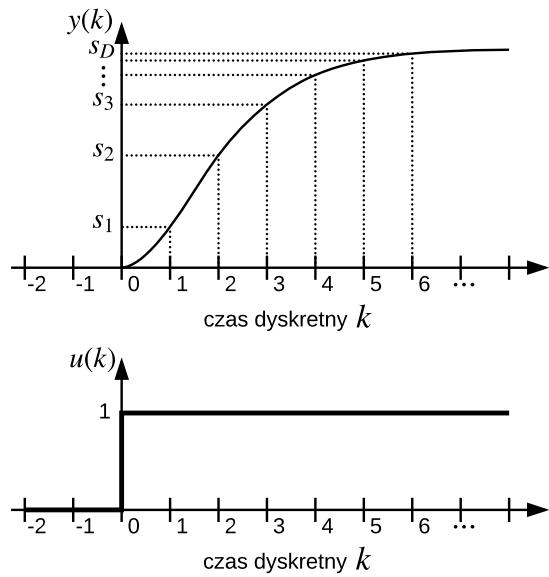
Taka odpowiedź skokowa jest jeszcze nieużyteczna – wymaga ona przeskalowania. W algorytmie DMC korzystamy z odpowiedzi będącej reakcją na **jednostkową** zmianę wartości sygnału sterowania (tj. zmianę z 0 na 1). Należy więc sprowadzić posiadaną odpowiedź do takiej postaci poprzez pomniejszenie każdego jej elementu o wartość pomiaru wyjścia obiektu regulowanego w stanie przed skokiem wartości sterowania. Zależność między czasem aplikacji nowej wartości sterowania a chwilą pomiaru wyjścia obiektu regulacji przedstawia Rys. 49.

Skalowanie pomiarów tak, aby uzyskać odpowiedź skokową polega na odjęciu od wszystkich pomiarów wartości pomiaru wyjścia obiektu regulowanego przed zmianą wartości sterowania i podzieleniu wynikowych elementów przez przyrost sterowania. W powyższym przykładzie wartości odczytane z mikrokontrolera zostały zapisane w wektorze y , gdzie wartość $y(1)$ odpowiada pomiarowi w chwili zmiany wartości sterowania. Oznacza to, że przejście z pomiarów na odpowiedź skokową wymaga następującego przekształcenia w MATLAB-ie:

```
s=(y(2:100)-y(1))/1000; % przeskalone pomiary = jednostkowa odpowiedz skokowa
```

Przy czym należy pamiętać, że element s_1 jest to przyrost wartości wyjściowej obiektu regulacji w **następnej** chwili po zmianie wartości sterowania (wyraźnie to widać na Rys.49). Z tego właśnie powodu wykorzystany został wektor $y(2:100)$ a nie $y(1:100)$.

UWAGA! Tak uzyskana odpowiedź skokowa jest odpowiednia tylko i wyłącznie wtedy, gdy czas oblicze-



Rysunek 49: Schemat zależności czasowych między zmianą sygnału sterowania a pomiarem wyjścia obiektu regulowanego

czeń algorytmu DMC jest znikomy w stosunku do czasu między wykonaniem kolejnych pomiarów (a co za tym idzie między kolejnymi iteracjami algorytmu DMC). Jeśli czas ten nie jest wystarczająco krótki i wprowadzone zostało sztuczne opóźnienie aplikacji do procesu nowej wartości sygnału sterującego, także i odpowiedź skokowa należy opóźnić o jeden takt. Sprowadza się to do dodania jednego zera na początku wektora odpowiedzi skokowych. Bez tego obiekt oraz model nie będą ze sobą spójne, co będzie powodować niepoprawne działanie algorytmu. W zależności od tego czy zostało zastosowane sztuczne opóźnienie uzyskana odpowiedź skokowa powinna rozpoczynać się od wartości różnej od zera (w przypadku braku opóźnienia) lub równej zero (w przypadku jego zastosowania).

Komunikacja na przykładzie nieustannego odczytu pomiarów Aby przekazać pomiar sygnału wyjściowego obiektu regulowanego oraz wartość sterowania do komputera, warto rozważyć przesyłanie wiadomości o treści np. "Y=%4d;U=%4d;". Gdzie oczywiście pierwszą liczbą w tym formacie jest odczyt wartości sygnału wyjściowego obiektu regulacji, natomiast drugim jest wartość sygnału sterującego. Taka wiadomość przesyłana była po każdej iteracji algorytmu regulacji. Ponieważ algorytm regulacji działać będzie w pętli nieskończonej – nie jest ważne, którą wiadomość odbierzemy jako pierwszą. W związku z tym kod skryptu, w którym dokonywane będzie zbieranie danych wygląda następująco:

```
delete(instrfindall); % zamkniecie wszystkich polaczen szeregowych
clear all;
close all;
s = serial('COM9'); % COM9 to jest port utworzony przez mikrokontroler
```

```

set(s,'BaudRate',115200);
set(s,'StopBits',1);
set(s,'Parity','none');
set(s,'DataBits',8);
set(s,'Timeout',1);
set(s,'InputBufferSize',1000);
set(s,'Terminator',13);
fopen(s); % otwarcie kanalu komunikacyjnego

Tp = 0.01; % czas z jakim probkuje regulator
y = []; % wektor wyjsc obiektu
u = []; % wektor wejsc (sterowan) obiektu
while length(y)~=100 % zbieramy 100 pomiarow
    txt = fread(s,14); % odczytanie z portu szeregowego
                % txt powinien zawierać Y=%4d;U=%4d;
                % czyli np. Y=1234;U=3232;
    eval(char(txt'));% wykonajmy to co otrzymalismy
    y=[y;Y]; % powiekszamy wektor y o element Y
    u=[u;U]; % powiekszamy wektor u o element U
end

figure; plot((0:(length(y)-1))*Tp,y); % wyswietlamy y w czasie
figure; plot((0:(length(u)-1))*Tp,u); % wyswietlamy u w czasie

```

Czyli jak widać utworzone przez wywołanie funkcji `eval` zmienne `Y` oraz `U` są dodawane do odpowiednich tablic zgodnie z zamysłem użytkownika skryptu. Poprzedni skrypt zawierał już informację o tym, który element wektora `y` należy uzupełnić odczytem z ADC, co pozwalało na synchronizację. Ponieważ tutaj synchronizacja nie jest potrzebna/konieczna – każdą wiadomość traktujemy identycznie.

5.1 Zadanie do wykonania

Zadaniem, które będzie podlegało późniejszej ocenie jest implementacja na mikrokontrolerze rodzinny STM32 dwóch algorytmów regulacji. Pierwszym jest algorytm regulacji PID (*Proportional-Integral-Derivative*), drugim DMC (*Dynamic Matrix Control*) w wersji analitycznej. Zrealizowana implementacja będzie podstawą do realizacji następnego projektu, tak więc musi być pozbawiona wszelkich błędów.

Projekt kończy się sprawozdaniem, w którym należy poruszyć następujące kwestie:

- Algorytm PID:
 - Omówienie implementacji.
 - Porównanie trajektorii sygnału wyjściowego procesu regulowanego przy wyznaczeniu nastaw metodą Zieglera-Nicholsa.

- Porównanie trajektorii sygnału wyjściowego procesu regulowanego przy wyznaczeniu nastaw metodą „inżynierską”.
- Porównanie wyników obu powyższych metod.
- Porównanie trajektorii sygnału wyjściowego procesu regulowanego w zależności od parametru T_v (tj. parametru związanego z algorytmem *anti-windup*)
- Algorytm DMC:
 - Omówienie implementacji.
 - Porównanie trajektorii sygnału wyjściowego procesu regulowanego przy zastosowaniu różnych wartości horyzontu predykcji (i możliwe duży horyzont sterowania).
 - Porównanie trajektorii sygnału wyjściowego procesu regulowanego przy zastosowaniu różnych wartości horyzontu sterowania.
 - Porównanie trajektorii sygnału wyjściowego procesu regulowanego przy zastosowaniu różnych wartości Λ .
- Porównanie najlepszej realizacji algorytmu PID i DMC pod kątem:
 - odporności na zakłóczenia (realizowanego poprzez wcisnięcie klawisza na płycie z obiektem symulowanym),
 - przesterowania,
 - czasu ustalenia,
 - oscylacji (ocena „na oko”).

Określenie „omówienie” oznacza w powyższym tekście zarówno opis przeprowadzonych eksperymentów, jak i wnioski wynikające z wyników – mogą być one zaskakująco różne od oczekiwanych. Warto w takiej sytuacji zastanowić się co może być przyczyną i również to opisać. W razie większych wątpliwości – warto skonsultować się z prowadzącym. Należy sformułować także wnioski, które wydają się oczywiste lub w szczególności „widoczne z rysunku”. Zadanie dotyczące „implementacji”, wymaga także omówienia tej implementacji. Uwagi te mają zastosowanie do wszystkich sprawozdań.

5.2 Sugerowany przebieg projektu

Sugerowanym podejściem do projektu jest implementacja na pierwszych zajęciach algorytmu PID oraz akwizycja odpowiedzi skokowej. Między zajęciami warto zaimplementować w środowisku MATLAB (a najlepiej w C) algorytm DMC, aby na drugie zajęcia przyjść już z gotowym, działającym i przetestowanym kodem. Jest to również dobry czas na zaplanowanie sprawozdania i wykonanych do niego eksperymentów. Drugie zajęcia warto spędzić na pozyskiwaniu przebiegów eksperymentów (dla PID zmiany członów K , T_i , T_D , T_v , a dla DMC zmiany λ , N , N_u , reakcja na zakłócenie wyjścia itp.). Sprawozdanie warto dokończyć po drugich zajęciach.

Powyższa kolejność jest wyłącznie sugestią – poczynania studentów w ramach projektu są uzależnione tylko i wyłącznie od decyzji uczestników projektu.

6 Identyfikacja modeli (typu odpowiedzi skokowej) procesu laboratoryjnego, implementacja algorytmu regulacji DMC, dobór nastaw algorytmu, badania porównawcze

Celem tego projektu jest implementacja algorytmu regulacji DMC z użyciem obiektu rzeczywistego (tj. takiego, którego sygnały mają pewną reprezentację fizyczną). Wykorzystane zostanie stanowisko grzejaco-chłodzące wprowadzone w poprzednich ćwiczeniach. Ponieważ sama implementacja algorytmu była już przerabiana, dla urozmaicenia przebiegu ćwiczenia zaprojektować oraz zaimplementować należy interfejs użytkownika regulatora oraz obsłużyć wszelkie wykrywalne stany awaryjne obiektu regulacji.

Stanowisko grzejaco-chłodzące Stanowisko to zostało już omówione w kontekście poprzednich ćwiczeń. W stosunku do zawartych tam informacji warto wiedzieć o kilku dodatkowych aspektach tego obiektu.

Obiekt ten działa z okresem 1 s – oznacza to, że raz na sekundę dokonuje on aplikacji wartości sterowania zapisanej w jego pamięci oraz wykonuje odczyt pomiarów, zapisując wyniki do pamięci. Wszelkie operacje wykonywane na tym obiekcie dokonują zmian w pamięci – stąd czasem widoczne jest opóźnienie między wysłaniem wiadomości modyfikującej wartość sterowania, a jej przełożeniem na rzeczywistą zmianę stanu urządzenia. Oznacza to, że nie ma potrzeby odpytywać o pomiary i przesyłać nowej wartości sygnału sterującego do obiektu częściej niż raz na sekundę. Taki jest również oczekiwany okres interwencji regulatora (tj. wyznacza on nową wartość sygnału sterującego co sekundę).

Rozważane stanowisko przesyła wartości temperatury ze skońzoną dokładnością. Oznacza to, że jej odczyt będzie przyjmować pewne dyskretne wartości. Wynika to z charakteru zastosowanych (12-bitowych) czujników oraz ze sposobu przekazywania wartości pomiarów poprzez protokół MODBUS – są one bowiem przekazywane w setnych częściach stopni Celsjusza.

Stany awaryjne Rozważane stanowisko posiada dwa kluczowe stany awaryjne, które mogą zostać w łatwy sposób wykryte i jednoznacznie zidentyfikowane. Te dwa aspekty należy wyraźnie rozróżnić, gdyż świadomość, że coś nie działa nie oznacza niestety, że wiadomo co jest tego przyczyną.

Pierwszym stanem awaryjnym jest błąd wykonania pomiaru. W sytuacji, gdy czujnik zostanie uszkodzony lub z jakiegoś powodu odłączony (np. zostanie zjedzony w całości przez chomika), mikrokontroler zarządzający stanowiskiem zgłasza zamiast poprawnego odczytu wartość spoza zakresu pomiarowego tego czujnika (tj. od -55°C do 125°C). Tak spreparowany pomiar pozwala na jednoznaczne wskazanie czujnika, który został uszkodzony/odłączony. Symulacja tego stanu następuje poprzez ustawienie przełącznika w prawym położeniu („9999”). Powoduje to fizyczne odłączenie czujnika od płyty mikrokontrolera zarządzającego, który z kolei informuje o tym fakcie poprzez ustawienie wartości pomiaru spoza zakresu pomiarowego czujnika. Przełączniki odpowiadają czujnikom od T1 do T4, czujnik T5 z założenia jest niezniszczalny.

Drugim kluczowym stanem awaryjnym jest błąd komunikacji. Korzystając z protokołu MODBUS można zauważać, że każda wiadomość wysłana z urządzenia typu *master* musi spotkać się z odpowiedzią od urządzenia typu *slave*. Jeśli taka odpowiedź została uzyskana to należy sprawdzić, czy jest ona poprawna. Jeśli również i to jest prawdą, to można dokonać analizy treści odpowiedzi i przejść do dalszego wykonywania programu. Błędy wynikające z niepoprawnie sformułowanej wiadomości wychodzącej od urządzenia typu *master* obejmują takie sytuacje jak nieobsługiwana przez urządzenie typu *slave* funkcja, błędny zakres

adresów, niepoprawna wartość, błędne CRC czy ostatecznie błąd wykonania funkcji. Wymienione błędy nie powinny pojawić się w przypadku poprawnie napisanego programu regulatora (tj. urządzenia typu *master*). Wyjątkiem mogłyby być ostatnia z wymienionych sytuacji nieprzyjemnych – tj. błąd wykonania funkcji – lecz wspomniane stanowisko grzejaco-chłodzące nie przewiduje sytuacji, w której nie może dojść do poprawnego zakończenia obsługi poprawnej wiadomości.

Powyższe zachodzi pod warunkiem, że komunikacja została nawiązana i jest utrzymywana – nie zawsze to musi mieć miejsce. W przypadku utraty fizycznego połączenia między dwoma urządzeniami, tj. gdy kabel do tego służący zostanie odpięty lub zerwany, dojdzie do dość oczywistej sytuacji. Gdy urządzenie typu *master* wyśle wiadomość, nie doczeka się na nią odpowiedzi. Wynika to z faktu, iż po wysłaniu zapytania, urządzenie typu *master* oczekuje przez skończony czas na uzyskanie odpowiedzi – urządzenie typu *slave* musi w tym czasie otrzymaną wiadomość odczytać, sparsować, obsłużyć, skonstruować nową wiadomość i wysłać ją do nadawcy. W rozważanym urządzeniu czas ten jest rzędu 250-300 ms. Jeśli w tym czasie nie zostanie uzyskana odpowiedź, możliwości jest kilka: urządzenie typu *slave* zawiesiło się, zostało uszkodzone, skradzione lub pojawił się problem z kablem służącym do komunikacji (uszkodzony, nieobecny, obecny w połowie). Finalny efekt jest jednak jeden – brak kontaktu z urządzeniem typu *slave*. Najlepsze co w takiej sytuacji można uczynić to możliwie sprawnie zasygnalizować problem z komunikacją operatorowi regulatora, aby ten zajął się dalszymi poszukiwaniemi przyczyny.

Istnieje dodatkowa sytuacja awaryjna, która może mieć miejsce. Przy zbyt długim i intensywnym wykorzystaniu grzałek może dojść do przegrzania stanowiska oraz jego uszkodzenia. Dlatego też wprowadzone zostały elementy mające temu zapobiec – wyłączniki termiczne. Zamontowane są one na wierzchu grzałek i ich zadaniem jest odłączenie zasilania od grzałek w sytuacji przekroczenia pewnej temperatury (w rozważanych wyłącznikach jest to 90°C). Ponieważ wyłączniki te są zamontowane w tym miejscu, przy czujnikach temperatury odczyty nie będą sięgały aż 90°. Takie sprzętowe zabezpieczenie odciąża projektanta regulatora od obowiązku zabezpieczenia układu przed przegrzaniem, lecz z drugiej strony projektant nie otrzymuje żadnej informacji, iż taka sytuacja ma miejsce. Istnieją wprawdzie metody do detekcji uszkodzeń elementów wykonawczych, lecz w ramach tego ćwiczenia nie będą one implementowane – należy więc zaimplementować obsługę wyłącznie tych sytuacji awaryjnych, których występowanie może zostać jednoznacznie wykryte.

Wizualizacja W poprzednim projekcie płytka odpowiadająca za symulację procesu regulacji jednocześnie wizualizowała jego stan. W tym projekcie wizualizacja ma mieć miejsce na płytce realizującej zadanie regulatora. Metody wyświetlania poszczególnych wartości na ekranie zostały omówione przy okazji poprzednich ćwiczeń – nie będą więc powtarzane.

Wizualizacja powinna cechować się zarówno prostotą jak i precyzyjnością. Z jednej strony szczegółowe rysunków nie powinny zaciemniać obrazu całego procesu, a z drugiej muszą przekazywać wszystkie potrzebne dla operatora informacje. Stąd też systemy SCADA (*Supervisory Control And Data Acquisition*) nie pokazują obiektów z wysoką dokładnością (tj. nie są to obrazki o dokładności zdjęć), a ograniczają się do schematycznych kształtów, które pozwalają możliwie szybko odnaleźć interesującą operatora wielkość i nawet bez odczytywania określić jej przybliżoną wartość. Oznacza to więc, że należy pokazać operatorowi zarówno dokładną wartość pewnej wielkości, jak i jej graficzną reprezentację.

Dodatkowym elementem, dzięki któremu powstający regulator coraz bardziej naśladuje system SCADA, jest alarmowanie. W sytuacji, gdy jeszcze nie wystąpiło poważne zagrożenie, ale zdecydowanie proces znaj-

duje się za daleko od bezpiecznego punktu pracy warto zgłosić alarm. Alarm można głosić na podstawie czujnika temperatury T1, ale można również do tego celu użyć czujnik temperatury T5 (adres 4), który służy do odczytu wartości temperatury otoczenia. Ponieważ wiatraki chłodzą stanowisko wyłącznie powietrzem uzyskanym z otoczenia, to przy temperaturze T1 równej T5 można zgłosić alarm, iż stanowisko osiągnęło dolną granicę temperatury, jaką jest w stanie uzyskać. Przy takim wykorzystaniu czujnika T5 warto tę temperaturę również pokazać na ekranie regulatora.

DMC Algorytm DMC należy zaimplementować przyjmując pewne wstępne warunki:

- sygnałem wyjściowym obiektu regulacji jest pomiar temperatury T1 (adres 0),
- sygnałem wejściowym obiektu regulacji jest sterowanie grzałką H1 (adres 4),
- przez cały czas trwania regulacji wiatrak W1 (adres 0) jest włączony na 50% swojej mocy – pozwoli to na przyspieszenie opadania temperatury.

Należy oczywiście przyjąć logiczne ograniczenia na sygnał sterujący. Podobnie jak w przypadku poprzedniego projektu, tak i tutaj warto przyjąć jako wartość zerowego sterowania środkową wartość z przedziału dopuszczalnych jego wartości. Oznacza to, że należy przyjąć, że dla $u = 0$, grzałka grzeje z mocą 50%, dla $u = -50$ moc spada do 0%, a dla $u = 50$ moc rośnie do 100%. Wartość pomiaru wyjściowego jest niemal bez znaczenia ponieważ DMC wyznacza nową wartość sygnału sterującego w oparciu o uchyb – jedyne więc o czym należy pamiętać, to zachowanie tych samych jednostek przy wyznaczaniu wartości zadanych.

Ingerencja operatora Ponieważ płytka, na której implementowany jest regulator posiada ekran dotykowy, warto udostępnić jego możliwości operatorowi. Oznacza to, że operator musi mieć wpływ na działanie regulatora, np. poprzez ręczne manipulowanie wartościami sterowania, zmianę wartości zadanej lub choćby zmianę parametrów regulacji. Ten ostatni punkt jest łatwy do realizacji w przypadku algorytmu PID, natomiast przy DMC wymagałoby to wyznaczenia na nowo macierzy służących do wyznaczenia optymalnego przyrostu sterowania.

6.1 Przebieg ćwiczenia

Realizacja ćwiczenia ogranicza się do obsługi ekranu dotykowego, wyświetlacza oraz implementacji algorytmu regulacji. Kwestie związane z konfiguracją są już zrealizowane. W szczególności program początkowy zakłada:

- Komunikację (dwukierunkową) z użyciem UART1 między mikrokontrolerem a komputerem,
- Komunikację (dwukierunkową) z użyciem UART6 między mikrokontrolerem a obiektem regulacji – tutaj zaimplementowany został również protokół MODBUS,
- Obsługę wyświetlacza poprzez bieżące modyfikowanie wyświetlonego obrazu lub wykorzystanie przerwania HAL_LTDC_LineEvenCallback do modyfikacji wyświetlonego obrazu w czasie powrotu plamki,
- Obsługę ekranu dotykowego z użyciem okresowego odpytywania (szczególnego do decyzji studenta),

- Rozsądnie przydzielone priorytety poszczególnym przerwaniom – w programie nie występują zakleszczenia/zagłodzenia, a komunikacja z użyciem protokołu MODBUS działa bez błędów.

Student na tym etapie powinien posiadać wiedzę dotyczącą każdego z wymaganych aspektów, tj:

- Komunikacja z użyciem protokołu MODBUS,
- Przekazywanie danych do komputera z użyciem UART,
- Rysowanie na wyświetlaczu,
- Obsługa ekranu dotykowego,
- Implementacja algorytmu DMC,
- Zarządzanie czasem mikrokontrolera.

W razie wątpliwości zachęca się do skorzystania z instrukcji do poprzednich ćwiczeń.

W ramach tego ćwiczenia punkty będą przyznawane za realizację następujących zadań:

- Poprawna odpowiedź skokowa, tj. model regulatora DMC,
- Implementacja algorytmu DMC:
 - Poprawność implementacji – w tym jej omówienie,
 - Dobranie skutecznych parametrów – w tym omówienie procesu ich doboru,
- Interfejs użytkownika/operatora:
 - Intuicyjność – czy łatwo ocenić z daleka w jakim stanie znajduje się system,
 - Dokładność – czy można odczytać dokładny stan systemu,
 - Funkcjonalność – możliwość zmiany wartości zadanej poprzez wykorzystanie ekranu dotykowego oraz sterowanie ręczne grzałką, czyli sterowanie automatyczne i manualne,
- Obsługa sytuacji awaryjnych:
 - Błąd z komunikacją,
 - Odłączenie czujnika temperatury,
- Alarmy – osiągnięcie temperatury niekorzystnie wpływającej na działanie systemu,
- Kontekst – symboliczna reprezentacja obiektu regulacji,
- Ogólna jakość realizacji projektu – między innymi estetyka wykonania.

Podstawą do oceny jest zarówno podsumowanie w postaci sprawozdania (w tym opisu interfejsu użytkownika i sposobu jego implementacji), oraz krótka prezentacja działania. Ostatnie posłuży do sprawdzenia reakcji regulatora na stany awaryjne i sytuacje alarmowe. Oczywiście przetestowany zostanie również interfejs użytkownika i jego intuicyjność.

7 System operacyjny czasu rzeczywistego

Celem tego ćwiczenia jest zapoznanie się z możliwościami mikrokontrolerów rodziny STM32 pod kątem implementacji systemów operacyjnych czasu rzeczywistego. Wykorzystane wstawki assemblerowe pozwalają na wygodne przełączanie kontekstów oraz stosów między kolejnymi zadaniami, co jest podstawą pracy takiego systemu. W tym ćwiczeniu zadanie regulacji jest pominięte, jako że zastosowany system operacyjny czasu rzeczywistego jest mocno uproszczony, co powoduje, że implementacja wszystkich przydatnych (nie tylko koniecznych) funkcjonalności jest czasochłonna.

System operacyjny czasu rzeczywistego Systemy operacyjne czasu rzeczywistego (*Real Time Operating System – RTOS*) są to w uproszczeniu programy, które pozwalają na realizację pewnych zadań (*tasks*) z uwzględnieniem ograniczeń czasowych. Ponieważ zagadnienie to jest wyjątkowo obszerne, skupienie padnie bardziej na wsparcie mikrokontrolerów rodziny STM32 dla takich systemów oraz pokazanie, że implementacja prostego RTOS jest niewymagająca.

Dokładniejszą definicją RTOS, dobrze oddającą cel RTOS jest „system, którego poprawność działania nie zależy jedynie od poprawności logicznych rezultatów, lecz również od czasu, w jakim te rezultaty są osiągane”¹. Oznacza to, że poza poprawnym wynikiem działania potrzebne jest również zapewnienie, że wynik ten uda się otrzymać w odpowiednio krótkim czasie. Gdy system operacyjny uwzględnia wykonanie wielu zadań jednocześnie pojawia się potrzeba przyjęcia pewnej polityki dotyczącej kolejności ich wykonania. Systemy operacyjne czasu rzeczywistego można więc podzielić na trzy typy, w zależności od kosztów, wynikających z przekroczenia czasu wykonania zadania:

- Twarde (*hard*) – przekroczenie ograniczeń czasowych skutkuje katastrofalnymi stratami, przykładem mogą być systemy związane z bezpieczeństwem, np. działanie poduszek powietrznych w samochodzie – brak wystarczająco szybkiej reakcji może spowodować realne zagrożenie zdrowia lub życia, dlatego czas reakcji jest tutaj ważniejszy nawet niż precyzja działania. W takich systemach czas odpowiedzi jest ograniczony od góry i wiadome jest, że nie będzie dłuższy.
- Miękkie (*soft*) – przekroczenie ograniczeń czasowych skutkuje coraz większymi stratami wraz z oddalaniem się od oczekiwanej czasu odpowiedzi. Jako przykład można podać różne systemy służące do wizualizacji i interakcji, np. odtwarzacz DVD – opóźnienia powodują, że zadowolenie z takiego sprzętu maleje, lecz nie jest to w żaden sposób zagrożenie. W takich systemach nie ma gwarancji nieprzekraczalności terminów.
- Mocne (*firm*) – system pośredni pomiędzy twardym a miękkim. Przekroczenie ograniczenia czasowego powoduje, że wyniki przestają być użyteczne, ale sytuacja taka nie powoduje szkód. Jako przykład warto podać odtwarzacz muzyki – pominięcie pojedynczych dźwięków jest lepszym rozwiązaniem niż próba ich odegrania z opóźnieniem.

Warto pamiętać więc, że systemy czasu rzeczywistego nie mają za zadania działać „szybko”, lecz raczej deterministycznie szybko, tj. w określonym czasie od wystąpienia pewnego zdarzenia musi być gotowa odpowiedź na to zdarzenie.

¹P. Hambarde, R. Varma and S. Jha, ”The Survey of Real Time Operating System: RTOS,” 2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies, Nagpur, 2014, pp. 34-39.

Inną cechą systemu operacyjnego jest sposób w jaki zarządza zadaniami. Zadania mogą być realizowane jako odpowiedź na pewne zdarzenie, pod warunkiem, że nie wykonywane jest już zadanie ważniejsze. Podejście takie pozwala na rozpatrywanie sygnałów wejściowych jako trudne do przewidzenia, każde jest obsługiwane osobno i może przyjść w dowolnej chwili.

Innym pomysłem jest zastosowanie systemu operacyjnego z podziałem czasu – taki system rozpatrujemy w ramach tego ćwiczenia. Jest to taki system, który przydziela dostęp do procesora poszczególnym zadaniom na pewien czas, po czym oddaje go kolejnemu zadaniu. W ten sposób można wykonywać wiele zadań wspólnie (nie mylić z wykonywaniem równoległym, które wymaga istnienia wielu procesorów). Takie podejście pozwala na jednocześnie wykonywanie wielu zadań, które potencjalnie mogą nie mieć końca – takimi właśnie zajmować się będziemy w ramach tego ćwiczenia. Zadania te są jednak jedynie pozornie nieskończone, raczej należy rozumieć je jako wykonywane okresowo. Implementowane będą bowiem jako nieskończona pętla, w której realizowane jest kolejno właściwe zadanie, a następnie następuje oczekiwanie na ponowne wykonanie zadania. A więc nieskończoność tego zadania wynika właśnie z jego nieustannego powtarzania co pewien czas. Taka postać jest wygodna z punktu widzenia regulacji gdzie nowe wartości sterowania muszą być wyznaczane okresowo. Jeśli jednak jeden proces działa szybciej (jak symulowane stanowisko z poprzednich ćwiczeń), a drugi wolniej (jak stanowisko grzejaco-chłodzące), to zaplanowanie wykorzystania przerwań generowanych przez poszczególne timerы jest zadaniem co najmniej niełatwym (należy pamiętać, że timerы nie służą wyłącznie do wywoływania kolejnych iteracji algorytmu regulacji, ale także do obsługi np. protokołu MODBUS).

Prezentowane podejście jest oczywiście jednym z wielu, nie wspominając o gotowych implementacjach systemów czasu rzeczywistego, jak np. QNX, RT-Linux, FreeRTOS i wiele innych. Własna implementacja ma za zadanie pokazać, że sam pomysł systemu czasu rzeczywistego nie wymaga skomplikowanych operacji (pod warunkiem, że assembler nie jest nam obcy), nie mówiąc już o implementacji poszczególnych zadań takiego systemu.

7.1 Implementacja

Do implementacji systemu czasu rzeczywistego opartego na podziale czasu wykorzystanych zostało kilka funkcji assemblerowych. Funkcje te zostały napisane przez dra inż. Macieja Szumskiego i szerzej opisane są w jego książce „Systemy Mikroprocesorowe w Sterowaniu, Część I: ARM Cortex-M3”. Tutaj podany zostanie jedynie ogólny pomysł stojący za poszczególnymi funkcjami.

Przede wszystkim rozważany system obsługuje pewną stałą liczbę zadań nazywanych dalej taskami (spolszczenie angielskiego słowa *task* oznaczające zadanie). W tym celu globalnie zdefiniowana jest lista tych tasków jako:

```
task_table_t task_table[3]; // tablica taskow
```

gdzie typ `task_table_t` zdefiniowany jest jako

```
typedef struct { // structure of task_table
    uint32_t sp; // Task stack pointer
    int flags; // Task status flags
} task_table_t;
```

Jest to więc struktura zawierająca dwa atrybuty – wskaźnik na stos tego zadania oraz flagi z nim związane.

Tablicę z taskami należy zainicjalizować podając adres, gdzie ma zostać stworzony stos dla każdego z tych tasków, po czym wywołując funkcję `new_task(...)` stos ten jest tworzony pod wskazanym adresem. Utworzenie dwóch kolejnych tasków realizowane jest więc następująco

```
task_table[1].sp = (unsigned int) 0x20001700;
task_table[2].sp = (unsigned int) 0x20002700;
new_task(my_task_1, (uint32_t) &var1); // inicjalizuje Task1
new_task(my_task_2, (uint32_t) &var2); // inicjalizuje Task2
```

Drugim parametrem funkcji tworzącej nowy task jest wskaźnik na miejsce pamięci, gdzie zdefiniowane są parametry wywołania poszczególnych tasków. Powyżej została zastosowana pojedyncza zmienna będąca numerem taska – nic nie stoi jednak na przeszkodzie aby przekazać wskaźnik na tablicę lub strukturę. Należy zwrócić uwagę na fakt, iż nie został zainicjalizowany ani zdefiniowany task o indeksie 0. Jest to umyślne działanie, ponieważ taskiem tym będzie pętla główna programu zawarta w funkcji `main`. Stos tego taska jest umiejscowiony na końcu dostępnej pamięci (zgodnie z założeniami przyjętymi przez firmę ST), a więc rozpoczyna się pod adresem 0x200327B0 (wysoki adres wynika ze zwiększonego rozmiaru stosu mikrokontrolera).

Za przełączanie procesów odpowiada przerwanie SysTick (zgłasiane co 50 ms) zdefiniowane w pliku `rtos_asm.s`, gdzie kolejno blokowane są przerwania, testowane jest czy po obsłudze przerwania SysTick planowane jest powrócić do obsługi innego przerwania czy normalnego programu (a więc adres którego stosu zapisać) i w zależności od wyniku uaktualniany jest adres stosu i jego zawartość. Dalej uaktualniany jest adres stosu w strukturze (a więc zakończone zostało zapisywanie kontekstu), następuje przełączenie na nowy task i następuje odtwarzanie kontekstu dla nowego taska analogicznie jak w przypadku jego zapisu.

Same taski są zdefiniowane w osobnych plikach (każdy posiada odpowiadający mu plik nagłówkowy). Kod przykładowego taska (pomijając deklaracje i dołączanie nagłówków) wygląda następująco:

```
void my_task_1(void *data){
    float angle = 0.0f;
    float l = 50.0f;

    unsigned int varTask;

    varTask = *((unsigned int*)data);
    while(1){
        ++counter_task1;
        angle += dir1*0.001f;
        animation(angle,l);
    }
}
```

gdzie funkcja `animation(...)` służy do modyfikowania współrzędnych wierzchołków kwadratu, który następnie rysowany jest w przerwaniu wyświetlacza. Jak widać, ponieważ parametry przekazywane są jako `(void*)`, należy je zrzutować na odpowiedni typ, by wyciągnąć spod tego adresu właściwe wartości parametrów. Aby ułatwić implementację wszelkie zasoby są przypisane poszczególnym taskom i przerwaniom w taki sposób, aby zminimalizować ryzyko błędu wynikającego z dostępu do wspólnego zasobu. W szczególności zapisy dokonywane są przez poszczególne taski do osobnych fragmentów pamięci, a periferiale obsługiwane są osobno, każdy w dokładnie jednej funkcji obsługi przerwania.

Zaimplementowane są trzy kluczowe funkcje obsługi przerwań (poza SysTick-iem i dodatkowym przerwaniem o którym będzie mowa niżej):

- przepełnienie licznika TIM5,
- przerwanie zgłoszone przez ekran dotykowy,
- osiągnięcie zadanej linii przez sterownik wyświetlacza.

Przepełnienie licznika TIM5, działającego z częstotliwością 1 kHz ma na celu zastąpienie przerwania SysTick, które było dotychczas wykorzystywane przez HAL w celu odmierzania kwantów o długości 1 ms. Jest to więc zaledwie formalna zmiana.

Przerwanie zgłoszone przez ekran dotykowy powoduje odczyt stanu ekranu dotykowego i zmianę wartości zmiennych określających kierunek obrotu kwadratów. Jest to zrealizowane jako przycisk monostabilny, a więc póki wcisnięta zostaje jedna z połówek ekranu dotykowego, pół kierunek się zmienia.

Wyświetlacz w momencie, kiedy odświeża ostatnią linię obrazu wyświetlanego zgłasza przerwanie, w którym wykonywane jest czyszczenie ekranu (bardzo czasochłonne – warto unikać czyszczenia całości wyświetlacza), wyświetlenie kwadratów w obecnej pozycji oraz tekstu pokazującego ile iteracji pętli poszczególnych tasków zostało już wykonanych. Na koniec należy ponownie zażądać wyzwolenia tego przerwania pod tymi samymi warunkami – inaczej wywołanie tego przerwania będzie jednorazowe.

Jako dodatek zaimplementowana została obsługa jeszcze jednego przerwania – HardFault. Implementacja jest zrealizowana w pliku `rtos_asm.s`, lecz głównie wywołuje ona funkcję z pliku `rtos.c`. Ta ostatnia powoduje wyświetlenie niebieskiego ekranu, na którym białymi literami wyświetcone są niektóre kluczowe informacje oraz kilka słów pocieszenia dla użytkownika/programisty. Pojawienie się tego ekranu najczęściej oznacza błąd programowy – prawdopodobnie błąd z pamięcią, a nie jak to bywa w systemach Windows (z których rozwiązanie to zostało zaczerpnięte) ze sprzętem.

7.2 Zadanie do wykonania

Student ma wykonać następujące zadania:

- Poprawić obecną implementację, tak, aby kwadraty obracały się wyłącznie na dotknięcie ekranu:
 - lewa górną część – lewy kwadrat obraca się zgodnie z ruchem wskazówek zegara,
 - lewa dolna część – lewy kwadrat obraca się przeciwnie do ruchu wskazówek zegara,
 - prawa górną część – prawy kwadrat obraca się zgodnie z ruchem wskazówek zegara,
 - prawa dolna część – prawy kwadrat obraca się przeciwnie do ruchu wskazówek zegara.
- Prędkość obrotu kwadratów powinna być równa 1 obrót na sekundę i 2 obroty na sekundę dla odpowiednio lewego i prawego kwadratu.
- Zaimplementować trzeci task, który będzie realizował rysowanie znaku „+”, który będzie się nieustannie obracał zgodnie z ruchem wskazówek zegara z prędkością 1 obrotu na 5 s.
- Zrealizować powyższe w taki sposób, aby niezależnie od czasu trwania pojedynczej iteracji zadania prędkość była stała.